Università degli Studi di Milano

Laboratorio di Algoritmi e strutture dati

Roberto Cordone

– Anno Accademico 2022/23 –

Indice

Introduzione

Il corso di *Algoritmi e strutture dati* è associato a un sito sulla piattaforma Ariel:

`https://mgoldwurmasd.ariel.ctu.unimi.it`

alla pagina web gestita dal docente del modulo di teoria:

`http://users.mat.unimi.it/users/goldwurm/Algoritmi(Matematica)`

e a quella gestita dal docente del modulo di laboratorio:

`https://homes.di.unimi.it/cordone/courses/[ANNO]-algo/[ANNO]-algo.html`

dove `[ANNO]` va sostituito con l'anno di svolgimento del corso. Su questi siti sono pubblicati i materiali del corso.

In particolare, queste dispense relative al modulo di laboratorio accorpano tutti i materiali del modulo: i lucidi (sotto forma di figure), i testi degli esercizi implementativi svolti durante le lezioni e le soluzioni commentate organizzate per fasi successive, gli esercizi aggiuntivi proposti per lo studio in autonomia. Siccome integrano materiali diversi mantenendoli riconoscibili e siccome derivano dalla sbobinatura delle lezioni registrate nella primavera del 2020, nascono decisamente prolisse e potenzialmente cariche di incoerenze e piccoli errori. Verranno snellite e corrette via via nel tempo, con l'aiuto delle segnalazioni da parte degli studenti.

L'introduzione descrive brevemente di che cosa si occupa il modulo di laboratorio, com'è strutturato il corso, in che cosa consiste l'esame, che materiali didattici sono disponibili e che strumenti verranno impiegati.

Obiettivi e struttura del laboratorio

Il modulo di laboratorio tratta alcuni argomenti teorici, ma per lo più ha lo scopo di descrivere in dettaglio gli algoritmi e le strutture dati introdotti nelle lezioni di teoria. Si propone poi di legare l'aspetto teorico e quello pratico, dato che i due hanno forti relazioni, ma il passaggio dall'uno all'altro pone difficoltà tecniche importanti. Infine, intende illustrare l'applicazione di uno strumento pratico per realizzare algoritmi, che è il linguaggio di programmazione C.

In linea di principio, sarebbe augurabile dare per scontato un buon livello di conoscenza della programmazione. Siccome in pratica gli studenti hanno una preparazione molto variegata, si è ritenuto opportuno dedicare i primi capitoli a richiami di programmazione in C e riservare una parte di tutti gli altri capitoli a discutere anche dettagli tecnici sull'argomento. Nonostante ciò, si tenga ben presente un punto fondamentale, che ha un impatto sulla valutazione in sede di esame:

> Il corso di Algoritmi e strutture dati
> non è un corso di programmazione

Il modulo di laboratorio consiste in 48 ore, organizzate in 16 lezioni da 3 ore. In genere, c'è una pausa ad aprile per Pasqua e un'altra a maggio per esami. Il modulo si conclude intorno alla metà di maggio per consentire agli studenti di svolgere il progetto pratico e al docente di valutarlo entro la prima sessione di esami di giugno.

Il modulo di laboratorio si propone di

- descrivere in dettaglio alcuni algoritmi e strutture dati introdotti nelle lezioni teoriche
- legare le nozioni generali e astratte agli aspetti più tecnici e concreti
- realizzare in pratica gli algoritmi in un linguaggio di programmazione (nel caso specifico, il C)
  NON È UN CORSO DI PROGRAMMAZIONE!

Il modulo consiste in 48 ore che si sviluppano attraverso

- 11 lezioni da 3 ore
- una pausa per Pasqua
- 2 lezioni da 3 ore
- una pausa per esami
- 3 lezioni da 3 ore

Figura 1: Obiettivi e struttura del modulo di laboratorio

Modalità d'esame

L'esame è composto di due parti in successione: un progetto, relativo al modulo di laboratorio e valutato dal docente di laboratorio; una prova orale, relativa al modulo di teoria e valutata dal docente del modulo di teoria, che definisce e registra il voto complessivo. Le due parti si possono svolgere in appelli separati (le regole al riguardo sono definite dal docente di teoria, che è il titolare del corso).

Regole per il progetto

1. Il progetto deve essere svolto **individualmente**.
2. Il progetto deve essere svolto **interamente**.

Organizzazione

Circa un mese prima di ciascun appello d'esame, sulla pagina web del modulo di laboratorio viene pubblicata la descrizione di un problema. Il progetto consiste per prima cosa nel modellare questo problema concreto in termini di oggetti matematici astratti. Dopo di che, richiede di definire gli algoritmi opportuni per risolvere il problema e scegliere in base a questi le strutture dati opportune per rappresentare

gli oggetti astratti. Quindi, occorre realizzare algoritmi e strutture dati in un programma scritto nel linguaggio di programmazione C. Infine, si deve stendere una relazione. Per lo svolgimento del progetto sono concesse circa tre settimane.

Il testo fornisce indizi chiari, ma in genere non espliciti, sugli oggetti matematici per modellare il problema, sugli algoritmi per risolverlo e sulle strutture per rappresentare dati e risultati. In casi più rari, li indica esplicitamente: se si è tentati di seguire strade diverse, si consiglia di contattare il docente, dato che in genere i suggerimenti sono ben motivati.

Con il testo vengono forniti dati di esempio e soluzioni. Di solito, ottenere soluzioni diverse da quelle indicate è un indice di errore, mentre ottenere le stesse soluzioni è un segno confortante, ma non una garanzia di correttezza. Può accadere che esistano soluzioni corrette molteplici (in genere, questo è segnalato nel testo) o che le soluzioni pubblicate siano errate (in tal caso, segnalandolo verranno corrette al più presto).

L'esame si compone di due parti successive

1. presentazione di un progetto per il modulo di laboratorio
2. prova orale per il modulo di teoria

Il progetto cambia ad ogni appello ed è lo stesso per tutti gli iscritti

Richiede di

- modellare un problema "concreto"
- progettare un algoritmo per risolvere il modello
- scrivere un programma in C che realizza l'algoritmo
- stendere una relazione NON È UN MANUALE TECNICO!

Testo e data di scadenza sono pubblicati sulla pagina web del laboratorio

Entro la data di scadenza occorre spedire al docente di laboratorio

- il codice (in C) adeguatamente commentato
- una relazione (in PDF) che descriva:
  1. modello: descrizione motivata degli oggetti matematici usati per descrivere il problema
  2. algoritmo: descrizione e pseudocodice dell'algoritmo risolutivo, analisi della sua complessità computazionale (tempo e spazio)
  3. programma: aspetti tecnici legati al linguaggio (solo se importanti)

Figura 2: Struttura dell'esame

Valutazione

Durante la valutazione, relazione e codice vengono scorsi in parallelo, verificando che corrispondano nel modo più semplice e diretto, e i codici vengono compilati ed eseguiti su molti problemi diversi. Relazioni di lettura ostica o scorrelata al codice e codici che reagiscono a modo loro incagliano il processo e rendono più difficile mantenere un metro equanime di valutazione. Sia la relazione sia il codice vengono valutati in base a diversi criteri.

1. Completezza della relazione Il lettore della relazione non sa nulla del problema: vuole sapere com'è e come è stato risolto; sa tutto dei corsi di programmazione e di algoritmi. Quindi, la relazione soddisfa il criterio di completezza quando descrive sinteticamente:

1. il problema a livello concettuale (senza dettagli tecnici come i formati di ingresso e uscita);
2. il modello astratto adottato e i motivi di tale scelta;
3. gli algoritmi e le strutture dati adottate, i motivi di tali scelte e i costi temporali e spaziali (senza i dettagli trattati nel corso).

2. Correttezza della relazione La relazione soddisfa il criterio di correttezza quando, scorrendola in parallelo al codice, si verifica pagina dopo pagina che le operazioni e le analisi di complessità descritte corrispondono a quelle implementate e ai risultati della teoria.

Le analisi di complessità sono riferite alle quantità che descrivono la dimensione del problema, non a un generico indice n. Le semplificazioni che cancellano indici vanno motivate, e la validità dei motivi viene valutata.

3. Struttura della relazione La relazione soddisfa il criterio di struttura se l'esposizione è chiara, l'aspetto estetico apprezzabile e soprattutto ha un'organizzazione "top-down" parallela a quella del codice. Si legge una volta sola dal principio alla fine, senza dover saltare avanti e indietro fra sezioni diverse. Presenta prima il problema generale e poi i sottoproblemi particolari, prima i concetti astratti poi la realizzazione concreta. Fornisce un colpo d'occhio su ogni algoritmo e poi i dettagli organizzati in sezioni e sottosezioni. Allo stesso modo, il codice ha un programma principale con poche procedure di livello superiore, ciascuna contenente chiamate a poche procedure di livello inferiore, che a loro volta ne chiamano altre. Non occorre leggere subito le sezioni dettagliate, o conoscere il contenuto delle procedure di livello inferiore, per capire che cosa fa l'algoritmo. La relazione non descrive il codice istruzione per istruzione, come il codice non è una singola procedura con centinaia di istruzioni allo stesso livello.

L'analisi di complessità temporale e spaziale di ogni funzione deriva dalla descrizione delle operazioni, quindi la segue immediatamente e rispecchia anch'essa il codice. Ricomponendo la complessità delle singole parti, si ottiene quella dell'intero algoritmo.

È in genere inutile e dannoso diffondersi su dettagli tecnici di programmazione come i moduli del codice, le funzioni di libreria, i costrutti, i tipi delle variabili, ecc. . . , a meno che non condizionino la comprensione o le prestazioni dell'algoritmo.

Le sezioni **??** e **??** delle dispense riportano e commentano *schemi* o *tracce* di relazione, che non vanno intesi come *esempi* o *fac-simile*. Ci si accerti di conoscere le differenze tra questi vocaboli.

La relazione del progetto non è un manuale tecnico
La relazione del progetto non è un capitolo delle dispense
La relazione del progetto non è uno schema di relazione

4. Correttezza del codice Il codice soddisfa il criterio di completezza solo quando non contiene errori di programmazione o algoritmici.

Gli errori di programmazione più insidiosi sono l'uso di aree di memoria non allocate o lo sforamento di aree allocate, l'uso di variabili non inizializzate, lo sfruttamento di condizioni verificate dagli esempi pubblicati, ma non indicate nel testo del progetto. Sono errori insidiosi perché alcuni compilatori li compensano o perché i dati pubblicati li nascondono. Il compilatore e i dati usati per la valutazione invece li rivelano[1].

Esempi di errori algoritmici sono la scelta di un algoritmo scorretto o diverso da quello indicato nel testo (nei casi in cui questo avviene), l'uso di criteri di ordinamento diversi in parte o in tutto da quelli indicati nel testo, l'uso di algoritmi che risolvono un problema diverso da quello indicato nel testo.

5. Efficienza del codice Il codice soddisfa il criterio di efficienza quando usa gli algoritmi e le strutture dati più efficienti rispetto al tempo e allo spazio (privilegiando il primo in caso di conflitti). Esempi sono la scelta di algoritmi di complessità superiore, la scelta di strutture dati inadatte, il ricalcolo o la ricerca di informazioni già disponibili o che potrebbero essere disponibili usando strutture opportune.

6. Struttura del codice Il codice soddisfa il criterio di struttura quando è adeguatamente commentato, rispetta i formati di ingresso e uscita specificati nel testo del progetto, è ben strutturato e portabile.

Il rispetto dei formati comporta di caricare i dati da file attraverso una linea di comando (rispettando l'eventuale ordine dei file, se sono più d'uno) e di stampare i risultati a video come descritto nel testo. Il rispetto rigoroso del formato (maiuscole e minuscole, a capi, ecc. . .) permette di confrontare le soluzioni con quelle pubblicate: basta redirigere le stampe da video a un file di testo accodando alla linea di comando `>` e il nome del file, e confrontare il file ottenuto con quello pubblicato con gli strumenti offerti dal sistema operativo. I codici che non rispettano il formato vengono modificati a mano dal docente durante la valutazione e questo comporta una penalizzazione.

La portabilità su altri compilatori e sistemi operativi comporta l'uso dello standard adottato nel corso, cioè il C89. Nella valutazione vengono quindi penalizzate:

- la mescolanza fra parte dichiarativa ed esecutiva delle procedure;
- le dichiarazioni di contatori nei cicli;
- le dichiarazioni di vettori statici con lunghezza espressa da una variabile (che dovrebbero essere invece vettori dinamici).

Sarebbe lungo discutere qui i motivi tecnici della scelta di questo standard. Il principale è che semplifica l'analisi di complessità per gli studenti e semplifica la ricerca di errori per il docente.

[1] A puro titolo informativo (nessuna pubblicità), i progetti sono valutati compilandoli con Microsoft Visual Studio Express, che segnala o fa emergere durante l'esecuzione anche errori ignorati da `gcc`. Codici che funzionano sulle macchine degli studenti spesso non lo fanno sulla mia: c'è sempre dietro un errore.

Conviene quindi compilare il codice con le opzioni `-std=c89` e `-Wall` e rimediare agli avvertimenti così ottenuti, perché possono aiutare a scoprire errori sfuggenti. Può essere utile anche l'opzione `-pedantic`, anche se genera avvertimenti un po' eccessivi.

La valutazione terrà conto di:

1. relazione:
   a. completezza, cioè descrizione di ogni aspetto rilevante del progetto
   b. correttezza, cioè corrispondenza fra teoria, relazione e codice
   c. struttura, cioè organizzazione e aspetto estetico
2. codice:
   a. correttezza, cioè assenza di errori sintattici e semantici
   b. efficienza, cioè uso di strutture dati e algoritmi appropriati
   c. struttura, cioè organizzazione in moduli, funzioni, blocchi, portabilità e adeguatezza dei commenti

Figura 3: Criteri di valutazione dell'esame

Spedizione

Relazione e codice vanno raccolti in un file compattato (in formato .ZIP o .RAR o .GZ) e spediti all'indirizzo di posta elettronica del docente di laboratorio entro la mezzanotte del giorno indicato nel testo del progetto. Prima di spedire, si verifichi che l'allegato non contenga file eseguibili o ambigui, perché il server di posta cancella questi allegati. Si riceverà un messaggio di conferma, più o meno immediato (da pochi minuti dopo al giorno dopo). In caso di ritardi nella conferma della ricezione o nella pubblicazione del progetto, conviene contattare il docente di laboratorio: potrebbero esserci stati problemi tecnici.

Programma del modulo di laboratorio

Per prima cosa, discuteremo che cosa siano un problema, un algoritmo, un programma, usando termini abbastanza precisi, ma informali, dato che non si tratta di un corso di informatica teorica. Poi introdurremo il concetto di complessità, cioè di costo di esecuzione di un algoritmo, asintotico e nel caso pessimo. Faremo alcuni richiami di programmazione in C, in teoria non necessari, ma in pratica graditi a un buon numero di studenti. Per non essere fastidiosi, si cercherà di non ripetere il corso di programmazione a velocità accelerata, ma di approfondire aspetti trattati più in fretta nei corsi di programmazione di base. Quindi, vedremo strutture dati e algoritmi elementari, che raccolgono insiemi di dati in maniera organizzata (per esempio, vettori o *record*). A queste strutture sono tipicamente associati algoritmi di uso comune: per esempio, algoritmi di ordinamento per i vettori, algoritmi di inserimento e cancellazione per le liste, e così via. Poi ci saranno le strutture più avanzate, come grafi, alberi semplici e di ricerca, *heap*, ecc. . . e gli algoritmi più sofisticati.

Il modulo tratta i seguenti argomenti:

- Algoritmi e programmi
- Definizione e valutazione di complessità asintotica
- Richiami di programmazione in C
- Strutture dati e algoritmi elementari:
  - vettori, tabelle, stringhe e *record*
  - liste concatenate
  - algoritmi di ordinamento
- Strutture dati e algoritmi avanzati:
  - grafi: algoritmi di visita e componenti connesse
  - alberi di ricerca: algoritmi di costruzione e modifica
  - strutture per *union/find*: gestione di partizioni
  - code con priorità
  - algoritmi di *divide-et-impera*
  - algoritmi di programmazione dinamica
  - algoritmi *greedy*

Figura 4: Programma del modulo di laboratorio

Materiali di riferimento

I materiali disponibili sul sito Ariel e sulle due pagine web sono dispense, lucidi, codici, ecc.... Non ci sono testi ufficiali, ma la figura 5 riporta un elenco di testi da consultazione per approfondire argomenti.

Per quanto riguarda gli algoritmi e le strutture dati:

- dispense dal sito del docente di teoria
  `http://users.mat.unimi.it/users/goldwurm/Algoritmi(Matematica)/`
- lucidi dal sito del docente di laboratorio
  `https://homes.di.unimi.it/cordone/courses/[ANNO]-algo/[ANNO]-algo.html`

Ovviamente, è più semplice una ricerca con nome e cognome

Testi utili per consultazione:

- T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Introduzione agli algoritmi e strutture dati*, McGraw-Hill, 2010.
  (in inglese, *Introduction to algorithms*, 3rd edition)
- A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, 1974.
- C. Demetrescu, I. Finocchi, G. Italiano, *Algoritmi e strutture dati*, McGraw-Hill, 2004.
- A. Bertossi, *Algoritmi e strutture dati*, UTET Libreria, 2000.

Figura 5: Materiali utili per gli algoritmi

Per la programmazione esistono materiali in abbondanza in rete, ma la Figura 6 riporta la pagina web di un mio vecchio corso di programmazione in C, con lucidi, materiali ed esercizi, e alcuni testi comunemente usati nei corsi di base. Qualsiasi materiale di buon livello va bene.

Poiché non studieremo teoricamente, ma scriveremo programmi, occorre un compilatore C. In laboratorio sono disponibili macchine Linux con il classico compilatore `gcc`. Io uso un terminale testuale su un portatile Windows, con un compilatore `gcc` tradotto per Windows. Usare un compilatore con interfaccia grafica non è necessario, ma non è certamente vietato. Si noti che compilatori diversi producono messaggi diversi (più o meno chiari) e generano programmi eseguibili diversi. Purtroppo, come come già specificato[2], programmi errati possono funzionare comunque se compilati con alcuni compilatori ed eseguite su alcune macchine, e bloccarsi o terminare stampando solo in parte i risultati su altre.

[2] AGGIUNGERE RIFERIMENTO

Per la programmazione in C, ogni testo o materiale va bene

Esempi (reperibili in rete):

- lucidi ed esercizi dal sito del docente di laboratorio
  `https://homes.di.unimi.it/cordone/courses/2015-prog/2015-prog.html`
- K. N. King, *Programmazione in C*, W. W. Norton & Company, 2008. (in inglese, *C Programming: A Modern Approach*, 2nd edition)
- A. Kelley, I. Pohl, *C. Didattica e Programmazione*, Pearson, Italia, 2004. (in inglese, *A book on C*, 4th edition)
- B. W. Kernighan, D. M. Ritchie, *Linguaggio C*, Pearson, Italia, 2004. (in inglese, *The C Programming Language*, 2nd edition)

Figura 6: Materiali utili per la programmazione in linguaggio C

È sufficiente un compilatore C

- in ambiente Linux, `gcc` è installato su tutte le distribuzioni
- in ambiente Windows,
  - durante le lezioni userò `Dev-C++` senza interfaccia
  - per valutare il progetto userò `Visual Studio Express`
  - `MinGW` include una versione (datata) di `gcc`
    `https://www.cs.colorado.edu/~main/bgi/cs1300/`

Le versioni con interfaccia grafica sono consigliabili solo a utenti esperti
Durante il corso non le useremo (ovviamente non sono vietate)

Figura 7: Strumenti di laboratorio

Capitolo 1

Problemi, algoritmi, programmi, linguaggio C

Questo capitolo fornisce un'introduzione con informazioni di contesto, utili per capire il taglio generale del modulo di laboratorio. Non sono argomenti necessari per portare a termine il progetto, e sono solo un sostegno secondario per l'orale, il cui riferimento sono le dispense del modulo di teoria. Fanno però eccezione tre argomenti essenziali, che sono l'impostazione *top-down* del progetto e della descrizione di algoritmi, la struttura generale dei codici C e il meccanismo di compilazione per progetti costituiti da file multipli.

1.1 Algoritmi e problemi

Un *algoritmo* è uno strumento per risolvere problemi matematici (vedi Figura 1.1). È uno strumento formale, cioè la risoluzione avviene meccanicamente senza che l'algoritmo "comprenda" il problema. Il corso in generale, e il modulo di laboratorio in particolare, hanno lo scopo di insegnare a progettare gli algoritmi, cioè a trovare una maniera meccanica per ricavare la soluzione di un problema dato, ma anche di descrivere questo modo ad altri esseri umani in maniera comprensibile. Inoltre, si vuole insegnare ad analizzare un algoritmo, cioè, data la descrizione di un algoritmo (realizzata da chiunque), a capire

1. se l'algoritmo è corretto, cioè se funziona su tutti i problemi della famiglia per la quale è stato progettato; [1]

2. quanto costa eseguire l'algoritmo, definendo opportunamente il costo (anche detto *complessità*) in termini di spazio e di tempo.

Per definire questi concetti occorrerà un po' di formalismo, ma non trattandosi di un corso di informatica teorica, ne daremo una formalizzazione alla buona.

Un *problema* è una domanda su un sistema descritto in maniera quantitativa (vedi Figura 1.2). Con "quantitativa" non si intende semplicemente un insieme di numeri, ma di oggetti matematici in genere, dunque relazioni, funzioni, operazioni permesse o vietate. Una *soluzione* è la risposta a una domanda di questo genere, anch'essa costituita da oggetti matematici.

[1] Questo va messo dopo: Magari la tua funzione 99% delle volte è scorretto però è già sicuramente più utile che non un algoritmo che non si sa quante volte corretto e quante No

Algoritmo è uno strumento formale per risolvere un problema,
cioè un meccanismo che non richiede la comprensione del problema

In questo corso si vuole sviluppare la capacità di

- progettare algoritmi: passare dalla descrizione di un problema alla descrizione di un algoritmo che lo risolve
- analizzare algoritmi: passare dalla descrizione di un problema e di un algoritmo alla dimostrazione
  1. che l'algoritmo risolve il problema (correttezza)
  2. di quanto costa risolvere il problema con l'algoritmo (complessità)

Per farlo, occorre impostare con un po' di rigore le definizioni di

- problema
- risolvere
- algoritmo
- costo

Faremo solo qualche accenno, non una formalizzazione completa

Figura 1.1: Nozione di algoritmo

Quando si affronta un problema concreto (e tendenzialmente l'algoritmica è matematica applicata, anche se nulla vieta di avere algoritmi di matematica pura), questo non è fatto di oggetti matematici. Allo stesso modo, le azioni pratiche che vengono compiute per risolvere il problema concreto non sono oggetti matematici. Quindi, la risoluzione algoritmica di un problema è sempre preceduta da una fase di *modellazione* e sempre seguita da una fase di *interpretazione* dei risultati. Queste due fasi non sono in genere banali e sono molto importanti. È quindi giusto contestualizzare richiamando questi aspetti, anche se il corso è centrato sul passaggio da problema astratto a soluzione astratta, e discuteremo solo occasionalmente la modellazione, limitandoci a questo accenno iniziale all'interpretazione.

Problema è una domanda su un sistema descritto quantitativamente

- insiemi di grandezze numeriche, fisiche, logiche...
- relazioni fra grandezze (ordine, appartenenza, funzioni...)
- operazioni (regole di trasformazione del sistema)

Dunque è un insieme di oggetti matematici

Soluzione è la risposta, descritta quantitativamente, alla domanda

Dunque un secondo insieme di oggetti matematici legato al primo

Se si affrontano problemi concreti

- prima il problema concreto va modellato in un problema astratto
- poi la soluzione astratta va interpretata in una strategia concreta

Nel corso non approfondiremo questi aspetti, ma faremo esempi

Figura 1.2: Una definizione informale di problema

La Figura 1.3 raccoglie alcuni esempi di tipici problemi, più o meno pratici, che ammettono una risoluzione algoritmica.

Problema della primalità Questo problema consiste nel prendere un numero (per esempio, il numero di Mersenne $2^{17} - 1$) e chiedersi se sia un numero primo oppure no. La risposta potrà essere sì o no. Sostanzialmente, questo problema è un numero, e la sua soluzione è un valore logico, o booleano, che tradizionalmente viene descritto come 0 (no, falso) o 1 (sì, vero).

Problema del percorso minimo Questo problema consiste nel sapere in quanto tempo si può arrivare da un punto a un altro (per esempio, da qui a Piazza Duomo). La descrizione del problema richiede una serie di informazioni. Per prima cosa, richiede una rete stradale sensata (non tutta la Lombardia, ma Milano), ridotta in forma di modello, cioè di descrizione approssimata. Quale sia la descrizione appropriata è un problema di per sé. La rete va descritta dal punto di vista topologico, come insieme di connessioni fra posizioni nello spazio, ma anche dal punto di vista metrico, attraverso i tempi di percorrenza delle strade. Tutto ciò forma un insieme di oggetti matematici, che possiamo vedere come il problema. La soluzione, cioè la risposta alla domanda, è un numero (per esempio, 15 minuti).

Problema dei nuclei familiari Questo problema chiede di individuare i nuclei familiari degli abitanti di Milano. Si può modellare questo problema attraverso un insieme, che contiene i residenti, e una relazione binaria di parentela, che collega due residenti che appartengono alla stessa famiglia. La relazione è binaria per semplicità. La soluzione del problema è una partizione dell'insieme dei residenti in sottoinsiemi disgiunti, che complessivamente danno luogo all'insieme dei residenti. I sottoinsiemi corrispondono ai nuclei familiari (questa è la fase di interpretazione).

Problema dell'ordinamento alfabetico Questo problema consiste nel ricevere le 5 parole "zebra", "alce", "renna", "gnu" e "antilope" e indicare in che ordine si trovano nel dizionario. Il problema è definito da un insieme di oggetti. La soluzione è definita dallo stesso insieme, ma con una relazione d'ordine sovraimposta, quindi è una sequenza, ovvero permutazione, anziché un semplice insieme.

1. *Il numero* $2^{17} - 1$ *è un numero primo?*
   - La domanda è definita dal numero $2^{17} - 1$
   - La risposta è *Sì* oppure *No*
2. *In quanto tempo si arriva da qui in Piazza Duomo?*
   - La domanda è definita dalla rete stradale (topologia e tempi)
   - La risposta è un numero (ad es., in minuti)
3. *Quali sono i nuclei familiari residenti a Milano?*
   - La domanda è definita dall'insieme dei cittadini e delle loro relazioni di parentela e convivenza
   - La risposta è una collezione di sottoinsiemi di cittadini
4. *In che ordine sono disposte le parole* zebra, alce, renna, gnu, antilope *sul dizionario?*
   - La domanda è definita da una sequenza di parole
   - La risposta è una permutazione della sequenza data

Figura 1.3: Esempi di problemi

In tutti i casi, problema e soluzione sono insiemi di oggetti matematici e il passaggio dall'uno all'altra è mediato da un algoritmo.

A questo punto è opportuno modificare leggermente la definizione di problema per essere più precisi. Domandarsi se 2^{17} sia un numero primo è un problema fra infiniti problemi simili, perché su ogni numero intero maggiore di 1 ci si può porre la stessa domanda. Ciascuno di questi problemi ha una propria soluzione, ma sono molto simili fra loro. Ha senso allora definire *istanza* la domanda su un singolo specifico insieme di oggetti matematici, conservando la parola soluzione per la risposta alla specifica domanda che costituisce un'istanza (vedi Figura 1.4).

Con *problema*, invece, da ora in poi descriveremo il legame fra istanze e soluzioni, quindi, in sostanza, una funzione (vedi Figura 1.5). Ogni elemento I di un opportuno insieme $\mathcal{I}_P$ di istanze è legato da una relazione alla propria specifica soluzione S, che sta in un insieme $\mathcal{S}_P$ di soluzioni. Il problema è la corrispondenza fra i due insiemi, quindi una funzione da oggetti matematici a oggetti matematici.

Capita spesso di porre la stessa domanda su sistemi diversi

- 2 è un numero primo?
- 3 è un numero primo?
- 10^{300} è un numero primo?
- $2^{17} - 1$ è un numero primo?

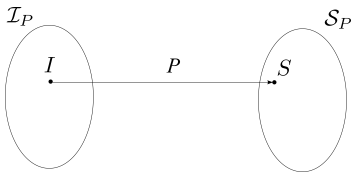
Definiremo

- istanza I la descrizione quantitativa dello specifico sistema su cui si pone una domanda
- soluzione S la descrizione quantitativa della specifica risposta a quella domanda

e riserveremo invece la parola

- problema P alla relazione che lega ogni istanza alla sua soluzione, ovvero un insieme di coppie istanza-soluzione

Figura 1.4: Istanze



- Ogni istanza I rappresenta i dati di uno specifico problema e l'insieme di tutte le istanze del problema è un dominio $\mathcal{I}_P$
- Ogni soluzione S rappresenta i risultati di uno specifico problema e l'insieme di tutte le soluzioni del problema è un dominio $\mathcal{S}_P$
- Il problema P è la corrispondenza fra istanze I e soluzioni S

$$P : \mathcal{I}_P \to \mathcal{S}_P$$

Quindi una funzione che lega insiemi matematici

Figura 1.5: Problemi e funzioni

Questo comincia a trasformare silenziosamente l'oggetto del nostro studio in qualcosa di completamente diverso. Consideriamo il problema della primalità e costruiamo una tabella (vedi Figura 1.6) che nella prima colonna contiene i numeri naturali maggiori di 1 e nella seconda le corrispondenti soluzioni del problema della primalità. Ovviamente, la tabella è infinita. Se codifichiamo le soluzioni in zeri e uni, come si fa classicamente, questa tabella diventa una funzione da numeri naturali a numeri naturali. Quindi, il problema della primalità, che è la corrispondenza fra istanze e soluzioni del problema stesso, è una funzione da quasi tutti i numeri naturali a un piccolo sottoinsieme dei numeri naturali. In generale, un problema è una generica funzione da numeri naturali a numeri naturali.

Definizione informale: *Il numero naturale n è primo?*

Definizione formale:

$\mathcal{I}_P$ (istanze)	$\mathcal{S}_P$ (soluzioni)
...	...
5	Sì
6	No
7	Sì
8	No
...	...

Introduciamo la classica corrispondenza *Sì* $= 1$ e *No* $= 0$

- le istanze sono tutti i numeri naturali positivi: $\mathcal{I}_P = \mathbb{N}^+$
- le soluzioni sono i numeri naturali 0 e 1: $\mathcal{S}_P = \{0, 1\}$

Il problema della primalità è una funzione $\mathbb{N}^+ \to \{0, 1\}$

Vedremo che tutti i problemi sono funzioni $\mathbb{N} \to \mathbb{N}$*!*

Figura 1.6: Esempio: il problema della primalità

Questo è un fantastico isomorfismo, che ha introdotto un fortissimo sviluppo nella matematica ai principi del '900, cioè l'idea che, invece di maneggiare oggetti matematici astratti, sia possibile codificarli in oggetti manipolabili meccanicamente (chiamiamoli simboli), che possono essere numeri, ma anche lettere, simboli matematici (come il prodotto cartesiano, l'intersezione, i quantificatori logici, ecc...). In pratica, l'idea è che si può scrivere il problema in un linguaggio opportuno e poi tradurlo in una forma particolarmente vincolata, cioè come una sequenza lineare di simboli (vedi Figura 1.7). Se la prima cosa in matematica è un'ovvietà, la seconda non è già più ovvia (si pensi all'armamentario di apici, pedici, frazioni, ecc... delle espressioni matematiche). L'uso di espressioni lineari serve a fornire il problema a una macchina, per la quale l'uso di due dimensioni introduce molte complicazioni pratiche. Se invece trasformiamo il problema (e la soluzione) in un file di testo, questo ha il fantastico vantaggio di rendere il meccanismo risolutore di problemi che stiamo cercando un meccanismo che manipola sequenze di simboli in sequenze di simboli, qualcosa di apparentemente molto più limitato che non risolvere problemi matematici.

La necessità di codificare istanze e soluzioni in simboli implica la necessità di definire un *alfabeto*, cioè l'insieme dei simboli utilizzabili (vedi Figura 1.8. Un esempio

La definizione informale è inadatta a una macchina

Occorre quindi

- codificare l'istanza affinché una macchina possa manipolarla
- codificare la soluzione affinché una macchina possa produrla
- definire il meccanismo di trasformazione da una codifica all'altra

La codifica di un'istanza e della corrispondente soluzione consiste nel

1. sostituire gli oggetti matematici con simboli elementari tratti da un alfabeto finito (di lettere, cifre, segni, ...)
2. raccogliere i simboli in una sequenza finita (stringa)

In pratica, un'istanza viene convertita in un file di testo, il meccanismo risolutivo lo manipola opportunamente e produce un file di testo con la soluzione

Figura 1.7: Codifica di un problema

è l'alfabeto che usiamo comunemente, magari arricchito dai caratteri di punteggiatura. Nei computer è di uso comune l'alfabeto ASCII da 128 caratteri, o quello esteso da 256. Però si puòanche pensare a un alfabeto binario, se si punta alla massima semplicità. In effetti, la scelta dell'alfabeto è secondaria, perché ogni alfabeto è convertibile meccanicamente in ogni altro. Qualunque carattere ASCII a 256 varianti si può tradurre in 8 bit, perché $2^8 = 256$ e si può quindi prendere un testo scritto con i caratteri ASCII e tradurlo in un testo scritto in binario sostituendo ogni carattere del primo con una sequenza di 8 caratteri del secondo. Viceversa, si può prendere un testo scritto in binario e convertirlo in un testo in ASCII, sostituendo ogni sequenza di 8 caratteri del primo con un carattere del secondo. È ovvio che il testo binario è più lungo, ma non è enormemente più lungo: il fattore di conversione è dall'alfabeto $\mathcal{A}_1$ all'alfabeto $\mathcal{A}_2$ è una costante:

$$\left\lceil \log_{|\mathcal{A}_2|} |\mathcal{A}_1| \right\rceil$$

che dipende dal numero di caratteri dei due alfabeti. Siccome la formula contiene un logaritmo, il fattore costante non è enorme anche se le cardinalità dei due alfabeti sono molto diverse tra loro. Se anche si passa da un alfabeto ricco di simboli, come quelli che gli esseri umani tendono a usare, all'alfabeto più povero di tutti, quello binario[2], i testi non si allungano troppo, né in ingresso né in uscita. E quindi manipolarli per risolvere il problema non richiederà un tempo lunghissimo per la semplice codifica dei dati e decodifica dei risultati.

Abbiamo detto che i problemi si possono vedere come funzioni sui numeri naturali. Ecco un esempio (vedi Figura 1.9: vogliamo ordinare un gruppo di lettere (per esempio, il gruppo `ab` rimane `ab`, mentre il gruppo `eb` diventa `be`, `vm` diventa `mv`, ecc... Nell'esempio usiamo gruppi di due lettere per semplicità, ma potrebbero

[2]Esiste anche un alfabeto *unario*, costituito da un solo simbolo, per il quale il significato della sequenza è interamente racchiuso nella sua lunghezza. Questo ha impatti rilevanti (anche se non rilevantissimi) sulla teoria, ma si esce dai confini del corso.

Gli alfabeti possibili sono infiniti:

- gli esseri umani (occidentali) usano spesso l'alfabeto ASCII (o l'extended ASCII), che è composto da 128 (o 256) simboli
- i computer usano l'alfabeto binario $\mathcal{A} = \{0, 1\}$

Ogni alfabeto $\mathcal{A}_1$ è traducibile meccanicamente in ogni altro alfabeto $\mathcal{A}_2$

- se $|\mathcal{A}_1| \geq |\mathcal{A}_2|$, si fissa una corrispondenza biunivoca convenzionale fra ogni simbolo di $\mathcal{A}_1$ e una stringa di $\left\lceil \log_{|\mathcal{A}_2|} |\mathcal{A}_1| \right\rceil$ simboli di $\mathcal{A}_2$

In particolare, l'alfabeto ASCII si può convertire in binario per

- trasmettere problemi ai computer
- ricevere soluzioni dai computer

Figura 1.8: Quale alfabeto?

essere sequenze di un numero qualunque di lettere. Se codifichiamo il dato `eb` in binario, otteniamo un filotto di zeri e uni, che però è anche un numero naturale. La soluzione è un altro filotto di zeri e uni, che però è un numero naturale. Tutti i problemi per i quali le istanze sono codificabili in simboli con un alfabeto finito, e quindi in binario, e quindi i numeri naturali, e lo stesso avviene per le soluzioni, sono isomorfi a funzioni da numeri naturali e numeri naturali.

L'aspetto critico è che, mentre alcune funzioni sono definite e si sa come calcolarle, altre sono definite, ma non si sa come calcolarle. Un algoritmo non è altro che un modo con il quale si prende un'istanza dall'insieme delle istanze, si manipola e si ottiene una soluzione nell'insieme delle soluzioni. Questa soluzione non è generica, ma è quella corrispondente all'istanza secondo il problema dato. Quindi, una volta definito l'alfabeto con cui si codificano i dati e le soluzioni e il cosiddetto *modello computazionale*, cioè la macchina che fa le operazioni su queste sequenze di simboli elementari (zeri e uni, o caratteri ASCII), un algoritmo introduce una corrispondenza fra istanze e soluzioni che ha la caratteristica aggiuntiva di essere calcolabile. Che cosa significa questo?

Un *algoritmo* (vedi Figura 1.11) è una sequenza finita di operazioni elementari che sono meccaniche. Queste operazioni trasformano le stringhe di simboli in ingresso (le istanze) in stringhe di simboli in uscita (le soluzioni). I simboli appartengono tutti a un alfabeto. Gli algoritmi hanno tre caratteristiche fondamentali. La prima è che le singole operazioni sono *elementari*, cioè è dato un elenco finito di operazioni eseguibili. Ad esempio, si possono fare somme, sottrazioni, confronti, ecc... La seconda caratteristica fondamentale è che queste operazioni sono *meccaniche*, cioè deterministicamente fissate dai dati. Non ci sono componenti di casualità, né di intuizione, né di volontà o libero arbitrio. La terza caratteristica è che queste operazioni formano una *sequenza finita*. In programmazione capita di scrivere dei cicli che non finiscono mai. Sono *programmi*, perché godono delle prime due caratteristiche fondamentali, ma non sono algoritmi. Se si vogliono calcolare le soluzioni delle istanze, la terminazione in tempo finito è necessaria.

Le dispense di teoria illustrano un *modello computazionale*, chiamato *macchina RAM*, che indica le operazioni consentite e le regole meccaniche che determinano i

Consideriamo un problema non numerico: ordinare un gruppo di lettere

Istanze		Soluzioni	
codifica ASCII	codifica binaria	codifica binaria	codifica ASCII
ab	1100001 1100010	1100001 1100010	ab
eb	1100101 1100010	1100010 1100101	be
vm	1110110 1101101	1101101 1110110	mv
df	1100100 1100110	1100100 1100110	df
...	...	...	...

Una volta codificate in binario

- le istanze sono isomorfe a un sottoinsieme di $\mathbb{N}$ $(12\,514, 13\,026, \ldots)$
- le soluzioni sono isomorfe a un sottoinsieme di $\mathbb{N}$ $(12\,514, 12\,645, \ldots)$

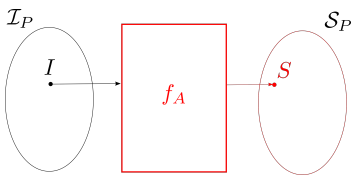
I problemi sono isomorfi a funzioni da numeri naturali a numeri naturali

In generale non sappiamo come calcolarle!

Figura 1.9: Problemi e funzioni su numeri naturali

Algoritmo è una sequenza finita di operazioni elementari meccaniche che trasforma stringhe di simboli in stringhe di simboli tratti da un alfabeto

Quindi un algoritmo è una funzione fra insiemi di stringhe/numeri



Per definire un algoritmo bisogna prima aver definito

- l'alfabeto di lavoro con cui si compongono le stringhe
- le operazioni elementari eseguibili, cioè un modello computazionale (lettura e scrittura di simboli, operazioni su numeri, valutazione di condizioni logiche, ecc...)

Nel modulo di teoria sarà presentata la macchina RAM

Figura 1.10: Algoritmi

Un algoritmo risolve un problema quando trasforma la stringa che codifica ogni istanza nella stringa che codifica la soluzione corrispondente

$$S_A(I) = S(I) \text{ per ogni } I \in \mathcal{I}_P$$

Un algoritmo computa la funzione che lega istanza e soluzione

Algoritmi e problemi sono legati, ma non identici

- un problema è una funzione
- un algoritmo è una funzione più il modo di computarla

In particolare, esistono

- problemi privi di algoritmi risolutivi
- problemi con algoritmi risolutivi diversi che hanno costo diverso

$$\begin{aligned} A: &\quad 7 + 4n + 2n^2 + n^3 \quad (3 \text{ somme e } 5 \text{ prodotti}) \\ A': &\quad 7 + n[4 + n(2 + n)] \quad (3 \text{ somme e } 2 \text{ prodotti}) \end{aligned}$$

$$S_A(\cdot) \equiv S_{A'}(\cdot) \text{ ma } A \neq A'$$

Figura 1.11: Algoritmi e problemi

risultati di tali operazioni. In laboratorio, usiamo un altro modello computazione, cioè il processore programmato in linguaggio C. A rigore, essendo un oggetto concreto, non è esattamente un modello computazionale, ma gli somiglia molto. Questi due modelli sono simili: ciascuno dei due è un po' più libero dell'altro per certe cose e un po' più limitato per altre. Commenteremo via via alcune differenze, che per lo più sono poco significative.

Si è detto che un algoritmo trasforma la stringa istanza in una stringa soluzione e un problema è una corrispondenza fra istanze e soluzioni. Verrebbe da pensare che siano la stessa cosa. Non lo sono perché l'algoritmo calcola la soluzione, cioè spiega in che maniera passare dagli argomenti al risultato. Esistono funzioni definite, ma che non si sa come calcolare. D'altra parte, ci sono funzioni che si possono calcolare in tanti modi. Analogamente, ci sono problemi che non hanno algoritmi risolutivi, e ci sono problemi che hanno molti algoritmi risolutivi. In questo corso, vedremo problemi che hanno due, quattro, o anche più algoritmi risolutivi. Per esempio, se si considera una classica espressione algebrica, come

$$7 + 4n + 2n^2 + n^3$$

che alle scuole superiori si insegna a considerare come una funzione, questa è una funzione, ma anche un algoritmo, perché indica esplicitamente come calcolare il valore della funzione: si prende il numero 7, poi si prende l'argomento e si moltiplica per 4 e si somma il risultato al 7, poi si prende l'argomentoe si eleva al quadrato e si moltiplica per 2 e si somma al risultato parziale precedente, e così via.

Questo esempio consente di introdurre il concetto di *costo*. Calcolare questa funzione "costa" 3 somme e 5 prodotti (che sono meno espliciti: 4 per n, 2 per n per n e n per n per n). In tempi meno fortunati, in cui bisognava calcolare a mano, o al limite con il regolo o le tabelle dei logaritmi, fare meno calcoli era

estremamente apprezzato. Infatti, ci si rese conto che la stessa funzione si può calcolare raccogliendo diversamente i suoi termini, cioè come

$$7 + n[4 + n(2 + n)]$$

La funzione è perfettamente identica, ma l'algoritmo di calcolo è diverso, e con esso il costo di calcolarla: oltre alle solite tre somme, occorrono solo due prodotti (il primo n per l'espressione fra parentesi quadre e il secondo n per l'espressione fra parentesi tonde). Questo esempio mostra visivamente come possano esistere algoritmi con prestazioni molto diverse per risolvere lo stesso problema. Uno degli scopi del corso è capire quanto costa un algoritmo e qual è l'algoritmo migliore per un problema. Questo conclude la carrellata sugli argomenti più teorici.

1.2 Processori e programmazione

Consideriamo ora gli strumenti tecnologici che possiamo usare per risolvere problemi matematici in modo algoritmico. Vogliamo realizzare un algoritmo in maniera da non eseguirlo a mano, anche se originariamente al-Khwārizmī insegnava ai contemporanei a fare le quattro operazioni aritmetiche in maniera più veloce usando le cifre arabe anziché quelle romane. A partire dall'800 ci si è resi conto che è possibile creare macchine che eseguono istruzioni formali in maniera meccanica, è possibile trasformare stringhe di simboli in altre stringhe di simboli, specialmente se entrambe sono binarie. Non solo è possibile farlo, ma è possibile farlo in maniera "universale": se ancora Pascal nel '600 progettava una macchina (la pascalina) che faceva le somme automaticamente girando una manovella, nell'Ottocento Babbage cominciava a pensare a una macchina che fosse programmabile per fare calcoli diversi ogni volta, e nel '900 Turing, Von Neumann e altri riuscirono effettivamente a realizzarla. L'idea è costruire una macchina che realizzasse fisicamente un meccanismo di calcolo universale, che sulla base di un programma, cioè di istruzioni codificate in maniera rigorosa, imponesse una relazione fissa fra le stringhe di bit fornitele in ingresso e le stringhe di bit ottenute in uscita. Nel momento in cui si riesce a codificare l'algoritmo in istruzioni espresse in un linguaggio simbolico, questo consente di codificare in binario non solo i dati e i risultati, ma anche le istruzioni stesse, e quindi di avere un mondo che vive nel binario (vedi Figura 1.12).

Perché concentrarsi su un mondo binario? Perché i sistemi fisici reagiscono più facilmente a livelli alti e bassi di tensione, di carica magnetica, a polarizzazioni opposte, piuttosto che a quantità analogiche. I computer analogici sono esistiti, e avevano anche dei vantaggi, ma sono nettamente più scomodi per le applicazioni più generali.

Allora l'idea è codificare le istruzioni simboliche che formano l'algoritmo anch'esse in un linguaggio binario, detto *linguaggio macchina*, per ottenere un *processore*. La Figura 1.13 ne dà una rappresentazione astratta, come una "scatola", che riceve un'istanza codificata in binario e restituisce una soluzione codificata in binario sulla base di un programma codificato in binario. La cosa più interessante è che, se il programma risolve il problema della primalità, quindi forza il processore a restituire 1 o 0 in uscita secondo che il numero fornito in ingresso sia primo o no, cambiando il programma la stessa macchina potrebbe indicare se il numero in ingresso è un quadrato perfetto o no, oppure che ne calcola la radice quadrata, o che mette in ordine alfabetico una data sequenza di parole. L'uscita infatti può tranquillamente essere una stringa più o meno lunga di bit.

Questo dispositivo è potentissimo, ma usarlo è tutt'altro che banale, perché richiede all'utente di fornire i dati in binario e riconvertire le soluzioni dal binario

Come si realizza un algoritmo?

Come si costruisce una macchina che trasforma simboli in simboli?

I processori trasformano stringhe binarie in stringhe binarie,
eseguendo istruzioni espresse come stringhe binarie

Codificando la macchina stessa come stringa binaria si ottiene
un meccanismo di calcolo universale, cioè
si possono realizzare diversi algoritmi sulla stessa macchina

Linguaggio macchina è la codifica binaria delle istruzioni per il processore

Figura 1.12: Linguaggio macchina

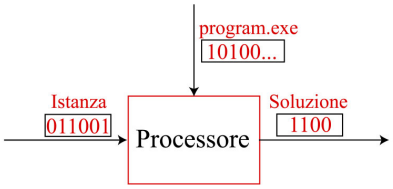


Figura 1.13: Processore

(e fin lì ci si arriva), ma soprattutto di programmare in binario, che è una cosa assolutamente contraria alla mente umana. Il linguaggio macchina è ostico.

Inoltre, la singola macchina avrà il suo linguaggio, strutturato in dettaglio sullo specifico funzionamento interno della macchina , sulle sue strutture e sul modo in cui le operazioni vengono eseguite. Ci si rese ben presto conto che sarebbe stato meglio avere un linguaggio simbolico, che avesse quanto meno un'espressione a parole per ciascuna operazione elementare eseguibile dalla macchina stessa. Questo linguaggio viene comunemente definito assembly. Per esempio, volendo sommare due numeri, la macchina deve prima caricarli in opportuni registri, poi esegure la somma e infine salvare il risultato in una terza locazione. Invece di scrivere la lista di uni e zeri che forza la macchina a eseguire queste quattro operazioni (si veda la seconda colonna della tabella centrale in Figura 1.14), si possono scrivere quattro operazioni testuali: `LOAD A`, `LOAD B`, `ADD` e `STORE C`. Non è chiarissimo, ma con un po' di addestramento si intuisce che il significato è: caricami un dato in un primo registro, caricane un altro in un altro registro, fai la somma del contenuto dei due registri e infine memorizza il risultato in una terza locazione. Invece di ricordare a memoria o consultare su un manuale le stringhe binarie, si può scrivere un programma in *assembly* e affidare la conversione in binario a un altro programma. Il linguaggio *assembly* è più leggibile, ma ha anche un altro vantaggio: se cambia il processore, non occorre cambiare il programma o il linguaggio *assembly*, purché la nuova macchina possa eseguire lo stesso insieme di operazioni. La macchina può essere diversa, ma deve avere un sommatore e dei registri indicizzati in un certo modo, in cui si possono caricare i dati e salvare i risultati. Se la macchina ha questa struttura, non importa che fisicamente l'operazione desiderata si scriva `0110` o `1010`. L'importante è che l'operazione si chiama `LOAD` e produca il caricamento di un certo dato nel posto in cui si esegue una certa operazione. Di conseguenza, è possibile imparare un linguaggio e tenerlo valido per tutte le macchine della stessa famiglia. Non sono tutte le macchine, ma tutte quelle pensate con uno stesso disegno di base. Finché non cambia la famiglia di processori, si tiene lo stesso linguaggio *assembly*.

Cambiando invece la struttura della macchina, introdurremo un nuovo linguaggio *assembly*, ma nel frattempo abbiamo risparmiato molta fatica. Da un punto di vista formale, stiamo chiudendo nella "scatola" rossa della Figura 1.15 non solo il processore, ma anche una prima scatola a sinistra, che converte un'istanza fatta di simboli scritti in caratteri ASCII in un'istanza binaria, una seconda "scatola" a destra, che riconverte all'indietro la soluzione binaria in caratteri, e una terza "scatola", la più importante, che converte il programma scritto in linguaggio *assembly* in un programma scritto in linguaggio macchina binario, dunque eseguibile. Chi rimane fuori della scatola rossa apparentemente vede una macchina che riceve ed emette testo ed è programmabile con testo, quindi una macchina molto più amichevole di quella che in realtà è nascosta all'interno. L'ideale sarebbe non dover mai aprire la "scatola" rossa e goderne tutti i vantaggi.

Il principio si può approfondire, dato che anche un codice in *assembly* non è amichevolissimo. Se per sommare due numeri occorrono quattro operazioni, figuriamoci per fare cose più complicate. Ben presto, quindi, si inventarono i *linguaggi ad alto livello* (vedi Figura 1.16), nei quali una singola operazione corrisponde a più operazioni elementari legate da un obiettivo comune. Caricare i numeri a e b, sommarli e copiare il risultato in c sono operazioni diverse, ma l'obiettivo è unitario, e scrivere `c = a + b` è molto più comodo che scrivere `LOAD A`, `LOAD B`, ecc... Si tratta di un'unità logica, anche se non funzionale. I linguaggi ad alto livello hanno lo scopo di accorpare istruzioni di base in istruzioni più comprensibili. Così facendo, in generale introducono una relazione "uno a molti", fra un'istruzione ad alto livello e molte istruzioni *assembly*, dunque non così immediata e meccanica come la relazione "uno a uno" fra l'istruzione *assembly* e quella binaria. Nonostante ciò,

Il linguaggio macchina ha evidenti svantaggi

- è ostico per un essere umano
- ogni processore ha il suo linguaggio specifico

Linguaggio assembly: per ogni istruzione macchina elementare definisce un'istruzione simbolica corrispondente

1. si scrive un testo che descrive il programma (codice o listato del programma)
2. un programma (assembler) traduce il codice in linguaggio macchina

LOAD A	1010...00101
LOAD B	1010...01101
ADD	0011...00101
STORE C	1001...10101

Vantaggi:

- il linguaggio assembly è più leggibile per un essere umano
- si può usare un solo linguaggio assembly per diverse macchine purché le macchine abbiano le stesse istruzioni elementari (basta cambiare traduttore)

Figura 1.14: Linguaggio assembly

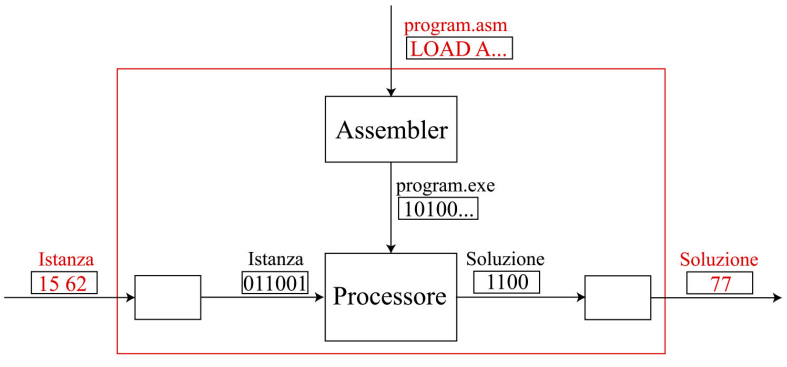


Figura 1.15: Assembler

ci sono programmi che sono in grado di fare questa conversione, che è una conversione di testo in testo, e che si chiamano *compilatori*. Ci sono compilatori per ogni linguaggio.

Svantaggi

- anche operazioni banali richiedono più istruzioni
- il programma è molto lungo
- il significato del programma non è evidente

Linguaggio ad alto livello: definisce una sola istruzione simbolica per ogni sequenza di istruzioni assembly con uno scopo comune

- si scrive un testo che descrive il programma (codice o listato del programma)
- un programma (compiler) traduce il codice in linguaggio macchina (di solito passando per l'assembly)

Operazione	Alto livello	Assembly
Somma a e b	$c = a + b$	LOAD A LOAD B ADD STORE C

Figura 1.16: Linguaggio ad alto livello

La struttura questo punto si complica ulteriormente ma l'idea la stessa. Si veda la Figura 1.17: la "scatola" rossa "parla" nel linguaggio C, cioè riceve un programma in C e dei dati in formato testo, e restituisce dei risultati in formato testo. La manipolazione dei dati fino a ottenere il risultato avviene sulla base di un programma che non è scritto in linguaggio naturale, ma in un linguaggio che gli somiglia abbastanza da consentire a chiunque, con un minimo di istruzione e allenamento, di servirsene. Sarebbe molto buono che la "scatola" rossa venisse chiusa e non ne uscisse mai nulla, cioè che il compilatore, l'*assembler*, i convertitori di ingresso e uscita e il processore si comportassero perfettamente come desiderato, ma purtroppo non si riesce per motivi tecnologici. Facciamo due esempi, legati a quello riportato nella Figura 1.17. Se sommiamo 15 e 62, il risultato è 77. Ma se sommiamo 2 miliardi e 2 miliardi, il risultato dovrebbe essere 4 miliardi, ma se il processore non dedica abbastanza spazio di memoria a conservare il singolo numero intero da poter contenere il numero 4 miliardi (per esempio, perché dedica 32 bit, ovvero cifre binarie, ai numeri interi positivi e negativi, e quindi li può rappresentare solo fra $-2^{31}+1$ e 2^{31}, allora il risultato dell'operazione è indeterminato, oppure viene determinato alla buona con l'aritmetica modulare e si ottiene un numero negativo. Questo succede normalmente. Chi usa la "scatola rossa" come se fosse davvero capace di sommare numeri interi, scrivendo `c = a + b` deve sapere che se a e b sono numeri grandi, può rischiare di ottenere un risultato scorretto. Questo perché il meccanismo ha dei limiti. Un altro esempio (provato personalmente): se si sommano due numeri reali $a + b$ è uguale a $b + a$, ma se si usa un processore per farlo questo non è sempre vero. Infatti, i numeri reali sono rappresentati con una precisione finita che dipende dai numeri stessi, e le due somme con i termini scambiati possono dare risultati

diversi, ovviamente nelle ultime cifre decimali. Se è importante rispettare la proprietà commutativa, bisogna prestare particolare attenzione. Questo non significa che la "scatola" rossa sia un oggetto inutile, ma che bisogna essere consapevoli del fatto che esistono limiti da non oltrepassare. Questo libera il 99% del tempo per pensare effettivamente ad algoritmi e oggetti matematici, mantenendo un 1% di sorveglianza sui problemi più tecnologici. Il modulo di laboratorio è dedicato a mostrare in che maniera si cerca faticosamente di astrarre il più possibile le strutture dati e gli algoritmi, in maniera tale che si possano usare come oggetti matematici, tenendo presenti i limiti che presentano. È per questo che in un corso di algoritmi è opportuno astrarre, ma anche conoscere il contenuto della macchina per affrontare in maniera adeguata i problemi che ne possono derivare.

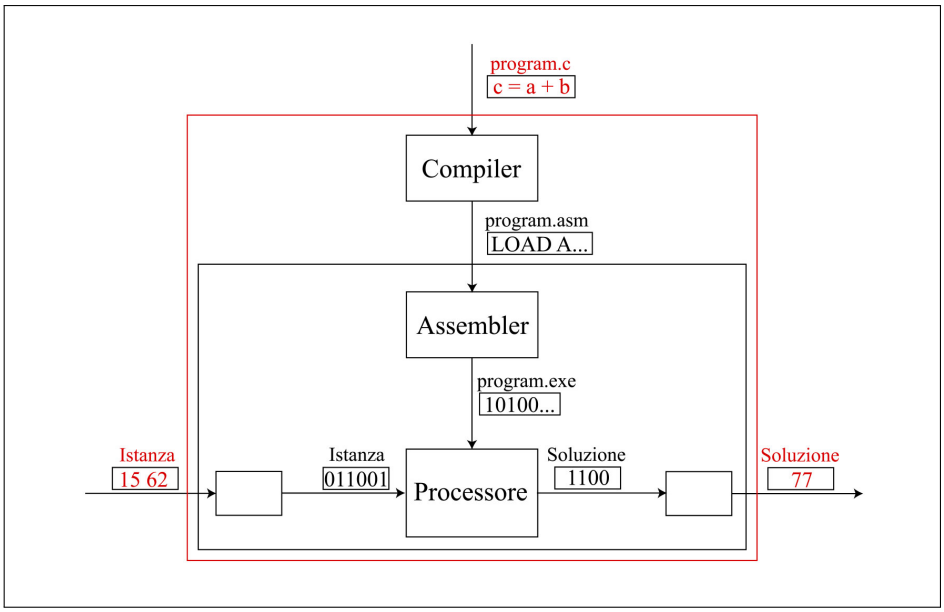


Figura 1.17: Compilatore

Riassumendo, avremo a che fare con un processore che riceve da un compilatore (eventualmente passando attraverso un *assembler*) un programma che pare scritto in un linguaggio testuale, ma che in realtà è scritto in alfabeto binario, e che maneggia delle stringhe che paiono scritte in caratteri ASCII, ma che in realtà sono stringhe di bit.

1.3 Il progetto top-down di algoritmi

Si è detto che un problema ammette in genere molti algoritmi risolutivi, più o meno costosi. È anche vero che lo stesso algoritmo può essere realizzato in un linguaggio di programmazione in molti modi diversi, alcuni migliori di altri. Quindi, è opportuno acquisire uno stile anche nella scrittura dei programmi[3].

[3]L'uso oscillante dei termini "programma" e "algoritmo" andrà corretto nel rivedere le dispense, ma in parte deriva dalla stretta relazione che li lega. Un programma è una sequenza di istruzioni elementari deterministiche, un algoritmo è la stessa cosa, ma in più termina certamente. Quindi, speriamo che i nostri programmi siano sempre algoritmi. D'altra parte, per programma si intende spesso il testo concreto, e come algoritmo l'aspetto più concettuale. È chiaro che le due cose devono assolutamente andare di pari passo

Come è opportuno realizzare un algoritmo in un linguaggio di programmazione? Nello stesso modo usato nel risolvere problemi matematici. Se si cerca di dimostrare un teorema, tipicamente ci si appoggia su lemmi, cioè su enunciati più semplici, spesso dimostrati da altri, che consentono di non "riscoprire la ruota" ogni volta, ma di procedere. In campo algoritmico succede lo stesso (vedi Figura 1.18: se si vuole risolvere un problema, conviene ricondurlo a sottoproblemi più semplici, le cui soluzioni siano magari già note o comunque realizzabili, e che possano poi essere ricombinate e manipolate in modo da ottenere la soluzione del problema di partenza. Siccome i sottoproblemi potrebbero essere troppo complicati a loro volta, si può procedere riconducendoli a sottosottoproblemi. È chiaro che questo dà luogo a una struttura gerarchica ad albero, con una radice che è il problema originale, un primo livello di sottoproblemi figli, un secondo livello di sottosottoproblemi, ecc... finché si arriva a foglie, che corrispondono a problemi talmente elementari che si dà per scontato che siano risolubili. Quando avviene questo? Quando esiste già del codice per risolverli, vuoi nel linguaggio vuoi in qualche libreria scritta da qualcuno in precedenza. Giunti a questi problemi elementari, non si va oltre. Convenzionalmente, l'albero è rovesciato, cioè si immagina che abbia la radice in alto e poi scenda sino alle foglie, che stanno in basso.

Questo approccio alla soluzione del problema non è necessario, ma è fortemente opportuno che sia perfettamente rispecchiato nella scrittura del codice. Si può dimostrare il teorema dei quattro colori partendo dagli assiomi della matematica degli insiemi, ma non ha senso farlo. Di solito, si procede sfruttando risultati già noti. Analogamente, volendo scrivere del codice per risolvere un problema, non ha senso scrivere le istruzioni elementari. Conviene scrivere istruzioni che risolvono sottoproblemi, combinarle, e poi, qualora quelle istruzioni non esistano già perché non corrispondono a funzioni elementari, realizzarle a posteriori.

Questo è noto come *approccio top-down*, perché parte dalla radice, dal problema da risolvere, che sta in cima, e termina nelle foglie dell'albero, i problemi elementari, giù in basso.

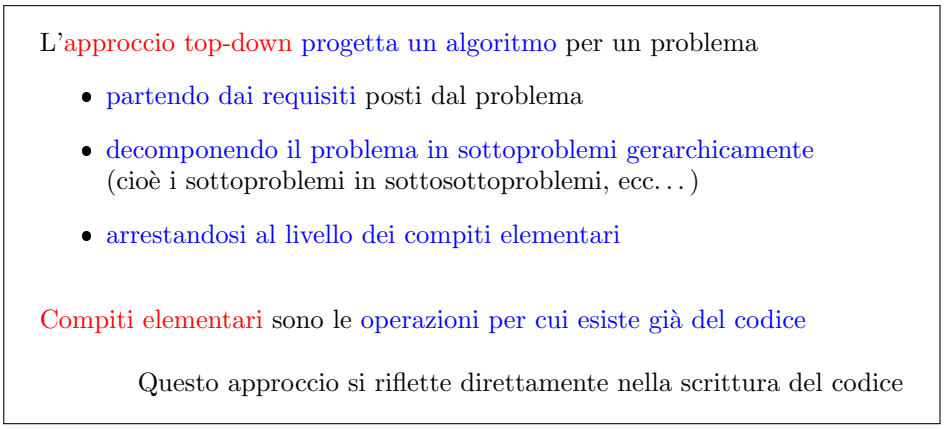


Figura 1.18: Approccio top-down

Vediamo un esempio di assoluta banalità, proprio per sottolineare gli aspetti fondamentali dell'approccio. Il problema si potrebbe risolvere in poche decine di righe di codice, ma farlo in un altro modo sottolinea l'importanza della struttura e rende più facile applicare lo stesso metodo a problemi più sofisticati. Ne approfitteremo anche per introdurre e discutere caratteristiche del linguaggio di programmazione C che spesso vengono trascurate nei corsi di programmazione di base per l'urgenza di chiarire le parole chiave e le istruzioni. In particolare, sottolineeremo il fatto che

i programmi C hanno una vera e propria struttura gerarchica, che può essere usata per aiutare a risolvere i problemi.

Siccome il C è un linguaggio molto flessibile, buona parte di questa struttura non è imposta dalle regole di sintassi, ma deriva da un impegno del programmatore. Esattamente come *ex nimia licentia nascitur tyrannus* (Cicerone), dall'usare eccessiva flessibilità nella programmazione derivano i bachi e le sofferenze. Usando semplici e banali regole, che sono intuitive se si ragiona in termini di riduzione di problemi a sottoproblemi, si può ottenere il risultato soffrendo di meno. Il ripasso di C ha non solo lo scopo di richiamare cose in parte già note, ma anche di sottolineare questo principio di base.

L'esempio è ispirato al classico primo programma che si scrive in C, cioè quello in cui si stampa sul video `Hello, world!` ("Ciao a tutti!"). Lo faremo in modo un po' più sofisticato del solito (vedi Figura 1.19).

Vogliamo stampare non "Hello, world!", ma un saluto generico indicato dall'utente all'interno di una cornicetta testuale realizzata con un carattere indicato dall'utente. È un problema leggermente più sofisticato. È veramente un problema, cioè una corrispondenza fra istanze e soluzioni, una funzione da numeri naturali a numeri naturali? Se si considerano il saluto e il carattere della cornice come una sequenza di simboli, quindi una sequenza di bit, quindi un numero naturale, e la stampa sul video (completa di a capi) come una sequenza di simboli, quindi di bit, quindi un numero naturale, la risposta è sì. Quindi, stiamo cercando il modo di calcolare questa inconsueta funzione. Il problema affrontato in questa forma diventa complicato, ma possiamo ridurlo a sottoproblemi, e vedremo che si semplifica molto rapidamente.

La decomposizione di un problema in sottoproblemi è in buona parte una questione di gusto e di intuizione, ma questo non significa che non ci siano decomposizioni migliori di altre. Vediamone una e vediamo perché è un'ottima decomposizione (Figura 1.19). Per prima cosa, l'utente deve indicare il carattere da usare per la cornice. Questo è un problema di per sé: ricavare il carattere desiderato dall'utente per la cornice. Analogamente, il secondo sottoproblema è ricavare dall'utente il saluto che bisogna stampare. Sono sottoproblemi diversi, perché la soluzione del primo è un carattere, quello del secondo è una sequenza di caratteri, però si somigliano molto. Questo è indice del fatto che i loro figli, cioè i sottosottoproblemi che ne risultano, potrebbero essere identici. Se questo avvenisse, risparmieremmo fatica. Il terzo sottoproblema è quello principale: stampare il saluto con la cornice. Come giudicare se la decomposizione è buona? Per prima cosa, i tre compiti sono chiaramente distinti fra loro: hanno dati diversi, risultati diversi, sono indipendenti, sono descrivibili chiaramente.

Scendiamo al livello inferiore. Consideriamo l'acquisizione del carattere desiderato. Potrebbe essere già un problema elementare, ma lo decomponiamo ancora in due mettendosi nei panni dell'utente, che lancia il comando e si trova di fronte a un programma che sta fermo in attesa del carattere richiesto e non sa che cosa stia succedendo. Per non richiedere troppa competenza all'utente, sarebbe opportuno che il programma gli spiegasse che cosa fare. Quindi, decomponiamo il problema dell'acquisizione in un'istruzione da dare all'utente, che è una stampa, e in un'effettiva acquisizione del carattere richiesto (vedi Figura 1.20). Per il saluto, avremo la stessa struttura: la stampa di un'istruzione (la situazione è diversa, ma è sempre una stampa) e l'acquisizione di una sequenza di caratteri. La stampa del saluto è molto più sofisticata e interessante, e richiede un po' di riflessione. Vogliamo stampare una cornice con dentro il saluto. Se dovessimo farlo a mano o con un terminale grafico, probabilmente divideremmo la cornice dal saluto e poi divideremmo la cornice in due righe orizzontali (una sopra e una sotto) e due verticali (una a sinistra e

Supponiamo di voler far stampare sul video un saluto scelto dall'utente incorniciato con un carattere scelto dall'utente

Questo compito si può scomporre in

1. acquisire dall'utente il carattere desiderato per la cornice (`'*'`)
2. acquisire dall'utente il saluto desiderato (`Hello, world!`)
3. stampare il saluto incorniciato

```
******************
* Hello, world!  *
******************
```

Intuiamo che la scomposizione è corretta perché

- ogni sottocompito è chiaro e indipendente dagli altri
- alcuni sottocompiti sono simili, se non identici (acquisizione di uno o più caratteri da tastiera)

Lo stesso avviene ai livelli inferiori

Figura 1.19: Esempio di approccio top-down (1)

una a destra) e in mezzo. Sono cinque operazioni ragionevolmente elementari. Non avendo un terminale grafico, dobbiamo fare i conti col modello computazionale, cioè con quello che la macchina è in grado di fare. La macchina stampa a video su terminale testuale, partendo dall'alto a sinistra e arrivando in basso a destra riga per riga. Può solo andare da sinistra a destra e poi scendere nella riga successiva. Non può tornare indietro[4]. Quindi la decomposizione considerata è impossibile: bisogna seguire un ordine diverso, meno naturale per noi, ma più naturale per lo strumento a disposizioe.

La decomposizione che consideriamo (vedi Figura 1.20) apparentemente parte stampando la parte alta della cornice, ma questo richiede di conoscere quanto è larga la cornice, cioè di quanti caratteri è composta. Questo dipende dalla lunghezza del saluto: `Hello, world!` consiste di 13 caratteri, a cui vanno sommati due spazi sui lati e i caratteri della cornice sinistra e destra, in tutto 17 caratteri. Se il saluto fosse `Ciao`, basterebbero $4 + 2 + 2 = 8$ caratteri. Quindi, la larghezza della cornice va correlata alla lunghezza del saluto. Primo sottoproblema è quindi calcolare la larghezza della cornice, poi si ha la stampa della cornice superiore, quindi andare a capo, stampare un pezzo di cornice sinistra, uno spazio vuoto, stampare il saluto, un altro spazio, un pezzo della cornice di destra, andare a capo e infine stampare tutta la cornice inferiore.

Perché questo è convincente? Per prima cosa, sono tutte operazioni chiare e ben distinte. Poi sono operazioni che si ripetono: l'operazione b e l'operazione j nell'elenco sono identiche, come pure lo sono la c e la i, la d e la h, la e e la g. Questo è il segno che contraddistingue una buona decomposizione, che consente di riciclare molto codice e risolvere tante volte gli stessi sottoproblemi, eventualmente in condizioni leggermente diverse.

[4] A rigore, può muoversi verso sinistra sulla stessa riga, ma cancellando quello che ha scritto attraverso la stampa di un carattere *backspace* (`'\b'`).

Infatti, le due acquisizioni si scompongono in

a comunicare all'utente che cosa gli si chiede di fornire

b acquisire l'informazione richiesta
(il carattere `cornice` o la riga `saluto`)

La stampa del saluto incorniciato si scompone in

a determinare la `larghezza` della cornice (lunghezza di `saluto` più 4)

b stampare la cornice superiore
(stampare `larghezza` volte il carattere `cornice`)

c andare a capo

d stampare la cornice laterale sinistra (stampare il carattere `cornice`)

e stampare uno spazio separatore (stampare uno spazio)

f stampare il saluto (stampare la stringa `saluto`)

g stampare uno spazio separatore (stampare uno spazio)

h stampare la cornice laterale destra (stampare il carattere `cornice`)

i andare a capo

j stampare la cornice inferiore
(stampare `larghezza` volte il carattere `cornice`)

I compiti ripetuti abbondano!

Figura 1.20: Esempio di approccio top-down (2)

A questo punto possiamo chiederci se fornire istruzioni all'utente sia un compito elementare o da decomporre ulteriormente in sottoproblemi (per esempio, stampare una parola, o addirittura una lettera per volta). Questo dipende dalla macchina. Supponiamo per semplicità di avere una *libreria*, cioè concretamente un file `advio.c` (*advanced input-output*), il cui scopo è consentire alcune operazioni di lettura e scrittura avanzate rispetto alle funzioni primitive del linguaggio C. Questa libreria fornisce un elenco di operazioni (riportate nella Figura 1.21), fra le quali la possibilità di leggere da tastiera delle righe di testo. Quindi, se l'utente scrive il saluto sulla tastiera, siamo in grado di acquisirlo. Allo stesso modo, siamo in grado di leggere dalla tastiera singoli caratteri (per esempio, il carattere della cornice). Siamo in grado di valutare la lunghezza di una stringa di caratteri, e quindi, una volta letto il saluto, di sapere quanti caratteri contiene. Siamo in grado di stampare stringhe di caratteri, dunque le istruzioni, e singoli caratteri ripetuti un numero qualsiasi di volte, quindi la cornice superiore, quella inferiore e gli spezzoni delle cornici laterali, nonché gli spazi bianchi fra cornice e saluto. Siamo anche in caso di andare a capo[5]

È interessante osservare che l'uso di librerie con operazioni, ovvero sottoproblemi risolti, un po' più sofisticate di quelli del linguaggio base, si indica comunemente come *approccio bottom-up*. Questo approccio è complementare dell'approccio *top-down* fin qui perseguito. Quello consisteva nel prendere un problema complessivo e decomporlo in sottoproblemi fino ad arrivare alle foglie. L'approccio *bottom-up* consiste nel partire dalle foglie, cioè dalla capacità di fare alcune cose con lo strumento disponibile, e combinare alcune foglie in un nodo a livello superiore, ovvero costruire una funzione che fa operazioni più sofisticate. Per esempio, ima funzione che legge un'intera riga di testo, o che stampa un certo numero di caratteri identici anziché uno solo. Poi si possono combinare fra loro nodi per ottenere altri nodi a un livello superiore. La domanda è: chi indica quali foglie combinare e quale funzione realizzare? Lo vedremo nel Capitolo **??**

Ci sono quindi due modi complementari di progettare un algoritmo: partire dal problema e decomporlo fino ad arrivare alle frattaglie risolubili, oppure partire dalle frattaglie risolubili e metterle insieme. L'approccio *bottom-up* richiede una forte intuizione, perché bisogna sapere dove si sta andando per non costruire nodi a caso e produrre funzioni che risolvono i problemi che non servono a niente. Sia chiaro che i due modi non sono opposti o alternativi, ma complementari: quando si applica l'approccio *top-down*, può capitare che, arrivati abbastanza in basso, ci si renda conto che alcuni sottoproblemi ancora aperti sono facilmente ricostruibili dal basso. Allora ci si può fermare quei nodi, partire dalle foglie e raggiungerli con l'approccio *bottom-up*, chiudendo l'albero di risoluzione. Un secondo aspetto è che dopo aver risolto molti problemi diversi, ci si rende conto che alcuni sottoproblemi tornano spesso, e avere già pronti gli algoritmi per risolverli sarebbe conveniente. Quindi, si interrompe l'approccio *top-down* e si dedica un certo tempo a costruire librerie di funzioni che non hanno uno scopo immediato e specifico, ma in base all'esperienza si intuisce che prima o poi serviranno effettivamente. Si può pensare che la libreria `advio.c` sia stata costruita in questo modo, e che ora si possa sfruttarla per interrompere l'approccio *top-down* al secondo livello anziché procedere ulteriormente.

Riassumiamo ciè che abbiamo fatto (vedi Figura 1.22). Abbiamo diviso il problema (che definiamo `main` usando il nome della funzione che lo risolve) in tre unità logiche, o sottoproblemi: acquisire la cornice, acquisire il saluto, stampare il saluto. Anche a questi sottoproblemi ho dato il nome delle funzioni che li risolvono. L'acquisizione della cornice richiede di stampare una stringa per dare istruzioni all'utente e

[5]Chi conosce il C sa già che questo equivale a stampare il carattere `'\n'`, ma non si è voluto introdurre dettagli tecnici in questa introduzione.

Quali sono i compiti elementari dove si arresta la scomposizione?
Forse la lettura e la scrittura di caratteri singoli?

Se supponiamo di avere una libreria `advio.h` che fornisca procedure per

- leggere da tastiera righe di testo
- leggere da tastiera singoli caratteri
- valutare la lunghezza di una stringa di caratteri
- stampare a video stringhe di caratteri
- stampare a video caratteri ripetuti
- andare a capo

possiamo arrestare la scomposizione considerando elementari tali compiti

L'approccio bottom-up consiste nel dotarsi di procedure che consentono di arrestare la scomposizione in anticipo

Figura 1.21: Esempio di approccio top-down (3)

di leggere un carattere. I nomi dei sottoproblemi e delle funzioni `StampaStringa` e `LeggeCarattere` vengono dalla libreria `advio`, e sono quindi piuttosto generici. Per acquisire il saluto si stampa una stringa e si legge una riga. Per stampare il saluto, bisogna calcolare la lunghezza della stringa `saluto` (e derivarne la larghezza della cornice), stampare il carattere della cornice un numero opportuno di volte, andare a capo, stampare il carattere della cornice, uno spazio, la stringa con il saluto, uno spazio, il carattere della cornice, andare a capo e stampare più volte il carattere della cornice. Il tutto si risolve con 6 funzioni di base, avendo solo 6 problemi elementari, nonostante in apparenza la decomposizione potesse portare a un'esplosione del numero dei sottoproblemi. Non c'è stata esplosione perché la decomposizione è buona e ripropone sempre gli stessi sottoproblemi.

1.4 Struttura modulare del codice

È abbastanza chiaro che è utile avere un codice che riflette il procedimento mentale seguito per risolvere il problema, perché sarà più facile controllare che la realizzazione del codice corrisponda al progetto ideale, e sarà più facile apportare modifiche a entrambi, qualora fosse necessario. Sarà ovviamente anche più facile descrivere algoritmo e codice ad altri.

Ora realizzeremo il codice e vedremo che la struttura ad albero del progetto concettuale si ritrova in qualche modo nel codice stesso, dato che avremo una funzione che risolve ciascuno dei sottoproblemi. Il codice sarà strutturato in blocchi, ciascuno dei quali si occupa di uno dei sottoproblemi.

Passiamo in rassegna altre regole di stile, che non sono requisiti tecnicamente indispensabili a poter operare, ma sono talmente utili a comunicare quello che si è fatto ad altri da diventare indispensabili in pratica (vedi Figura 1.23). Con "altri" si intende non solo altre persone, ma anche sé stessi dopo un po' di tempo, quando occorre tornare sul proprio lavoro per modificarlo o proseguirlo. Siccome è difficile che, a distanza di tempo, si ricordi in dettaglio quel che si pensava scrivendolo, è

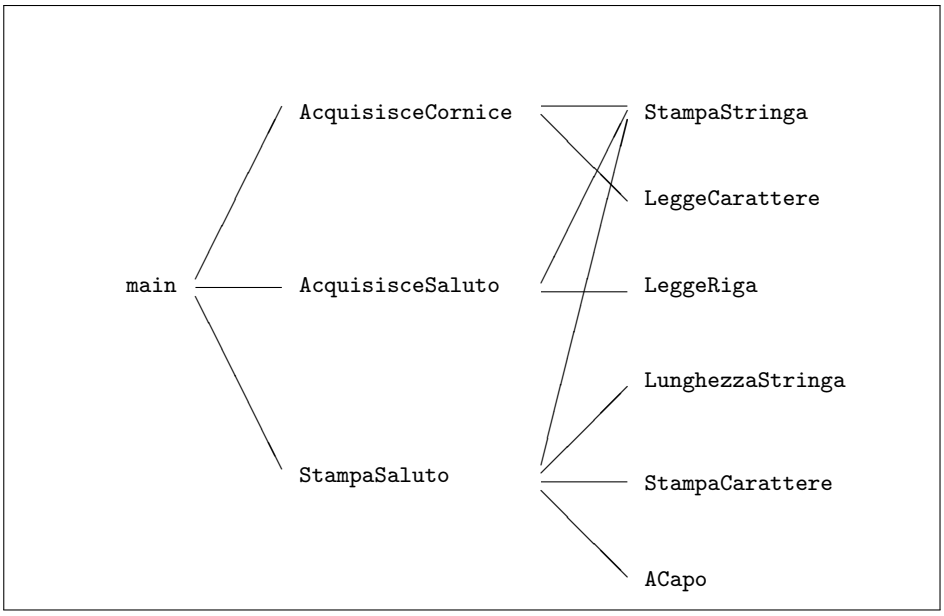


Figura 1.22: Esempio di approccio top-down (4)

necessario che sia il programma stesso a ricordare che cosa si pensava. Per rendere il codice comprensibile, bisogna che esso imiti in qualche modo il più possibile il linguaggio umano, che i nomi delle funzioni, delle variabili, ecc... siano autoesplicativi, che ci sia una chiara corrispondenza fra oggetti e variabili. Per esempio, anche se la parola "intero" comincia per "i", non chiameremo `i` tutti gli interi, perché non è sufficiente a chiarire il loro ruolo. Potremo chiamare `i` gli indici, ma le somme di altri numeri si chiameranno `somma` o `tot`, e così via.

I singoli moduli devono essere il più possibile separati e deve essere molto chiaro quali sono i dati e i risultati di ciascuno, perché questo riduce il rischio che pasticciando da una parte si abbiano effetti imprevisti da altre parti. Se si commette un errore, infatti, si cerca di limitarne gli effetti il più possibile.

In conclusione, bisogna che la compresione di un programma si riduca semplicemente alla comprensione dei singoli moduli e del modo in cui interagiscono fra loro. Non è banale imparare questi elementi di stile, occorre molta esperienza pratica e rimane comunque una fortissima componente di gusto personale: c'è chi decompone i problemi in molti livelli, chi in un numero inferiore. Entro certi limiti, va tutto bene.

1.4.1 Suddivisione dei programmi in moduli

Il principio di divisione del codice in blocchi viene portato alle conseguenze logiche dividendolo in file separati. Nella Figura 1.17, la macchina programmabile in C (la "scatola" rossa) riceveva un file `.C` che contiene tutte le istruzioni necessarie a risolvere il problema. Se il programma è di dimensioni non elementari, però, questo è sconsigliabile per diversi motivi (vedi Figura 1.24). Per prima cosa, una sola persona che lavori su un programma di grandi dimensioni non è in grado di ricordare dove stanno le varie componenti. Se ci lavorano più persone, lavorare tutti sullo stesso file diventa caotico, per il semplice motivo che se le righe aggiunte da qualcuno fanno scorrere verso il basso i pezzi di codice su cui stanno lavorando gli altri. Ovviamente, esistono strumenti informatici per aiutare la navigazione del

Il progetto top-down si riflette nella struttura modulare del codice

Il codice viene strutturato in modo da essere gestibile

1. si scompone il codice in moduli o blocchi strutturati gerarchicamente (esattamente come il problema è scomposto in sottoproblemi)
2. si rendono i moduli comprensibili adottando convenzioni
   - dichiarazioni che imitano il linguaggio umano
   - nomi autoesplicativi per funzioni, macro e variabili
   - corrispondenza biunivoca fra variabili e oggetti
   - usare spazi, a capi e indentazioni per chiarire il senso
   - commenti (ultima risorsa)
3. si rendono i moduli controllabili
   - definendoli in modo che abbiano poche interazioni fra loro
   - esplicitando i requisiti di ogni modulo (dati e risultati)

Si vuole ridurre la comprensione di un programma a

- comprensione di ciascun modulo
- comprensione dei rapporti fra moduli

Non esistono regole meccaniche: la semplicità è una conquista

Figura 1.23: Stile modulare di programmazione

codice, ma aggiungere strumenti a mo' di stampella è in genere un'idea peggiore che risolvere alla radice il danno ed evitare l'uso di stampelle. Per rendere il codice controllabile, tipicamente, si procede a dividerlo in file, in modo che ognuno possa lavorare su un file diverso (oppure il singolo, quando lavora su una questione, possa lavorare sul file specifico che la affronta).

A questo punto, file che risolvono sottoproblemi utili in altre situazioni, possono essere facilmente riutilizzati. Se un codice risolve molti problemi in un unico file, usarlo interamente solo per poter accedere a un piccolo brano di codice è uno spreco, e spesso è tecnicamente complicato farlo. Se invece il codice è diviso, si può prendere il file interessante, copiarlo in un altro progetto e usarlo senza preoccuparsi delle relazioni con il resto del codice originale. Dal punto di vista del progetto d'esame, buona parte delle librerie realizzate nelle lezioni di laboratorio sono utilizzabili, a volte direttamente senza modifiche, altre volte con adattamenti minori. Ci si potrà rendere conto per esperienza che la cosa paga.

Un altro aspetto, di natura tecnica, è che dividere il codice in librerie consente di tradurre ciascuna di loro (e non l'intero programma) in un eseguibile in linguaggio macchina e di prendere questi file binari e usarli direttamente senza doverli ricompilare: si risparmia tempo di compilazione (non sembra molto, ma non molti anni fa compilare un programma lungo poteva richiedere diversi minuti e farlo molte volte al giorno diventava un costo rilevante. la cosa era uno spreco. Avere pezzi di codice già compilato è anche un vantaggio commerciale, perché si possono vendere a chi ne ha bisogno. L'utente di queste librerie compilate può applicarle ai propri problemi, ma non acquisire il *know-how* che ne è alla base, perché ha le istruzioni binarie, illeggibili, e non il codice. Il codice è competenza e conoscenza, e ha un valore superiore al programma compilato, che è semplicemente capacità di risolvere un problema specifico.

Dividere un programma in tanti file, però, pone anche dei problemi. In generale, il programma eseguibile deve essere uno solo per poterlo lanciare dal sistema operativo[6]. Allora il compilatore che fa da intermediario fra i molti file di testo forniti alla macchina programmabile in C e il processore che riceve un solo file binario in linguaggio macchina, deve operare la conversione dagli uni all'altro. È come rimontare un mobile avendone comprato i singoli pezzi, con la differenza che non viene fornito un libretto di istruzioni: le istruzioni per combinare i singoli pezzi, cioè i singoli file, devono essere contenute al loro interno.

1.5 Il processo di compilazione

Discutiamo ora come ricomporre e tradurre un programma costituito da diversi file. Lo vedremo brevemente a livello astratto e lo faremo in pratica sull'esempio del programma per stampare il saluto incorniciato. "Compilare" significa tradurre il codice dal linguaggio ad alto livello scritto in testo al programma scritto in linguaggio macchina, cioè binario. Questo procedimento avviene in più fasi perché è piuttosto sofisticata, soprattutto a causa della divisione su diversi file.

Ci sono tre fasi (Figura 1.25). La prima viene detta di *precompilazione* (in inglese, *preprocessing*), la seconda di *compilazione* vera e propria e la terza di *collegamento* (in inglese, *linking*). Non è difficile capire il senso e memorizzare il ruolo delle tre fasi: basta ricordare da dove si parte e dove si arriva, e che tipo di relazione esiste fra ingresso e uscita di ciascuna fase (vedi Tabella 1.1). In particolare, la fase di compilazione trasforma file di testo in altri file di testo, facendo una pura manipolazione testuale, e la relazione fra ingresso e uscita è in generale una relazione

[6]Sorvoliamo su dettagli tecnici come le *Dynamic-Link Libraries* (*DLL*).

Il codice di un programma complesso viene distribuito in più file perché sarebbe troppo lungo, anche se scritto in linguaggio ad alto livello

- il singolo file è gestibile da un essere umano
- si possono assegnare brani di codice a gruppi indipendenti
- si possono riusare brani di codice in altri programmi senza copiarli (librerie)
- si possono tradurre librerie in linguaggio macchina
  - per evitare di tradurle ogni volta col resto del codice
  - per poterle vendere indipendentemente

Siccome i singoli file non corrispondono a programmi funzionanti

- prima di eseguirli, vanno ricollegati in un codice binario unico
- i singoli file devono contenere indicazioni su come ricollegarli

Figura 1.24: Suddivisione dei programmi in moduli

molti a uno. Nella Figura 1.26 si vede un certo numero di file ingresso con estensioni `.c` e `.h`: i primi contengono il codice, i secondi sono file di *intestazione* (*header file*), e vedremo fra poco il loro scopo. La relazione di precompilazione prende a volte solo un file `.c`, a volte un file `.c` e un file `.h`, a volte un file `.c` e molti file `.h`. L'unica regola è che riceve sempre esattamente un file `.c`. Li fonde in un unico file precompilato, che è un file di testo[7]. L'aspetto importante è che si passa da testo a testo: si fa aggregano eventualmente dei file `.h` a un file `.c` che fa da base. Tutti questi file possono essere stati scritti da noi o forniti da altri (per esempio, la libreria standard `stdio.h`, ovvero "standard input-output", è presente in ogni sistema per programmare in C).

La seconda fase di compilazione, invece, converte codice testuale in linguaggio binario ed applica una relazione uno a uno: ogni singolo file testuale in C viene tradotto in un singolo file binario. Infine, c'è la fase di collegamento, che prende tutti i file binari ottenuti sinora (di solito definiti *file oggetto*, con estensione `.o` o `.obj` secondo il compilatore, l'ambiente e il sistema operativo usati) e li fonde in un eseguibile, ancora binario. Anche in questa fase, può succedere che si aggiungano altri file binari, acquistati o regalati. Al termine, si ha un eseguibile unico, che contiene tutto l'occorrente per funzionare.

Ordine	Nome	Ingresso	Uscita	Relazione
1	Precompilazione	Testo	Testo	Molti a uno
2	Compilazione	Testo	Binario	Uno a uno
3	Collegamento	Binario	Binario	Molti a uno

Tabella 1.1: Le tre fasi del processo di compilazione

[7] L'ho indicato con lo stesso nome del file `.c` di partenza per semplicità. Questo non vuol dire che il file originale venga sovrascritto: il tutto avviene in memoria senza lasciare tracce permanenti.

Compilazione è il processo di traduzione

- dal codice in linguaggio ad alto livello (uno o più file di testo)
- al programma in linguaggio macchina (un file eseguibile)

Attraversa tre fasi

1. precompilazione o preprocessing (da codici a codice): modifica o cancella brani di codice ad alto livello e ne fonde diversi in uno solo
2. compilazione (da codice a oggetto): traduce un file codice in un file binario (file oggetto), di solito passando per il codice assembly
3. collegamento o linking (da oggetti a eseguibile): lega diversi file oggetto ed eventuali librerie esterne in un solo programma eseguibile

Figura 1.25: Il processo di compilazione

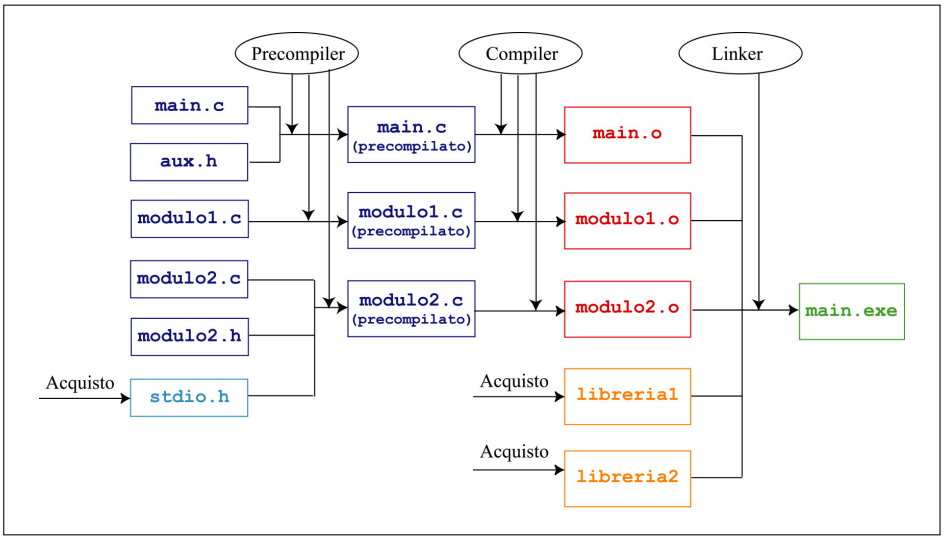


Figura 1.26: Schema del processo di compilazione

1.6 Struttura dei listati C

Un altro aspetto da discutere prima di verificare concretamente su un esempio quanto detto è la struttura dei codici in linguaggio C. Ogni file `.c` è tipicamente formato da 4 componenti (vedi Figura 1.27). La prima componente sono le così dette *direttive*, quindi ci sono i *prototipi*, o dichiarazioni delle procedure secondarie. Poi c'è il *programma principale* (o *main*), cioè quella parte del codice che risolve il problema, e infine le definizioni, ovvero i corpi, delle *procedure secondarie*, che risolvono i sottoproblemi. Le procedure secondarie compaiono sia nella seconda sia nella quarta componente per un motivo tecnico: il compilatore scorre il file una sola volta. Quando incontra un oggetto, deve avere tutte le informazioni strettamente necessarie a gestirlo. Quando nel programma principale il compilatore trova le chiamate delle procedure secondarie, deve già conoscerne i nomi e sapere quali sono i loro dati e risultati, mentre non ha bisogno di sapere che operazioni eseguono. Quindi, ha bisogno dei loro prototipi (che costituiscono la seconda componente, che precede il programma principale), ma non dei loro corpi.

Per lo stesso motivo legato all'ordine di scorrimento, ogni procedura (principale e secondarie) contiene prima una *parte dichiarativa* e poi una *parte esecutiva*, cioè prima una parte che elenca le strutture dati utilizzate, cioè le celle di memoria che saranno necessarie, e poi una parte che descrive le istruzioni da eseguire. Infatti, siccome il compilatore scorre il file una volta sola, quando arriva a un'istruzione che usa una variabile deve sapere dove si trova nella memoria e quanta ne occupa. Queste informazioni sono specificate in precedenza dalla parte dichiarativa. Delle quattro componenti, solo il programma principale è indispensabile: ciascuna delle altre può eventualmente mancare. Il loro ordine non è in effetti rigorosamente necessario, ma rispettarlo (salvo per esigenze molto raffinate) è una buona norma di stile.

Ogni file in linguaggio C (listato) ha una struttura regolare

1. direttive
2. dichiarazioni delle procedure secondarie (prototipi)
3. programma principale (`main`)
   a) parte dichiarativa
   b) parte esecutiva
4. definizioni delle procedure secondarie
   a) parte dichiarativa
   b) parte esecutiva

La struttura è strettamente legata al processo di compilazione

- ciascuna componente può esserci o mancare
  (per es., il `main` compare in un solo file dell'intero progetto)
- l'ordine non è rigoroso, ma violarlo può far fallire la compilazione

Figura 1.27: Struttura dei listati C

1.7 Laboratorio

A questo punto vediamo un esempio pratico della divisione in file, del processo di compilazione, della quadripartizione di ciascun file e della realizzazione di un algoritmo con un approccio *top-down*. L'esempio è disponibile in accompagnamento a queste dispense e consiste di cinque file:

1. `hello0.c` è il punto di partenza dell'esercizio, che esibisce già la struttura del listato, ma non contiene le istruzioni che risolvono il problema, che aggiungeremo nel corso dell'esercizio;

2. `advio.c` è la libreria che consente di gestire la lettura da tastiera e la scrittura su video senza scendere al livello elementare offerto dal linguaggio C, e di mostrare come si gestisce un progetto su più file;

3. `advio.h` è il file di intestazione di tale libreria;

4. `istruzioni.txt` è un file di dati che useremo nelle fasi finali dell'esercizio;

5. `hello.c` è la soluzione dell'esercizio[8].

Un possibile approccio è risolvere l'esercizio autonomamente e consultare la soluzione e la sua descrizione a posteriori per meditare sulle differenze fra il proprio approccio e quello qui indicato. Un altro, analogo a quello tenuto nelle lezioni in laboratorio è alternare brevi fasi di realizzazione autonoma a fasi di confronto con la soluzione suggerita. Questo approccio serve a tenere la classe il più possibile unita, nonostante le forti differenze nella conoscenza della programmazione C da parte degli studenti.

Gli scopi di questo esercizio sono:

1. notare la struttura di un listato C (quali parti lo compongono e perché), per rendersi sensibili a questa struttura (sempre una buona cosa per un matematico o tecnico);

2. notare lo stile con il quale si realizza del codice con un approccio *top-down*, compresi i "gesti" che tipicamente si eseguono più e più volte e, fatti correttamente, offrono qualche protezione contro gli errori di programmazione (come quando si esegue ripetutamente il servizio nel tennis, il muro nella pallavolo, ecc... fino a impararli in automatico);

3. notare il ruolo delle tre fasi di compilazione nel trasformare passo per passo i molteplici file di testo scritti dal programmatore nell'unico file binario eseguibile che risolve effettivamente il problema.

1.7.1 Analisi della struttura del codice

Aprendo il file `hello0.c`, che è il punto di partenza dell'esercizio, notiamo un po' della struttura tipica di un file C (vedi Codice 1.1), in particolare la sua divisione in quattro parti sopra elencate, più una che in realtà non ha uno scopo direttamente funzionale.

[8] Una possibile soluzione: tutti gli esercizi di laboratorio ammettono moltissime soluzioni corrette, più o meno efficienti e più o meno ben strutturate.

0. Commenti I commenti, infatti, non fanno parte di alcuna delle quattro parti del codice prima elencate. In C tutto ciò che è compreso fra * e *\ è un commento, cioè un pezzo di scrittura che è destinato alla lettura di esseri umani, ma viene ignorato dal compilatore (vedi Figura 1.28). Vedremo infatti che uno dei compiti del precompilatore è cancellarli per non ostacolare l'opera del compilatore vero e proprio. D'altra parte, i commenti possono essere fondamentali al programmatore per ricordare e spiegare ad altri l'algoritmo realizzato.

I commenti sono spiegazioni del codice ad uso degli utenti racchiuse fra /* e */ (non si possono annidare!)

- sono eliminati dal precompilatore (la macchina non li usa)
- sono essenziali per lavorare in gruppo o a distanza di tempo
- esistono strumenti software che creano automaticamente il manuale del programma partendo dai commenti

Commento è anche il testo compreso fra // e la fine della riga
a partire dallo standard C99 (*noi seguiremo quasi in tutto il C89*)

Figura 1.28: Commenti

1. Direttive La prima parte vera e propria del codice è formata dalle direttive (vedi Figura 1.29, che sono riconoscibili perché cominciano con il carattere diesis (`#`) che chi non conosce la notazione musicale chiama anche "cancelletto". Le direttive più utili e comuni sono `#include`, che compare anche qui, e `#define` che vedremo in seguito.

Servono a includere file di intestazione e a definire costanti simboliche

POVA

Figura 1.29: Direttive

Servono anche a non includere più volte lo stesso file di intestazione

Figura 1.30: Direttive2

2. Prototipi La seconda parte qui è marcata con un commento, ma al momento è vuota, e sarà costituita dai prototipi, o dichiarazioni delle procedure secondarie. Siccome il problema è stato decomposto in sottoproblemi, e quindi il programma principale utilizzerà delle procedure secondarie, in questa sezione bisognerà comunicare al compilatore quali sono i sottoprogrammi usati, in modo che il compilatore possa scorrere il codice una volta sola (durante la compilazione vera e propria) e

Cominciano con il carattere speciale `#`

Danno istruzioni al precompilatore

- inclusione di altri file (`#include` *nomefile*):
  si sostituisce la direttiva con l'intero contenuto del file (ricorsivamente)
- espansione di macro (`#define` *macro valore*):
  si sostituisce ogni occorrenza della macro col valore (ricorsivamente)
- compilazione condizionale
  `#ifdef` *macro codice* `#endif`:
  il codice si conserva se la macro è definita, altrimenti si cancella
  `#ifndef` *macro codice* `#endif`:
  il codice si conserva se la macro non è definita, altrimenti si cancella

Per vedere in dettaglio l'azione del precompilatore, si esegue il comando

Figura 1.31: Direttive3

Figura 1.32: Struttura: prototipi delle procedure

quando arriva nella terza sezione, il programma principale, sappia già che le procedure secondarie esistono, quali dati richiedono e quali risultati restituiscono. La seconda sezione serve proprio a comunicare queste informazioni al compilatore. Vedi Figure 1.33) e 1.34).

3. Programma principale La terza parte del codice è il programma principale, costituito dalla funzione `main`, che risolve il problema nel suo complesso (vedi Figura 1.35). Tutte le funzioni hanno la stessa struttura: la prima riga fa da intestazione, le righe seguenti, comprese fra le parentesi graffe aperta ({) e chiusa (}), formano il corpo. Ogni coppia di parentesi corrispondenti racchiude un modulo o blocco. L'intestazione dà un nome alla funzione (per il programma principale, il nome `main` è obbligatorio, dato che serve a identificarla), dà dei nomi ai dati della funzione[9] e ne specifica il tipo, cioè il dominio di definizione, e infine definisce il tipo del risultato della funzione[10]. Entrando nel corpo del programma principale (o di qualsiasi altra funzione), cioè fra le corrispondenti parentesi graffe, si hanno due parti consecutive (vedi Figura 1.36):

1. la *parte dichiarativa* elenca i blocchi di memoria usati dal programma, con i relativi nomi e *tipi*, che indicano il dominio di definizione di quel blocco di memoria e la sua dimensione spaziale (vedi Figure 1.37 e 1.38)
2. la *parte esecutiva* contiene le istruzioni, compresa quella finale, che restituisce il risultato alla procedura chiamante (o al sistema operativo) (vedi Figure 1.39 e 1.40).

[9] Discuteremo nel Capitolo ?? quali dati abbia il programma principale.
[10] Discuteremo nel Capitolo ?? che genere di risultato restituisca il programma principale.

Presentano le funzioni o procedure: brani di codice abbastanza importanti da avere un nome, dei dati e dei risultati, come i programmi

Il prototipo (dichiarazione) di una funzione ne specifica

1. il tipo del risultato (`void` se non c'è risultato)
2. il nome simbolico usato per chiamarla
3. il tipo e il nome dei dati (parametri di ingresso)

La definizione della funzione (il brano di codice) è in un'altra sezione

Figura 1.33: Struttura: prototipi delle procedure

I file *header* inclusi dalla direttiva `#include` spesso contengono prototipi di funzioni definite in librerie esterne

Figura 1.34: Struttura: prototipi inclusi dai file intestazione

Una o l'altra di queste parti può mancare: se manca la prima, significa che la procedura è abbastanza semplice da non richiedere risultati intermedi; se manca la seconda, significa che la procedura non fa nulla (cosa in genere assurda, salvo che sia stata scritta in preparazione di una stesura, come faremo spessissimo durante gli esercizi).

Figura 1.35: Struttura: main

4. Definizioni delle procedure secondarie La quarta parte contiene le definizioni delle procedure secondarie, dichiarate nella seconda parte e usate nella terza dal programma principale, o nella quarta da altre procedure secondarie. Questa parte serve a descrivere non più solo che dati e che risultati hanno le procedure, ma come fanno a passare dai dati ai risultati. Quindi, ciascuna procedura secondaria riporterà un insieme di istruzioni. La forma delle procedure secondarie è identica a quella del programma principale: hanno un'intestazione, un corpo racchiuso fra parentesi graffe e contenente una parte dichiarativa e una parte esecutiva (vedi Figure **??**.

Tutta questa struttura è fissa e regolare perché il compilatore è una macchina, e funziona agevolmente con strutture fisse e regolari. Se si cambia l'ordine di questi elementi, il compilatore riesce a gestirle entro certi limiti, perché il compilatore C in realtà è un oggetto molto flessibile, ma è consigliabile non sfruttare troppo questa flessibilità e forzarsi a seguire delle regole perché nella gran maggioranza dei casi

Uno e uno solo dei moduli C contiene il programma principale (`main`)

La sua intestazione consiste ancora in

1. tipo del risultato (sempre `int`)
2. nome del programma principale
   (sempre `main`, per distinguerlo dalle altre funzioni)
3. tipi e nomi dei dati (sempre `argc` e `argv`)

La definizione è racchiusa fra parentesi graffe ({})

a) parte dichiarativa: introduce le variabili,
   cioè gli oggetti manipolati dal programma

b) parte esecutiva: introduce le istruzioni,
   cioè le operazioni compiute dal programma

Figura 1.36: Parte dichiarativa e parte esecutiva del `main`

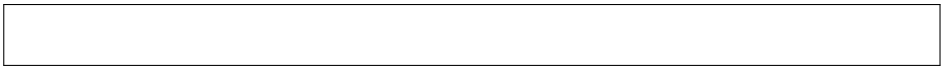


Figura 1.37: Struttura: parte dichiarativa

Le variabili sono risultati parziali; di ognuna si specifica

- il tipo, che identifica l'insieme dei valori che può assumere,
  le operazioni che si possono compiere con essa
  e lo spazio che occupa in memoria
- il nome, cioè un identificatore simbolico usato per manipolarla

Figura 1.38: Struttura: parte dichiarativa

Figura 1.39: Struttura: parte esecutiva

Le istruzioni possono essere

- operazioni definite nel linguaggio (per es., `=`)
- funzioni tratte da librerie standard (per es., `printf`)
- funzioni definite dall'utente (per es., `StampaStringa`)

Ogni istruzione termina con un punto e virgola (`;`)

La dichiarazione di una variabile o di una procedura è un'istruzione

- aggiunge il nome della variabile o procedura
  alla tabella dei simboli utilizzabili dal programma
- quindi, termina con `;`

Figura 1.40: Struttura: parte esecutiva

Figura 1.41: Struttura: definizioni delle procedure secondarie

Figura 1.42: Struttura: definizioni delle procedure secondarie

La definizione è il brano di codice preannunciato dal prototipo

Ha la stessa struttura del main

- intestazione: copia del prototipo (senza `;`)
- parte dichiarativa: tipo e nome delle variabili
- parte esecutiva: istruzioni

Figura 1.43: Struttura: definizioni delle procedure secondarie

questa struttura garantisce che tutto funzioni. Per agire diversamente bisogna avere buoni motivi e sapere che cosa si sta facendo.

Listing 1.1: File iniziale `hello0.c`

```
1  /* hello.c */
2
3  /* Direttive */
4  #include <stdlib.h>
5  #include "advio.h"
6
7
8  /* Prototipi delle procedure secondarie */
9
10 /* Acquisisce il carattere che si vuole usare per la cornice */
11 char AcquisisceCornice ();
12
13 /* Acquisisce il saluto che si vuole stampare */
14 void AcquisisceSaluto (char *saluto);
15
16 /* Stampa il saluto incorniciato */
17 void StampaSaluto (char cornice, char *saluto);
18
19
20 /* Programma principale */
21 int main (int argc, char *argv[])
22 {
23   /* Parte dichiarativa */
24   char cornice;
25   char saluto[ROW_LENGTH];
26
27
28   /* Parte esecutiva */
29
30   /* Acquisisce il carattere che si vuole usare per la cornice */
31   cornice = AcquisisceCornice();
32
33   /* Acquisisce il saluto che si vuole stampare */
34   AcquisisceSaluto(saluto);
35   ACapo();
36
37   /* Stampa il saluto incorniciato */
38   StampaSaluto(cornice,saluto);
39
40   return EXIT_SUCCESS;
41 }
42
43
44 /* Definizione delle procedure secondarie */
45
46 /* Acquisisce il carattere che si vuole usare per la cornice */
47 char AcquisisceCornice ()
48 {
49   char cornice;
50
51   StampaStringa("Inserire il carattere per la cornice: ");
52   cornice = LeggeCarattere();
53   return cornice;
54 }
55
56
57 /* Acquisisce il saluto che si vuole stampare */
58 void AcquisisceSaluto (char *saluto)
59 {
60   StampaStringa("Inserire il saluto da incorniciare: ");
61   LeggeRiga(saluto);
62 }
63
```

```
64
65 /* Stampa il saluto incorniciato */
66 void StampaSaluto (char cornice, char *saluto)
67 {
68   int larghezza;
69
70   /* determinare la larghezza della cornice (lunghezza di saluto piu'
       4) */
71   larghezza = LunghezzaStringa(saluto)+4;
72
73   /* stampare la cornice superiore (stampare larghezza volte il
       carattere cornice) */
74   StampaCarattere(cornice,larghezza);
75
76   /* andare a capo */
77   ACapo();
78
79   /* stampare la cornice laterale sinistra (stampare il carattere
       cornice) */
80   StampaCarattere(cornice,1);
81
82   /* stampare uno spazio separatore (stampare uno spazio) */
83   StampaCarattere(SPAZIO,1);
84
85   /* stampare il saluto (stampare la stringa saluto) */
86   StampaStringa(saluto);
87
88   /* stampare uno spazio separatore (stampare uno spazio) */
89   StampaCarattere(SPAZIO,1);
90
91   /* stampare la cornice laterale destra (stampare il carattere
       cornice) */
92   StampaCarattere(cornice,1);
93
94   /* andare a capo */
95   ACapo();
96
97   /* stampare la cornice inferiore */
98   StampaCarattere(cornice,larghezza);
99 }
```

Il codice comprende altri due file. Apriamo `advio.h` (vedi Codice 1.2). Questo è un file di intestazione, e svolge una funzione diversa dal file precedente, ma ha struttura simile: comincia con delle direttive (diverse da quelle che viste), seguite da un elenco di prototipi di procedure (nomi di procedure seguiti dai tipi e nomi dei dati e preceduti dal tipo del risultato[11]). La terza e la quarta sezione, cioè programma principale e definizioni delle procedure secondarie, mancano. La terza sezione, infatti, è presente solo nel file principale dell'intero progetto. La quarta sezione manca sempre nei file di intestazione, che contengono solo direttive e prototipi, con l'eccezione di alcune istruzioni che vedremo nel seguito e che non sono contenute in alcuna procedura.

Listing 1.2: File iniziale `advio.h`

```
1 #ifndef __advio_h
2 #define __advio_h
3
4 #define SPAZIO      ' '
5 #define ROW_LENGTH 256
6
7 /* Stampa a video la stringa di caratteri "s" */
8 void StampaStringa (char *s);
9
```

[11] La parola chiave `void` si usa quando la procedura non ha alcun risultato, per esempio perché si limita a stampare qualcosa.

```
10 /* Legge da tastiera la stringa di caratteri "s" */
11 void LeggeRiga (char *s);
12
13 /* Legge la prima parola inserita da tastiera e la scrive nella
       stringa "s" */
14 void LeggeParola (char *s);
15
16 /* Stampa a video il carattere "c" ripetuto "num" volte */
17 void StampaCarattere (char c, int num);
18
19 /* Legge il primo carattere inserito da tastiera */
20 char LeggeCarattere ();
21
22 /* Va a capo */
23 void ACapo ();
24
25 /* Stampa il numero intero "n" */
26 void StampaIntero (int n);
27
28 /* Legge da tastiera un numero intero */
29 int LeggeIntero ();
30
31 /* Stampa il numero naturale "n" */
32 void StampaNaturale (unsigned int n);
33
34 /* Legge da tastiera un numero naturale */
35 unsigned int LeggeNaturale ();
36
37 /* Stampa il numero long "n" */
38 void StampaLong (long n);
39
40 /* Legge da tastiera un numero intero di tipo long */
41 long LeggeLong ();
42
43 /* Stampa il numero "f", reale di tipo float */
44 void StampaFloat (float f);
45
46 /* Legge da tastiera un numero reale di tipo float */
47 float LeggeFloat ();
48
49 /* Stampa il numero "d", reale di tipo double */
50 void StampaDouble (double d);
51
52 /* Legge da tastiera un numero reale di tipo double */
53 double LeggeDouble ();
54
55 /* Stampa l'indirizzo "i" */
56 void StampaIndirizzo (void *i);
57
58 /* Stampa la cifra in denaro rappresentata dal numero reale "f" */
59 void StampaCifraDenaro (float f);
60
61 /* Legge da tastiera una cifra in denaro e la converte in un numero
       reale di tipo float */
62 float LeggeCifraDenaro ();
63
64 /* Converte una stringa numerica in un numero intero (in caso di
       errore, termina il programma) */
65 int ConverteStringaInIntero (char *s);
66
67 /* Determina il numero di caratteri di una stringa data */
68 int LunghezzaStringa (char *s);
69
70 #endif
```

Apriamo infine `advio.c` (vedi Codice 1.3), che tipicamente contiene la prima e la quarta parte, cioè direttive e definizioni di procedure secondarie.

Listing 1.3: File iniziale `advio.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5
6 /* Stampa a video la stringa di caratteri "s" */
7 void StampaStringa (char *s)
8 {
9   printf("%s",s);
10 }
11
12
13 /* Legge da tastiera la stringa di caratteri "s" */
14 void LeggeRiga (char *s)
15 {
16   fgets(s,256,stdin);
17   s[strlen(s)-1] = '\0';
18 }
19
20
21 /* Legge la prima parola inserita da tastiera e la scrive nella
       stringa "s" */
22 void LeggeParola (char *s)
23 {
24   char riga[256];
25
26   fgets(s,256,stdin);
27   sscanf(riga,"%s",s);
28 }
29
30
31 /* Stampa a video il carattere "c" ripetuto "num" volte */
32 void StampaCarattere (char c, int num)
33 {
34   int i;
35
36   for (i = 1; i <= num; i++)
37     printf("%c",c);
38 }
39
40
41 /* Legge il primo carattere inserito da tastiera */
42 char LeggeCarattere ()
43 {
44   char riga[256];
45
46   fgets(riga,256,stdin);
47   return riga[0];
48 }
49
50
51 /* Va a capo */
52 void ACapo ()
53 {
54   printf("\n");
55 }
56
57
58 /* Stampa il numero intero "n" */
59 void StampaIntero (int n)
60 {
61   printf("%d",n);
62 }
63
64
65 /* Legge da tastiera un numero intero */
```

```
66 int LeggeIntero ()
67 {
68   int res, i;
69
70   res = scanf("%d",&i);
71   if (res != 1)
72   {
73     fprintf(stderr,"Non si e' inserito un numero intero corretto!");
74     exit(EXIT_FAILURE);
75   }
76
77   return i;
78 }
79
80
81 /* Stampa il numero naturale "n" */
82 void StampaNaturale (unsigned int n)
83 {
84   printf("%u",n);
85 }
86
87
88 /* Legge da tastiera un numero naturale */
89 unsigned int LeggeNaturale ()
90 {
91   int res;
92   unsigned int i;
93
94   res = scanf("%u",&i);
95   if (res != 1)
96   {
97     fprintf(stderr,"Non si e' inserito un numero naturale corretto!");
98     exit(EXIT_FAILURE);
99   }
100
101  return i;
102 }
103
104
105 /* Stampa il numero long "n" */
106 void StampaLong (long n)
107 {
108   printf("%ld",n);
109 }
110
111
112 /* Legge da tastiera un numero intero di tipo long */
113 long LeggeLong ()
114 {
115   int res;
116   long i;
117
118   res = scanf("%ld",&i);
119   if (res != 1)
120   {
121     fprintf(stderr,"Non si e' inserito un numero intero long
          corretto!");
122     exit(EXIT_FAILURE);
123   }
124
125   return i;
126 }
127
128
129 /* Stampa il numero "f", reale di tipo float */
130 void StampaFloat (float f)
131 {
```

```
132   printf("%f",f);
133 }
134
135
136 /* Legge da tastiera un numero reale di tipo float */
137 float LeggeFloat ()
138 {
139   int res;
140   float f;
141
142   res = scanf("%f",&f);
143   if (res != 1)
144   {
145     fprintf(stderr,"Non si e' inserito un numero reale corretto!");
146     exit(EXIT_FAILURE);
147   }
148
149   return f;
150 }
151
152
153 /* Stampa il numero "d", reale di tipo double */
154 void StampaDouble (double d)
155 {
156   printf("%f",d);
157 }
158
159
160 /* Legge da tastiera un numero reale di tipo double */
161 double LeggeDouble ()
162 {
163   int res;
164   double d;
165
166   res = scanf("%lf",&d);
167   if (res != 1)
168   {
169     fprintf(stderr,"Non si e' inserito un numero reale double
          corretto!");
170     exit(EXIT_FAILURE);
171   }
172
173   return d;
174 }
175
176
177 /* Stampa l'indirizzo "i" */
178 void StampaIndirizzo (void *i)
179 {
180   printf("%p",i);
181 }
182
183
184 /* Stampa la cifra in denaro rappresentata dal numero reale "f" */
185 void StampaCifraDenaro (float f)
186 {
187   printf("%.2f euro",f);
188 }
189
190
191 /* Legge da tastiera una cifra in denaro e la converte in un numero
      reale di tipo float */
192 float LeggeCifraDenaro ()
193 {
194   float f;
195
196   if (scanf("%f",&f) != 1)
```

```
197   {
198     fprintf(stderr,"Non si e' inserita una cifra in denaro
          corretta!");
199     exit(EXIT_FAILURE);
200   }
201
202   return f;
203 }
204
205
206 /* Converte una stringa numerica in un numero intero (in caso di
      errore, termina il programma) */
207 int ConverteStringaInIntero (char *s)
208 {
209   int n;
210
211   if (sscanf(s,"%d",&n) != 1)
212   {
213     StampaStringa("Il parametro non e' un numero intero!\n");
214     exit(EXIT_FAILURE);
215   }
216
217   return n;
218 }
219
220
221 /* Determina il numero di caratteri di una stringa data */
222 int LunghezzaStringa (char *s)
223 {
224   return strlen(s);
225 }
```

La Tabella 1.2 riassume quanto visto. Ricordiamo che non si tratta di regole ferree, ma dell'uso comune, motivato dal ruolo dei diversi tipi di file entro il progetto complessivo.

File	Estensione	Direttive	Prototipi	`main`	Altre procedure
Listato principale	`.c`	×	×	×	×
Listati secondari	`.c`	×			×
Intestazioni	`.h`	×	×		

Tabella 1.2: Struttura dei diversi file in un progetto C tipico

1.7.2 Stesura *top-down* del codice

Avendo già progettato la soluzione del problema in modalità *top-down*, ci resta solo da realizzarla. Dividiamo il problema in tre sottoproblemi: acquisire il carattere desiderato per la cornice, acquisire il saluto da stampare e stampare il saluto incorniciato, assegnando loro dei nomi simbolici. Il codice `hello0.c` contiene questa soluzione sotto forma di commenti.

A ciascun sottoproblema, cioè a ciascun commento, facciamo corrispondere la chiamata a una procedura, nella quale indicheremo i dati e il risultato:

```
cornice = AcquisisceCornice();
```

Non ci sono dati, perché il carattere da usare per la cornice non viene ricavato con un calcolo, bensì chiedendo all'utente di indicarlo da tastiera. L'assenza di dati è indicata dalle parentesi tonde vuote. Si tratta di una procedura abbastanza "degenere", ma non è l'unica in questo esempio estremamente semplice. C'è un

risultato, il carattere richiesto, che viene assegnato con un operatore `=` a un'area di memoria. Quest'area deve avere un nome, che per noi sarà `cornice`. L'istruzione è terminata da un punto e virgola (`;`), che è il segno distintivo delle operazioni nel linguaggio C. L'istruzione nel suo complesso esegue la funzione `AcquisisceCornice`, che non ha bisogno di dati, ne ricava un carattere, lo assegna a un'area di memoria che si chiama `cornice`.

Limitarsi ad aggiungere questa istruzione non è accettabile dal compilatore perché:

1. non sa che cos'è `cornice`;
2. non sa che cos'è `AcquisisceCornice`.

Siccome il compilatore traduce il codice in binario riga per riga, quando arriva a questa riga si ferma e stampa errori, che vedremo nella Sezione **??**.

Per poter usare il simbolo `cornice`, bisogna subito aggiungere un'istruzione di dichiarazione:

```
char cornice;
```

nella parte dichiarativa della procedura, cioè prima della parte esecutiva che contiene la chiamata. Si noti il `;` che termina l'istruzione e la qualifica come un'istruzione. Mentre è ovvio che la precedente fosse un'istruzione, lo è molto meno che dichiarare una variabile sia un'istruzione, ma in effetti lo è. Infatti, il compilatore che incontra questa istruzione aggiorna un'opportuna *tabella dei simboli*, dove conserva tutti i simboli che indicano aree di memoria, con i loro nomi e tipi, ma soprattutto assegna a questo simbolo l'area di memoria che verrà usata dal simbolo stesso quando si compieranno operazioni su di esso, in modo che nessun altro simbolo possa usare la stessa area di memoria. Vedremo come avvenga questo assegnamento nel Capitolo **??**. L'indicazione del tipo della variabile (`char`) serve a determinare quanto è grande quest'area di memoria: esattamenta la dimensione necessaria a contenere un carattere.

Per poter usare il simbolo `AcquisisceCornice`, bisogna fare qualcosa di simile, ma più complicato. Bisogna aggiungere un prototipo nella seconda parte del codice e una definizione nella quarta. Aggiungere il prototipo significa aggiungere l'istruzione:

```
char AcquisisceCornice ();
```

magari copiando anche il commento, e adattandolo affinché valga in una generica occasione in cui la procedura viene chiamata, anziché nella specifica occasione che stiamo considerando ora. Il prototipo fornisce alla procedura un nome, i nomi e i tipi dei dati (qui nessuno) e il tipo del risultato. Il risultato non ha un nome, perché ad ogni diversa chiamata della funzione possiamo assegnarlo a un'area diversa. Anche questa è un'istruzione che indica al compilatore di aggiungere un simbolo alla tabella dei simboli, specificando il funzionamento della procedura, in termini di grandezza delle aree di memoria coinvolte, ai soli "morsetti esterni" (dati e risultati). Il codice non è ancora completo, perché bisogna specificare che cosa fa la procedura. Per il momento, non lo sappiamo ancora, ma la correttezza sintattica del tutto richiede di avere non solo la dichiarazione, ma anche la definizione. Copiamo quindi l'intestazione (e anche il commento, per completezza) nella quarta parte del codice. Togliamo il punto e virgola perché la definizione di una procedura non è un'istruzione (all'interno del suo corpo ci saranno istruzioni, ma nel complesso non è un'istruzione). All'intestazione facciamo seguire le parentesi graffe che individuano il corpo della funzione, ma lasciamo il corpo vuoto.

```
char AcquisisceCornice ()
{
}
```

Il risultato è una procedura correttamente definita, che non fa nulla (per ora). A rigore, questa funzione per definizione dovrebbe restituire un carattere, ma al momento non sta restituendo nulla. Possiamo scegliere se ignorare temporaneamente la cosa (che il compilatore segnalerà come avvertimento, non come errore) o se aggiungere un'istruzione che restituisca un carattere per *default* (che in gergo significa "per difetto", o "in assenza" di un vero risultato). La libreria `advio.h` ha un possibile suggerimento: la direttiva

```
#define SPAZIO ' '
```

indica che la costante simbolica `SPAZIO` (anche detta *macro*) equivale alla stringa `' '` che la segue (e quindi può essere usata al posto). Tale stringa indica nel linguaggio C il carattere spazio bianco: gli apici che racchiudono il carattere segnalano che va inteso e usato letteralmente. A questo punto, possiamo includere nella funzione l'istruzione

```
return SPAZIO;
```

che indica di restituire uno spazio bianco come carattere per la cornice, in assenza di indicazioni più precise fornite dall'utente. Lo scopo è sostanzialmente di avere un codice sintatticamente corretto, anche se incompleto. Sarebbe del tutto equivalente scrivere

```
return ' ';
```

ma la prima scrittura è più chiara e consente, eventualmente, di modificare a piacere la definizione di `SPAZIO` senza dover cercare in giro per il codice (che potrebbe essere diventato nel frattempo enorme) una per una tutte le occorrenze di `' '`.

1.7.3 Compilazione

Vediamo ora che cosa succede fisicamente al codice di esempio quando si compila. La prima fase è la precompilazione (vedi Figura 1.26), che manipola testualmente un file `.c` e un numero qualsiasi (eventualmente, nullo) di file `.h` in modo da ottenere un nuovo file di testo, che in generale non viene salvato su disco, ma tenuto in memoria pronto per la fase successiva. Per osservare che cosa succede, faremo salvare il file di testo su disco con il comando

```
gcc -E hello0.c
```

che esegue il compilatore `gcc`[12] sul file `hello0.c` limitandosi alla precompilazione (opzione `-E`). Se lanciamo semplicemente questo comando, il risultato viene stampato a video e non sarà facile capire molto, dato che sarà molto lungo. Quindi chiederemo di salvare il risultato su un file di testo, attraverso l'opzione `-o`, seguita dal nome del file stesso:

```
gcc -E hello0.c -o hello0.txt
```

Si veda la Figura 1.44 per un sommario delle opzioni di compilazione. Aprendo questo file (vedi Listato 1.4), si osservano molte scritte complicate (e in parte dipendenti dalla macchina su cui si sta lavorando). Il motivo è che il precompilatore

[12] Questo è il compilatore che usiamo in laboratorio: altri avranno altri nomi e altre opzioni.

scorre il codice una volta sola (come farà poi il compilatore) e manipola direttive e commenti. La prima cosa che trova nel file `hello0.c` è una serie di direttive `#include`. Il precompilatore le esegue recuperando i file che seguono la direttiva stessa. Alcuni sono file standard, forniti insieme all'ambiente di compilazione (questo è indicato dal fatto che i loro nomi sono racchiusi fra `<` e `>`). Il precompilatore sa quindi dove trovarli, li apre e ne copia il contenuto all'interno del nuovo file al posto della direttiva stessa. Infatti in `hello0.txt` si vedono elenchi di funzioni, molto più complicate di quelle che abbiamo visto prima, ma concettualmente analoghe.

Dopo il file `stdio.h` viene incluso il file `advio.h`. Questa volta, il nome è racchiuso fra virgolette (`"`), dato che si tratta di una libreria scritta da noi. Riconosciamo il suo contenuto dalla comparsa delle procedure `StampaStringa`, `LeggeRiga`, ecc... Si può anche notare che i commenti non sono stati inclusi, ma semplicemente cancellati. Infatti, servono agli utenti umani, ma sono solo d'impiccio per la macchina.

Il precompilatore fa anche un'altra cosa importante, che si vede nelle ultime righe del file precompilato: sostituisce le costanti simboliche con i loro valori; in gergo, *espande le macro*. Il file `advio.h`, infatti, includeva diverse procedure `#define`, che non vengono semplicemente incluse nel file precompilato, ma processate. Questo avviene con un'operazione di sostituzione testuale, che cerca ogni occorrenza della costante simbolica (successiva alla definizione) e la rimpiazza con il suo valore. Quindi, la costante `EXIT_SUCCESS`, che è definita in `stdlib.h`, viene sostituita dal valore `0` (e infatti l'ultima istruzione del `main` precompilato è `return 0;`, anziché `return EXIT_SUCCESS;`). Allo stesso modo, la costante `SPAZIO`, definita in `advio.h`, viene sostituita dal valore `' '`. Infatti, l'ultima istruzione di `AcquisisceCornice` diventa `return ' ';`

Listing 1.4: Risultato della precompilazione di `hello0.c`

```
1 # 1 "hello0.c"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 1 "hello0.c"
5
6
7
8 # 1 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 1 3
9 # 9 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
10 # 1 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/crtdefs.h" 1 3
11 # 10 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/crtdefs.h" 3
12 # 1 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 1 3
13 # 12 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 3
14 # 1 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw_mac.h" 1 3
15 # 46 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw_mac.h" 3
16
17 # 55 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw_mac.h" 3
18
19 # 13 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 2 3
20 # 1 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw_secapi.h" 1
     3
21 # 14 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 2 3
22 # 282 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 3
23 # 1 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/vadefs.h" 1 3
24 # 9 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/vadefs.h" 3
25 # 1 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 1 3
26 # 686 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 3
27 # 1
     "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/sdks/_mingw_directx.h"
     1 3
28 # 687 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 2 3
29 # 1 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/sdks/_mingw_ddk.h"
     1 3
```

```
30 # 688 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 2 3
31 # 10 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/vadefs.h" 2 3
32
33
34 #pragma pack(push,_CRT_PACKING)
35 # 22 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/vadefs.h" 3
36   typedef __builtin_va_list __gnuc_va_list;
37
38
39
40
41
42
43   typedef __gnuc_va_list va_list;
44 # 101 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/vadefs.h" 3
45 #pragma pack(pop)
46 # 283 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 2 3
47
48
49 #pragma pack(push,_CRT_PACKING)
50 # 377 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 3
51 __extension__ typedef unsigned long long size_t;
52 # 387 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 3
53 __extension__ typedef long long ssize_t;
54 # 399 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 3
55 __extension__ typedef long long intptr_t;
56 # 412 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 3
57 __extension__ typedef unsigned long long uintptr_t;
58 # 425 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 3
59 __extension__ typedef long long ptrdiff_t;
60 # 435 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 3
61 typedef unsigned short wchar_t;
62
63
64
65
66
67
68
69 typedef unsigned short wint_t;
70 typedef unsigned short wctype_t;
71 # 463 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 3
72 typedef int errno_t;
73
74
75
76
77 typedef long __time32_t;
78
79
80
81
82 __extension__ typedef long long __time64_t;
83
84
85
86
87
88
89
90 typedef __time64_t time_t;
91 # 656 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/_mingw.h" 3
92 void __attribute__((__cdecl__)) __debugbreak(void);
93 extern __inline__ __attribute__((__always_inline__,__gnu_inline__))
       void __attribute__((__cdecl__)) __debugbreak(void)
94 {
95   __asm__ __volatile__("int {$}3":);
```

```
 96 }
 97
 98
 99
100
101 const char *__mingw_get_crt_info (void);
102
103
104
105
106
107
108 #pragma pack(pop)
109 # 11 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/crtdefs.h" 2 3
110 # 26 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/crtdefs.h" 3
111 typedef size_t rsize_t;
112 # 153 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/crtdefs.h" 3
113 struct threadlocaleinfostruct;
114 struct threadmbcinfostruct;
115 typedef struct threadlocaleinfostruct *pthreadlocinfo;
116 typedef struct threadmbcinfostruct *pthreadmbcinfo;
117 struct __lc_time_data;
118
119 typedef struct localeinfo_struct {
120   pthreadlocinfo locinfo;
121   pthreadmbcinfo mbcinfo;
122 } _locale_tstruct,*_locale_t;
123
124
125
126 typedef struct tagLC_ID {
127   unsigned short wLanguage;
128   unsigned short wCountry;
129   unsigned short wCodePage;
130 } LC_ID,*LPLC_ID;
131
132
133
134
135 typedef struct threadlocaleinfostruct {
136   int refcount;
137   unsigned int lc_codepage;
138   unsigned int lc_collate_cp;
139   unsigned long lc_handle[6];
140   LC_ID lc_id[6];
141   struct {
142     char *locale;
143     wchar_t *wlocale;
144     int *refcount;
145     int *wrefcount;
146   } lc_category[6];
147   int lc_clike;
148   int mb_cur_max;
149   int *lconv_intl_refcount;
150   int *lconv_num_refcount;
151   int *lconv_mon_refcount;
152   struct lconv *lconv;
153   int *ctype1_refcount;
154   unsigned short *ctype1;
155   const unsigned short *pctype;
156   const unsigned char *pclmap;
157   const unsigned char *pcumap;
158   struct __lc_time_data *lc_time_curr;
159 } threadlocinfo;
160 # 10 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 2 3
161 # 1
      "D:/Dev-Cpp/MinGW64/lib/gcc/x86_64-w64-mingw32/4.9.2/include-fixed/limits.h"
```

```
      1 3 4
162 # 34
      "D:/Dev-Cpp/MinGW64/lib/gcc/x86_64-w64-mingw32/4.9.2/include-fixed/limits.h"
      3 4
163 # 1
      "D:/Dev-Cpp/MinGW64/lib/gcc/x86_64-w64-mingw32/4.9.2/include-fixed/syslimits.h"
      1 3 4
164
165
166
167
168
169
170 # 1
      "D:/Dev-Cpp/MinGW64/lib/gcc/x86_64-w64-mingw32/4.9.2/include-fixed/limits.h"
      1 3 4
171 # 168
      "D:/Dev-Cpp/MinGW64/lib/gcc/x86_64-w64-mingw32/4.9.2/include-fixed/limits.h"
      3 4
172 # 1 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/limits.h" 1 3 4
173 # 169
      "D:/Dev-Cpp/MinGW64/lib/gcc/x86_64-w64-mingw32/4.9.2/include-fixed/limits.h"
      2 3 4
174 # 8
      "D:/Dev-Cpp/MinGW64/lib/gcc/x86_64-w64-mingw32/4.9.2/include-fixed/syslimits.h"
      2 3 4
175 # 35
      "D:/Dev-Cpp/MinGW64/lib/gcc/x86_64-w64-mingw32/4.9.2/include-fixed/limits.h"
      2 3 4
176 # 11 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 2 3
177
178
179
180
181
182 #pragma pack(push,_CRT_PACKING)
183 # 40 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
184   typedef int (__attribute__((__cdecl__)) *_onexit_t)(void);
185 # 50 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
186   typedef struct _div_t {
187     int quot;
188     int rem;
189   } div_t;
190
191   typedef struct _ldiv_t {
192     long quot;
193     long rem;
194   } ldiv_t;
195
196
197
198
199
200 #pragma pack(4)
201   typedef struct {
202     unsigned char ld[10];
203   } _LDOUBLE;
204 #pragma pack()
205
206
207
208   typedef struct {
209     double x;
210   } _CRT_DOUBLE;
211
212   typedef struct {
213     float f;
```

```
214   } _CRT_FLOAT;
215
216
217
218
219   typedef struct {
220     long double x;
221   } _LONGDOUBLE;
222
223
224
225 #pragma pack(4)
226   typedef struct {
227     unsigned char ld12[12];
228   } _LDBL12;
229 #pragma pack()
230 # 105 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
231   extern int * __imp___mb_cur_max;
232 # 131 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
233   typedef void (__attribute__((__cdecl__)) *_purecall_handler)(void);
234
235   __attribute__ ((__dllimport__)) _purecall_handler
        __attribute__((__cdecl__))
        _set_purecall_handler(_purecall_handler _Handler);
236   __attribute__ ((__dllimport__)) _purecall_handler
        __attribute__((__cdecl__)) _get_purecall_handler(void);
237
238   typedef void (__attribute__((__cdecl__))
        *_invalid_parameter_handler)(const wchar_t *,const wchar_t
        *,const wchar_t *,unsigned int,uintptr_t);
239   __attribute__ ((__dllimport__)) _invalid_parameter_handler
        __attribute__((__cdecl__))
        _set_invalid_parameter_handler(_invalid_parameter_handler
        _Handler);
240   __attribute__ ((__dllimport__)) _invalid_parameter_handler
        __attribute__((__cdecl__)) _get_invalid_parameter_handler(void);
241
242
243
244   __attribute__ ((__dllimport__)) extern int
        *__attribute__((__cdecl__)) _errno(void);
245
246   errno_t __attribute__((__cdecl__)) _set_errno(int _Value);
247   errno_t __attribute__((__cdecl__)) _get_errno(int *_Value);
248
249   __attribute__ ((__dllimport__)) unsigned long
        *__attribute__((__cdecl__)) __doserrno(void);
250
251   errno_t __attribute__((__cdecl__)) _set_doserrno(unsigned long
        _Value);
252   errno_t __attribute__((__cdecl__)) _get_doserrno(unsigned long
        *_Value);
253
254
255
256
257   extern __attribute__ ((__dllimport__)) char *_sys_errlist[1];
258   extern __attribute__ ((__dllimport__)) int _sys_nerr;
259 # 171 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
260   extern int * __imp___argc;
261
262
263
264
265
266
267
```

```
268   extern char *** __imp___argv;
269
270
271
272
273
274
275
276   extern wchar_t *** __imp___wargv;
277 # 199 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
278   extern char *** __imp__environ;
279 # 208 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
280   extern wchar_t *** __imp__wenviron;
281 # 217 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
282   extern char ** __imp__pgmptr;
283 # 226 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
284   extern wchar_t ** __imp__wpgmptr;
285
286
287
288   errno_t __attribute__((__cdecl__)) _get_pgmptr(char **_Value);
289   errno_t __attribute__((__cdecl__)) _get_wpgmptr(wchar_t **_Value);
290
291
292
293
294   extern int * __imp__fmode;
295
296
297
298   __attribute__ ((__dllimport__)) errno_t __attribute__((__cdecl__))
        _set_fmode(int _Mode);
299   __attribute__ ((__dllimport__)) errno_t __attribute__((__cdecl__))
        _get_fmode(int *_PMode);
300
301
302
303
304
305   extern unsigned int * __imp__osplatform;
306 # 256 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
307   extern unsigned int * __imp__osver;
308 # 265 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
309   extern unsigned int * __imp__winver;
310 # 274 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
311   extern unsigned int * __imp__winmajor;
312 # 283 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
313   extern unsigned int * __imp__winminor;
314
315
316
317
318   errno_t __attribute__((__cdecl__)) _get_osplatform(unsigned int
        *_Value);
319   errno_t __attribute__((__cdecl__)) _get_osver(unsigned int *_Value);
320   errno_t __attribute__((__cdecl__)) _get_winver(unsigned int
        *_Value);
321   errno_t __attribute__((__cdecl__)) _get_winmajor(unsigned int
        *_Value);
322   errno_t __attribute__((__cdecl__)) _get_winminor(unsigned int
        *_Value);
323 # 306 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
324   void __attribute__((__cdecl__)) __attribute__ ((__nothrow__))
        exit(int _Code) __attribute__ ((__noreturn__));
325   void __attribute__((__cdecl__)) __attribute__ ((__nothrow__))
        _exit(int _Code) __attribute__ ((__noreturn__));
326
```

```
327
328
329   void __attribute__((__cdecl__)) _Exit(int) __attribute__
          ((__noreturn__));
330
331
332
333
334
335
336
337
338   void __attribute__((__cdecl__)) __attribute__((noreturn))
          abort(void);
339
340
341
342
343   __attribute__ ((__dllimport__)) unsigned int
          __attribute__((__cdecl__)) _set_abort_behavior(unsigned int
          _Flags,unsigned int _Mask);
344
345
346
347   int __attribute__((__cdecl__)) abs(int _X);
348   long __attribute__((__cdecl__)) labs(long _X);
349
350
351   __extension__ long long __attribute__((__cdecl__)) _abs64(long
          long);
352
353   extern __inline__ __attribute__((__always_inline__,__gnu_inline__))
          long long __attribute__((__cdecl__)) _abs64(long long x) {
354     return __builtin_llabs(x);
355   }
356
357
358   int __attribute__((__cdecl__)) atexit(void
          (__attribute__((__cdecl__)) *)(void));
359
360
361   double __attribute__((__cdecl__)) atof(const char *_String);
362   double __attribute__((__cdecl__)) _atof_l(const char
          *_String,_locale_t _Locale);
363
364   int __attribute__((__cdecl__)) atoi(const char *_Str);
365   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
          _atoi_l(const char *_Str,_locale_t _Locale);
366   long __attribute__((__cdecl__)) atol(const char *_Str);
367   __attribute__ ((__dllimport__)) long __attribute__((__cdecl__))
          _atol_l(const char *_Str,_locale_t _Locale);
368
369
370   void *__attribute__((__cdecl__)) bsearch(const void *_Key,const
          void *_Base,size_t _NumOfElements,size_t _SizeOfElements,int
          (__attribute__((__cdecl__)) *_PtFuncCompare)(const void *,const
          void *));
371   void __attribute__((__cdecl__)) qsort(void *_Base,size_t
          _NumOfElements,size_t _SizeOfElements,int
          (__attribute__((__cdecl__)) *_PtFuncCompare)(const void *,const
          void *));
372
373   unsigned short __attribute__((__cdecl__)) _byteswap_ushort(unsigned
          short _Short);
374   unsigned long __attribute__((__cdecl__)) _byteswap_ulong (unsigned
          long _Long);
375   __extension__ unsigned long long __attribute__((__cdecl__))
```

```
        _byteswap_uint64(unsigned long long _Int64);
376   div_t __attribute__((__cdecl__)) div(int _Numerator,int
        _Denominator);
377   char *__attribute__((__cdecl__)) getenv(const char *_VarName) ;
378   __attribute__ ((__dllimport__)) char *__attribute__((__cdecl__))
        _itoa(int _Value,char *_Dest,int _Radix);
379   __extension__ __attribute__ ((__dllimport__)) char
        *__attribute__((__cdecl__)) _i64toa(long long _Val,char
        *_DstBuf,int _Radix) ;
380   __extension__ __attribute__ ((__dllimport__)) char
        *__attribute__((__cdecl__)) _ui64toa(unsigned long long
        _Val,char *_DstBuf,int _Radix) ;
381   __extension__ __attribute__ ((__dllimport__)) long long
        __attribute__((__cdecl__)) _atoi64(const char *_String);
382   __extension__ __attribute__ ((__dllimport__)) long long
        __attribute__((__cdecl__)) _atoi64_l(const char
        *_String,_locale_t _Locale);
383   __extension__ __attribute__ ((__dllimport__)) long long
        __attribute__((__cdecl__)) _strtoi64(const char *_String,char
        **_EndPtr,int _Radix);
384   __extension__ __attribute__ ((__dllimport__)) long long
        __attribute__((__cdecl__)) _strtoi64_l(const char *_String,char
        **_EndPtr,int _Radix,_locale_t _Locale);
385   __extension__ __attribute__ ((__dllimport__)) unsigned long long
        __attribute__((__cdecl__)) _strtoui64(const char *_String,char
        **_EndPtr,int _Radix);
386   __extension__ __attribute__ ((__dllimport__)) unsigned long long
        __attribute__((__cdecl__)) _strtoui64_l(const char
        *_String,char **_EndPtr,int _Radix,_locale_t _Locale);
387   ldiv_t __attribute__((__cdecl__)) ldiv(long _Numerator,long
        _Denominator);
388   __attribute__ ((__dllimport__)) char *__attribute__((__cdecl__))
        _ltoa(long _Value,char *_Dest,int _Radix) ;
389   int __attribute__((__cdecl__)) mblen(const char *_Ch,size_t
        _MaxCount);
390   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
        _mblen_l(const char *_Ch,size_t _MaxCount,_locale_t _Locale);
391   __attribute__ ((__dllimport__)) size_t __attribute__((__cdecl__))
        _mbstrlen(const char *_Str);
392   __attribute__ ((__dllimport__)) size_t __attribute__((__cdecl__))
        _mbstrlen_l(const char *_Str,_locale_t _Locale);
393   __attribute__ ((__dllimport__)) size_t __attribute__((__cdecl__))
        _mbstrnlen(const char *_Str,size_t _MaxCount);
394   __attribute__ ((__dllimport__)) size_t __attribute__((__cdecl__))
        _mbstrnlen_l(const char *_Str,size_t _MaxCount,_locale_t
        _Locale);
395   int __attribute__((__cdecl__)) mbtowc(wchar_t * __restrict__
        _DstCh,const char * __restrict__ _SrcCh,size_t _SrcSizeInBytes);
396   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
        _mbtowc_l(wchar_t * __restrict__ _DstCh,const char *
        __restrict__ _SrcCh,size_t _SrcSizeInBytes,_locale_t _Locale);
397   size_t __attribute__((__cdecl__)) mbstowcs(wchar_t * __restrict__
        _Dest,const char * __restrict__ _Source,size_t _MaxCount);
398   __attribute__ ((__dllimport__)) size_t __attribute__((__cdecl__))
        _mbstowcs_l(wchar_t * __restrict__ _Dest,const char *
        __restrict__ _Source,size_t _MaxCount,_locale_t _Locale);
399   int __attribute__((__cdecl__)) rand(void);
400   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
        _set_error_mode(int _Mode);
401   void __attribute__((__cdecl__)) srand(unsigned int _Seed);
402 # 404 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
403   double __attribute__((__cdecl__)) __attribute__ ((__nothrow__))
        strtod(const char * __restrict__ _Str,char ** __restrict__
        _EndPtr);
404   float __attribute__((__cdecl__)) __attribute__ ((__nothrow__))
        strtof(const char * __restrict__ nptr, char ** __restrict__
        endptr);
```

```
405
406   long double __attribute__((__cdecl__)) __attribute__
          ((__nothrow__)) strtold(const char * __restrict__ , char **
          __restrict__ );
407
408
409   extern double __attribute__((__cdecl__)) __attribute__
          ((__nothrow__))
410   __strtod (const char * __restrict__ , char ** __restrict__);
411
412
413
414
415   float __attribute__((__cdecl__)) __mingw_strtof (const char *
          __restrict__, char ** __restrict__);
416   double __attribute__((__cdecl__)) __mingw_strtod (const char *
          __restrict__, char ** __restrict__);
417   long double __attribute__((__cdecl__)) __mingw_strtold(const char *
          __restrict__, char ** __restrict__);
418
419   __attribute__ ((__dllimport__)) double __attribute__((__cdecl__))
          _strtod_l(const char * __restrict__ _Str,char ** __restrict__
          _EndPtr,_locale_t _Locale);
420   long __attribute__((__cdecl__)) strtol(const char * __restrict__
          _Str,char ** __restrict__ _EndPtr,int _Radix);
421   __attribute__ ((__dllimport__)) long __attribute__((__cdecl__))
          _strtol_l(const char * __restrict__ _Str,char ** __restrict__
          _EndPtr,int _Radix,_locale_t _Locale);
422   unsigned long __attribute__((__cdecl__)) strtoul(const char *
          __restrict__ _Str,char ** __restrict__ _EndPtr,int _Radix);
423   __attribute__ ((__dllimport__)) unsigned long
          __attribute__((__cdecl__)) _strtoul_l(const char * __restrict__
          _Str,char ** __restrict__ _EndPtr,int _Radix,_locale_t _Locale);
424
425
426   int __attribute__((__cdecl__)) system(const char *_Command);
427
428   __attribute__ ((__dllimport__)) char *__attribute__((__cdecl__))
          _ultoa(unsigned long _Value,char *_Dest,int _Radix) ;
429   int __attribute__((__cdecl__)) wctomb(char *_MbCh,wchar_t _WCh) ;
430   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
          _wctomb_l(char *_MbCh,wchar_t _WCh,_locale_t _Locale) ;
431   size_t __attribute__((__cdecl__)) wcstombs(char * __restrict__
          _Dest,const wchar_t * __restrict__ _Source,size_t _MaxCount) ;
432   __attribute__ ((__dllimport__)) size_t __attribute__((__cdecl__))
          _wcstombs_l(char * __restrict__ _Dest,const wchar_t *
          __restrict__ _Source,size_t _MaxCount,_locale_t _Locale) ;
433
434
435
436   void *__attribute__((__cdecl__)) calloc(size_t
          _NumOfElements,size_t _SizeOfElements);
437   void __attribute__((__cdecl__)) free(void *_Memory);
438   void *__attribute__((__cdecl__)) malloc(size_t _Size);
439   void *__attribute__((__cdecl__)) realloc(void *_Memory,size_t
          _NewSize);
440   __attribute__ ((__dllimport__)) void *__attribute__((__cdecl__))
          _recalloc(void *_Memory,size_t _Count,size_t _Size);
441
442
443
444
445
446
447   __attribute__ ((__dllimport__)) void __attribute__((__cdecl__))
          _aligned_free(void *_Memory);
448   __attribute__ ((__dllimport__)) void *__attribute__((__cdecl__))
```

```
        _aligned_malloc(size_t _Size,size_t _Alignment);
449
450
451
452   __attribute__ ((__dllimport__)) void *__attribute__((__cdecl__))
        _aligned_offset_malloc(size_t _Size,size_t _Alignment,size_t
        _Offset);
453   __attribute__ ((__dllimport__)) void *__attribute__((__cdecl__))
        _aligned_realloc(void *_Memory,size_t _Size,size_t _Alignment);
454   __attribute__ ((__dllimport__)) void *__attribute__((__cdecl__))
        _aligned_recalloc(void *_Memory,size_t _Count,size_t
        _Size,size_t _Alignment);
455   __attribute__ ((__dllimport__)) void *__attribute__((__cdecl__))
        _aligned_offset_realloc(void *_Memory,size_t _Size,size_t
        _Alignment,size_t _Offset);
456   __attribute__ ((__dllimport__)) void *__attribute__((__cdecl__))
        _aligned_offset_recalloc(void *_Memory,size_t _Count,size_t
        _Size,size_t _Alignment,size_t _Offset);
457
458
459
460
461
462   __attribute__ ((__dllimport__)) wchar_t *__attribute__((__cdecl__))
        _itow(int _Value,wchar_t *_Dest,int _Radix) ;
463   __attribute__ ((__dllimport__)) wchar_t *__attribute__((__cdecl__))
        _ltow(long _Value,wchar_t *_Dest,int _Radix) ;
464   __attribute__ ((__dllimport__)) wchar_t *__attribute__((__cdecl__))
        _ultow(unsigned long _Value,wchar_t *_Dest,int _Radix) ;
465
466   double __attribute__((__cdecl__)) __mingw_wcstod(const wchar_t *
        __restrict__ _Str,wchar_t ** __restrict__ _EndPtr);
467   float __attribute__((__cdecl__)) __mingw_wcstof(const wchar_t *
        __restrict__ nptr, wchar_t ** __restrict__ endptr);
468   long double __attribute__((__cdecl__)) __mingw_wcstold(const
        wchar_t * __restrict__, wchar_t ** __restrict__);
469 # 482 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
470   double __attribute__((__cdecl__)) wcstod(const wchar_t *
        __restrict__ _Str,wchar_t ** __restrict__ _EndPtr);
471   float __attribute__((__cdecl__)) wcstof(const wchar_t *
        __restrict__ nptr, wchar_t ** __restrict__ endptr);
472
473
474   long double __attribute__((__cdecl__)) wcstold(const wchar_t *
        __restrict__, wchar_t ** __restrict__);
475
476   __attribute__ ((__dllimport__)) double __attribute__((__cdecl__))
        _wcstod_l(const wchar_t * __restrict__ _Str,wchar_t **
        __restrict__ _EndPtr,_locale_t _Locale);
477   long __attribute__((__cdecl__)) wcstol(const wchar_t * __restrict__
        _Str,wchar_t ** __restrict__ _EndPtr,int _Radix);
478   __attribute__ ((__dllimport__)) long __attribute__((__cdecl__))
        _wcstol_l(const wchar_t * __restrict__ _Str,wchar_t **
        __restrict__ _EndPtr,int _Radix,_locale_t _Locale);
479   unsigned long __attribute__((__cdecl__)) wcstoul(const wchar_t *
        __restrict__ _Str,wchar_t ** __restrict__ _EndPtr,int _Radix);
480   __attribute__ ((__dllimport__)) unsigned long
        __attribute__((__cdecl__)) _wcstoul_l(const wchar_t *
        __restrict__ _Str,wchar_t ** __restrict__ _EndPtr,int
        _Radix,_locale_t _Locale);
481   __attribute__ ((__dllimport__)) wchar_t *__attribute__((__cdecl__))
        _wgetenv(const wchar_t *_VarName) ;
482
483
484   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
        _wsystem(const wchar_t *_Command);
485
```

```
486   __attribute__ ((__dllimport__)) double __attribute__((__cdecl__))
          _wtof(const wchar_t *_Str);
487   __attribute__ ((__dllimport__)) double __attribute__((__cdecl__))
          _wtof_l(const wchar_t *_Str,_locale_t _Locale);
488   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
          _wtoi(const wchar_t *_Str);
489   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
          _wtoi_l(const wchar_t *_Str,_locale_t _Locale);
490   __attribute__ ((__dllimport__)) long __attribute__((__cdecl__))
          _wtol(const wchar_t *_Str);
491   __attribute__ ((__dllimport__)) long __attribute__((__cdecl__))
          _wtol_l(const wchar_t *_Str,_locale_t _Locale);
492
493   __extension__ __attribute__ ((__dllimport__)) wchar_t
          *__attribute__((__cdecl__)) _i64tow(long long _Val,wchar_t
          *_DstBuf,int _Radix) ;
494   __extension__ __attribute__ ((__dllimport__)) wchar_t
          *__attribute__((__cdecl__)) _ui64tow(unsigned long long
          _Val,wchar_t *_DstBuf,int _Radix) ;
495   __extension__ __attribute__ ((__dllimport__)) long long
          __attribute__((__cdecl__)) _wtoi64(const wchar_t *_Str);
496   __extension__ __attribute__ ((__dllimport__)) long long
          __attribute__((__cdecl__)) _wtoi64_l(const wchar_t
          *_Str,_locale_t _Locale);
497   __extension__ __attribute__ ((__dllimport__)) long long
          __attribute__((__cdecl__)) _wcstoi64(const wchar_t
          *_Str,wchar_t **_EndPtr,int _Radix);
498   __extension__ __attribute__ ((__dllimport__)) long long
          __attribute__((__cdecl__)) _wcstoi64_l(const wchar_t
          *_Str,wchar_t **_EndPtr,int _Radix,_locale_t _Locale);
499   __extension__ __attribute__ ((__dllimport__)) unsigned long long
          __attribute__((__cdecl__)) _wcstoui64(const wchar_t
          *_Str,wchar_t **_EndPtr,int _Radix);
500   __extension__ __attribute__ ((__dllimport__)) unsigned long long
          __attribute__((__cdecl__)) _wcstoui64_l(const wchar_t *_Str
          ,wchar_t **_EndPtr,int _Radix,_locale_t _Locale);
501
502
503
504
505   __attribute__ ((__dllimport__)) char *__attribute__((__cdecl__))
          _fullpath(char *_FullPath,const char *_Path,size_t
          _SizeInBytes);
506   __attribute__ ((__dllimport__)) char *__attribute__((__cdecl__))
          _ecvt(double _Val,int _NumOfDigits,int *_PtDec,int *_PtSign) ;
507   __attribute__ ((__dllimport__)) char *__attribute__((__cdecl__))
          _fcvt(double _Val,int _NumOfDec,int *_PtDec,int *_PtSign) ;
508   __attribute__ ((__dllimport__)) char *__attribute__((__cdecl__))
          _gcvt(double _Val,int _NumOfDigits,char *_DstBuf) ;
509   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
          _atodbl(_CRT_DOUBLE *_Result,char *_Str);
510   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
          _atoldbl(_LDOUBLE *_Result,char *_Str);
511   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
          _atoflt(_CRT_FLOAT *_Result,char *_Str);
512   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
          _atodbl_l(_CRT_DOUBLE *_Result,char *_Str,_locale_t _Locale);
513   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
          _atoldbl_l(_LDOUBLE *_Result,char *_Str,_locale_t _Locale);
514   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
          _atoflt_l(_CRT_FLOAT *_Result,char *_Str,_locale_t _Locale);
515
516
517
518
519
520   __extension__ unsigned long long __attribute__((__cdecl__))
```

```
        _lrotl(unsigned long long _Val,int _Shift);
521     __extension__ unsigned long long __attribute__((__cdecl__))
        _lrotr(unsigned long long _Val,int _Shift);
522
523
524
525
526
527
528
529     __attribute__ ((__dllimport__)) void __attribute__((__cdecl__))
        _makepath(char *_Path,const char *_Drive,const char *_Dir,const
        char *_Filename,const char *_Ext);
530     _onexit_t __attribute__((__cdecl__)) _onexit(_onexit_t _Func);
531
532
533
534     void __attribute__((__cdecl__)) perror(const char *_ErrMsg);
535
536     __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
        _putenv(const char *_EnvString);
537
538
539
540
541     __extension__ unsigned long long __attribute__((__cdecl__))
        _rotl64(unsigned long long _Val,int _Shift);
542     __extension__ unsigned long long __attribute__((__cdecl__))
        _rotr64(unsigned long long Value,int Shift);
543
544
545
546
547
548
549     unsigned int __attribute__((__cdecl__)) _rotr(unsigned int _Val,int
        _Shift);
550     unsigned int __attribute__((__cdecl__)) _rotl(unsigned int _Val,int
        _Shift);
551
552
553     __extension__ unsigned long long __attribute__((__cdecl__))
        _rotr64(unsigned long long _Val,int _Shift);
554     __attribute__ ((__dllimport__)) void __attribute__((__cdecl__))
        _searchenv(const char *_Filename,const char *_EnvVar,char
        *_ResultPath) ;
555     __attribute__ ((__dllimport__)) void __attribute__((__cdecl__))
        _splitpath(const char *_FullPath,char *_Drive,char *_Dir,char
        *_Filename,char *_Ext) ;
556     __attribute__ ((__dllimport__)) void __attribute__((__cdecl__))
        _swab(char *_Buf1,char *_Buf2,int _SizeInBytes);
557
558
559
560     __attribute__ ((__dllimport__)) wchar_t *__attribute__((__cdecl__))
        _wfullpath(wchar_t *_FullPath,const wchar_t *_Path,size_t
        _SizeInWords);
561     __attribute__ ((__dllimport__)) void __attribute__((__cdecl__))
        _wmakepath(wchar_t *_ResultPath,const wchar_t *_Drive,const
        wchar_t *_Dir,const wchar_t *_Filename,const wchar_t *_Ext);
562
563
564     __attribute__ ((__dllimport__)) void __attribute__((__cdecl__))
        _wperror(const wchar_t *_ErrMsg);
565
566     __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
        _wputenv(const wchar_t *_EnvString);
```

```
567   __attribute__ ((__dllimport__)) void __attribute__((__cdecl__))
          _wsearchenv(const wchar_t *_Filename,const wchar_t
          *_EnvVar,wchar_t *_ResultPath) ;
568   __attribute__ ((__dllimport__)) void __attribute__((__cdecl__))
          _wsplitpath(const wchar_t *_FullPath,wchar_t *_Drive,wchar_t
          *_Dir,wchar_t *_Filename,wchar_t *_Ext) ;
569
570
571   __attribute__ ((__dllimport__)) void __attribute__((__cdecl__))
          _beep(unsigned _Frequency,unsigned _Duration) __attribute__
          ((__deprecated__));
572
573   __attribute__ ((__dllimport__)) void __attribute__((__cdecl__))
          _seterrormode(int _Mode) __attribute__ ((__deprecated__));
574   __attribute__ ((__dllimport__)) void __attribute__((__cdecl__))
          _sleep(unsigned long _Duration) __attribute__
          ((__deprecated__));
575 # 607 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
576   char *__attribute__((__cdecl__)) ecvt(double _Val,int
          _NumOfDigits,int *_PtDec,int *_PtSign) ;
577   char *__attribute__((__cdecl__)) fcvt(double _Val,int _NumOfDec,int
          *_PtDec,int *_PtSign) ;
578   char *__attribute__((__cdecl__)) gcvt(double _Val,int
          _NumOfDigits,char *_DstBuf) ;
579   char *__attribute__((__cdecl__)) itoa(int _Val,char *_DstBuf,int
          _Radix) ;
580   char *__attribute__((__cdecl__)) ltoa(long _Val,char *_DstBuf,int
          _Radix) ;
581   int __attribute__((__cdecl__)) putenv(const char *_EnvString) ;
582
583
584
585   void __attribute__((__cdecl__)) swab(char *_Buf1,char *_Buf2,int
          _SizeInBytes) ;
586
587
588   char *__attribute__((__cdecl__)) ultoa(unsigned long _Val,char
          *_Dstbuf,int _Radix) ;
589   _onexit_t __attribute__((__cdecl__)) onexit(_onexit_t _Func);
590
591
592
593
594
595   typedef struct { __extension__ long long quot, rem; } lldiv_t;
596
597   __extension__ lldiv_t __attribute__((__cdecl__)) lldiv(long long,
          long long);
598
599   __extension__ long long __attribute__((__cdecl__)) llabs(long long);
600
601
602
603
604   __extension__ long long __attribute__((__cdecl__)) strtoll(const
          char * __restrict__, char ** __restrict, int);
605   __extension__ unsigned long long __attribute__((__cdecl__))
          strtoull(const char * __restrict__, char ** __restrict__, int);
606
607
608   __extension__ long long __attribute__((__cdecl__)) atoll (const
          char *);
609
610
611   __extension__ long long __attribute__((__cdecl__)) wtoll (const
          wchar_t *);
612   __extension__ char *__attribute__((__cdecl__)) lltoa (long long,
```

```
        char *, int);
613   __extension__ char *__attribute__((__cdecl__)) ulltoa (unsigned
        long long , char *, int);
614   __extension__ wchar_t *__attribute__((__cdecl__)) lltow (long long,
        wchar_t *, int);
615   __extension__ wchar_t *__attribute__((__cdecl__)) ulltow (unsigned
        long long, wchar_t *, int);
616 # 665 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 3
617 #pragma pack(pop)
618
619 # 1
      "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/sec_api/stdlib_s.h"
      1 3
620 # 9
      "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/sec_api/stdlib_s.h"
      3
621 # 1 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 1 3
622 # 10
      "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/sec_api/stdlib_s.h"
      2 3
623 # 668 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 2 3
624 # 1 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/malloc.h" 1 3
625 # 11 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/malloc.h" 3
626 #pragma pack(push,_CRT_PACKING)
627 # 46 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/malloc.h" 3
628   typedef struct _heapinfo {
629     int *_pentry;
630     size_t _size;
631     int _useflag;
632   } _HEAPINFO;
633
634
635   extern unsigned int _amblksiz;
636 # 65 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/malloc.h" 3
637
638
639 # 97 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/malloc.h" 3
640
641
642
643
644
645
646 void * __mingw_aligned_malloc (size_t _Size, size_t _Alignment);
647 void __mingw_aligned_free (void *_Memory);
648 void * __mingw_aligned_offset_realloc (void *_Memory, size_t _Size,
      size_t _Alignment, size_t _Offset);
649 void * __mingw_aligned_realloc (void *_Memory, size_t _Size, size_t
      _Offset);
650
651
652
653   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
        _resetstkoflw (void);
654   __attribute__ ((__dllimport__)) unsigned long
        __attribute__((__cdecl__)) _set_malloc_crt_max_wait(unsigned
        long _NewValue);
655
656   __attribute__ ((__dllimport__)) void *__attribute__((__cdecl__))
        _expand(void *_Memory,size_t _NewSize);
657   __attribute__ ((__dllimport__)) size_t __attribute__((__cdecl__))
        _msize(void *_Memory);
658
659
660
661
662
```

```
663
664   __attribute__ ((__dllimport__)) size_t __attribute__((__cdecl__))
          _get_sbh_threshold(void);
665   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
          _set_sbh_threshold(size_t _NewValue);
666   __attribute__ ((__dllimport__)) errno_t __attribute__((__cdecl__))
          _set_amblksiz(size_t _Value);
667   __attribute__ ((__dllimport__)) errno_t __attribute__((__cdecl__))
          _get_amblksiz(size_t *_Value);
668   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
          _heapadd(void *_Memory,size_t _Size);
669   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
          _heapchk(void);
670   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
          _heapmin(void);
671   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
          _heapset(unsigned int _Fill);
672   __attribute__ ((__dllimport__)) int __attribute__((__cdecl__))
          _heapwalk(_HEAPINFO *_EntryInfo);
673   __attribute__ ((__dllimport__)) size_t __attribute__((__cdecl__))
          _heapused(size_t *_Used,size_t *_Commit);
674   __attribute__ ((__dllimport__)) intptr_t __attribute__((__cdecl__))
          _get_heap_handle(void);
675 # 144 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/malloc.h" 3
676   static __inline void *_MarkAllocaS(void *_Ptr,unsigned int _Marker)
          {
677     if(_Ptr) {
678       *((unsigned int*)_Ptr) = _Marker;
679       _Ptr = (char*)_Ptr + 16;
680     }
681     return _Ptr;
682   }
683 # 163 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/malloc.h" 3
684   static __inline void __attribute__((__cdecl__)) _freea(void
          *_Memory) {
685     unsigned int _Marker;
686     if(_Memory) {
687       _Memory = (char*)_Memory - 16;
688       _Marker = *(unsigned int *)_Memory;
689       if(_Marker==0xDDDD) {
690  free(_Memory);
691       }
692
693
694
695
696
697     }
698   }
699 # 209 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/malloc.h" 3
700 #pragma pack(pop)
701 # 669 "D:/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include/stdlib.h" 2 3
702 # 5 "hello0.c" 2
703 # 1 "advio.h" 1
704
705
706
707
708
709
710
711 void StampaStringa (char *s);
712
713
714 void LeggeRiga (char *s);
715
716
```

```
717 void LeggeParola (char *s);
718
719
720 void StampaCarattere (char c, int num);
721
722
723 char LeggeCarattere ();
724
725
726 void ACapo ();
727
728
729 void StampaIntero (int n);
730
731
732 int LeggeIntero ();
733
734
735 void StampaNaturale (unsigned int n);
736
737
738 unsigned int LeggeNaturale ();
739
740
741 void StampaLong (long n);
742
743
744 long LeggeLong ();
745
746
747 void StampaFloat (float f);
748
749
750 float LeggeFloat ();
751
752
753 void StampaDouble (double d);
754
755
756 double LeggeDouble ();
757
758
759 void StampaIndirizzo (void *i);
760
761
762 void StampaCifraDenaro (float f);
763
764
765 float LeggeCifraDenaro ();
766
767
768 int ConverteStringaInIntero (char *s);
769
770
771 int LunghezzaStringa (char *s);
772 # 6 "hello0.c" 2
773
774
775
776
777
778 char AcquisisceCornice ();
779
780
781
782 int main (int argc, char *argv[])
783 {
```

```
784
785   char cornice;
786
787
788
789
790
791   cornice = AcquisisceCornice();
792
793
794
795
796
797   return 0;
798 }
799
800
801
802
803
804 char AcquisisceCornice ()
805 {
806 }
```

Riassumendo, il precompilatore:

1. cancella i commenti
2. include fisicamente il contenuto dei file di intestazione indicati dalle direttive `#include`
3. sostituisce meccanicamente le macro definite dalle direttive `#define` con la loro espansione.

Passiamo alla compilazione vera e propria. Non esiste un'opzione per partire dal file precompilato e applicargli la compilazione. Esiste però l'opzione `-c` per eseguire in sequenza le prime due fasi (vedi Figura 1.44):

```
gcc -c hello0.c
```

Questo crea il file precompilato e lo trasforma in binario. Il risultato non viene stampato a video (non avrebbe molto senso stampare un file binario), ma automaticamente salvato su un file che ha lo stesso nome dell'unico file `.c` e ha estensione `.o`[13]. Volendo, si può specificare esplicitamente il nome del file binario (che in gergo si chiama *file oggetto*), facendolo precedere dall'opzione `-o`. Questo serve in particolare se si vuole cambiarne il nome:

```
gcc -c hello0.c -c hello.o
```

L'automatismo è reso possibile dalla relazione biunivoca fra file `.c` e file `.o`. Aprendo il file oggetto con un *editor*, non si riesce a leggere quasi nulla, salvo qualche parola testuale qua e là. Infatti, non è un file di testo e non è destinato agli utenti umani, ma al processore.

Il progetto, però, è composto da altri file: c'è anche un file `advio.c`, che va a sua volta precompilato e compilato, producendo un file `advio.o`. E poi i due file oggetto vanno collegati in un unico file eseguibile.

La terza fase di compilazione, cioè il collegamento, fonde i due file oggetto in un file eseguibile. Ancora una volta, il compilatore che usiamo non consente di eseguire solo la terza fase, ma consente di eseguirle tutte in sequenza: basta lanciare

[13] In molti ambienti Windows, l'estensione predefinita è invece `.obj`.

il comando senza opzioni, ma riportando l'elenco di tutti i file `.c` (in un ordine qualsiasi).

```
gcc -c hello0.c advio.c
```

Se non si specifica il nome del file eseguibile, per convenzione questo viene battezzato `a.` (in ambiente Windows, sarà `a.exe`). Siccome converrà dare un nome significativo all'eseguibile, aggiungiamo la solita opzione `-o` per poterlo fare[14]:

```
gcc -c hello0.c advio.c -o hello
```

Per trasformare un programma C in eseguibile si usa il programma `gcc`

- `gcc -E modulo.c -o modulo_precompilato.c`
  esegue solo la precompilazione su un modulo C
  Non si usa praticamente mai
- `gcc -c modulo.c -o modulo.o`
  esegue precompilazione e compilazione su un modulo C
  producendo un file oggetto
- `gcc modulo.c modulo2.c modulo3.c -o prog`
  esegue precompilazione, compilazione e collegamento sui moduli
  producendo un file eseguibile (*in Windows*, `prog.exe`)

Figura 1.44: Istruzioni di compilazione

Possiamo anche eseguire questo file, che ovviamente non fa nulla, perché abbiamo solo aggiunto la funzione `AcquisisceCornice`, che si limita a restituire uno spazio bianco. Però abbiamo visto che tutto funziona.

Ora è interessante osservare che cosa succederebbe se avessimo inserito la chiamata ad `AcquisisceCornice` senza definire la variabile `cornice` e la procedura stessa. Togliamo le due dichiarazioni e la definizione della funzione secondaria. Si ottiene il messaggio[15]

```
hello0.c: In function 'main':
hello0.c:21:3: error: 'cornice' undeclared (first use in this function)
cornice = AcquisisceCornice();
^
hello02.c:21:3: note: each undeclared identifier is reported only once
for each function it appears in
```

È fondamentale imparare a leggere i messaggi di errore (vedi Figura 1.45). Nella riga 21 (e colonna 3) del file `hello0.c` il simbolo `cornice` non è dichiarato. Infatti, in quella posizione compare il simbolo, nell'istruzione che gli assegna il risultato di `AcquisisceCornice`. Questo simbolo è stato usato, ma non dichiarato, per cui il compilatore non sa che cosa significhi. Aggiungendo la dichiarazione di `cornice`, il messaggio cambia.

```
hello02.c: In function 'main':
hello02.c:22:3: warning: implicit declaration of function 'AcquisisceCornice'
```

[14] In ambiente Windows il file eseguibile dovrà avere estensione `.exe`, altrimenti non sarà utilizzabile.

[15] Ovviamente, i messaggi variano con il compilatore, e anche con la singola sua versione.

```
[-Wimplicit-function-declaration]
cornice = AcquisisceCornice();
^
hello02.c:16:8:  warning:  variable 'cornice' set but not used [-Wunused-but-set-variable]
char cornice;
^
C:\Users\utente\AppData\Local\Temp\ccYqmo5h.o:hello02.c:(.text+0x15):  undefined
reference to ``AcquisisceCornice''
collect2.exe:  error:  ld returned 1 exit status
```

Questa volta, è più criptico: dice che la funzione `AcquisisceCornice` era stata precedentemente dichiarata in modo "implicito", anche se in realtà non lo è stata. Notiamo anche che si tratta di un avvertimento (*warning*), e non di un errore. Il motivo è che le funzioni potrebbero essere dichiarate anche in altri file, per cui la seconda fase di compilazione non è certa che la dichiarazione manchi, ma certamente manca un prototipo. Per convenzione in C una funzione non dichiarata si presuppone che restituisca un intero. Sono residui di vecchie convenzioni che conviene non sfruttare. Segue un altro avvertimento: che la variabile `cornice` ha ricevuto un valore, ma non viene usata. Non è un errore, ma è una stranezza, e i buoni compilatori le segnalano perché potrebbero essere indice di errori semantici (cioè concettuali) compiuti dal programmatore. Infine, le ultime due righe, che vengono dalla terza fase di compilazione, segnalano un errore vero e proprio. Eseguendo il collegamento, infatti, si scopre che effettivamente `AcquisisceCornice` non è definita in alcuno dei file del progetto, e questo è certamente un errore.

In linea di principio, ogni messaggio di errore, ma anche di avvertimento, va preso in seria considerazione e compreso nel suo significato. Per aiutarsi, conviene quindi chiedere al compilatore la massima pignoleria. I compilatori offrono la possibilità di modulare la precisione del proprio lavoro di verifica con opportune opzioni. Durante il corso useremo quindi le opzioni

- `-Wall` per indicare di stampare tutti i messaggi possibili;
- `-pedantic` per essere molto pedante nell'esame del codice.

1.7.4 Stesura top-down del codice (seguito)

Riprendiamo a stendere il codice, passando all'acquisizione del saluto. Verrebbe naturale fare una cosa simile all'acquisizione della cornice, come:

```
saluto = AcquisisceSaluto();
```

ma non si può per motivi tecnici: le procedure C non possono restituire risultati multipli, e un saluto è una sequenza di caratteri, non un carattere singolo. La soluzione, per motivi che vedremo nel Capitolo **??** è scrivere i risultati composti come se fossero dati:

```
AcquisisceSaluto(saluto);
```

A parte questa stranezza, procediamo in tre passi come prima: aggiungiamo la chiamata nel corpo del programma principale (e la dichiarazione della variabile `saluto`), il prototipo della funzione nella parte precedente e la definizione nella parte successiva. La dichiarazione del saluto assume la forma

```
char saluto[ROW_LENGTH];
```

Il compilatore fa un controllo sugli errori sintattici del codice e li segnala

Alcuni costrutti sintattici possono essere corretti sintatticamente,
ma sospetti, perché spesso abbinati a errori semantici
(*per es., usare variabili non inizializzate*)

Il programma `gcc` segnala alcuni di questi costrutti con degli *warning*:
conviene tenerli d'occhio e risolverli per

- ridurre gli errori semantici
- aumentare la portabilità

Le seguenti opzioni regolano gli avvertimenti

- `-Wall`: indica di stampare tutti i possibili messaggi di avvertimento
- `-pedantic`: indica di richiedere un'aderenza stretta al C standard

Figura 1.45: Errori e avvertimenti

per indicare che è un vettore composto da un certo numero di caratteri. Siccome non conosciamo la lunghezza del saluto, ne faremo una stima per eccesso e le daremo un nome simbolico (`ROW_LENGTH`), attraverso una macro già definita nel file `advio.h`, in modo da poter cambiare il valore in modo semplice, qualora fosse necessario.

Quindi, il prototipo fornirà nome della procedura, dati (assenti) e risultato, spostato nella posizione dei dati per il problema prima introdotto. Mancando quindi un risultato "ufficiale", sostituiamo il tipo del risultato con la parola chiave `void`.

```
void AcquisisceSaluto (char *saluto);
```

Del risultato bisogna specificare il tipo, che apparentemente dovrebbe essere il vettore di caratteri già dichiarato nel programma principale. In realta', per un motivo tecnico che sarà anch'esso discusso nel Capitolo **??** quando un vettore viene passato a una procedura, il suo tipo diventa un puntatore, e quindi cambia forma, da `char [ROW_LENGTH]` a `char *`.

Infine, si definisce la procedura copiando la dichiarazione senza punto e virgola e aggiungendo un corpo, per ora vuoto.

```
void AcquisisceSaluto (char *saluto)
{
}
```

Questa volta non occorre nemmeno restituire un valore convenzionale perché la procedura non ha risultati.

Faremo gli stessi tre passi per risolvere in modalità *top-down* il problema di stampare il saluto. Aggiungiamo una chiamata

```
StampaSaluto(saluto,cornice)
```

che ha come dati il saluto e la cornice e non ha risultati (non si calcola nulla: si fa solo una stampa). Poi aggiungiamo il prototipo

```
void StampaSaluto (char *saluto, char cornice);
```

con i due dati e i loro tipi già discussi e il risultato assente. Infine, aggiungiamo una definizione vuota

```
void StampaSaluto (char *saluto, char cornice)
{
}
```

Il codice nel suo complesso è sintatticamente corretto, dunque compila senza messaggi di errore o avvertimento. Ovviamente, non fa nulla, ma la cosa importante è che il programma non si blocca e non dà errori nemmeno in esecuzione. Il vantaggio di questo approccio è che ricompilando e rieseguendo ogni poche righe si può capire immediatamente se si stanno aggiungendo errori e si possono correggere prima che interagiscano fra loro e diventino difficili da risolvere e richiedano l'analisi di un codice molto lungo. Al contrario, scrivere molte righe di codice senza verificarle produce anche l'effetto che, commesso un errore, il compilatore non riesca più a dare un significato al resto del codice, anche se corretto, e quindi stampi centinaia di messaggi, di cui solo il primo è motivato. Purtroppo, non esistono solo errori sintattici, ma anche semantici, e capita che ricompilando e rieseguendo non si trovino immediatamente tutti gli errori, ma in generale le ultime righe aggiunte, se non sono scorrette sono quanto meno quelle che consentono all'errore di manifestarsi, e quindi dovrebbero avere un legame concettuale con l'errore stesso.

Si può osservare che la scelta dei nomi delle procedure e delle variabili è tale che, anche se non ci fossero commenti, un ipotetico lettore potrebbe capire che cosa sta facendo il codice.

L'approccio *top-down* richiede ora di risolvere i sottoproblemi, cioè di passare a realizzare le procedure di livello inferiore. Abbiamo deciso di decomporre l'acquisizione della cornice nella stampa di una stringa di istruzioni sul video e nella lettura di un carattere da tastiera. Fortunatamente, entrambi i sottoproblemi sono già risolti da procedure della libreria `advio.h`. Si tratta quindi semplicemente di recuperare la procedure dalla libreria e usarle. La procedura `StampaStringa` richiede un `char *`, che abbiamo già visto essere un vettore di caratteri, cioè una stringa. Può essere una variabile o una costante. In questo secondo caso, che ci interessa qui, bisogna indicare al compilatore che i caratteri della stringa vanno intesi esattamente come sono, e non come istruzioni o nomi di variabili o di procedure. Per farlo, basta racchiuderli fra virgolette (`""`). La procedura `LeggeCarattere`, da parte sua, aspetta che da tastiera l'utente inserisca un carattere (seguito da invio). Questo carattere va salvato temporaneamente da qualche parte in modo da poterlo restituire alla procedura chiamante. Basterà dichiarare (nella parte dichiarativa, ovviamente) una variabile `cornice`[16] di tipo `char` e assegnarle il risultato della procedura. Al termine, si restituirà `cornice` al posto di `SPAZIO`.

```
char AcquisisceCornice ()
{
/* Parte dichiarativa */
char cornice;

/* Parte esecutiva */
StampaStringa("Indicare il carattere da usare per la cornice:  ");
cornice = LeggeCarattere();
return cornice;
}
```

[16] Ma il nome può essere anche diverso: vedremo che non è la stessa area di memoria usata nel programma principale dalla variabile dello stesso nome.

Notiamo che questa volta non procediamo con le solite tre fasi (chiamata, dichiarazione e definizione) per il semplice motivo che le due funzioni chiamate hanno già un'intestazione nella sezione dei prototipi, grazie alla direttiva che include il file `advio.h` (basta ricompilare con l'opzione `-E` per sincerarsene) e hanno già una definizione nella sezione delle definizioni, anche se questa sta nel file `advio.c`. La direttiva di inclusione permette di includere un gran numero di prototipi di funzioni che sarebbe fastidioso includere esplicitamente.

Se compiliamo ed eseguiamo il codice, questa volta compare l'indicazione di inserire un carattere. Non succede altro perché mancano le altre due procedure. Per acquisire il saluto, procediamo in modo del tutto analogo, con poche differenze. Stamperemo un'istruzione diversa, ma sempre usando `StampaStringa`. Poi useremo `LeggeRiga` per ottenere il saluto inserito dall'utente con la tastiera, notando che questa funzione ha il proprio risultato nell'elenco dei dati, per la ragione tecnica cui già si è accennato. La variabile `saluto` in cui salvare il dato non va dichiarata, perché già compare nell'elenco dei dati della procedura. Infine, non occorre l'istruzione `return`, dato che ufficialmente la procedura non ha risultati. Quindi la parte dichiarativa questa volta è vuota.

```
void AcquisisceSaluto (char *saluto)
{
/* Parte esecutiva */
StampaStringa("Indicare il saluto da stampare:  ");
LeggeRiga(saluto);
}
```

Veniamo al sottoproblema più interessante, che è diviso in 10 sottoproblemi (vedi Figura 1.20). Avendo già progettato la divisione, è sensato usarla come una serie di commenti che guidino la stesura del codice in modo da non dover saltabeccare fra codice e appunti, da avere già pronti i commenti e (come vedremo) anche una traccia per eventuali relazioni sul proprio lavoro.

```
 /* Determina la larghezza della cornice */
/* Stampa la cornice superiore */
/* Va a capo */
/* Stampa la cornice sinistra */
/* Stampa uno spazio */
/* Stampa il saluto */
/* Stampa uno spazio */
/* Stampa la cornice destra */
/* Va a capo */
/* Stampa la cornice inferiore */
```

Il primo sottoproblema è determinare la larghezza della cornice, che è pari alla lunghezza del saluto più 4 caratteri corrispondenti ai due spazi bianchi che lo separano dalla cornice e ai due caratteri della cornice laterale. Occorre sapere la lunghezza di una stringa che conosciamo. La funzione `LunghezzaStringa` fornisce questa informazione. Quindi:

```
int larghezza;
larghezza = LunghezzaStringa(saluto) + 4;
```

dove ovviamente bisogna dichiarare nome e tipo della variabile `larghezza` che contiene l'informazione prima di assegnarle un valore.

Ora bisogna stampare la cornice superiore, cioè una sequenza di caratteri identici a `cornice` in numero pari a `larghezza`. Consultiamo la libreria e scopriamo

che esiste una procedura `StampaCarattere` che lo fa. Se non ci fosse, potremmo rimandare il problema a un altro livello chiamando una funzione che ancora non esiste, aggiungendo prototipo e definizione vuota e rimandando a una fase successiva il riempimento della funzione stessa. Per andare a capo c'è già una funzione (vedremo poi come fare senza la libreria).

```
StampaCarattere(cornice,larghezza);
ACapo();
```

Il compilatore è in grado di manipolare il tutto perché le funzioni sono già dichiarate e definite in luoghi accessibili al compilatore (se includiamo l'intestazione e compiliamo entrambi i file `.c`).

La stampa della cornice di sinistra è semplicemente un'altra chiamata a `StampaCarattere`, sempre con il carattere `cornice`, ma questa volta con cardinalità pari a 1. La stampa dello spazio bianco è la stessa cosa, cambiando `cornice` con `SPAZIO`. La stampa del saluto è la stampa di una stringa, che abbiamo già usato per stampare le istruzioni per l'utente (la differenza è che non si stampa una stringa costante esplicita, ma la stringa variabile contenuta in `saluto`). Seguono ancora la stampa di uno spazio bianco, della cornice destra e l'andare a capo.

```
StampaCarattere(cornice,1);
StampaCarattere(SPAZIO,1);
StampaStringa(saluto);
StampaCarattere(SPAZIO,1);
StampaCarattere(cornice,1);
ACapo();
```

Infine, procediamo per la cornice inferiore come per quella superiore.

```
StampaCarattere(cornice,larghezza);
ACapo();
```

A questo punto, possiamo compilare ed eseguire più volte il codice, provando con diversi saluti e diversi caratteri e constatare che il risultato si adegua alle istruzioni. Come già osservato, questo è un algoritmo, da un punto di vista teorico, perché in base all'istanza fornita dall'utente restituisce soluzioni diverse.

1.7.5 Redirezione dell'ingresso e dell'uscita

Consideriamo ora il file `istruzioni.txt`. Il codice realizzato chiede all'utente di inserire a mano i dati. Questa è una pessima idea, perché costringe il programmatore a rispondere al codice ogni volta che lancia il programma, cosa che abbiamo consigliato di fare molto frequentemente durante la stesura. Ci sono però strumenti che consentono di evitare questo problema, offerti dal sistema operativo: la redirezione dell'ingresso e dell'uscita.

Il file `istruzioni.txt` contiene:

```
:
Ciao
```

cioè esattamente ciò che l'utente dovrebbe battere a tastiera: un carattere, un invio, un saluto, un altro invio. Sarebbe comodo poter indicare al programma che riceva i dati da questo file anziché da tastiera. È quello che fa la redirezione dell'ingresso.

```
hello < istruzioni.txt
```

specifica che l'ingresso del programma non avverrà da tastiera ma dal file `istruzioni.txt`. Si può provare, e si vede che la cosa funziona. Un piccolo difetto è che la stampa delle istruzioni all'utente sposta verso destra la cornice superiore. Questo si può risolvere togliendo le stampe oppure (se si vuole conservare la possibilità di fornire l'ingresso anche da tastiera avendo delle istruzioni esplicite) andare a capo subito dopo aver acquisito la cornice e il saluto.

La redirezione può funzionare anche per l'uscita, cioè si può indicare al programma di non stampare a video, ma su un file di testo

```
hello > risultato.txt
```

consente di stampare su file il saluto incorniciato, invece che a video. In realtà, in questo caso vengono stampati su file i messaggi di acquisizione, per cui l'utente deve sapere che cosa ci si aspetta da lui e il risultato conterrà anche le istruzioni. Ovviamente, è possibile combinare le due cose.

```
hello < istruzioni.txt > risultato.txt
```

Listing 1.5: Soluzione dell'esercizio (`hello.c`)

```
1  /* hello.c */
2
3  /* Direttive */
4  #include <stdlib.h>
5  #include "advio.h"
6
7
8  /* Prototipi delle procedure secondarie */
9
10 char AcquisisceCornice ();
11
12 void AcquisisceSaluto (char *saluto);
13
14 void StampaSaluto (char *saluto, char cornice);
15
16
17 /* Programma principale */
18 int main (int argc, char *argv[])
19 {
20   /* Parte dichiarativa */
21   char cornice;
22   char saluto[ROW_LENGTH];
23
24
25   /* Parte esecutiva */
26
27   /* Acquisisce il carattere che si vuole usare per la cornice */
28   cornice = AcquisisceCornice();
29
30   /* Acquisisce il saluto che si vuole stampare */
31   AcquisisceSaluto(saluto);
32
33   /* Stampa il saluto incorniciato */
34   StampaSaluto(saluto,cornice);
35
36   return EXIT_SUCCESS;
37 }
38
39
40 /* Definizione delle procedure secondarie */
41 char AcquisisceCornice ()
42 {
43   /* Parte dichiarativa */
```

```
44   char cornice;
45
46   /* Parte esecutiva */
47   StampaStringa("Inserire il carattere da usare per la cornice: ");
48   cornice = LeggeCarattere();
49   return cornice;
50 }
51
52
53 void AcquisisceSaluto (char *saluto)
54 {
55   /* Parte dichiarativa */
56
57   /* Parte esecutiva */
58   StampaStringa("Inserire il saluto da stampare: ");
59   LeggeRiga(saluto);
60 }
61
62
63 void StampaSaluto (char *saluto, char cornice)
64 {
65   int larghezza;
66
67
68   /* Determina la larghezza della cornice */
69   larghezza = LunghezzaStringa(saluto) + 4;
70
71   /* Va a capo */
72   ACapo();
73
74   /* Stampa la cornice superiore */
75   StampaCarattere(cornice,larghezza);
76
77   /* Va a capo */
78   ACapo();
79
80   /* Stampa la cornice sinistra */
81   StampaCarattere(cornice,1);
82
83   /* Stampa uno spazio */
84   StampaCarattere(' ',1);
85
86   /* Stampa il saluto */
87   StampaStringa(saluto);
88
89   /* Stampa uno spazio */
90   StampaCarattere(' ',1);
91
92   /* Stampa la cornice destra */
93   StampaCarattere(cornice,1);
94
95   /* Va a capo */
96   ACapo();
97
98   /* Stampa la cornice inferiore */
99   StampaCarattere(cornice,larghezza);
100 }
```

Capitolo 2

Stringhe, stream, parsing, parametri del main

Questo capitolo richiama alcuni aspetti del linguaggio C che non sono strettamente algoritmici, ma che è necessario padroneggiare per poter usare il linguaggio come strumento per risolvere problemi algoritmicamente. Lo faremo in modo molto pratico, partendo da un esempio estremamente rudimentale, che di algoritmico ha poco, ma di cui cercheremo di sottolineare quel poco per quanto possibile.

2.1 Un esercizio sull'acquisizione di dati

Si tratta di gestire una specie di catalogo di merci. Vorremmo leggere da un file di testo un elenco di articoli di un negozio che hanno un nome, un prezzo e una data di acquisto, e vorremmo riformattare queste informazioni stampandole a video nello stesso ordine, ma in modo più leggibile. Vogliamo che siano incolonnate su tre colonne:

- una prima colonna di 10 caratteri, che riporta i nomi degli articoli (allineati a sinistra);
- una seconda colonna di 9 caratteri, che contiene i prezzi;
- una terza colonna di 11 caratteri, che contiene le date, non nella forma verbale (cioè con il nome del mese), ma in quella interamente numerica.

Evidentemente, è un pretesto per ripassare il caricamento di dati da file interpretandoli (tecnicamente, il *parsing*), la stampa formattata (scrivere incolonnati i dati), la manipolazione di stringhe (in particolare, il meccanismo di conversione dei mesi da parole a numeri). È tutto molto banale ma lo scopo è capire che approccio usare. Per essere coerenti, dovremmo usare l'approccio *top-down*, come nel capitolo precedente e nel resto della dispensa. Dovremmo cioè dividere il problema in sottoproblemi e questi ulteriormente in sottosottoproblemi. Per esempio, potremmo determinare il nome del file contenente il catalogo e gestire gli articoli uno per uno. La gestione di un articolo comporterebbe il riconoscimento del nome, del prezzo, del giorno, del mese e dell'anno, la conversione del mese, la stampa nel formato richiesto. Questa volta però faremo un'eccezione (sperando di non creare confusione), adottando l'approccio *bottom-up*, cioè partendo da ciò che si sa fare a livello bassissimo e aggregando queste cose in componenti via via più complicate e sofisticate fino ad ottenere l'algoritmo che risolve il problema desiderato.

Nei capitoli seguenti, adotteremo questo approccio in modo saltuario, non sin dal principio, ma solo quando nel corso dell'approccio *top-down* avremo stabilito di adoperare certe strutture dati astratte (concetto che introdurremo nel Capitolo **??**, e quindi sapremo che sarà necessario dotarci di una serie di operazioni. Sia chiaro che anche per questo esercizio si potrebbe adottare la modalità *top-down* e che qui non lo facciamo per fornire un esempio nel quale fare altrimenti può avere senso. La logica è partire non dal problema dati (leggere un file contenente un intero catalogo e convertirlo), ma da un problema decisamente più elementare e probabilmente contenuto nel primo: leggere un solo articolo da tastiera (dato che conosciamo una serie di procedure per gestire la tastiera). Ne approfitteremo per sostituire la libreria `advio.h` usata nello scorso capitolo con la libreria `stdio.h` che il linguaggio C fornisce, e ripassarne il contenuto. Ripasseremo le procedure di ingresso e uscita, quelle di gestione delle stringhe e la gestione dei parametri del `main`, cioè tutto ciò che riguarda l'acquisizione dei dati e la produzione dei risultati.

```
zaino      12.34 4  maggio 1679
pere 2.5 23 settembre 2390
tavolo 99.99 21 agosto 2019
sedie 45 1 aprile 2018
STOP
```

Figura 2.1: Esempio di istanza del problema (file `catalogo.txt`)

I codici di accompagnamento consistono in sette file con estensione `.c` e un file di testo `catalogo.txt`. I file `.c` forniscono le successive fasi di realizzazione della soluzione dell'esercizio. Qualche commento su questo punto: in laboratorio, questi file servono per consentire agli studenti che non riescono a star dietro all'esercizio stesso di tenere da parte il loro codice non funzionante e continuare a seguire la lezione a partire da un codice che funziona sicuramente. In più, in aula io procedo a braccio, a volte in base a suggerimenti degli studenti, e quindi a volte scrivo soluzioni leggermente diverse da quelle riportate nei file di accompagnamento. Questo può portare a confusione (e talolta a errori), ma anche a soluzioni diverse degli stessi problemi, a volte migliori o comunque interessanti. Il concetto che la stessa cosa si possa fare in tanti modi, con vantaggi da alcuni punti di vista e svantaggi da altri, è utile e prezioso. Le diverse fasi di svolgimento dell'esercizio sono marcate nel seguito da sottosezioni, che corrispondono a ciascuno dei file in questione. Cercherò di essere chiaro su un punto: dividere la descrizione dell'esercizio in fasi serve a guidare gli studenti che lo stanno realizzando e che stanno imparando e ripassando questioni algoritmiche e di programmazione. La relazione del progetto d'esame serve a spiegare a un lettore che conosce gli algoritmi e la programmazione come funziona l'algoritmo che ha risolto un dato problema. Non serve a spiegare le fasi di realizzazione e non serve a spiegare il codice.

La relazione del progetto non è una traccia di lezione

Il file di testo è una possibile istanza del problema (vedi Figura 2.1): contiene quattro articoli con il prezzo e la data di acquisizione, terminati dalla parola chiave `STOP`. Questa serve a indicare al programma che i dati sono conclusi. Se i dati sono in un file, in realtà, non è necessaria questa parola chiave (quando è finito il file, è finito il catalogo), ma siccome partiamo in modalità *bottom-up*, per un po' lavoreremo fornendo i dati da tastiera e l'ingresso da tastiera non ha una conclusione ovvia, per cui occorre introdurne una convenzionale. Ipotizziamo che l'utente scriva

Parsing (analisi sintattica) è il processo che analizza una data sequenza di simboli per determinarne la struttura grammaticale

Questa operazione è fondamentale perché serve ad acquisire i dati del programma da uno stream in ingresso

- dati digitati su tastiera
- stringhe di caratteri
- file su disco
- altre fonti esterne

Parser è un programma che esegue tale analisi non banale

Figura 2.2: *Parsing*

a mano un articolo e costruiamo un programma che lo legga e lo traduca nel formato richiesto.

Il nome di un articolo sarà una stringa, o vettore di caratteri; il prezzo sarà un numero reale; la data sarà formata da un giorno, che sarà un intero, un mese (ancora una stringa) e infine un anno (ancora un numero intero). Faremo un po' di controlli, convertiremo il mese in un numero e stamperemo il tutto nel formato rigido che è stato prescritto. Abbiamo tre sottoproblemi (e rispunta fuori ancora l'approccio *top-down*): l'acquisizione dei dati, la loro interpretazione con una serie di controlli e la stampa. L'acquisizione non è banale, perché il processore riceve da tastiera una sequenza di caratteri, ma non ha modo di sapere che cosa questi significhino cioè che i primi formano il nome di un articolo, che lo spazio successivo va ignorato, che potrebbe esserci un numero qualsiasi di spazi[1], che quanto segue è un numero reale (ma come si fa a sapere che `12.34` è un numero reale, e non piuttosto un numero intero, seguito da un punto e da un altro numero intero? O addirittura una sigla di qualche genere, una stringa di caratteri alfanumerici?), e così via. Queste sono tutte domande aperte. La risposta a queste domande è l'operazione di *parsing*, ovvero interpretazione (vedi Figura 2.2).

Questa operazione consiste nel ricevere uno *stream* in ingresso (vedi Figura 2.3), cioè un flusso monodimensionale di dati prodotti da una sorgente e interpretarli facendo due cose:

1. verificando che siano i dati giusti, attesi dall'algoritmo;
2. determinando come impiegarli, cioè scrivendoli nelle aree di memoria adatte al loro uso successivo.

Sappiamo che vogliamo un nome un prezzo un giorno un mese un anno e che nome e mese sono stringhe, il prezzo è un numero reale, giorno e anno sono numeri interi. Quindi ci siamo preparati dichiarando delle variabili che riservano gli spazi opportuni in memoria. Come si fa a convertire i caratteri che arrivano in ingresso nei dati corretti in memoria? La libreria `stdio.h` fornisce una fondamentale funzione che esegue il *parsing*, cioè questa operazione. La funzione si chiama `scanf` ed è la croce e la delizia dei programmatori in C, dato che consente operazioni molto

[1] Limitare troppo il formato dell'ingresso da tastiera è sbagliato in linea di principio, perché l'utente considera le limitazioni fastidiose e il programma che le impone inutilizzabile.

Si dice *stream* qualsiasi sorgente di dati in ingresso e qualsiasi destinazione per i risultati in uscita

- tastiera
- video
- file su disco, CD, DVD, memorie flash
- dispositivi di comunicazione (porte di rete, stampanti, ecc. . .)

La libreria `stdio.h` tratta tutti gli *stream* allo stesso modo (per quanto possibile)

- rappresentandoli con puntatori a file (`FILE *`)
- su cui operano funzioni simili o identiche

Figura 2.3: Gli *stream*

sofisticate, ma è piuttosto complicata, forse persino ostica, nel suo comportamento. È fondamentale capire come funziona (vedi Figura 2.4).

Questa funzione sfrutta una bella teoria matematica, che è nota come *teoria dei linguaggi formali*. La funzione `scanf` richiede per prima cosa una stringa di caratteri che descriva il formato dei dati che ci si aspetta di leggere da tastiera (vedi Figura 2.5).

Apriamo il file `catalogo0.c` (e verifichiamo che ha la solita struttura: direttive, un insieme per ora vuoto di prototipi, il programma principale e un insieme per ora vuoto di procedure secondarie. Nel `main` aggiungiamo l'istruzione per leggere da tastiera ciò che ci aspettiamo di ricevere dall'utente. La stringa di formato codifica in qualche modo queste attese. Per prima cosa, occorre leggere una parola, cioè una sequenza di caratteri "significativi", non semplici *separatori* (ovvero spazi bianchi, a capi e tabulazioni)[2]. Ci sono molte possibili stringhe che obbediscono a questa regola: un intero insieme di stringhe. Questo insieme è codificato come `%s`, dove il carattere `%` indica che si sta usando una così detta *specifica di conversione* (vedi Figura 2.6). Aprendo il file di esempio, si vede che la prima parola è `zaino`, terminata da uno spazio bianco che la separa dai dati successivi.[3]

A questo punto, ci si aspetta di leggere un numero reale. Che cos'è un numero reale in termini di caratteri? È una sequenza di cifre (da `0` a `9`), eventualmente preceduta da un segno (`+` o `-`), eventualmente seguita da un punto (`.`)[4] ed eventualmente seguita da un'altra sequenza di cifre. Ci sono molte possibili stringhe di caratteri che corrispondono a questa descrizione. L'insieme di tali stringhe è codificato come `%f`.

Nell'esempio, però, si può vedere che nel primo articolo il numero reale segue la parola dopo molti spazi bianchi, mentre nel secondo ce n'è uno solo. Come si rappresenta questo? Siccome non si vuole imporre un formato rigido al punto da fissare il numero di separatori (gli utenti commetterebbero errori, o comunque ne sarebbero infastiditi), la codifica `%f` rappresenta anche stringhe che cominciano con

[2] Ogni articolo è descritto da una singola parola, per semplicità.

[3] Devo anticipare qui la questione degli spazi bianchi, così la sbrighiamo subito e vale per tutte le specifiche.

[4] In inglese si usa il punto per separare la parte intera da quella frazionaria, anziché la virgola.

La principale funzione per il parsing di testo fornito da tastiera è

```
int scanf(char *formato, ...)
```

- ha un numero variabile di parametri ($p \geq 1$)
  - la stringa di formato definisce il *pattern* che si cerca di riconoscere nello stream in ingresso
  - $p - 1$ puntatori sono gli indirizzi delle celle dove conservare i valori riconosciuti nello stream in ingresso
- termina alla fine della stringa o al primo oggetto non riconosciuto
- restituisce il numero di oggetti del *pattern* ritrovati nello stream

Il resto dello stream resta disponibile per la chiamata seguente

Figura 2.4: La funzione `scanf`

un numero qualsiasi di separatori (eventualmente nullo) prima dei caratteri che descrivono il numero reale vero e proprio. Quindi contiene stringhe molto diverse, come `12.34`, `12`, `+12.3`. `.3`, e via dicendo.

Analogamente, `%d` rappresenta i numeri interi, eventualmente preceduti da un numero qualsiasi di separatori ed eventualmente con un segno. Queste codifiche, che iniziano tutte col carattere `%` si definiscono *specifiche* e rappresentano insiemi di stringhe.

Nel complesso, per leggere i dati di un articolo basta usare la stringa di formato `"%s %f %d %s %d"`. Gli spazi fra una specifica e l'altra sono sostanzialmente inutili, dato che la specifica prevede che ci possano essere separatori prima del dato vero e proprio. Servono solo a ricordare che qualsiasi sequenza di spazi (uno o più) nella stringa di formato rappresenta una sequenza di separatori (zero o più) che ci si attende di ricevere. Si noti la differenza: i dati ricevuti possono non contenere separatori, anche se la stringa di formato li prevede. In sostanza `"%s %f %d %s %d"` equivale perfettamente a `"%s%f%d%s%d"` e usare una o l'altra dipende solo dal gusto personale.

La teoria dei linguaggi formali descrive le proprietà di insiemi di stringhe di simboli tratti da un opportuno alfabeto. In particolare, descrive la possibilità di rappresentare in modo compatto insiemi potenzialmente anche infiniti (l'insieme delle parole di qualsiasi lunghezza, quelli delle rappresentazioni decimali dei numeri reali e dei numeri interi), e quindi di riconoscere se una data stringa appartiene o no a un dato insieme. Per esempio, la stringa `zaino` è compatibile con la specifica `%s`, ma non con le specifiche `%f` e `%d`. Nel nostro esempio la stringa `"%s %f %d %s %d"` è una rappresentazione compatta, "grammaticale", di tutte le stringhe che costituiscono un dato accettabile. La funzione `scanf` realizza un algoritmo di riconoscimento, che confronta la stringa di formato con lo *stream* di ingresso e valuta se il secondo è compatibile con la prima, cioè se la parte iniziale di quanto viene fornito da tastiera (lo *stream* è infinito, potenzialmente: si può sempre aggiungere altri dati, finché il processore è acceso) appartiene all'insieme rappresentato dalla stringa di formato. È un problema sofisticato e richiede una serie di risultati teorici riguardo come sia possibile farlo ed entro quali limiti: alcuni insiemi di stringhe saranno riconoscibili, altri no, secondo le operazioni consentite all'algoritmo che esegue il riconoscimento.

Stringa di formato e stream in ingresso sono compatibili quando

- ogni carattere non separatore nella stringa, cioè diverso da spazio, a capo e tabulazione, corrisponde allo stesso carattere nello stream
- uno o più separatori consecutivi nella stringa corrispondono a zero o più separatori consecutivi nello stream
- le specifiche di conversione nella stringa (`%...`) corrispondono a zero o più separatori consecutivi e una sequenza di caratteri che descrive un oggetto del tipo indicato dalla specifica
  (numero intero, numero reale, singolo carattere, parola, ...)

Esempio:

La stringa di formato "`Numero  %d/%d`" corrisponde allo stream "`Numero`" seguito da zero o più spazi e da due interi separati da `'/'`

Quindi è compatibile con

"`Numero3/5`", "`Numero 15/ 2`", "`Numero -6/ 0`", ecc...

Non è compatibile con "`Numero 10 /5`"

Figura 2.5: La stringa di formato

Indica il tipo di oggetto atteso nella posizione corrente dell'ingresso

`%d`	`int` decimale
`%u`	`unsigned int` decimale
`%f`	`float` in notazione decimale
`%e`	`float` in notazione scientifica
`%c`	`char`
`%s`	stringa priva di separatori
`%[`*set*`]`	stringa fatta con i caratteri elencati in *set*

Ogni puntatore indica dove copiare l'oggetto corrispondente

L'istruzione `scanf("%d/%d/%d",&giorno,&mese,&anno);`

- cerca tre numeri interi separati da `'/'`
- li assegna alle variabili `giorno`, `mese` e `anno`
- restituisce il numero di interi trovati

Figura 2.6: Specifica di conversione

Se la funzione `scanf` facesse solo questo, sarebbe solo un analizzatore lessicale (o *lexical analyzer*, in breve *lexer*) e la sua utilità sarebbe solo teorica. In realtà fa anche un'altra cosa. Una volta riconosciuto che da tastiera sono stati forniti dati compatibili con quelli attesi, cioè una volta isolate le unità di significato e riconosciuta la loro correttezza rispetto al bisogno, va a convertire ciascuno di questi dati dalla sottostringa di caratteri che è attualmente in una struttura dati in memoria, cioè nella corrispondente rappresentazione interna binaria[5] e va a scrivere questa rappresentazione in un'opportuna cella della memoria. Questo spiega gli argomenti successivi della funzione `scanf`, che sono tutti indirizzi di celle di memoria: le celle in cui i singoli risultati vanno conservati. E spiega anche il risultato restituito da `scanf`. Siccome la stringa di formato prevede in generale più dati (nell'esempio, 5), è possibile che la funzione ne riconosca alcuni, ma non altri. Il risultato della funzione `scanf` è un numero intero, pari al numero di dati riconosciuti ed elaborati (cioè convertiti e salvati in celle di memoria).

Per esempio, se da tastiera l'utente scrivesse `zaino 12.34 pippo` anziché `zaino 12.34 4`, la funzione `scanf` fallirebbe nel riconoscimento del terzo dato e si arresterebbe, restituendo il valore 2 (numero dei dati riconosciuti) e assegnando il valore corretto solo ai primi due argomenti. Con i dati dell'esempio, invece, l'articolo viene letto interamente e la funzione restituisce il valore 5. Si noti che la funzione termina al primo problema, cioè non procede col riconoscimento dei dati successivi al primo errore, anche se potenzialmente validi.

Ovviamente, i dati vanno scritti in variabili precedentemente dichiarate. Quindi la semplice chiamata a `scanf` non è sufficiente: occorre aggiungere nella parte dichiarativa del programma principlale le dichiarazioni opportune. Il nome dell'articolo è un vettore di caratteri, che secondo il testo può arrivare a 10 elementi. Lo dichiareremo come

```
char articolo[10+1];
```

aumentando di 1 il numero di elementi per il motivo seguente. Un vettore di 11 caratteri non necessariamente li usa tutti per rappresentare una parola. Infatti, la parola `zaino` è di 5 lettere, che occuperanno i primi 5 caratteri del vettore stesso. Come si fa a sapere che i caratteri seguenti non sono significativi? Molto semplice: in C ogni stringa di caratteri termina con un carattere particolare, indicato come `'\0'` e chiamato *terminatore*. Si veda la Figura 2.7: un vettore di 7 caratteri contiene i caratteri `'p'`, `'r'`, `'o'`, `'\0'`, `'v'`, `'a'` e `'\0'`, convenzionalmente si intende che contenga la stringa `"pro"`. Più precisamente, trattandolo come vettore di caratteri (come vedremo nel Capitolo **??**) si potrà accedere a tutti i suoi elementi, ma trattandolo come stringa, cioè usando le funzioni specificamente dedicate alle stringhe (che ripasseremo nella Sezione **??**), queste avranno accesso solo ai caratteri che precedono la prima occorrenza del terminatore. In questo esercizio, useremo i nomi degli articoli e dei mesi come stringhe, e quindi prevederemo che essi contengano alla fine un terminatore. Ma allora il vettore deve avere spazio sufficiente a ospitare non solo i 10 caratteri potenzialmente utilizzabili per un articolo, ma anche il carattere aggiuntivo necessario per il terminatore. L'uso dell'espressione `10+1` anziché `11` ha solo motivi didattici di chiarezza.

Analogamente, siccome il mese col nome più lungo è `settembre`, che contiene 9 lettere, la variabile corrispondente sarà dichiarata:

```
char mese[9+1];
```

[5] I numeri interi o reali sono rappresentati da stringhe binarie completamente diverse da quelle che rappresentano i caratteri della loro forma decimale.

In C stringa è

- qualsiasi sequenza di caratteri
- terminata dal carattere `'\0'` (terminatore)
  che non è significativo

Per rappresentarla esplicitamente, basta racchiuderla fra virgolette
Esempio: `"Questa e' una stringa."`
(*il terminatore segue l'ultimo carattere ed è implicito*)

Le variabili che conservano stringhe si dichiarano come vettori di caratteri
`char s[ROW_LENGTH];`
con lo spazio sufficiente a contenere anche il terminatore

Il primo terminatore contenuto nel vettore tronca la stringa

'p'	'r'	'o'	'\0'	'v'	'a'	'\0'
0	1	2	3	4	5	6

vale ''pro''

Non c'è controllo che un vettore di caratteri contenga un terminatore
(*questo può causare errori*)

Figura 2.7: Stringhe di caratteri

Avremo poi bisogno di un numero reale per il prezzo e due numeri interi per il giorno e l'anno.

```
float prezzo;
int giorno;
int anno;
```

Se vogliamo controllare che i dati siano stati tutti riconosciuti correttamente, possiamo aggiungere una variabile intera `n` a cui assegnare il valore restituito dalla funzione `scanf`:

```
int n;
```

Notiamo che la funzione `scanf` ha un solo dato, che è la stringa di formato (lo *stream* in ingresso è un dato, ma viene dalla tastiera, non è un'area di memoria, quindi non compare fra gli argomenti). Le 5 aree di memoria in cui salvare i dati sono risultati per la funzione `scanf`, esattamente come il loro numero. Come sempre in C, quando i risultati sono multipli, solo uno può figurare esplicitamente, mentre gli altri devono comparire nella lista degli argomenti, come se fossero dati. Aggiungiamo un aspetto fondamentale: quando si scrive nella chiamata a una funzione un dato che in realtà è un risultato, occorre marcare questo facendo precedere il nome del risultato da un carattere `&`, per motivi che vedremo nella Sezione **??** che tratta il passaggio di argomenti alle funzioni.

```
int n;

n = scanf("%s %f %d %s %d",articolo,&prezzo,&giorno,mese,&anno);
```

Perché non tutti i risultati hanno il carattere `&`? Perché `articolo` e `mese` non sono dati elementari, ma vettori, e il meccanismo che descriveremo in seguito non

richiede questo carattere ai vettori (anzi, lo vieta, nel senso che aggiungere una `&` cambierebbe il significato dell'operazione).

A questo punto, potremmo verificare la correttezza dei dati letti. Per prima cosa, possiamo chiederci se i dati sono stati riconosciuti tutti, ed eventualmente segnalare all'utente eventuali errori. Per stampare a video, basta usare la funzione `printf` mettendo il messaggio da stampare come argomento (il carattere speciale `'\n'` rappresenta l'andare a capo).

```
if (n != 5)
{
printf("Errore nel formato dei dati!\n");
exit(EXIT_FAILURE);
}
```

La funzione `exit` indica al processore di terminare il programma e restituire al sistema operativo come risultato dell'intero programma il valore `EXIT_FAILURE` al posto del risultato `EXIT_SUCCESS` che indica una terminazione corretta. A questo punto, sta al sistema operativo (o meglio agli *script* che eventualmente gestiscono l'esecuzione del programma, se questo non è stato banalmente lanciato a mano come stiamo facendo qui) decidere che cosa fare.

Se vogliamo sottolineare il fatto che questa stampa non è un risultato, ma un messaggio di errore, conviene usare un'altra funzione (`fprintf`) che non stampa rigorosamente a video, ma consente di stampare su un file (da cui la `f` iniziale). Questo file in realtà sarà il così detto *standard error* (`stderr`) che è lo *stream* riservato agli errori.

```
fprintf(stderr,"Errore nel formato dei dati!\n");
```

In pratica, la scrittura avviene comunque a video. Che cosa è cambiato? Che la redirezione dell'output agisce sulle stampe a video, ma non sulle stampe di errore. In questo modo, evitiamo che i messaggi di errore finiscano nei risultati: continueranno a finire sul video. Volendo, si possono redirigere su un altro file, diverso da quello usato per l'output (o si può lasciare l'output sul video e redirigere i messaggi di errore su un file). Si veda la Figura 2.8.

Possiamo fare molti test di correttezza sui dati letti. Ci limitiamo a un esempio incompleto: sappiamo che il giorno è un numero intero, ma possiamo controllare che sia compreso fra 1 e 31.

```
if ( (giorno < 1) || (giorno > 31) )
{
printf("Errore nel formato dei dati!\n");
exit(EXIT_FAILURE);
}
```

Comunque, dobbiamo convertire il mese nella sua forma numerica. Per farlo occorre confrontare la stringa `mese` con ciascuna delle dodici stringhe che contengono i nomi corretti dei mesi. Il confronto di stringhe si può fare con la funzione `strcmp` (vedi Figura 2.9).

Applicata a due stringhe, per esempio la stringa variabile `mese` e la stringa costante `"gennaio"`, questa funzione restituisce 0 se le due stringhe sono identiche,

Esistono tre *stream* standard, che non occorre definire, aprire e chiudere

- lo standard input (`stdin`), ovvero la tastiera
- lo standard output (`stdout`), ovvero il video
- lo standard error (`stderr`), ovvero il video

Quando si chiama un programma, il sistema operativo può reindirizzare gli *stream* standard, cioè modificarne il significato

- *programma* `<` *nomefile* indica che si ricevono i dati dal file *nomefile* anziché da tastiera (`stdin` punta il file *nomefile*)
- *programma* `>` *nomefile* indica che si stampano i risultati sul file *nomefile* anziché a video (`stdout` punta il file *nomefile*)
- *programma* `2>` *nomefile* indica che si stampano i messaggi di errore sul file *nomefile* anziché a video (`stderr` punta il file *nomefile*)

Figura 2.8: Gli *stream* standard

Non si possono confrontare stringhe con l'operatore `==`

`strcmp(`*s1*`,`*s2*`)`

confronta le stringhe *s1* e *s2* in modo lessicografico (dizionario)

1. scorre in parallelo le due stringhe per $i \geq 0$ caratteri fino a trovare
   - due caratteri diversi (`s1[i] != s2[i]`)
   - oppure il termine di una delle due stringhe
2. restituisce
   - valore nullo se entrambe le stringhe sono terminate
   - valore negativo se *s1* termina o `s1[i] < s2[i]`
   - valore positivo se *s2* termina o `s1[i] > s2[i]`

Figura 2.9: Confronto fra stringhe

un numero negativo se la prima stringa precede alfabeticamente la seconda[6] e un numero positivo se la prima stringa segue alfabeticamente la seconda. Quindi quel che ci serve è il confronto con 0 e una variabile intera `m` a cui assegnare l'indice del mese:

```
if (strcmp(mese,"gennaio") == 0)
m = 1;
```

Si può fare lo stesso per tutti i mesi, concatenando le condizioni, mutuamente esclusive, con i costrutti `else if`. Se tutti i confronti falliscono, la stringa `mese` non è un mese, e quindi c'è ancora un errore nel formato dei dati. È un pezzo di codice abbastanza lungo e sofisticato da poter meritare di diventare una funzione (`m = ConverteMese(mese);`), ma lo lasciamo come esercizio.

Altri controlli potrebbero riguardare il fatto che il prezzo sia positivo, che l'anno cada in un certo intervallo di valori interi, e così via.

A questo punto bisogna stampare a video il risultato in un formato piuttosto preciso. La stampa sfrutta la funzione `printf` già discussa, con una serie di strumenti aggiuntivi che consentono di introdurre precisazioni sul modo di scrivere i dati stessi. Per esempio, scriviamo una parola, ma imponiamo la lunghezza della parola stessa; scriviamo un numero reale o intero, ma precisiamo quanto spazio occuperemo nel farlo. Per ottenere questo risultato, basta aggiungere un numero (due, nel caso dei numeri reali) alla specifica. Così

```
printf("%-10s %9.2f %02d\%02d\%04d\n",articolo,prezzo,giorno,m,anno);
```

indicherà

1. che il nome dell'articolo deve occupare 10 caratteri: se ne occupa di meno, verranno aggiunti spazi bianchi al termine; se ne occupa di più (cosa esclusa dal testo dell'esercizio, ma che in altri casi potrebbe accadere) si useranno quelli strettamente necessari; il `-` indica che gli spazi bianchi eventualmente necessari vanno al termine della stampa (per avere un incolonnamento naturale dei nomi);

2. che il prezzo deve occupare almeno 9 caratteri (se ne occorrono di più, se ne usa il numero minimo possibile, come sopra), compreso il separatore decimale e l'eventuale segno, dei quali 2 sono cifre decimali (dato che i prezzi arrivano fino al centesimo); qui gli spazi bianchi eventualmente necessari vanno al principio della stampa (per avere un incolonnamento naturale dei numeri reali in corrispondenza al separatore decimale);

3. che la data consiste in tre numeri interi separati da barre, dei quali i primi due sono espressi con due cifre e il terzo con quattro cifre, accumulando le cifre al termine della stampa e usando il carattere `0` come riempitivo al posto dello spazio bianco se per caso un numero richiede meno cifre di quanto indicato.

Tutto questo funziona correttamente fin tanto che i prezzi non sono negativi e non eccedono le 6 cifre intere e le date sono formate da numeri positivi con anni non superiori al 9 999. A questo punto potremmo tornare indietro e introdurre questi controlli specifici nella lettura.

[6] Il numero non è precisato dallo standard: ogni compilatore potrebbe restituire un numero diverso. Il caso più comune è che si restituisca la distanza lungo l'alfabeto ASCII delle prime lettere diverse che le due stringhe contengono. Non si può fare affidamento su questo se si vuole scrivere codice che funzioni sicuramente su ogni macchina.

Compiliamo il codice per verificare che il tutto funzioni.

```
gcc -Wall -pedantic catalogo0.c -o catalogo.exe
```

ed eseguiamo diversi esempi, distribuendo spazi in vario modo e scrivendo nomi di diversa lunghezza, prezzi diversi e date differenti. I risultati sono incolonnati correttamente e nel formato desiderato.

Supponiamo ora di commettere errori di battitura

```
zaino 4.56 xx aprile 2023
zaino 4.56 1 apriglio 2023
```

Questo errore vengono rilevati in punti diversi del codice: il primo quando la funzione `scanf` riconosce due termini anziché cinque nell'input, dato che il terzo termine è una stringa anziché un numero intero; il secondo quando si cerca di convertire il mese nel suo indice, e non si riesce a trovare una corrispondenza. Possiamo quindi dire che il problema è risolto nella sua forma più semplice.

2.1.1 Costruzione di una procedura a partire da un blocco

Ora possiamo procedere *bottom-up* complicando il problema. Invece di eseguire la conversione su un articolo solo, eseguiamola su molti. Ovviamente, quanto già scritto risolve un sottoproblema di questo problema, e quindi viene naturale considerarlo come una procedura. Fra l'altro, il programma principale sta diventando lungo (una schermata è il limite ragionevole, dato che superarlo impedisce al programmatore di avere un colpo d'occhio unitario sull'algoritmo), ed è quindi tempo di raccogliere operazioni logicamente collegate in una procedura.

Se avessimo proceduto *top-down*, avremmo definito a priori una funzione vuota per gestire un articolo e l'avremmo riempita in seguito. Procedendo *bottom-up*, prima scriviamo e poi ci accorgiamo di possibili accorpamenti. Come si procede?

Basta identificare il blocco di codice che si vuole accorpare, aggiungere una chiamata a funzione, con un nome, argomenti e un risultato opportuno, riportare il prototipo di tale funzione nella sezione dei prototipi, aggiungere una definizione vuota nella sezione delle definizioni di procedure secondarie e copiare il blocco di codice nel corpo vuoto della funzione stessa. Probabilmente, sarà anche necessario spostare variabili dalla parte dichiarativa del programma principale alla parte dichiarativa della nuova procedura[7]

Nel nostro caso, la chiamata sarà

```
GestisceArticolo();
```

senza dati, perché si ricevono da tastiera, e senza risultati, perché il risultato viene stampato, non usato per calcolare qualcosa. Ne deriva il prototipo:

```
void GestisceArticolo ();
```

e la definizione:

```
void GestisceArticolo ()
{
}
```

[7] Può capitare che una variabile sia usata sia nel blocco che si vuole spostare sia altrove. Di solito, questo è il segno che tale variabile è un dato o un risultato della funzione stessa.

dove il corpo contiene le variabili precedentemente definite nel programma principale (e ora non più necessarie lì) e le operazioni di lettura, controllo, conversione e stampa.

Perché bisogna spostare le variabili? Perché una variabile è utilizzabile (in gergo, spesso si dice "visibile") solo all'interno del blocco nel quale è dichiarata, dove con *blocco* si intende la parte di codice racchiusa fra una parentesi graffa aperta ({) e la corrispondente parentesi graffa chiusa (}).

Ricompilando ed eseguendo alcuni esempi si può avere qualche conferma del fatto di non aver introdotto errori.

2.1.2 Iterazione della conversione

Diventa ora semplice risolvere il problema complessivo, cioè la lettura ed elaborazione iterata di molti articoli, interrotta dall'utente battendo a tastiera la parola chiave `STOP`. Invece di modificare il programma principale, per poi trovarsi a costruire una funzione a posteriori, introduciamo direttamente una nuova funzione `GestisceCatalogo` (chiamata, prototipo e definizione vuota), e riempiamo la sua definizione usando la funzione che gestisce un singolo articolo.

```
void GestisceCatalogo ();
```

Per ora, anche `GestisceCatalogo` non ha dati né risultati. Il corpo contiene un ciclo, che si occupa di iterare il procedimento, contenente `GestisceArticolo`.

Che genere di ciclo usiamo? I cicli sono in realtà tutti vicendevolmente trasformabili gli uni negli altri, ma qui sembra naturale usare un ciclo a condizione finale, dato che si vuole eseguire ripetutamente la gestione del singolo articolo finché il risultato indica che l'utente vuole proseguire. Quindi bisogna che `GestisceArticolo` restituisca un risultato di tipo logico (vero o falso). Scriveremo:

```
do
{
risultato = GestisceArticolo();
} while (risultato == TRUE);
```

il che, però scatena una serie di operazioni aggiuntive da compiere. Per prima cosa, bisogna definire la variabile logica `risultato`

```
boolean risultato;
```

ma in C89 non esistono i valori logici: le espressioni `boolean` e `TRUE` sono sconosciute al compilatore. Quindi, dobbiamo simulare i valori logici con i valori interi 0 (per il falso) e 1 (per il vero), come si fa correntemente dai tempi di Boole. Come si fa a dare il significato desiderato alle nuove espressioni?

Bisogna consentire al compilatore di leggere `boolean`, `TRUE` e `FALSE` e intendere `int`, `1` e `0`. Un modo per farlo è usare la direttiva `#define`

```
#define boolean int
#define FALSE 0
#define TRUE 1
```

mentre un altro modo è usare l'istruzione `typedef` e i così detti *tipi enumerativi*, che consentono di dare un nome a opportuni sottoinsiemi finiti di numeri interi

```
typedef boolean;
DA COMPLETARE
```

Va notato che la prima soluzione usa le direttive, e quindi impegna il precompilatore in una sostituzione testuale di codice con altro codice, mentre la seconda usa istruzioni (si noti il `;`) e quindi impegna il compilatore in operazioni che riguardano la tabella dei simboli[8]. Qualunque sia la soluzione scelta, è comunque ragionevole andare a modificare la sezione iniziale del codice, perché i nuovi simboli siano disponibili in tutto il seguito.

Poi bisogna che la dichiarazione e la definizione di `GestisceArticolo` siano modificati in modo che restituisca un risultato. Quindi, la dichiarazione diventa:

```
boolean GestisceArticolo();
```

e corrispondentemente cambia l'intestazione della definizione. Deve poi cambiare il corpo, aggiungendo un'istruzione terminale che restituisca il valore "vero" quando la gestione è andata a buon fine:

```
return TRUE;
```

ma anche istruzioni che consentano di restituire il valore "falso" quando l'utente ha inserito la parola chiave che lo richiede. Questo è il caso in cui i dati non contengono le cinque informazioni che caratterizzano un articolo, ma solo la parola chiave `STOP`. Ora scriviamo una soluzione scorretta per fissare le idee, poi discuteremo perché è scorretta: quando il numero di oggetti riconosciuti da `scanf` è 1, anziché 5, e quest'unico oggetto, che è stato inserito nella variabile `articolo` coincide con la stringa `STOP`, allora la funzione deve terminare restituendo `FALSE`.

```
if ( (n == 1) && (strcmp(articolo,"STOP") == 0) ) return FALSE;
```

Purtroppo, come dicevamo, questa soluzione è errata per un motivo tecnico. In realtà, se l'utente batte a tastiera `STOP` e preme l'invio, la funzione `scanf` non restituisce 1, ma rimane in attesa dei quattro oggetti successivi. Lo *stream* in ingresso è potenzialmente illimitato e il fatto che l'utente abbia fornito un solo dato non significa che non possa fornirne altri in futuro. L'invio, infatti, è un semplice separatore, non il segnale che l'ingresso è terminato. La funzione `scanf` si ferma anticipatamente solo se c'è un dato in ingresso, ma non corrisponde alla stringa di formato. D'altra parte, quella è una situazione di errore, non una terminazione corretta.

Una possibile soluzione valida è di leggere da tastiera l'intera riga e poi elaborarla. Il vantaggio è che in questo modo si lavora non sullo *stream* in ingresso, ma su una stringa, che è intrinsecamente limitata dal proprio terminatore. Per leggere un'intera riga basta usare la funzione `gets`, che raccoglie quanto l'utente batte a tastiera fino al primo invio (dunque, l'invio in questo caso termina effettivamente l'ingresso), e dichiarare una stringa di lunghezza sufficiente a contenerlo.

```
char riga[ROW_LENGTH];
...
gets(riga);
```

dove `ROW_LENGTH` è una costante simbolica dichiarata con una direttiva `#define`. Ovviamente, se l'utente scrive da tastiera un numero di caratteri più alto, si va a sforare la memoria e nascono problemi, nei quali per ora non ci addentriamo.

[8] La distinzione è facilmente visibile precompilando i due codici che ne derivano con l'opzione `-E`.

Anche le stringhe di caratteri si possono vedere come *stream*
e quindi si possono gestire con funzioni simili

```
int sscanf(char *stringa, char *formato, ...)
```

- riconosce nella *stringa* (primo parametro)
- gli elementi forniti dalla stringa di *formato* (secondo parametro)
- assegna gli oggetti riconosciuti ai puntatori (parametri successivi)

Esempio:

```
char s[10+1];
int giorno, mese, anno;

strcpy(s,"14/05/2010");
sscanf(s,"%d/%d/%d",&giorno,&mese,&anno);
```

`sprintf` genera stringhe formattate come `printf` stampa a video

Figura 2.10: Parsing di stringhe

Ora dobbiamo leggere i cinque oggetti non da tastiera, ma dalla stringa `riga`. Per fortuna, la funzione `scanf` ha due funzioni sorelle che fanno le stesse cose operando non sullo *stream* in ingresso, ma su un file (`fscanf`) o su una stringa (`sscanf`). L'unica differenza è che queste due funzioni richiedono di specificare il file o la stringa su cui devono operare, per cui hanno un argomento aggiuntivo, che precede gli altri (vedi Figura 2.10):

```
int n;

n = sscanf(riga,"%s %f %d %s %d",articolo,&prezzo,&giorno,mese,&anno);
```

Ora il codice funziona, perché `sscanf` non si ferma solo in caso di riconoscimenti errati, ma anche nel caso in cui la stringa in ingresso su cui opera è terminata. Lo stesso succede quando `fscanf` legge un file: anche i file hanno un carattere terminatore (detto `EOF` e diverso da quello delle stringhe).

A questo punto, il codice funziona correttamente. Avrete certamente notato che realizzare un algoritmo in modalità *bottom-up* richiede, oltre che di guardare lontano per sapere dove si sta andando, anche di rincorrere i problemi che si incontrano, modificando e adattando i pezzi di codice scritti in precedenza. Richiede un certo grado di concentrazione e memoria ed è consigliabile solo per problemi semplici e interventi limitati.

Notiamo che abbiamo avuto bisogno solo di due funzioni in lettura (`scanf` e `gets`), a cui corrispondono due funzioni in scrittura (`printf` e `puts`), con le loro meccaniche varianti dedicate a file (`fscanf` e `fgets`, `fprintf` e `fputs`) e stringhe (`sscanf` e `sprintf`: mancano le altre, perché si ritiene che nelle stringhe non abbia particolare senso leggere o scrivere "righe"). Con queste si può fare praticamente tutto quel che occorre per gestire ingresso e uscita. Esistono molte altre funzioni, ma servono a scopi decisamente più sofisticati e imbarcarsi a usarle senza motivo di

solito porta solo a un lungo, sanguinoso e insensato arrabattarsi.

2.1.3 Elaborazione da file

In realtà, nel problema originale il catalogo si supponeva contenuto in un file di testo, e non inserito da tastiera. Si potrebbe ovviare con la redirezione dell'input. Infatti

```
catalogo < istruzioni.txt
```

funziona perfettamente. Ma supponiamo di volere un programma che lavori solo da file, in modo da ripassare anche la gestione dei file.

2.1.4 Caricamento dei dati da un file

Nella versione successiva, il programma dovrà quindi aprire un file, leggerlo, caricarne il contenuto in memoria ed elaborarlo come già visto. La funzione `GestisceCatalogo`, quindi, non è più priva di dati, ma ha come dato il nome di un file. Sempre procedendo *bottom-up*, semplifichiamo il problema, cioè supponiamo di conoscere già il nome del file (per esempio, `catalogo.txt`) e passiamo la relativa stringa come argomento nella chiamata della funzione.

```
GestisceCatalogo("catalogo.txt");
```

Prototipo e intestazione della definizione devono cambiare di conseguenza, includendo un tipo e un nome per il nuovo dato.

```
void GestisceCatalogo (char *NomeCatalogo);
```

con il punto e virgola nel prototipo e senza nell'intestazione.

A questo punto, dobbiamo modificare il corpo della funzione in modo che apra il file dato il suo nome e legga i dati da esso. Il nome del file e il file su disco sono due cose diverse: il nome serve a sapere dove si trova il file, ma il file è un'altra variabile *ad hoc*, con un proprio tipo, e va dichiarata nella parte dichiarativa della funzione `GestisceCatalogo`.

```
FILE *fCatalogo;
```

dove si vede che la nuova variabile `*fCatalogo`[9] è un puntatore all'area di memoria che contiene tutte le informazioni di cui il processore ha bisogno per gestire il file stesso, e che vengono passate a tutte le funzioni di gestione del file, in modo che il programmatore non abbia bisogno di preoccuparsi esplicitamente di tutti i dettagli tecnici.

Le operazioni più comuni per la gestione dei file sono sostanzialmente quattro: aprire, chiudere, leggere e scrivere. In questo esercizio, useremo solo le prime tre. Per aprire un file (vedi Figure 2.11 e 2.12) basta usare la funzione `fopen`, che ha due dati e un risultato. Il primo dato è il nome del file, il secondo dato è una stringa che indica in che modo il file va aperto: se per leggerlo o per scriverci, e in questo secondo caso se ci si scrive partendo da zero o accodando in fondo a ciò che c'è già scritto qualche altra informazione. In questo esercizio, leggiamo soltanto, e quindi scriviamo la stringa `"r"`.

Il risultato è un puntatore a `FILE` che possiamo assegnare alla variabile sopra introdotta. Se l'apertura del file fallisce, la funzione restituisce un puntatore specifico

[9] Spesso per i file si usa il generico nome `fp`, che sta per *file pointer*; qui si è voluto puntare alla massima chiarezza.

I file di testo sono costituiti da sequenze di caratteri organizzati in righe, separate dal carattere speciale `'\n'`

Per usare un file occorre aprirlo con il comando

`FILE *fopen(char *`*nomefile*`, char *`*modo*`)`

che specifica

- il nome del file da aprire e la posizione su disco (*path*)
- il modo in cui usarlo
  - `"r"`: in lettura, ponendosi al principio del file
  - `"w"`: in scrittura, ponendosi al principio del file
  - `"a"`: in accodamento, ponendosi alla fine del file

Il *path* può essere assoluto o relativo (al file eseguibile o al progetto)

Figura 2.11: Apertura di un file di testo (1)

detto `NULL`. È buona norma sempre verificare che questo sia successo, ed eventualmente stampare un messaggio di errore per comunicare il fallimento all'utente, preferibilmente con il nome del file che non si è riusciti ad aprire.

```
fp = fopen(NomeCatalogo,"r");
if (fp == NULL)
{
fprintf(stderr,"Errore nell'apertura del file %s!\n",NomeCatalogo);
exit(EXIT_FAILURE);
}
```

I motivi per il fallimento possono essere molti, dall'aver usato un nome sbagliato o una cartella sbagliata all'aver cercato di aprire un file che già altri programmi stanno tenendo aperto in un modo che blocca l'accesso ad altri, e così via. Aperto il file, dovremo passarlo alla funzione `GestisceArticolo`, in modo che possa lavorarci. Questo comporta l'aggiunta di un dato nella chiamata

```
risultato = GestisceArticolo(fp);
```

nella dichiarazione e nella definizione

```
boolean GestisceArticolo (FILE *fp);
```

dove si può osservare (in attesa che chiariamo il meccanismo di passaggio degli argomenti alle funzioni nel Capitolo **??**), che il nome del file è diventato il più generico `fp`. Queto è consentito dal fatto che in realtà l'argomento è una cosa diversa dalla variabile usata nella chiamata (sono fisicamente celle diverse, in punti diversi della memoria), e può avere senso (non è obbligatorio: è una questione di gusti e dipende dalla situazione) assegnare loro nomi diversi. Prototipo e intestazione della definizione, ovviamente, devono invece essere rigorosamente uguali (salvo il punto e virgola finale).

Al termine, è buona norma chiudere il file per una serie di motivi tecnici: lo spazio per l'apertura dei file è limitato; tenere un file aperto significa bloccare

La funzione `fopen` restituisce un puntatore al file per poterlo usare

- se il file non esiste
  - in lettura, restituisce `NULL`
  - in scrittura e accodamento, ne crea uno vuoto
- se il file non può essere aperto o creato
  - restituisce `NULL`

Aperto un file, la posizione accessibile è

- il principio del file se si è aperto il file in lettura o scrittura
- la fine del file se si è aperto il file in accodamento

Dopo l'uso, il file va chiuso con l'istruzione

```
int fclose(FILE *stream)
```

Figura 2.12: Apertura di un file di testo (2)

l'accesso ad altri programmi che possono averne bisogno; un file aperto è più esposto a danneggiamenti se il processore avesse dei problemi.

Ora bisogna leggere i dati dal file, anziché da tastiera. Anziché `gets`, si userà `fgets` (vedi Figura 2.13), che ha alcune differenze: anziché solo la stringa nella quale andrà copiata la riga letta, bisogna passarle il puntatore al file di testo da cui leggere la riga e un numero intero, che specifica quanti caratteri al massimo si potranno leggere. Si era detto che la funzione `gets` è pericolosa in quanto consente di caricare più caratteri di quelli che la stringa in memoria è in grado di contenere. La nuova funzione impedisce questo specificando che, se la riga supera il numero massimo di caratteri ammesso, la lettura si fermerà a tale numero. Siccome la stringa è stata definita di `ROW_LENGTH` caratteri, fisseremo questo valore come secondo argomento della chiamata.

```
fgets(riga,ROW_LENGTH,fp);
```

e questo conclude la modifica del codice affinché legga da file, anziché da tastiera.

La Figura 2.14 discute la funzione `fscanf`, che corrisponde a `scanf`, ma opera su file di testo.

2.1.5 Interpretazione della linea di comando

Manca un ultimo passaggio: il nome del file che contiene il catalogo non può essere fissato una volta per tutte, ma vogliamo che sia deciso dall'utente e trasmesso durante il lancio del programma passandolo dalla linea di comando. Invece di scrivere `catalogo` e automaticamente lavorare su `catalogo.txt`, vogliamo scrivere

```
catalogo catalogo.txt
```

che il sistema operativo interpreta come:

Si può leggere un'intera riga da tastiera con l'istruzione

```
char *gets(char *s)
```

- opera sullo *stream* `stdin`
- legge una riga compreso il carattere terminale `'\n'`
- restituisce la stringa `s` escluso `'\n'`; se fallisce, restituisce `NULL`

Si può leggere un'intera riga da file con l'istruzione

```
char *fgets(char *s, int n, FILE *stream)
```

- opera sullo stream specificato
- legge una riga compreso il carattere terminale `'\n'`,
  ma legge al massimo `n` caratteri e non include `'\n'`
- restituisce la stringa `s` compreso il icarattere terminale `'\n'`
- restituisce la stringa `s`; se fallisce, restituisce `NULL`

È consigliabile sostituire `gets(s)` *con* `fgets(s,n,stdin)`

Figura 2.13: Lettura di righe

Tutti gli stream di ingresso sono gestiti allo stesso modo

```
int fscanf(FILE *stream, char *formato, ...)
```

funziona come `scanf`, salvo richiedere il puntatore allo stream usato:

- interpreta il contenuto dello *stream*
- in base alla stringa di formato
- assegna gli oggetti riconosciuti ai puntatori che seguono
- restituisce il numero di oggetti assegnati

Esempio:

```
FILE *fp;
int giorno, mese, anno;
fp = fopen("prova.txt","r");
fscanf(fp,"%d/%d/%d",&giorno,&mese,&anno);
```

Figura 2.14: Parsing di file di testo

Anche il `main` è una funzione, chiamata dal sistema operativo

- ha dei dati, descritti dai parametri formali nell'intestazione
- ha un risultato, descritto dal tipo restituito nell'intestazione

Tuttavia

- il numero e tipo dei parametri non è noto a priori
- il risultato è sempre intero

Questo apre una serie di domande

- come si definisce il numero e tipo dei parametri?
- come si passano i parametri attuali?
- a che serve e come si recupera il risultato?

Figura 2.15: Il `main` come funzione

1. fai partire l'eseguibile `catalogo.exe` nella cartella corrente;
2. passagli come argomento la stringa `catalogo.txt`.

Questo consente all'utente di cambiare l'argomento del programma stesso, cioè di lanciare il programma su file di qualsiasi nome. In altre parole, vogliamo che non solo le singole funzioni, ma anche il programma principale abbia dei dati. In effetti, il programma principale, o `main`, è una funzione come le altre, e quindi ha dati e risultati (vedi Figura 2.15). È però una funzione piuttosto peculiare. La caratteristica principale che lo distingue dalle altre funzioni è che a priori non si sa quanti saranno i parametri e di che tipo saranno: l'utente potrebbe scrivere qualsiasi cosa sulla linea di comando dopo il nome del programma. Quindi, la parte relativa ai dati (quella racchiusa fra parentesi tonde) deve avere flessibilità sufficiente a poter ricevere qualsiasi cosa. Programmi diversi hanno un numero diverso di parametri, dal significato e tipo diverso (parole, numeri interi o reali, ecc...).

La soluzione che si è scelta è interpretare i parametri come stringhe di caratteri (che è poi il loro formato originale, essendo battuti da tastiera dall'utente). Sarà il programma stesso a manipolare queste stringhe per interpretarle (vedi Figura 2.16). Le strutture in cui vengono conservati i parametri sono due:

1. la variabile intera `argc` contiene il numero di parole[10] scritte nella linea di comando;
2. la variabile `argv` è un vettore di stringhe di caratteri (da cui la barocca definizione di tipo `char *argv[]` che non approfondiremo).

Ognuna delle parole scritte nella linea di comando occupa una stringa del vettore stesso, partendo dal nome stesso del programma, usato per lanciarlo, che occupa l'elemento di indice 0, per proseguire con le altre `argc-1` parole.

[10] Con "parola" si intende una sequenza di caratteri senza separatori, come nella specifica `%s`.

```
int main (int argc, char *argv[])
```

I parametri formali non sono definiti direttamente, bensì attraverso

- `int argc`, che è il numero dei parametri
- `char *argv[]`, che è un vettore dinamico di lunghezza `argc` composto da stringhe che descrivono i parametri

I parametri attuali vengono passati attraverso la linea di comando

- `argv[0]` è il nome del programma
- `argv[1]` è il primo parametro
- `argv[2]` è il secondo parametro
- ...

Figura 2.16: Parametri formali e attuali

Il risultato, invece, sarà sempre un numero intero, perché verrà gestito (ammesso che lo sia: spesso non lo è) da *script*, cioè altri programmi al livello del sistema operativo scritti dagli amministratori di sistema. Di solito, ciò che interessa a questo livello è sapere se il programma ha funzionato correttamente oppure no. Nel caso non abbia funzionato, potrebbe essere interessante sapere perché. Tutto questo è facilmente codificabile in un numero intero (vedi Figura 2.17). La libreria standard `stdlib.h`, che viene inclusa in tutti i programmi C con un'opportuna direttiva, comprende direttive che definiscono due costanti simboliche fondamentali a questo riguardo, che abbiamo già utilizzato: la costante `EXIT_SUCCESS` che indica successo, e la costante `EXIT_FAILURE` che indica fallimento (se non si è riusciti ad aprire un file, ad allocare memoria, se i dati letti da file o i parametri letti da linea di comando contengono errori, ecc...). Queste costanti simboliche tipicamente valgono 0 e -1, rispettivamente, ma questi sono dettagli tecnici che non devono interessarci.

Come si usano i parametri del programma principale? La prima cosa da fare è controllare se il numero di parametri è corretto, cioè se siano effettivamente due, perché l'utente ha chiamato il programma passando un nome per il file dei dati oltre al nome del programma. In caso di errore, nel messaggio è sensato spiegare all'utente come doveva essere la linea di comando: avrebbe dovuto contenere il nome del programma (`argv[0]`) seguito dal nome del file dei dati (che non conosciamo: un uso abbastanza comune è mettere fra parentesi quadre la spiegazione di che cosa si deve trovare in un dato posto).

```
if (argc != 2)
{
DA COMPLETARE
Errore nella linea di comando
}
```

Se la linea di comando è corretta, bisogna catturare il primo parametro, cioè la seconda parola `argv[1]` e interpretarla come nome del file dei dati, semplicemente copiandola in un'opportuna stringa variabile. Per copiare una stringa in un'altra,

Il risultato di un programma C è sempre un intero

La libreria `stddef.h` definisce le costanti simboliche

- `EXIT_SUCCESS` (0) da usare se il programma ha avuto successo
- `EXIT_FAILURE` (-1) da usare in caso di errore

Si usa specificare il tipo di errore definendo altre costanti simboliche

Il valore numerico viene restituito al sistema operativo

- alcuni ambienti di compilazione lo indicano all'utente
- i file batch o script lo possono usare

Figura 2.17: Risultato

basta usare la funzione `strcpy` (vedi Figura 2.18) che ha la stringa destinazione come primo parametro e quella originale come secondo.

```
char NomeFile[ROW_LENGTH];

strcpy(NomeFile,argv[1]);
```

Questo funziona finché, come nel caso presente, i parametri sono effettivamente stringhe. Se si trattasse di altri tipi di dati, per esempio numeri interi o reali, abbiamo già visto come si possa convertire una stringa che contiene un numero intero o reale e trasformarla nel numero corrispondente: basta usare la funzione `sscanf`, usando come primo argomento l'elemento desiderato del vettore `argv`, una stringa di formato opportuna, e le variabili in cui andare a salvare i dati in memoria (preceduti da `&` per sottolineare che sono risultati della funzione, e non dati). Vedremo come farlo in dettaglio nel Capitolo **??**.

Ora si può sostituire la stringa `"catalogo.txt"` con la variabile `NomeFile` nella chiamata di `GestisceCatalogo`, compilare ed eseguire il codice, e verificare che tutto funziona correttamente, e consente di usare file di testo diversi con qualsiasi nome.

Per concludere, conviene osservare che le operazioni di interpretazione della linea di comando non stanno bene nel programma principale, perché costituiscono una chiara unità logica. Il programma non è modulare (cosa abbastanza facile procedendo *bottom-up*, mentre la modalità *top-down* suggerisce per natura la costruzione di un albero di procedure controllabile). Conviene quindi trasferire queste operazioni in una funzione `InterpretaLineaComando`. I dati di questa funzione sono `argc` e `argv`. Il risultato è `NomeFile`, che non è elementare, in quanto è una stringa, e quindi va riportato nell'elenco dei dati. Non è preceduto da `&` perché si tratta di un vettore.

```
InterpretaLineaComando(argc,argv,NomeFile);
```

Corrispondentemente, ci sarà una dichiarazione

```
void InterpretaLineaComando (int argc, char *argv[], char *NomeFile);
```

La libreria `string.h` fornisce funzioni per gestire stringhe di caratteri
Per poterle usare bisogna includere la libreria: `#include <string.h>`

La funzione

`strlen(`*s*`)`

fornisce la lunghezza di una stringa

Non si possono copiare stringhe con l'operatore di assegnamento (`=`)

La funzione

`strcpy(`*dest,orig*`)`

copia la stringa *orig* nella stringa *dest*

- il C non controlla che la stringa *dest* possa contenere *orig*:
  se *dest* è più corta, la copia eccede i limiti e sporca altri dati

  La funzione `strncpy(`*dest,orig,n*`)` copia al max. i primi n caratteri

Figura 2.18: Lunghezza di una stringa e copia

Esempio

Ingresso	Istruzione	n	i	j
12 , 34	n = scanf("%d%d",&i,&j);	1	12	invariato
12 , 34	n = scanf("%d,%d",&i,&j);	1	12	invariato
12 , 34	n = scanf("%d ,%d",&i,&j);	2	12	34
12 , 34	n = scanf("%d, %d",&i,&j);	1	12	invariato
12 , 34	n = scanf("%d , %d",&i,&j);	2	12	34

Suggerimento: si noti la posizione della virgola nella stringa di formato

e una definizione, che contiene le istruzioni che avevamo scritto nel programma principale.

Da ora in poi, useremo sempre una funzione per manipolare la linea di comando ed estrarne le istruzioni che l'algoritmo deve eseguire.

TRACCIA?

2.2 Esercizi

Esercizio 1 Si realizzi la funzione `m = ConverteMese(mese);`), che riceve la stringa di caratteri che descrive il nome del mese e restituisce il corrispondente indice numerico e la si inserisca nella posizione corretta del codice.

Esercizio 2 Si realizzi il codice dell'esercizio "catalogo" applicando la progettazione *top-down* anziché *bottom-up*, vale a dire:

1. affrontando direttamente il problema complessivo (lettura di un catalogo da file di testo il cui nome venga indicato nella linea di comando);

Raffinamenti

I modificatori di lunghezza alterano il tipo dell'oggetto

`%hd`	`short` decimale
`%hu`	`unsigned short` decimale
`%ld`	`long` decimale
`%lu`	`unsigned long` decimale
`%lf`	`double`
`%Lf`	`long double`

Un numero (larghezza massima) indica che l'oggetto trovato in ingresso deve contenere un massimo numero dato di caratteri: con ingresso "2010", l'istruzione `scanf("%2d",&i);` assegna a `i` il valore 20

Un asterisco (soppressore) indica che l'oggetto trovato in ingresso non va assegnato ad alcun puntatore: `scanf("%*d",&i);` non altera `i`

Esempio

Ingresso	Istruzione	Risultati
`12 34`	`n = scanf("%*d%d",&i);`	`n = 1` `i = 34`
`Un due tre`	`n = scanf("%*s%s",s);`	`n = 1` `s = "due"`

`12345`	`n = scanf("%1d%2d%3d",&i,&j,&k);`	`n = 3`
		`i = 1`
		`j = 23`
		`k = 45`

`123456`	`n = scanf("%2d%2s%2d",&i,s,&j);`	`n = 3`
		`i = 12`
		`s = "34"`
		`j = 56`

Si noti la mancanza della `&` prima di `s`

Uscita formattata

```
int printf(char *formato, ...)
```

è la principale funzione per la stampa a video

Funziona esattamente come la funzione `scanf`

Ha un numero variabile di parametri $p \geq 1$

- la stringa di formato definisce un *pattern* che si ricostruisce in uscita
- $p - 1$ oggetti dai quali la funzione trae i risultati da stampare

Restituisce il numero di oggetti del *pattern* stampati in uscita

Terminazione di un file

Se si arriva al termine di un file

- la funzione `fscanf` restituisce il numero di oggetti assegnati
- se non ne ha assegnati, restituisce la costante simbolica `EOF`

```
int fscanf(FILE *stream, char *formato, ...)
```

N.B.: `fscanf` restituisce `EOF` se si trova esattamente al termine,
non se il file termina durante il *parsing*

Dopo il fallimento di un'operazione di lettura

- la funzione `feof` restituisce vero, cioè un valore intero non nullo

```
int feof(FILE *stream)
```

Scrittura su file

```
int fprintf(FILE *stream, char *formato, ...)
```

funziona esattamente come `printf` e `sprintf`:

- scrive sullo *stream*
- nel *formato* specificato dalla relativa stringa
- il valore degli oggetti che seguono

```
int *fputs(char *s, FILE *stream)
```

scrive la stringa `s` sullo *stream* di uscita senza aggiungere `'\n'`

```
int *puts(char *s)
```

scrive la stringa `s` sullo `stdout` con un `'\n'` aggiuntivo

Restituiscono `EOF` se falliscono, un valore non negativo altrimenti

2. dividendolo in sottoproblemi, e questi a loro volta in sottoproblemi ancora più semplici;

3. risolvendo ciascuna fase con chiamate a funzioni che risolvano i problemi di livello inferiore, in modo che il codice sia sempre sintatticamente corretto.

2.3 Esercizi sul *parsing*

Esercizio 1 Si supponga di eseguire l'istruzione `n = scanf("%d%f%d",&i,&r,&j);` dove `i`, `j` e `n` sono variabili intere e `r` è una variabile reale. Supponendo di aver scritto da tastiera i seguenti ingressi, si indichi il valore delle quattro variabili e il contenuto dello *stream* di ingresso dopo la chiamata.

- `10 20 30`
- `1.0 2.0 3.0`
- `0.10 0.20 0.30`
- `a.1 .2 3`

Esercizio 2 Si scriva un programma MAGAZZINO.C che riceve da tastiera un articolo alla volta, nel formato

articolo prezzo giorno anno

dove *articolo* è una singola parola di lunghezza massima pari a 10, *prezzo* è un numero reale minore di 1000.0, *giorno* e *anno* sono numeri interi che indicano il giorno e l'anno di una data e *mese* è il nome di uno dei dodici mesi (in minuscolo). Il termine del catalogo di articoli viene rappresentato dall'inserimento da tastiera di una riga contenente solo la parola chiave `STOP`. Si può dare per scontato che non vi siano più di 10 articoli.

Il programma deve poi stampare a video il catalogo in tre colonne, rispettivamente di 10, 9 e 11 caratteri, secondo il formato dell'esempio seguente:

```
Articolo  Prezzo    Data
Articolo1     12.50 21/05/2010
Articolo2    199.90 01/04/2010
```

Esercizio 3 Si scriva un programma DURATA.C che riceve da tastiera due orari, nel formato

ora1 : *minuto1* *ora1* : *minuto2*

con un numero qualsiasi di spazi fra loro, e restituisce la differenza in minuti fra i due orari.

Esercizio 4 Si scriva un programma ESTENSIONE.C che riceve da tastiera il nome di un file, ne cambia l'estensione in `.txt` e restituisce a video il risultato. Si definisca estensione quella parte del nome del file che sta dopo l'ultimo punto. Si tenga conto del fatto che il nome originale del file può contenere più di un punto, ma anche nessun punto.

Suggerimento: si può costruire il risultato sia con la funzione `strcat` sia con la funzione `sprintf`.

Esercizio 5 Si scriva un programma PARAM.C che riceve da linea di comando una qualsiasi sequenza di parametri e la stampa a video, una riga alla volta, nel formato:

```
printf("Il parametro n.  %d e' uguale a %s\n",p,argv[p]);
```

Si sostituisca `%d` con:

- `%2d` per avere una larghezza minima di 2 cifre: per numeri da 1 cifra, questo significa aggiungere uno spazio PRIMA del numero (con 10 o più parametri, questo consente di avere un allineamento perfetto dell'uscita)
- `%-2d` per allineare i numeri a sinistra anziché a destra
- `%02d` per aggiungere uno 0 prima del numero anziché uno spazio

Esercizio 6 Si scriva un programma DATA.C che riceva da linea di comando una data nel formato

`GG/MM/AAAA`

dove `GG` è un numero di due cifre che indica il giorno, `MM` un numero di due cifre che indica il mese e `AAAA` un numero di quattro cifre che indica l'anno, e la trasforma nella corrispondente data in formato letterale, cioè con il mese scritto a parole, verificando che i tre parametri siano effettivamente tre numeri, che siano compresi negli intervalli corretti (da 1 a 12 i mesi, da 1 a 28, 29, 30 o 31 i giorni, ecc. . .).

2.4 Esercizi sui parametri del `main`

Esercizio 1 Scrivere un programma `LUNGHEZZE.C` che stampa a video i parametri della linea di comando, escluso il nome del programma stesso, con la loro lunghezza.

Variante più complessa: scrivere un programma che stampa a video i parametri in ordine di lunghezza crescente, sfruttando gli algoritmi di ordinamento descritti nella lezione sulla ricorsione.

Esercizio 2 Scrivere un programma `SOMMA.C` che stampa a video la somma dei numeri interi passati come parametri nella linea di comando.

Esercizio 3 Scrivere un programma `ESEGUI.C` che riceve da linea di comando una somma o un prodotto nel formato `n1 + n2` o `n1 x n2`, con `n1` e `n2` valori interi, riconosce l'operazione da eseguire e ne restituisce il risultato corretto[11].

Esercizi sulla gestione dei file

Esercizio 1 Si scriva il codice LUNGHEZZA_RIGA.C, che riceve da linea di comando il nome di un file di testo e un numero intero `n` e stampa a video la lunghezza (numero di caratteri escluso l'a capo finale) della riga `n`-esima di tale file. Se il file contiene meno di `n` righe, si deve stampare 0.

Esercizio 2 Si scriva il codice APRIBILE.C, che riceve da linea di comando un numero qualsiasi di nomi di file di testo e stampa a video l'elenco di quelli che si possono aprire e di quelli che non si possono aprire. Il codice deve restituire al sistema operativo il valore `EXIT_SUCCESS` se tutti i file si possono aprire, `EXIT_FAILURE` se almeno uno dei file non si può aprire.

Esercizio 3 Si scriva il codice MAIUSCOLO.C, che riceve da linea di comando il nome di un file di testo e stampa a video il file stesso, convertendo tutte le lettere minuscole in maiuscole. Si considerino le due varianti nelle quali si legge una parola alla volta o una riga alla volta.

Esercizio 4 Si scriva il codice CONCATENA.C, che riceve da linea di comando i nomi di diversi file di testo esistenti e di un file da creare (l'ultimo) e salva nell'ultimo file la concatenazione del contenuto dei file iniziali, nell'ordine con cui compaiono nella linea di comando. Si usi un solo puntatore a file e l'apertura in accodamento.

Esercizio 5 Si scriva il codice LEGGEPROBLEMA.C, che riceve da linea di comando il nome di un file di testo, che contiene i dati per il seguente problema: una fabbrica produce n prodotti utilizzando m componenti. Ciascuna unità del prodotto j garantisce un profitto p_j, ma consuma una quantità a_{ij} del componente

[11] La moltiplicazione va indicata con il carattere `x` anziché il consueto `*` perché in ambiente Linux (in teoria, non in ambiente Windows) il carattere `*` viene usato dal sistema operativo per indicare la lista dei file contenuti nella cartella, e viene convertito in tale lista prima di passarlo all'eseguibile, che quindi non riceve due interi separati da un operatore, ma due interi separati da una lista di nomi di file.

i. In magazzino sono presenti b_i unità del componente i. I dati n, m, p_j, a_{ij} e b_i sono rappresentati secondo le specifiche del linguaggio di modellazione *MathProg*. Un esempio di tale linguaggio è riportato qui di seguito:

```
param NumProdotti := 3 ;
param NumComponenti := 4 ;

param Profitto :=
1 50
2 30
3 30
;

param Consumo :=
[1,1] 2 [1,2] 1 [1,3] 1
[2,1] 1 [2,2] 0 [2,3] 1
[3,1] 0 [3,2] 1 [3,3] 1
[4,1] 1 [4,2] 2 [4,3] 1
;

param Magazzino :=
1 1000
2 400
3 700
4 1200
;

end;
```

Il codice da scrivere deve caricare i valori di n e m in opportune variabili, quelli di p_j e b_i in opportuni vettori e quelli di a_{ij} in un'opportuna matrice. Vettori e matrice devono essere allocati dinamicamente. Quindi il programma deve stampare i dati a video.

Suggerimento: Si veda anche la dispensa `Leggere.pdf` sul sito del corso di programmazione.

Capitolo 3

Complessità computazionale

Questo capitolo inframmezza il ripasso di programmazione in linguaggio C con un argomento teorico che prende il nome di complessità computazionale. Riguarda il fatto che quando ci si trova di fronte a un algoritmo la cosa fondamentale è chiaramente sapere se l'algoritmo risolve il problema cui è dedicato, ma un aspetto non trascurabile è che l'algoritmo potrebbe farlo richiedendo un tempo assolutamente non pratico (anni, se non secoli) o risorse di calcolo molto superiori a quelle disponibili. Il problema, in tal caso, è teoricamente risolto, ma in pratica è come se non lo fosse. La complessità computazionale si occupa del costo di risolvere un problema con un algoritmo, un aspetto che è importante quasi quanto la correttezza.

Questo aspetto interessa gli studenti sia per l'orale di teoria sia per il progetto d'esame. All'orale è richiesto di conoscere e dimostrare la complessità degli algoritmi di base descritti nel corso. Nel progetto d'esame, la relazione deve analizzare l'algoritmo realizzato fornendone il costo.

Nel Capitolo 1[1] si è descritto sia il contesto teorico sia quello tecnologico della risoluzione di un problema con un algoritmo: si tratta di eseguire una sequenza finita e deterministica di operazioni elementari, che si possono per convenzione definire *passi*. Queste operazioni elementari vengono eseguite da una macchina, nel mondo fisico, quindi richiedono tempo, consumando una risorsa scarsa. Inoltre, siccome risolvere un problema significa trasformare la stringa di simboli che codifica l'istanza (i dati del problema) nella stringa di simboli che codifica la soluzione (i risultati), e siccome questo avviene per passaggi intermedi, generando risultati parziali, in generale occorrerà uno spazio dove conservare tutte queste sequenze di simboli: dati, risultati intermedi e risultati finali. Per convenzione, questo spazio è definito *memoria* e organizzato in unità elementari dette *celle*. Avremo quindi un costo in termini di risorse temporali e un costo in termini di risorse spaziali.

Definizione 1 *Diremo* complessità temporale *di un algoritmo A nel risolvere un'istanza I di un dato problema, denotandola con $T_A(I)$, il numero di passi elementari che A esegue per passare dalla istanza I alla soluzione $S(I)$.*

Definizione 2 *Diremo* complessità spaziale *di un algoritmo A nel risolvere un'istanza I di un dato problema, denotandola con $S_A(I)$, il numero di celle di cui A ha bisogno per risolvere l'istanza I.*

Si noti che, siccome i passi elementari dell'algoritmo sono in sequenza, i relativi tempi si sommano. Nel caso dello spazio, invece, non si tratta di una somma, ma

[1] Riferimento

Applicare un algoritmo significa

- eseguire una sequenza finita di operazioni elementari (passi)
- manipolare una sequenza finita di simboli (celle di memoria), che include I all'inizio, S alla fine e risultati parziali nei passi intermedi

Per risolvere un'istanza I con un algoritmo A, quindi si paga un costo

- temporale $T_A(I)$, pari al

  Definizione 3 *numero di passi eseguiti*

  spaziale $S_A(I)$*, pari al*

  Definizione 4 *numero massimo di celle usate in un passo*

È intuitivo che questi costi (complessità) dipendono

- dalle operazioni elementari disponibili (modello computazionale)
- dai simboli disponibili (alfabeto)

ma si dimostra che la dipendenza non è fortissima

Comunque, useremo quasi sempre la macchina RAM

Figura 3.1: Complessità di un algoritmo (costo) (CITARE NEL TESTO)

di un massimo, perché (contrariamente ai passi di tempo) le celle di memoria sono riutilizzabili: quando una cella non serve più, torna libera e ci si può conservare altra informazione; quando un istante di tempo non è più necessario, lasciarlo libero e usarlo per fare qualcos'altro è impossibile. Quindi la differenza fra complessità temporali e spaziali è netta: le prime si sommano, delle seconde si considera il massimo. Entrambe sono però numeri interi positivi.

La complessità temporale e quella spaziale dipendono da vari aspetti. Per prima cosa, dipendono dalle operazioni elementari disponibili. Si tratta di determinare se possiamo prendere n numeri e ordinarli istantaneamente, oppure solo sommarne due, o addirittura solo lavorare su una singola cifra o bit. Questo aspetto è determinato dalla macchina usata e dall'alfabeto che essa adopera, cioè dal fatto che si stiano rappresentando i dati come stringhe di bit (0 o 1) oppure stringhe di caratteri ASCII.

Lasciando i dettagli ai corsi di informatica teorica, basti dire che si può dimostrare che tutti i modelli computazionali e tutti gli alfabeti che hanno senso pratico sono riducibili l'uno all'altro, in modo più o meno sofisticato, così che la relazione fra le complessità temporali e spaziali con cui due diversi modelli computazionali con i relativi alfabeti applicano lo stesso algoritmo per risolvere una data istanza di un dato problema non è enorme: la maggiore delle due è al più un polinomio nella minore. In questo corso, non ci poniamo il problema perché le lezioni di teoria usano la macchina *RAM* e quelle di laboratorio il processore programmato in linguaggio C. Si tratta di due macchine molto simili (anche se non identiche), per le quali le operazioni elementari e i dati elementari sono piuttosto chiari e si corrispondono quasi uno a uno.

A che serve definire la complessità di un algoritmo? Per prima cosa serve a stabilire se un algoritmo è meglio o peggio di un altro. Quindi, ci può servire a stabilire delle relazioni fra algoritmi Poi potremmo dare anche una valutazione assoluta della qualità di un algoritmo, ma cominciamo confrontando due algoritmi, e concentriamoci sul tempo. Vedremo che per lo spazio si possono praticamente ripetere le stesse definizioni. I risultati saranno però in generale diversi. Se si supera una certa soglia di efficienza, i due aspetti cominciano a diventare conflittuali: se si vuole risparmiare tempo, si deve impiegare più spazio; se si vuole risparmiare spazio, si deve impiegare più tempo.

Che significa che un algoritmo A è meglio di un algoritmo A'? Se abbiamo una sola istanza, la definizione è molto semplice: $A(I)$ domina su $A'(I)$ se e solo se il tempo che A impiega a risolvere I è non superiore al tempo che A' impiega a risolvere I, cioè

$$T_A(I) \leq T_{A'}(I)$$

Vorremmo estendere questa definizione da singole istanze a interi problemi, perché sapere che un algoritmo è ideale per risolvere uno specifico caso non è interessante. Sarebbe interessante sapere che è ideale per una famiglia di casi, meglio ancora se infinita. Quel che si vorrebbe ottenere è una *relazione di ordine debole*:

1. che sia *riflessiva* sembra ragionevole, dato che sembra assurdo non poter dire che un algoritmo sia meglio di sé stesso (ovviamente intendendo "non peggio" con l'espressione "meglio")

$$A \preceq A \text{ per ogni algoritmo } A$$

2. che sia *transitiva* sembra anche importante: se non ci fosse la transitività, perderemmo una delle basi della razionalità scientifica (se diciamo che un algoritmo A è meglio di un algoritmo A', e A' è meglio di A'', ci aspettiamo

Ora vogliamo definire il costo degli algoritmi che risolvono un problema P in modo che A sia meglio di A' quando impiega meno tempo (o spazio)

Per una singola istanza I è facile:

$$A(I) \preceq A'(I) \Leftrightarrow T_A(I) \leq T_{A'}(I)$$

Vorremmo estendere il confronto da singole istanze all'intero problema P, istituendo una relazione di ordine debole dotata di:

1. riflessività: $A \preceq A$
2. transitività: $A \preceq A'$ e $A' \preceq A'' \Rightarrow A \preceq A''$
3. completezza: $A \not\preceq A' \Rightarrow A' \preceq A$

per ogni terna di algoritmi A, A' e A'' che risolvono P

Figura 3.2: Confronto fra algoritmi: proprietà richieste (CITARE NEL TESTO)

che A debba essere meglio di A'')

$$A \preceq A' \text{ e } A' \preceq A'' \Rightarrow A \preceq A'' \text{ per ogni terna di algoritmi } A, A', A''$$

3. che sia *completa* sembra anche necessario, per poter dire sempre, dati due algoritmi A e A', o che è meglio A, o che è meglio A', o al limite che sono equivalenti

$$A \not\preceq A' \Rightarrow A' \preceq A \text{ per ogni coppia di algoritmi } A, A'$$

Queste proprietà dovrebbero valere per tutti gli algoritmi corretti per un dato problema, e per ogni problema che ammetta algoritmi risolutivi. Vasto programma, che non realizzeremo del tutto, ma a cui cercheremo di avvicinarci, raggiungendolo quasi del tutto a fini pratici.

Ci sono varie definizioni che vanno nella direzione tracciata. Ne discutiamo tre (vedi Figura 3.3). La prima e più naturale consiste nel dire che un algoritmo è meglio di un altro se impiega meno tempo su tutte le istanze del problema:

$$T_A(I) \leq T_{A'}(I) \text{ per ogni } I \in \mathcal{I}_P$$

Questa è una definizione quasi banale, ma molto complicata da verificare, perché richiede una dimostrazione che prende in considerazione tutte le infinite istanze di un problema. Inoltre, non è praticamente mai possibile garantire questa proprietà, per cui la relazione che ne deriva è lontana dall'essere completa. A meno che non si confronti un algoritmo molto cattivo, è difficile che un altro algoritmo sia sempre meglio: se si ragiona algoritmi ragionevoli, ci sarà sempre l'istanza in cui va meglio uno e l'istanza in cui va meglio l'altro. Quindi, dobbiamo indebolire questa definizione.

La seconda idea abbastanza spontanea è di considerare un caso medio. Il problema del caso medio è che, come sa chi mastica un po' di statistica, la sua definizione richiede una sommatoria (le istanze formano un insieme discreto, per cui non si tratta di un integrale), in generale infinita (i problemi interessanti tendono ad avere un

Ci sono tre definizioni naturali per la relazione d'ordine $A \preceq A'$

1. su tutte le istanze: $T_A(I) \leq T_{A'}(I)$ per ogni $I \in \mathcal{I}_P$
   - è molto complicata da verificare
   - l'ordine non è quasi mai completo
     A sarà migliore su alcune istanze, A' su altre

2. nel caso medio: $E\left[T_A(I)\right] \leq E\left[T_{A'}(I)\right]$
   - richiede di considerare tutte le istanze
   - richiede una distribuzione di probabilità delle istanze
   - richiede calcoli complicati

     È una buona definizione, ma complicata e in parte arbitraria

3. nel caso pessimo: $\max_{I \in \mathcal{I}_P} T_A(I) \leq \max_{I \in \mathcal{I}_P} T_{A'}(I)$
   - spesso è facile identificare le istanze peggiori
   - fornisce un limite superiore, che è un'informazione comunque utile
   - in alcuni problemi il caso pessimo è abbastanza frequente e quindi
     la complessità nel caso pessimo è simile a quella nel caso medio
     (ad es., l'insuccesso in una ricerca)

     È una definizione sbilanciata, ma utile in pratica

Figura 3.3: Confronto fra algoritmi: definizioni possibili

insieme infinito di istanze) su tutte le istanze. Quindi, si ripresenta il problema della prima definizione, di dover considerare tutte le istanze. In aggiunta, la definizione richiede una distribuzione di probabilità. Qual è la distribuzione di probabilità delle istanze? Che probabilità ha ciascuna delle infinite istanze del problema di presentarcisi? Chi affronta lo studio di complessità degli algoritmi nel caso medio di solito fa una ipotesi molto semplice che è un'ipotesi di distribuzione uniforme: tutte le istanze hanno la stessa probabilità. Ne deriva comunque uno studio molto complicato. C'è anche un terzo aspetto, che a rigore rende impossibile la definizione stessa di caso medio, e che riprenderemo fra poco per poterlo risolvere.

Infine, si può pensare di considerare il caso pessimo. Si tratta cioè di considerare tutte le istanze, risolverle con entrambi gli algoritmi, concentrarsi sulla peggiore istanza (quella che richiede più tempo) per ciascuno dei due algoritmi (in generale, si tratterà di istanze diverse), e confrontare i due massimi.

$$\max_{I \in \mathcal{I}_P} T_A(I) \leq \max_{I \in \mathcal{I}_P} T_{A'}(I)$$

Questa definizione è un poco più maneggevole perché il concetto di istanza pessima è spesso più facile da gestire, e non richiede veramente di considerare una per una tutte le istanze del problema. Inoltre, l'istanza pessima dà un limite superiore al tempo di risoluzione delle altre, che può essere molto cattivo, ma è comunque un'informazione utile. Infine, per molti problemi (anche se non per tutti), il caso pessimo (o casi molto cattivi) tende ad essere frequente, cioè può presentarsi frequentemente. Un esempio tipico è la ricerca di una parola in un dizionario: se la "parola" è generata da un processo poco affidabile e molto casuale, è probabile che sia un'accozzaglia di lettere non presente nel dizionario. Questo tipo di parole richiederà probabilmente le ricerche più lunghe e complesse, dato che non può capitare di trovarla per fortuna ai primi tentativi, cosa che può succedere con le vere parole. Quindi, la complessità nel caso pessima è una definizione sbilanciata, ma utile in pratica.

A questo punto, va notato che abbiamo commesso un errore di fondo nelle definizioni introdotte via via. Un insieme infinito di istanze conterrà istanze via via sempre più difficili o grandi, che richiedono un tempo via via crescente, in generale senza un estremo superiore finito. In tal caso, non è possibile definire un valore massimo, e spesso (questo dipende dalle probabilità) nemmeno un valore medio. Come si può ovviare a questo problema?

Evidentemente, dobbiamo tener conto della dimensione delle istanze: istanze più grandi in genere richiedono un tempo maggiore, ma non è questo che intendiamo veramente quando parliamo di complessità: vogliamo definire un concetto che sia piuttosto indipendente dalla semplice dimensione. A questo scopo, ci serve prima introdurre il concetto di dimensione di un'istanza I, che indicheremo con $|I|$, come se fosse una cardinalità, anche se in generale non lo è (vedi Figura 3.4).

In informatica teorica, si farebbe riferimento a un alfabeto usato per codificare l'istanza e si definirebbe la dimensione dell'distanza come il numero di simboli tratti dell'alfabeto che costituiscono la stringa di simboli che codifica l'istanza (in pratica, il numero di caratteri del file di testo che la descrive). Questa è la definizione formale e precisa. In pratica, è poco utile perché molte stringhe diverse consentono di descrivere la stessa istanza. È più utile e intuitivo fare riferimento agli oggetti matematici che compongono l'istanza stessa. Per esempio, il problema può riguardare un insieme di oggetti (si devono ordinare n parole, o numeri interi). È spontaneo considerare il numero n di oggetti come la dimensione dell'istanza, anche se il numero di caratteri del file è maggiore, perché occorrono molti caratteri per ciascun oggetto dell'insieme. In generale, però, esiste un legame forte fra il numero di elementi e il numero dei simboli usati per descriverli. Durante questo corso, quindi, assumeremo impropriamente (ma con buona approssimazione) come concetto di dimensione del-

Caso medio e caso pessimo hanno però un difetto fondamentale:

$$\sup_{I \in \mathcal{I}_P} T_A(I) = +\infty$$

cioè non esiste un tempo massimo su $\mathcal{I}_P$ e spesso neppure medio, perché il problema include infinite istanze, senza limite sul tempo di risoluzione

Si può legare il tempo $T_A(I)$ alla dimensione $|I|$ dell'istanza I, definita

- secondo la teoria, come numero di simboli della codifica di I
- in pratica, attraverso un indice dal significato concreto (o più indici)
  - se il problema riguarda insiemi, il numero di elementi (n)
  - se il problema riguarda relazioni (grafi), il numero di elementi/-nodi (n) e/o il numero di coppie in relazione/archi (m)

Figura 3.4: Complessità e dimensione (1)

l'istanza per un problema che riguarda un insieme il numero di elementi dell'insieme stesso (vedi Figura 3.5). Esistono problemi che riguardano oggetti più complessi, per esempio relazioni. Vedremo un oggetto matematico, detto *grafo*, che consiste in un insieme base di elementi (*nodi*) e un insieme di coppie che sono in relazione fra loro (dette *archi*). In questo caso, sia il numero dei nodi sia quello degli archi si candidano a esprimere la dimensione del problema. Useremo come dimensione dell'istanza una coppia (n, m) di numeri interi, dove n è il numero di nodi e m è il numero degli archi. Questo ovviamente complica la trattazione, ma la rende più fisica e intuitiva, quindi più semplice.

Introdotto il concetto di dimensione, possiamo ragionare, anziché su tutte le istanze di un problema, su un sottoinsieme di istanze a dimensione fissata. Per ogni valore della dimensione n, consideriamo tutte e sole le istanze del problema che hanno dimensione uguale a n e massimizziamo il tempo (oppure ne calcoliamo il valore medio) su quel sottoinsieme di istanze. Ne ricaviamo una definizione di complessità nel caso pessimo e nel caso medio che non si riferiscono all'intero problema, ma al sottoinsieme di stanze di una certa dimensione. Quindi, non un singolo numero, ma nemmeno una funzione con dominio sull'insieme delle istanze, che sono oggetti astratti. Si tratta di una funzione della dimensione, cioè di un numero naturale. Queste funzioni si possono anche disegnare. Nella Figura 3.5 si vedono i profili delle funzioni $T_A(n)$ per due algoritmi, l'algoritmo rosso A e l'algoritmo blu A'. Possiamo disquisire su quale dei due sia meglio dell'altro (sempre tenendo conto che abbiamo scelto una prospettiva, cioè quella del caso pessimo o quella del caso medio. In questo corso adotteremo quasi sempre la prospettiva del caso pessimo, salvo casi episodici in cui si dirà qualche cosa di più sul caso medio.

Dalla figura si vede che i due algoritmi si intrecciano: su alcune dimensioni è meglio A, su altre A'. Abbiamo già visto il problema dell'incompletezza quando cercavamo di introdurre una definizione che valesse istanza per istanza. Anche procedere dimensione per dimensione è troppo dettagliato, cioè chiedere che un algoritmo sia meglio di un altro per tutte le dimensioni per poter dire che lo domina porta a una definizione che in pratica non vale mai, e che richiede di testare infiniti valori di dimensione, anziché infinite stringhe di simboli che codificano le istanze.

Definita la dimensione di ogni istanza

- si considerano le istanze di ogni dimensione n fissata:

$$\mathcal{I}_P^{(n)} = \{I \in \mathcal{I}_P : |I| = n\}$$

- si determina il caso medio o il caso pessimo per ciascuna dimensione

$$T_A(n) = \frac{\sum_{I \in \mathcal{I}_P^{(n)}} T_A(I)}{|\mathcal{I}_P^{(n)}|} \text{ oppure } T_A(n) = \max_{I \in \mathcal{I}_P^{(n)}} T_A(I) \text{ per ogni } n \in \mathbb{N}$$

- si confrontano le funzioni $T_A(n)$ per ogni dimensione n
  Ma anche le funzioni $T_A(n)$ sono ordinate molto raramente

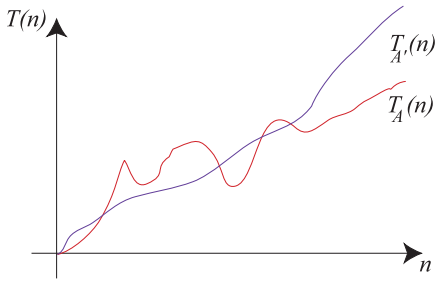


Figura 3.5: Complessità e dimensione (2)

Allora semplifichiamo ulteriormente, introducendo la così detta *complessità asintotica* (vedi Figura 3.6). Questa si basa sul principio che è più importante e utile risolvere velocemente le istanze grande di quelle piccole. Se un algoritmo risolve in microsecondi istanze che un altro algoritmo risolve in secondi, ma impiega giorni su istanze che l'altro riolve in ore, i due algoritmi non sono inconfrontabili: è meglio il secondo, quello che va più veloce sulle istanze grandi. Di solito, infatti, si ipotizza di poter avere la sfortuna di trovarsi a dover risolvere istanze grandi, e il costo risparmiato su queste compensa abbondantemente quello pagato in più sulle altre. Ci interessa quindi quel che succede non per tutti i valori di n, ma solo per n non inferiore a una data soglia. Fissare una soglia precisa uguale per tutti i problemi non ha ovviamente senso (nessuna soglia specifica ha motivi forti per essere scelta), per cui la definizione consente di scegliere una soglia *ad hoc* per il problema che ci interessa. Secondo la definizione, quindi un algoritmo A è meglio di un algoritmo A^{prime} se e solo se esiste un opportuno valore n_0 tale che il massimo tempo che A impiega a risolvere problemi di dimensione n è minore uguale al tempo massimo che A' impiega a risolvere problemi della stessa dimensione n per tutte le dimensioni maggiori o uguali di n_0. Questo valore n_0 non è fissato a priori, ma va scoperto analizzando i due algoritmi e il problema.

Conta di più risolvere in fretta le istanze "grandi" che quelle "piccole"

Impiegare due giorni anziché uno è peggio che due secondi anziché uno

$$A \preceq A' \Leftrightarrow T_A(n) \leq T_{A'}(n) \text{ per ogni } n \geq n_0$$

A è meglio di A' quando usa meno tempo (o spazio)

- sulla peggior istanza di dimensione n
- per ogni valore di $n \geq n_0$, per un opportuno valore di $n_0 \in \mathbb{N}$

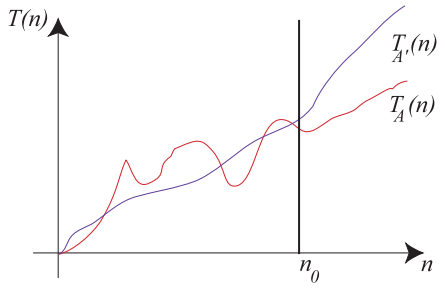


Figura 3.6: Complessità asintotica

Questo concetto è relativo, cioè introduce una relazione fra coppie di algoritmi. Avviciniamoci a una definizione invece assoluta di complessità. Per farlo, introdurremo degli *spazi funzionali*, cioè insiemi di funzioni. L'idea è assumere delle funzioni di riferimento scelte arbitrariamente in base alla loro semplicità e accorpare loro, ovvero approssimare con loro, tutte le funzioni che asintoticamente non sono né migliori né peggiori. Si veda la Figura 3.7: la retta $f(n) = n$ sale linearmente con la pendenza di 45 gradi. Lo spazio funzionale $\Theta(f(n))$ è l'insieme di tutte le funzioni che si comportano asintoticamente (cioè per ogni n maggiore o uguale a un opportuno valore n_0). "Comportarsi" significa non essere né molto più piccole né molto più grandi.

Più nel dettaglio, devono esistere due parametri reali c_1 e c_2, entrambi positivi, oltre al parametro naturale n_0, tale che $c_1 f(n) \leq T(n)$ e $T(n) \leq c_2 f(n)$ per tutti i valori $n \geq n_0$. I valori c_1, c_2 e n_0 non sono dati a priori, ma devono essere fissi e indipendenti da n. Si può riassumere la definizione dicendo che il tempo di esecuzione $T(n)$ dell'algoritmo per la peggiore istanza di ciascuna dimensione n (o il valor medio di tale tempo, se stiamo studiando l'altra definizione di complessità) può comportarsi in modo estemporaneo solo fino a un dato valore n_0. Da lì in poi, la funzione è racchiusa nella regione compresa come in un *sandwich* fra i profili delle funzioni $c_1 f(n)$ e $c_2 f(n)$, entrambi proporzionali a $f(n)$. Si noti che questi due profili possono essere entrambi inferiori a quello di $f(n)$, come nella figura, oppure entrambi superiori o uno inferiore e l'altro superiore. Quello che succede nella parte iniziale del profilo, prima di n_0, non importa.

L'idea fondamentale è *ignorare i fattori moltiplicativi*. Questo consente di dire che, se $T(n) \in \Theta(f(n))$, allora per istanze grandi il tempo di calcolo dell'algoritmo è proporzionale ai valori della funzione $f(n)$, non con una costante di proporzionalità fissa, ma con un fattore variabile compresa fra due costanti. Il fattore, quindi, non è totalmente arbitrario. Questa definizione è stranamente vaga, perché i tre parametri che postula sono liberi, ma l'aspetto fondamentale è che sono costanti.

Data una funzione approssimante $f(n)$

$$T(n) \in \Theta(f(n))$$

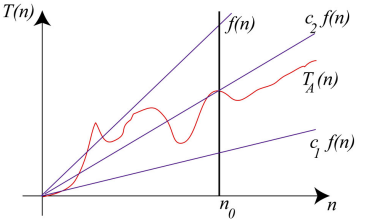
significa formalmente che

$$\exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N} : c_1\, f(n) \leq T(n) \leq c_2\, f(n) \text{ for all } n \geq n_0$$

dove c_1, c_2 e n_0 sono indipendenti da n

$T(n)$ è *"chiusa a sandwich"* fra $c_1 f(n)$ e $c_2 f(n)$

- per qualche valore *"piccolo"* di c_1
- per qualche valore *"grande"* di c_2
- per ogni valore *"grande* di n
- per qualche definizione di *"piccolo" e "grande"*



Asintoticamente, $f(n)$ stima $T(n)$ a meno di un fattore moltiplicativo:

- per istanze grandi, il tempo di calcolo è almeno e al massimo proporzionale ai valori della funzione $f(n)$

Figura 3.7: Notazione Θ

Dalla definizione dello spazio funzionale $\Theta(f(n))$ derivano banalmente altre due definizioni, ignorando uno dei due limiti imposti su $T(n)$. La notazione O (vedi Figura 3.8) è spesso più nota della notazione Θ si ottiene semplicemente ignorando il limite inferiore, cioè richiedendo solo l'esistenza di n_0 e di c_2, che a questo punto chiameremo semplicemente c, tali che $T(n) \leq c f(n)$ per tutti i valori $n \geq n_0$. Questo significa che $T(n)$ viene dominata da un'opportuna amplificazione di $f(n)$ (anche se non necessariamente da $f(n)$: nel disegno si vede che $T(n)$ supera $f(n)$ anche dopo n_0)[2]

[2]Nei disegni attuali la funzione $f(n)$ è sempre la stessa, il che forse confonde le idee.

Data una funzione approssimante $f(n)$

$$T(n) \in O(f(n))$$

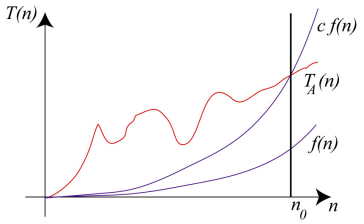
significa formalmente che

$$\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} : T(n) \leq c\, f(n) \text{ for all } n \geq n_0$$

dove c, e n_0 sono indipendenti da n

$T(n)$ è *"dominata"* da $cf(n)$

- per qualche valore *"grande"* di c
- per ogni valore *"grande* di n
- per qualche definizione di *"grande"*



Asintoticamente, $f(n)$ stima $T(n)$ per eccesso a meno di un fattore moltiplicativo:

- per istanze grandi, il tempo di calcolo è al massimo proporzionale ai valori della funzione $f(n)$

Figura 3.8: Notazione O

Lo spazio funzionale $\Omega(f(n))$ ha ovviamente una definizione complementare: si impone un limite solo inferiore, chiedendo che sia $T(n) \geq c_1 f(n)$ per ogni $n \geq n_0$, con un opportuno valore c_1, che a questo punto possiamo chiamare semplicemente c, e un opportuno valore di n_0. La scelta di n_0, c_1 e c_2 è completamente libera.

A che serve tutto questo? A consentire l'analisi di algoritmi senza dover precisare esattamente il numero di passi che compie su ciascuna istanza. Con ragionamenti abbastanza elementari, infatti, si può concludere che questo numero (incognito) non sarà mai maggiore di o minore di opportune soglie espresse come funzioni della dimensione n. Questo già basta a dimostrare l'appartenenza di $T(n)$ a uno di questi tre spazi funzionali. A loro volta, gli spazi funzionali consentono di sostituire all'espressione di $T(n)$, potenzialmente molto complicata, delle funzioni approssimanti tipicamente molto semplici. Queste approssimanti danno un'idea elementare del costo di esecuzione dell'algoritmo. Inoltre, gli spazi funzionali consentono confrontare algoritmi in modo semplice, perché basta confrontare le approssimanti anziché le espressioni esatte delle rispettive complessità. Per esempio, una funzione quadratica prima o poi diventa più grande di una lineare, qualunque siano gli specifici coefficienti usati nelle loro espressioni.

Perché stiamo ignorando le costanti moltiplicative? Non solo per semplificarsi la vita, ma anche perché il tempo fisicamente impiegato a eseguire delle operazioni dipende da come è fatto il processore, cioè da un gran numero di dettagli tecnici, ma noi non vogliamo studiare l'algoritmo in maniera il più possibile indipendente dalla macchina che lo esegue: vogliamo una teoria che valga in generale. Di conseguenza, non vogliamo stimare il tempo fisico in secondi di calcolo, ma il numero di operazioni elementari richiesto, e questo non dipende dalla tecnologia ed è lo stesso per tutte le macchine che hanno lo stesso insieme di operazioni elementari (vedi Figura 3.10). Sostituendo il tempo con il numero di operazioni libera dalla necessità di specificare la macchina utilizzata e permette di considerare uniforme

Data una funzione approssimante $f(n)$

$$T(n) \in \Omega(f(n))$$

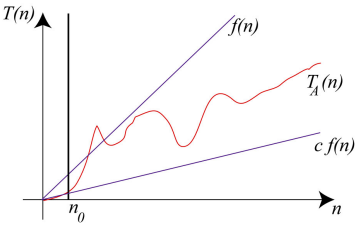
significa formalmente che

$$\exists c > 0, n_0 \in \mathbb{N} : T(n) \geq c\, f(n) \text{ for all } n \geq n_0$$

dove c e n_0 sono indipendenti da n

$T(n)$ *"domina"* $cf(n)$

- per qualche valore *"piccolo"* di c
- per ogni valore *"grande* di n
- per qualche definizione di *"piccolo" e "grande"*



Asintoticamente, $f(n)$ stima $T(n)$ per difetto a meno di un fattore moltiplicativo:

- per istanze grandi, il tempo di calcolo è almeno proporzionale ai valori della funzione $f(n)$

Figura 3.9: Notazione Ω

il tempo richiesto da tutte le operazioni elementari. Questo non è vero: fare un prodotto è più difficile che fare una somma; se si considerano le operazioni bit per bit cambia addirittura la funzione approssimante (quadratica anziché lineare), ma finché adottiamo quello che nelle lezioni di teoria viene introdotto come *criterio di costo uniforme*, potremo dire che tutte le operazioni elementari hanno la stessa durata. In realta', un prodotto richiede magari decine di volte il tempo di una, ma questo rapporto è costante e le costanti si possono ignorare perché i coefficienti c_1 e c_2 nei confronti e nelle valutazioni di complessità sono definibili a piacere. Quindi, non siamo più interessati a conoscerne il valore esatto. Inoltre, se compriamo un processore più rapido, l'analisi non cambia di una virgola, perché non è riferita al tempo fisico di calcolo, che è cambiato, ma alle operazioni, che non lo sono.

È veramente utile una definizione così vaga? Consideriamo l'esempio pratico della Figura 3.11: supponiamo di avere a disposizione un giorno di calcolo e due possibili macchine. La prima è più lenta, e richiede un microsecondo per ogni operazione elementare (potrebbe essere una degli anni '80-'90, con una frequenza di *clock* dell'ordine dei Megahertz). Abbiamo anche sei algoritmi per risolvere un problema, corrispondenti alle colonne della tabella. Un algoritmo di complessità lineare ha tempo di esecuzione appartenente allo spazio funzionale $\Theta(n)$, cioè esegue un numero di operazioni compreso fra $c_1 n$ e $c_2 n$, con c_1 opportunamente piccolo e c_2 opportunamente grande. Semplifichiamo e supponiamo che il numero sia esattamente n. Avendo a disposizione un giorno di tempo, una macchina che fa un'operazione ogni microsecondo e un algoritmo che richiede esattamente n operazioni per risolvere un'istanza di dimensione n, qual è l'istanza più grande che possiamo risolvere? Ovviamente, l'istanza di dimensione pari al numero di microsecondi contenuti in un giorno, cioè $8.64 \cdot 10^{10}$, che è decisamente grande. Considerando la stessa macchina e lo stesso tempo, ma un algoritmo un po' meno efficiente, per esempio in $\Theta(nlogn)$ (e ipotizziamo per semplicità che le operazioni siano esattamente $n \log_2 n$, in un

Il tempo di calcolo effettivo è il prodotto del numero di operazioni elementari T_A per il tempo γ richiesto da ciascuna

$$T_{\text{eff}} = T_A \, \gamma$$

Il tempo γ richiesto per un'operazione elementare

- dipende dalla tecnologia
- non è rigorosamente uguale per tutte le operazioni

Se cambia la specifica macchina usata, ma non la sua struttura

- il tempo γ richiesto per ciascuna operazione può cambiare
- il numero T_A di operazioni elementari rimane uguale

Un'analisi che ignora i fattori moltiplicativi è valida per tutte le macchine che aderiscono allo stesso modello computazionale

Figura 3.10: Perché ignorare le costanti moltiplicative?

giorno riusciremo a fare meno operazioni, e quindi a risolvere solo problemi un più piccoli. Il problema di dimensione massima che riusciamo a risolvere ha dimensione $2.75 \cdot 10^9$, cioè $30-40$ volte più piccolo. Se l'algoritmo diventa quadratico, riusciamo a risolvere solo istanze di dimensione $2.94 \cdot 10^5$ alla quinta; se è cubico, $4.42 \cdot 10^3$. Sono ancora istanze abbastanza grandi, ma nettamente meno. Se l'algoritmo diventa esponenziale e richiede 2^n operazioni, allora l'istanza più grande che si può risolvere ha dimensione 36, che è piccola. Se si devono riordinare le parole di un vocabolario con un algoritmo lineare o quasi (come faremo durante il corso), potremo trattare vocabolari di dimensioni galattiche. Se l'algoritmo è quadratico (come anche faremo durante il corso), tratteremo dizionari tipici, da $100\,000 - 200\,000$ parole. Se l'algoritmo è esponenziale, ordineremo dizionarietti di una trentina di parole, e con 3^n addirittura di 23 parole.

Una possibile reazione è procurarsi una macchina più potente, come quelle moderne, che hanno frequenze dell'ordine dei Gigahertz, per esempio una che richiede un nanosecondo per operazione, e quindi esegue un miliardo di operazioni al secondo. L'algoritmo lineare va $1\,000$ volte più veloce e risolve istanze $1\,000$ volte più grandi, dunque sfrutta pienamente la nuova macchina. L'algoritmo di complessità $n \log_2 n$ diventa qualche centinaio di volte più veloce, e l'istanza massima cresce in corrispondenza. L'algoritmo quadratico diventa una trentina di volte più veloce, e quindi il vantaggio offerto da una macchina tanto più veloce è già molto meno interessante. L'algoritmo cubico diventa 10 volte più veloce. Gli algoritmi esponenziali ordinando dizionarietti di 46 parole (anziché 36) o 29 parole (anziché 22), e quindi il vantaggio è minimo.

Le conclusioni sono due. La prima è che un algoritmo buono compensa ampiamente una macchina scadente. La seconda è che un algoritmo buono sfrutta meglio una macchina buona. Se la tecnologia è sicuramente importante, l'aspetto teorico, algoritmico e computazionale, lo è di più.

Supponiamo di avere

- un giorno di calcolo a disposizione
- due macchine, il cui tempo medio per operazione elementare è pari, rispettivamente, a $\gamma_1 = 1\mu$s/op e $\gamma_2 = 1$ ns/op
- algoritmi di complessità diversa per lo stesso problema

La massima dimensione trattabile nel tempo disponibile è

$$\bar{n} = \max\left\{n \in \mathbb{N} : T_A(n)\,\gamma \leq 8.64 \cdot 10^{10}\mu\text{s} = 1\,\text{giorno}\right\}$$

	$\bar{n}$					
$T_A(n)$	n	$n \log_2 n$	n^2	n^3	2^n	3^n
$\gamma = 1\mu$s/op	$8.64 \cdot 10^{10}$	$2.75 \cdot 10^9$	$2.94 \cdot 10^5$	$4.42 \cdot 10^3$	36.33	22.92
$\gamma = 1$ ns/op	$8.64 \cdot 10^{13}$	$8.29 \cdot 10^{11}$	$9.30 \cdot 10^6$	$4.42 \cdot 10^4$	46.30	29.21

Se ne deduce facilmente che

- un algoritmo migliore surclassa una macchina migliore
- una macchina più veloce è utile solo se si usa un algoritmo veloce

Figura 3.11: Impatto pratico della complessità

Riassumendo, lo scopo delle definizione di complessità asintotica nel caso pessimo (sia per lo spazio sia per il tempo) è aggregare funzioni diverse (cioè algoritmi diversi) in una famiglia che contenga esemplari praticamente equivalenti, dove l'equivalenza consiste nel richiedere tempi che hanno una costante come rapporto tra loro. La costante può essere grande a piacere (un algoritmo può essere miliardi di volte più lento), ma non deve dipendere dalla istanza stessa: ci dev essere un limite massimo al rapporto di velocità. Dal punto di vista del progresso tecnologico, gli algoritmi della stessa famiglia possono essere resi confrontabili usando macchine sufficientemente più veloci; per gli algoritmi di famiglie diverse la tecnologia può fare qualcosa per ovviare alla diversa complessità, ma non molto

3.0.1 Esercizio 1

Questo esercizio mostra come condurre in pratica una dimostrazione di appartenenza a uno spazio funzionale $\Theta(\cdot)$[3]. Si veda la Figura 3.12.

Supponiamo di avere un algoritmo che impiega un tempo (o uno spazio: non cambierebbe nulla) pari a $T(n) = 3n^2 + 7n + 8$ per risolvere la peggiore istanza di dimensione n di un problema. È una funzione precisissima, cosa che non avremo mai, ma la definizione suppone di disporre di questa funzione. Cerchiamo di dimostrare che questa funzione appartiene allo spazio funzionale $\Theta(n^2)$, cioè in gergo l'algoritmo è quadratico. Dal punto di vista della definizione, la tesi da dimostrare è che esistano due costanti reali c_1 e c_2, entrambe positive, e una costante intera positiva n_0, tali che $c_1 n^2 \leq 3n^2 + 7n + 8 \leq c_2 n^2$ per tutti i valori di $n \geq n_0$, cioè

[3] Questi esercizi andranno spostati in appendice al capitolo, con un rimando là dove si ha la definizione, e integrati con gli esercizi già esistenti.

che l'algoritmo risolva tutte le istanze di dimensione $n \geq n_0$ in un tempo che al massimo (caso pessimo!) è compreso $c_1 n^2$ e $c_2 n^2$.

La dimostrazione è costruttiva, cioè consiste nel trovare i valori di c_1, c_2 e n_0 che soddisfano questa espressione. Merita osservare che, trovata una terna di valori che la soddisfano, se ne trovano infinite, perché tutti i valori di c_1 più piccolo e tutti i valori di c_2 e di n_0 più grandi la soddisfano a maggior ragione. Di conseguenza, il procedimento consiste nell'"indovinare" i valori di c_1, c_2 e n_0: se si ha successo, ci si ferma; se si fallisce, si diminuisce c_1 o si aumenta c_2 o si aumenta n_0 finché non si ha successo. Di solito, intuizione ed esperienza bastano a garantire una scelta corretta al primo passo.

Più in dettaglio, faremo un'ipotesi sui valori di c_1 e c_2 e, sostituiti i valori nella doppia disequazione, trovare se esiste un valore minimo n_0 che garantisca di soddisfarla con tutti i valori più grandi. Nel nostro caso, se si pone $c_1 = 3$, è chiaro che $3n^2 + 7n + 8 \geq 3n^2$ per ogni valore di n, dato che la prima disuguaglianza diventa $7n + 8 \leq 0$, che è vera per ogni numero intero positivo n. Quindi la scelta di c_1 è praticamente ovvia e suggerisce come potenziale valore per n_0 il primo numero intero positivo, cioè $n_0 = 1$. Più difficile è scegliere il valore di c_2, ma la forma della funzione $T(n)$ suggerisce che qualsiasi numero maggiore di 3 sia già sufficiente, purché n sia abbastanza grande. Proviamo con 4 per ragionare con numeri interi, e quindi forse mantenere calcoli più semplici. La seconda disuguaglianza diventa $3n^2 + 7n + 8 \leq 4n^2$, cioè $n^2 - 7n - 8 \geq 0$, che vale per $n \leq -1$ oppure per $n \geq 8$. Il primo intervallo è inutile, sia perché riguarda numeri negativi sia perché è superiormente limitato, mentre il secondo intervallo ha la forma che desideriamo ($n \geq n_0$), semplicemente a patto di porre $n_0 = 8$. Per imporre entrambe le disuguaglianze, scegliamo la condizione più stretta su n_0, dunque $n_0 = 8$. Riassumendo, con i valori $c_1 = 3$, $c_2 = 4$ e $n_0 = 8$ è facile dimostrare la tesi.

Dimostrare che $T(n) = 3n^2 + 7n + 8 \in \Theta\left(n^2\right)$, cioè che

$$\exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N}^+ : c_1 n^2 \leq 3n^2 + 7n + 8 \leq c_2 n^2 \quad \forall n \geq n_0$$

Il procedimento è semplice

1. si fa un'ipotesi sul valore di c_1 e c_2,
   basata su regole generali, intuizione o semplici tentativi
2. si ricava n_0 in modo da rispettare la tesi

Poniamo $c_1 = 3$ e $c_2 = 4$:

- la prima disuguaglianza diventa

  $$0 \leq 7n + 8 \qquad \text{che vale per ogni } n \geq 1$$

- la seconda disuguaglianza diventa

  $$7n + 8 \leq n^2 \qquad \text{che vale per ogni } n \geq 8$$

Di conseguenza, $c_1 = 3$, $c_2 = 4$ e $n_0 = 8$ soddisfano la definizione

Figura 3.12: Esercizio 1

3.0.2 Esercizio 2

Passiamo ora a una dimostrazione di non appartenenza, che è un po' più raffinata, ma usa esattamente gli stessi strumenti. Vogliamo dimostrare che $T(n) = 3n^2 - 7n + 2$ non appartenga a $\Theta(n)$, cioè che non è lineare. È abbastanza evidente che è così, essendo quadratica, ma vediamo la dimostrazione (Figura 3.13).

Questa volta dobbiamo dimostrare che non esistono due numeri c_1 e c_2 reali positivi e un numero n_0 intero positivo, tali che $c_1 n \leq 3n^2 - 7n + 2 \leq c_2 n$ per ogni $n \geq n_0$. È una tesi abbastanza sofisticata, con dei quantificatori di esistenza negati e un quantificatore universale. La logica indica che è possibile trasformare questa espressione in un'espressione equivalente secondo la quale esiste un valore $n \geq n_0$ tale che almeno una delle due disuguaglianze è violata, cioè $c_1 n > 3n^2 - 7n + 2$ oppure $3n^2 - 7n + 2 > c_2 n$, e questo vale per tutti i valori reali positivi di c_1 e c_2 e per tutti i valori interi positivi di n_0.

Si può visualizzare questa situazione come una specie di gioco, nel quale il nostro avversario cerca di assegnare a c_1, c_2 e n_0 dei valori nei rispettivi dominii in maniera da impedirci di trovare un valore n che sia non inferiore a n_0 (e questo sarà facile) e che violi almeno una delle due disuguaglianze. Questo è più complicato, ma possiamo almeno scegliere quale delle due disuguaglianze violare. La prima ($c_1 n > 3n^2 - 7n + 2$) è poco promettente, perché è difficile rendere un'espressione lineare maggiore di un'espressione quadratica per valori di $n \geq n_0$, con un avversario che può alzare a piacere n_0 costringendoci a prendere valori di n alti. Seguiamo la seconda strada, cioè cerchiamo di dimostrare che un'espressione lineare è più piccola di una quadratica: $c_2 n < 3n^2 - 7n + 2$. Nel "gioco", il nostro avversario sceglie per primo c_1, c_2 e n_0, e poi noi scegliamo n. Quindi, abbiamo la facoltà di assegnare a n un valore intero positivo funzione di c_1, c_2 e n_0. Risolvendo in modo parametrico la disequazione, si ottengono due casi possibili:

1. per $(7 + c_2)^2 - 24 < 0$, la disuguaglianza è sempre verificata, e quindi vinciamo il gioco semplicemente imponendo $n \geq n_0$;
2. per $(7 + c_2)^2 - 24 \geq 0$, la disuguaglianza è verificata per $n < \frac{7 + c_2 - \sqrt{\phi(c_2)}}{6}$ oppure per $n > \frac{7 + c_2 + \sqrt{\phi(c_2)}}{6}$, dove $\phi(c_2) = (7 + c_2)^2 - 24$.

Ancora una volta, possiamo scegliere, ma è chiaro che la prima condizione, che limita superiormente n, non ci fa gioco, perché contrasta con la condizione che n sia non inferiore a un n_0 fissabile a piacere. Sceglieremo la seconda condizione, $n > \frac{7 + c_2 + \sqrt{\phi(c_2)}}{6}$, e la combineremo con $n \geq n_0$ in un'espressione che permette di soddisfare tutte le condizioni richieste:

$$n = \max\left(n_0, \left\lceil \frac{7 + c_2 + \sqrt{\phi(c_2)}}{6} \right\rceil + 1\right)$$

In generale, si procede in modo molto più spiccio, lasciando implicite le tecnicalità.

La Figura 3.14 riporta una serie di altri possibili esercizi del genere di quelli visti.

3.1 Proprietà fondamentali

Gli spazi funzionali che abbiamo introdotto hanno una serie di interessanti proprietà, che costituiscono teoremi facilmente dimostrabili, e la cui dimostrazione può essere un utile esercizio per verificare la propria comprensione dei concetti sopra introdotti.

Dimostrare che $T(n) = 3n^2 - 7n + 2 \notin \Theta(n)$, cioè che

$$\nexists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N}^+ : c_1 n \leq 3n^2 - 7n + 2 \leq c_2 n \quad \forall n \geq n_0$$

Questo equivale a

$$\exists n \geq n_0 : c_1 n > 3n^2 - 7n + 2 \text{ oppure } 3n^2 - 7n + 2 > c_2 n \; \forall c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N}^+$$

Si deve trovare una n funzione di c_1, c_2 e n_0 che soddisfi la tesi

Basta soddisfare una delle due disuguaglianze: scegliamo la seconda

$$3n^2 - 7n + 2 > c_2 n$$

Se $(7 + c_2)^2 - 24 = \phi(c_2) < 0$, la disuguaglianza vale per ogni $n \in \mathbb{N}$.

Altrimenti, vale per $n < \frac{7+c_2-\sqrt{\phi(c_2)}}{6}$ oppure per $n > \frac{7+c_2+\sqrt{\phi(c_2)}}{6}$

Scegliamo la seconda disuguaglianza, e la combiniamo con $n \geq n_0$:

$$n = \max\left(n_0, \left\lceil \frac{7 + c_2 + \sqrt{\phi(c_2)}}{6} \right\rceil + 1\right)$$

Figura 3.13: Esercizio 2

Si dimostri che:

- $n^2 \in \Theta\left(n^2 + 4n + 3\right)$
- $2n^2 + 3n \in \Theta\left(n^2\right)$
- $6n^2 + 2n \in \Theta\left(n^2\right)$
- $n^2 \in O\left(n^2/4 - 2\right)$
- $2n^2 + 3n \in O\left(n^3\right)$
- $n^4 \in O\left(2^n\right)$
- $n \log_2 n \in O\left(n^2\right)$
- $n^2 \in \Omega\left(n^2 + 2n + 5\right)$
- $3n^2 - 2n \in \Omega\left(n^2\right)$
- $3n^5 \in \Omega\left(n^4\right)$
- $4n^2 \notin \Theta\left(n^3\right)$
- $n^2 \notin O\left(10^6 n\right)$
- $3^n \notin O\left(2^n\right)$
- $2n^2 - 3n \notin \Omega\left(n \log_2 n\right)$

Figura 3.14: Esercizi

Sono tutte proprietà decisamente intuitive. Per cominciare (vedi Figura 3.15), vale una proprietà di *riflessività* secondo la quale ogni funzione appartiene allo spazio funzionale associato la funzione stessa.

C'è una proprietà di *simmetria*, secondo la quale se una funzione appartiene allo spazio associata a un'altra, questa appartiene allo spazio associato alla prima. La proprietà vale per gli spazi Θ, la cui definizione è simmetrica, e non per gli altri due, che hanno definizioni asimmetriche. Esiste però una sorta di *simmetria trasposta*, per cui se una funzione appartiene allo spazio O associato a un'altra, questa appartiene allo spazio Ω associato alla prima: se una funzione è tendenzialmente (cioè asintoticamente e a meno di costanti moltiplicative) inferiore a un'altra, l'altra è tendenzialmente superiore alla prima. L'*antisimmetria* implica che una funzione appartenente sia allo spazio Ω sia allo spazio O di un'altra, allora appartiene allo spazio Θ, e viceversa.

Tutto questo richiama immediatamente in modo naturale un parallelo fra l'appartenenza allo spazio Θ e l'uguaglianza, l'appartenenza a O e la relazione $\leq$ e l'appartenenza a Ω e la relazione $\geq$. Questa è una buona guida mnemonica, ma bisogna tener presente che le relazioni di cui stiamo trattando sono leggemente più deboli, e non godono di tutte le proprietà di cui godono le relazioni alle quali le stiamo facendo corrispondere.

Riflessività

- $f(n) \in \Theta(f(n))$
- $f(n) \in O(f(n))$
- $f(n) \in \Omega(f(n))$

Simmetria (solo per Θ)

- $g(n) \in \Theta(f(n)) \Leftrightarrow f(n) \in \Theta(g(n))$

Simmetria trasposta (fra O e Ω)

- $g(n) \in O(f(n)) \Leftrightarrow f(n) \in \Omega(g(n))$

Antisimmetria

- $\begin{cases} f(n) \in O(g(n)) \\ f(n) \in \Omega(g(n)) \end{cases} \Leftrightarrow f(n) \in \Theta(g(n))$

Sono facili da ricordare, associando mentalmente Θ a $=$, O a $\leq$ e Ω a $\geq$

Figura 3.15: Proprietà fondamentali (1)

Un'altra proprietà che conferma queste corrispondenze è la *transitività*, per cui se una funzione appartiene a uno dei tre spazi associati a un'altra e quest'altra allo stesso spazio di una terza, allora la prima funzione appartiene a quello spazio associato alla terza. Viene invece violata la proprietà di *completezza*: non è vero che, date due funzioni, ciascuna delle due cade in uno dei tre spazi associati all'altra, mentre un numero è necessariamente minore, uguale o maggiore di qualsiasi altro. Parlando di due funzioni, non sempre una è più piccola, uguale o più grande dell'altra. È perfettamente possibile avere due funzioni che violano questa proprietà.

Per esempio, n e $n^{1+\sin\left(\frac{\pi}{2}n\right)}$ si "sfuggono" l'un l'altra anche considerando arbitrarie costanti moltiplicative. Infatti, per valori pari di n, le due funzioni sono identiche e cadono ciascuna nello spazio Θ dell'altra. Per valori dispari che diano resto 1 quando siano divisi per 4, la seconda funzione diventa n^2, ed è chiaramente in $\Omega(n) \setminus \Theta(n)$. Infine, per valori di n dispari che diano resto 3 quando siano divisi per 4, la seconda funzione diventa 1, che cade in $O(n) \setminus \Theta(n)$. Quindi, queste due funzioni hanno la caratteristica che periodicamente una delle due è nettamente inferiore all'altra, uguale o nettamente superiore, oltre qualsiasi costante moltiplicativa. È chiaro che sono funzioni strane, costruite a tavolino: gli algoritmi di solito non si comportano così, e questo è confortante, perché significa che la definizione di complessità asintotica nel caso pessimo, anche se in teoria non consente di ordinare completamente gli algoritmi dal più efficiente al meno efficiente, in pratica invece riesce a farlo.

Transitività

- $\begin{cases} f(n) \in \Theta(g(n)) \\ g(n) \in \Theta(h(n)) \end{cases} \Rightarrow f(n) \in \Theta(h(n))$
- $\begin{cases} f(n) \in O(g(n)) \\ g(n) \in O(h(n)) \end{cases} \Rightarrow f(n) \in O(h(n))$
- $\begin{cases} f(n) \in \Omega(g(n)) \\ g(n) \in \Omega(h(n)) \end{cases} \Rightarrow f(n) \in \Omega(h(n))$

Sono facili da ricordare, associando mentalmente Θ a $=$, O a $\leq$ e Ω a $\geq$

Non vale la completezza: esistono funzioni non confrontabili
(*sono rare e "strane": per esempio,* n *e* $n^{1+\sin\left(\frac{\pi}{2}n\right)}$)

Figura 3.16: Proprietà fondamentali (2)

Oltre ai tre spazi funzionali sopra introdotti, nei corsi di algoritmi sono di uso comune altri spazi, che vengono spesso adottati nei corsi di analisi matematica: gli spazi $o(\cdot)$ e $\omega(\cdot)$ (vedi Figura 3.17). Questi spazi funzionali sono definiti sulla base del concetto di limite: una funzione $T(n)$ appartiene a $o(f(n))$ quando il limite per n che tende a $+\infty$ del rapporto $T(n)/f(n)$ è nullo. In un certo qual modo,questo corrisponde a dire che $T(n)$ è asintoticamente più piccola di $f(n)$, ma il concetto è più forte che nel caso dello spazio $O(f(n))$. Si può infatti dimostrare (e la Figura 3.17 riporta una traccia di dimostrazione) che l'appartenenza a $o(f(n))$ è condizione sufficiente, anche se non necessaria, per l'appartenenza a $O(f(n))$. Questo consente di usare il limite anche come strumento per dimostrare l'appartenenza a spazi O. Lo stesso succede per la proprietà di *approssimazione asintotica*: quando il limite del rapporto $T(n)/f(n)$ è pari a 1, allora $T(n) \in \Theta(n)$, ma ancora una volta la condizione è solo sufficiente. Infine, quando il limite del rapporto è $+\infty$, questo è condizione sufficiente per l'appartenenza di $T(n)$ a $\Omega(f(n))$. Un modo di ricordare queste proprietà e il fatto che questi spazi sono più ristretti dei precedenti è di far corrispondere gli spazi funzionali agli operatori di disuguaglianza stretta: lo spazio funzionale $o(\cdot)$ all'operatore $<$ e lo spazio funzionale $\omega(\cdot)$ all'operatore $>$. Non esistendo un'"uguaglianza stretta", la corrispondenza con Θ viene un po' a mancare.

Ora consideriamo alcuni principi fondamentali nel manipolare questi spazi fun-

Un'altra terna di relazioni fra funzioni si basa sul concetto di limite

- $T(n) \in o(f(n))$ quando $\lim_{n\to+\infty} \frac{T(n)}{f(n)} = 0$
- $T(n) \sim f(n)$ quando $\lim_{n\to+\infty} \frac{T(n)}{f(n)} = 1$
- $T(n) \in \omega(f(n))$ quando $\lim_{n\to+\infty} \frac{T(n)}{f(n)} = +\infty$

Queste relazioni sono legate alle precedenti, ma sono più restrittive

- $T(n) \in o(f(n)) \Rightarrow T(n) \in O(f(n))$
- $T(n) \sim f(n) \Rightarrow T(n) \in \Theta(f(n))$
- $T(n) \in \omega(f(n)) \Rightarrow T(n) \in \Omega(f(n))$

Per dimostrarlo, basta applicare la definizione di limite

$$T(n) \in O(f(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} : T(n) \leq c\, f(n) \text{ for all } n \geq n_0$$
$$T(n) \in o(f(n)) \Leftrightarrow \forall \epsilon \in \mathbb{R}^+, \exists n_\epsilon \in \mathbb{N} : T(n) \leq \epsilon\, f(n) \text{ for all } n \geq n_\epsilon$$

Sono facili da ricordare, associando mentalmente o a $<$ e ω a $>$

Figura 3.17: Complessità asintotica e limiti

zionali (vedi Figura 3.18). Per prima cosa, se si moltiplica una funzione per un fattore costante positivo[4], la nuova funzione rimane nello stesso spazio funzionale e conserva tutte le relazioni della vecchia con qualsiasi altra funzione.

Lo stesso succede se si somma o si sottrae alla funzione un termine "dominato", cioè un termine che sia nello spazio $o(\cdot)$ della funzione di partenza. Per esempio, se $f \in \Theta(h(n))$ e si considera $g(n) \in o(f(n))$, sommare o sottrarre $g(n)$ a $f(n)$, anche moltiplicato per una costante c qualsiasi, produce un risultato che rimane in $\Theta(h(n))$. In breve, perturbazioni moltiplicative o additive dominate non fanno uscire dallo spazio funzionale corrente. Queste proprietà, ovviamente, non valgono per gli operatori di uguaglianza e disuguaglianza fra numeri, quindi la corrispondenza non è completa.

Il grosso vantaggio di questi principi sta nel fatto di poter usare come funzione rappresentativa del tempo o spazio di calcolo, la funzione il cui spazio O, Θ o Ω si considera, una funzione di riferimento molto semplice. Le useremo come etichette per classificare le funzioni originali, e quindi gli algoritmi.

Dai due principi sopra enunciati, derivano immediatamente due fondamentali semplificazioni che si applicano nell'analisi degli algoritmi (vedi Figura 3.19). Siccome i fattori moltiplicativi si possono ignorare, la base di qualsiasi termine logaritmico è irrilevante, dato che cambiare la base significa solo moltiplicare il valore per una costante. Infatti, parleremo sempre di algoritmi di complessità $\Theta(nlogn)$, senza specificare la base del logaritmo. Inoltre, qualsiasi polinomio è sostituibile con il suo termine di grado massimo, cancellando tutti quelli di grado inferiore, nonché il coefficiente, riducendo quindi il polinomio a una semplice potenza. Un ultimo aspet-

[4]Tutte le funzioni di cui stiamo ragionando sono tipicamente a valori positivi, dato che descrivono il tempo e lo spazio richiesti da un algoritmo per eseguire una computazione.

I fattori moltiplicativi costanti non sono significativi

- $f(n) \in O(h(n)) \Leftrightarrow c\, f(n) \in O(h(n))$ per ogni $c \in \mathbb{R}^+$
- $f(n) \in \Theta(h(n)) \Leftrightarrow c\, f(n) \in \Theta(h(n))$ per ogni $c \in \mathbb{R}^+$
- $f(n) \in \Omega(h(n)) \Leftrightarrow c\, f(n) \in \Omega(h(n))$ per ogni $c \in \mathbb{R}^+$

Aggiungere o sottrarre termini dominati non ha effetti significativi

- $f(n) \in O(h(n)) \Leftrightarrow (f(n) + c\, g(n)) \in O(h(n))$
- $f(n) \in \Theta(h(n)) \Leftrightarrow (f(n) + c\, g(n)) \in \Theta(h(n))$
- $f(n) \in \Omega(h(n)) \Leftrightarrow (f(n) + c\, g(n)) \in \Omega(h(n))$

per ogni $c \in \mathbb{R}$ e per ogni $g(n) \in o(f(n))$

Queste proprietà

- non valgono per le corrispondenti relazioni fra numeri
- consentono di usare approssimanti semplici per classificare le funzioni

Figura 3.18: Principi di sostituzione

to rilevanteè che tutte le funzioni limitate appartengono a $\Theta(1)$, e quindi qualsiasi arrotondamento all'intero si può tranquillamente applicare, dato che corrisponde a sommare o sottrarre un termine in $\Theta(1)$, dominato da qualsiasi altro termine eventualmente esistente. Questo principio verrà applicato molto spesso durante il corso: in particolare dividendo il valore della funzione per 2, anche se il risultato deve essere intero (perché esprime un numero di operazioni o di celle di memoria), spesso si ignora se l'arrotondamento sia per eccesso o per difetto, dato che si tratta comunque di sottrarre o sommare un termine pari a 0.5, trascurabile rispetto alla componente principale della funzione che si sta valutando. Fanno ovviamente eccezione gli arrotondamenti a zero, dato che allora il termine sommato o sottratto diventa tutt'altro che trascurabile. Ma funzioni che convergono asintoticamente a zero sono decisamente rare nel caso dell'analisi di algoritmi, dato che in genere un algoritmo ha complessità crescente con la dimensione.

A questo punto è possibile elencare le tipiche funzioni che fanno da etichetta per gli spazi funzionali usati nell'analisi di algoritmi (vedi Figura 3.20). In genere, qualunque algoritmo si può classificare in uno spazio rappresentato da una di queste funzioni, o da funzioni derivate moltiplicandole fra loro. Sono tre famiglie:

1. le funzioni *polilogaritmiche*, cioè le potenze di logaritmi di n[5]

$$(\log n)^r \qquad \text{con } r \geq 0;$$

2. le *potenze* di n

$$n^r \qquad \text{con } r > 0;$$

[5]Sfortunatamente, in matematica il termine "polilogaritmo" viene usato anche per indicare una generalizzazione del concetto di logaritmo basato su serie di potenze. Questa accezione non ha niente a che vedere con l'uso del termine nello studio degli algoritmi.

Dai principii di sostituzione deriva che

- i fattori moltiplicativi e la base dei logaritmi si possono ignorare

$$\log_b f(n) = \frac{\log_a f(n)}{\log_b a}$$

- di un polinomio si può considerare solo il termine direttore

$$\left(c_r n^r + c_{r-1} n^{r-1} + \ldots + c_1 n + c_0\right) \in \Theta\left(n^r\right)$$

Inoltre, per le funzioni che non convergono a 0

- gli arrotondamenti all'intero si possono ignorare
- le funzioni limitate appartengono tutte a $\Theta(1)$

Figura 3.19: Approssimazioni asintotiche

3. le *funzioni esponenziali* di n

$$r^n \qquad \text{con } r > 1.$$

Va notato che tutte le funzioni approssimabili con un polilogaritmo sono nello spazio $o(\cdot)$ di tutte le funzioni approssimabili con una potenza, e tutte le funzioni approssimabili con una potenza sono nello spazio $o(\cdot)$ di tutte le funzioni approssimabili con un'esponenziale, cioè ogni algoritmo polilogaritmico è strettamente migliore di ogni algoritmo polinomiale (approssimabile con una potenza) e qualunque algoritmo polinomiale è strettamente migliore di ogni algoritmo esponenziale. All'interno di ciascuna delle tre classi, le funzioni sono ordinate in base al parametro numerico: gli algoritmi migliorano al calare dell'esponente per i polilogaritmi e per le potenze, e migliorano al calare della base per le esponenziali. Tutto questo fa pensare che ci sia un ordinamento totale di approssimanti nel quale si possa incasellare qualsiasi algoritmo. Questo è vero se si riesce a classificare l'algoritmo con una di queste approssimanti, o anche con un prodotto di più approssimamenti: allora si possono ordinare gli algoritmi in maniera debole, cioè considerando equivalenti (nel senso già discusso relativo allo spazio $\Theta(\cdot)$) due algoritmi eventualmente diversi. Se l'algoritmo non cade in alcuna di queste classi, elementari o derivate, dipende dai casi.

La Figura 3.21 riassume l'intero procedimento di definizione della complessità asintotica nel caso pessimo, che serve a dare una misura del tempo o dello spazio di calcolo di un algoritmo. Per prima cosa, si misura il tempo con il numero di operazioni elementari eseguite e lo spazio con il numero di unità elementari di memoria (avendo scelto un modello computazionale ben preciso, dal quale tali numeri in generale dipendono). In questo modo, evitiamo ogni dipendenza della misura dalla tecnologia specifica adottata. Quindi, abbiamo definito per ogni istanza una misura di dimensione, attraverso un numero intero n. La definizione che useremo sarà più operativa di quella teorica (numero di simboli della codifica dell'istanza in un opportuno alfabeto), cioè il numero di elementi di un opportuno insieme che caratterizza il problema (elementi di un vettore, colonne di una matrice, nodi o archi di un grafo), ed eventualmente composta da più valori, anziché uno solo. Dopo di che, per ogni valore di n determiniamo il valore massimo o medio (questo in base a

Le approssimanti più usate per la complessità $T(n)$ di un algoritmo sono

- polilogaritmiche: $(\log n)^r$ con $r \geq 0$
- potenze: n^r con $r > 0$
- esponenziali: r^n con $r > 1$

oppure prodotti di tali approssimanti

Le approssimanti fondamentali si dominano in ordine lessicografico stretto

$$(\log n)^r \in o\big(n^s\big) \quad \text{per ogni } r \geq 0 \text{ e } s > 0$$
$$n^r \in o\,(s^n) \quad \text{per ogni } r > 0 \text{ e } s > 1$$

Ovviamente le funzioni con esponenti o basi più grandi dominano le altre

Gli algoritmi con queste approssimanti sono debolmente ordinati
(*quindi, non tutti gli algoritmi, ma quasi*)

Figura 3.20: Approssimanti di uso comune

un'opportuna distribuzione di probabilità) su tutte le istanze che hanno dimensione n, e lo consideriamo come il tempo di calcolo di riferimento, rispettivamente nel caso pessimo e nel caso medio. A questo punto, la complessità è una funzione che associa ad ogni dimensione un numero intero. Questa funzione va approssimata con una funzione più semplice, che in generale sarà un polilogaritmo, una potenza, un'esponenziale, o un prodotto di funzioni di queste famiglie, della quale non interessa l'andamento su tutto l'insieme dei numeri naturali, ma solo per valori grandi, in un intorno dell'infinito, cioè un intervallo $[n_0, +\infty)$, dove n_0 può essere definito a piacere. Questo significa che interessa sostanzialmente l'efficienza dell'algoritmo per istanze di grandi dimensioni. A questo punto, algoritmi che stanno in classi diverse sono in generale confrontabili, cioè uno è meglio dell'altro, e algoritmi che stanno nella stessa classe sono equivalenti, a meno di una costante moltiplicativa indeterminata.

3.2 Algoritmi iterativi e sommatorie

Una classe di algoritmi particolarmente importante pone un problema matematico di base nella valutazione di complessità. Lo affrontiamo ora in termini del tutto generali, prima di considerare esempi specifici nel seguito del corso. Un *algoritmo iterativo* (vedi Figura 3.22) è un algoritmo che ripete un blocco di operazioni su dati diversi più e più volte, fino a che non si verifica una condizione di termine. Si parla allora di *ciclo*, e daremo per scontata l'esistenza di cicli di diverso genere: a conteggio, a condizione iniziale e a condizione finale. Nei cicli a conteggio (spesso indicati come *cicli for*), il numero di iterazioni del ciclo è noto. Nei cicli a condizione iniziale (spesso indicati come *cicli while*), si verifica prima una condizione: se la condizione è soddisfatta, si eseguono le operazioni iterativamente finché la condizione non diventa violata. Infine, nei cicli a condizione finale, le operazioni vengono eseguite prima e la condizione valutata dopo, abbandonando il ciclo se essa è rispettata (nei *repeat ... until*) o violata (nei *do ... while*). I cicli a condizione iniziale o finale non hanno un conteggio esplicito delle iterazioni, ma è in generale possibile

La complessità asintotica di un algoritmo nel caso pessimo fornisce una misura del tempo di calcolo dell'algoritmo attraverso i seguenti passaggi

1. misuriamo il tempo col numero T di operazioni elementari eseguite
   (così la misura è indipendente dallo specifico meccanismo usato)
2. scegliamo un valore n che misuri la dimensione di un'istanza (per es., il numero di elementi dell'insieme, di righe o colonne della matrice, di nodi o archi del grafo)
3. troviamo il tempo di calcolo massimo o medio per ogni dimensione n

$$T_A(n) = \frac{\sum_{I \in \mathcal{I}_P^{(n)}} T_A(I)}{|\mathcal{I}_P^{(n)}|} \text{ o } T(n) = \max_{I \in \mathcal{I}_P^{(n)}} T(I) \qquad n \in \mathbb{N}$$

   (questo riduce la complessità a una funzione $T : \mathbb{N} \to \mathbb{N}$)
4. approssimiamo $T(n)$ con una funzione $f(n)$ più semplice,
   di cui interessa solo l'andamento per $n \to +\infty$
   (è più importante che l'algoritmo sia efficiente su dimensioni grandi)
5. raccogliamo le funzioni in classi con la stessa approssimante semplice
   (la relazione di approssimazione è una relazione di equivalenza)

Figura 3.21: Sommario

stimare (almeno per eccesso e difetto) il numero di iterazioni stesse, in maniera da poter impostare l'analisi di complessità. Si tratta di studiare le operazioni interne per dedurre dopo quante iterazioni la condizione di permanenza nel ciclo risulta certamente violata.

Il problema che affronteremo ora è come ottenere la complessità temporale dell'intero ciclo a partire da quella delle operazioni interno al ciclo (il così detto *corpo* del ciclo). Quindi, per semplicità ridurremo l'algoritmo a un semplice ciclo (in generale, bisognerà aggiungere al risultato le operazioni che lo precedono e lo seguono) e ipotizzeremo di conoscere esattamente il numero di iterazioni, eventualmente usando una stima.

Nel seguito c'è un problema di notazione. Ipotizzeremo che il ciclo da studiare abbia un indice che cresce da 1 a n. È chiaro che in generale, i valori estremi dell'indice potrebbero essere diversi, e che l'indice potrebbe decrescere, anziché crescere. Si tratta semplicemente di introdurre un indice ausiliario in modo da ricadere nella forma standard che useremo. Il problema di notazione sta nel fatto che nell'intero corso usiamo n per denotare la dimensione (o almeno uno dei termini di dimensione) dell'istanza, ma in questa sezione n denota il numero di iterazioni del ciclo. Spesso i due valori coincidono, ma non sempre. D'altra parte, introdurre una notazione più rigorosa le distanzierebbe troppo da quelle del modulo di teoria e complicherebbe la notazione, potenzialmente introducendo confusione. Quindi, accettiamo l'incoerenza.

Il corpo del ciclo consiste di operazioni, che ovviamente dipendono dall'istanza, ma in generale possono cambiare da iterazione a iterazione, in base ai risultati parziali delle iterazioni precedenti. Per esempio, molti algoritmi che ordinano un insieme di oggetti, procedono ordinando prima un sottoinsieme costituito da un solo oggetto, e poi allargando il sottoinsieme in modo che resti ordinato, finché

Un algoritmo iterativo ripete le stesse operazioni più volte su dati diversi

For $i := 1$ to n do

`Operazioni`(i, I)

Sia $f(i)$ la complessità di `Operazioni` al passo i

(*in generale dipende anche da* $|I|$*, ma qui semplifichiamo*)

La complessità dell'intero ciclo è

$$F(n) = \sum_{i=1}^{n} f(i)$$

Come approssimarla asintoticamente rispetto al numero di iterazioni n?

Figura 3.22: Algoritmi iterativi e sommatorie

non coincide con l'intero insieme. Questo significa che nelle diverse iterazioni l'algoritmo considera via via sottoinsiemi crescenti, e la complessità delle operazioni può crescere in corrispondenza. Per questo lo pseudocodice nella Figura 3.22 indica che `Operazioni` dipende dai dati, cioè dall'istanza `I`, ma anche dall'indice `i` dell'iterazione corrente.

Denotiamo con $f(i)$ la complessità delle operazioni eseguite all'interno del ciclo all'iterazione i-esima. Questa funzione in generale dipende anche dalla dimensione dell'istanza, ma lasciamo questa dipendenza implicita per evitare di introdurre un termine che in tutto il corso chiamiamo n, ma in questa sezione non possiamo chiamare così per evitare di confonderlo col numero totale di iterazioni.

Quando un algoritmo esegue una sequenza di operazioni, la complessità temporale delle singole operazioni si somma. Un ciclo è una sequenza di operazioni, anche se indicata in modo compatto, quindi la sua complessità è la somma di quelle delle singole iterazioni[6]. Di conseguenza, la complessità $F(n)$ di un ciclo può essere descritta come una sommatoria con un indice i che va da 1 a n della complessità $f(i)$ delle operazioni eseguite all'iterazione i-esima. La complessità temporale dell'intero algoritmo, $T(n)$, contiene gli eventuali termini aggiuntivi per le operazioni che precedono o seguono il ciclo stesso.

Tutto questo pone la domanda fondamentale: come si valuta una sommatoria? Questo è un oggetto di studio classico dell'analisi matematica, ma ricordiamo che non ci interessa la somma esatta, bensì una stima asintotica nel caso pessimo, il che consente una serie di semplificazioni. La prima è descritta nel teorema enunciato nella Figura 3.23, e dimostrato nelle dispense di teoria (pagina 20). Secondo questo teorema, è sempre possibile sostituire la complessità esatta delle operazioni nel corpo del ciclo con una loro approssimante, ottenendo un'approssimante corretta del risultato finale della sommatoria, cioè della complessità dell'intero algoritmo. Per esempio, se $f(i) = 3i^2 + 2i + \log i + 5$, ci si può limitare ad approssimarla con $g(i) = i^2$, eliminando i termini dominati e le costanti moltiplicative, e usare

[6] Una parentesi ovvia, ma importante, e fonte di errori abbastanza frequenti negli esami: la complessità temporale si somma; per quella spaziale, invece, quando le celle di memoria usate in passi diversi sono le stesse, non si somma, ma si considera l'occupazione massima. Ovviamente, se ad ogni iterazione si usano celle diverse, invece si somma, ma di solito usare celle diverse ad ogni passo è uno spreco.

$\sum_{i=1}^{n} i^2$ come approssimante corretta dell'intero ciclo. Questo è un enorme vantaggio, perché, siccome in generale le operazioni nel corpo del ciclo saranno approssimate da polilogaritmi, potenze, esponenziali e loro prodotti, studiare le sommatorie di polilogaritmi, potenze, esponenziali e loro prodotti risolve quasi tutti i problemi posti da algoritmi iterativi.

Teorema: si possono sostituire gli addendi con un'approssimante

$$f(i) \in \Theta(g(i)) \Rightarrow \sum_{i=1}^{n} f(i) \in \Theta\left(\sum_{i=1}^{n} g(i)\right)$$

Dimostrazione: *si veda a pag. 20 delle dispense*

Quindi studiamo le sommatorie di approssimanti fondamentali

Figura 3.23: Teorema fondamentale

Prima di considerare il caso specifico delle tre famiglie di approssimanti principali, consideriamo però alcune proprietà e metodi che valgono in generale.

3.2.1 Minorazioni e maggiorazioni di sommatorie

Siccome $f(i)$ non è una generica funzione a valori naturali di un numero naturale, possiamo introdurre alcune ipotesi e dedurne delle proprietà utili (vedi Figura 3.24). Si tratta del numero di passi elementari compiuti in un'iterazione del ciclo, quindi certamente un numero non negativo. Probabilmente (questo è meno certo in generale) si tratta di una funzione non decrescente: di solito l'algoritmo maneggia oggetti via via sempre più grandi e complicati, e quindi compie un numero di passi uguale o superiore[7]

Sotto queste ipotesi, è facile dimostrare che l'intera sommatoria si può banalmente minorare con uno qualsiasi dei suoi termini, in particolare il massimo, che è anche l'ultimo, cioè $f(n)$. D'altra parte, altrettanto banalmente si può maggiorare con il termine massimo moltiplicato per il loro numero, cioè $nf(n)$. Quindi, la complessità dell'intero ciclo è compresa fra quella dell'ultima iterazione e quella dell'ultima iterazione moltiplicata per il numero di iterazioni.

$$F(n) \in \Omega(f(n)) \qquad F(n) \in O(n f(n))$$

Questo fissa un intervallo di complessità ancora grossolano, ma piuttosto preciso. Vedremo che funzioni $f(i)$ diverse cadono più vicine a uno o all'altro estremo di questo intervallo: le funzioni esponenziali cadono sul primo estremo, perché crescono talmente in fretta che l'ultima iterazione soverchia le precedenti; le funzioni polilogaritmiche e polinomiali cadono sul secondo perché crescono abbastanza lentamente da consentire a ciascuno dei termini della sommatoria di contribuire con il proprio peso in modo sensibile e paragonabile a quello dell'ultimo e massimo.

[7] Si potrebbe anche osservare che una somma finita può essere eseguita in qualsiasi ordine, e quindi si può imporre la proprietà di monotonia. Va detto che cambiando l'ordine si perde quasi certamente l'approssimazione con polilogaritmi, potenze o esponenziali, ma questa proprietà prescinde da quella. Comunque, algoritmi che accelerano via via sono decisamente rari.

Vogliamo trovare un'espressione asintotica per la complessità

$$F(n) = \sum_{i=1}^{n} f(i)$$

Quasi sempre è possibile ipotizzare che $f(i)$ sia

1. non negativa: $f(i) \geq 0$ per ogni $i \in \mathbb{N}$
2. non decrescente: $f(i+1) \geq f(i)$ per ogni $i \in \mathbb{N}$

Sotto queste ipotesi valgono le banali stime per difetto e per eccesso:

$$f(n) \leq F(n) \leq n\, f(n)$$

e quindi

$$F(n) \in \Omega(f(n)) \qquad F(n) \in O(n\, f(n))$$

Tipicamente

- le funzioni esponenziali cadono in $\Theta(f(n))$
- le potenze e le funzioni polilogaritmiche cadono in $\Theta(n\, f(n))$

Esistono tecniche per ottenere stime più precise

Figura 3.24: Stima mediante minorazione e maggiorazione

3.2.2 Sommatorie di esponenziali

Per le sommatorie di funzioni esponenziali esiste una nota proprietà, detta della *somma geometrica*. La Figura 3.25 la illustra anche in forma grafica: un quadrato di lato 1 è accostato a un rettangolo di area doppia (2), che sta a fianco a un quadrato di area doppia (4), e così via. Disponendoli in modo intelligente, questi quadrati e rettangoli disegnare ad ogni passo delle figure molto vicine a un quadrato o a un rettangolo, a cui manca solo un quadrato di lato 1, in basso a sinistra. Tale quadrato o rettangolo ha un'area pari al doppio dell'ultima componente. Questo mostra plasticamente la proprietà di una sommatoria approssimabile con il proprio ultimo termine, semplicemente moltiplicato per una costante.

La dimostrazione algebrica fornisce un'espressione precisa della somma, che dal punto di vista della complessità asintotica non è importante, e si può tranquillamente sostituire con la proprietà

$$F(n) \in \Theta(f(n))$$

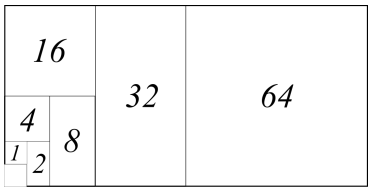
Questo risolve anche la piccola tecnicalità che la sommatoria viene valutata da 0 a n anziché da 1 a n: semplicemente andrebbe sottratto un termine costante, pari a 1, che non cambia l'espressione asintotica. La sommatoria sostanzialmente si comporta come l'ultimo suo termine $f(n)$, salvo qualche costante da moltiplicare e da sottrarre.

Per le somme di esponenziali esiste una nota soluzione in forma chiusa

$$\sum_{i=0}^{n} r^i = \frac{r^{n+1} - 1}{r - 1} \Rightarrow F(n) \in \Theta(f(n))$$

Dimostrazione:

$$\begin{aligned} r \quad F(n) &= r \quad + \quad r^2 + \ldots + r^n \quad + r^{n+1} \\ F(n) &= 1 \quad + \quad r \quad + \ldots + r^{n-1} + r^n \\ (r-1)\,F(n) &= r^{n+1} - 1 \end{aligned}$$



$$\sum_{i=0}^{6} 2^i = 2^7 - 1$$

Figura 3.25: Approssimanti esponenziali: somma geometrica

3.2.3 Sommatorie di potenze e stime mediante integrali

Un'altra tipica sommatoria trattata nei corsi di base di matematica è la *somma aritmetica*. La Figura 3.26 illustra il celebre aneddoto di Gauss bambino che alle scuole elementari scoprì come ridurre la somma dei primi n numeri interi alla metà del prodotto di n per $n + 1$. Questo valore è molto vicino alla metà dell'ultimo valore moltiplicato per il numero dei valori, cioè ricadiamo nella classe di funzioni

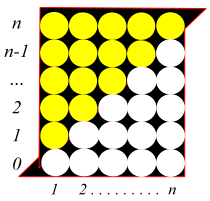
che crescono abbastanza lentamente da rendere tutti i termini degni di nota, e quindi la sommatoria approssimabile con $nf(n)$.

Per le somme di funzioni lineari, esiste una nota soluzione in forma chiusa

$$F(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \Rightarrow F(n) \in \Theta(nf(n))$$

Dimostrazione:

$$\begin{aligned} 2F(n) = \quad & 1 \quad + \quad 2 \quad + \ldots + (n-1) + \quad n \quad + \\ & n \quad + (n-1) + \ldots + \quad 2 \quad + \quad 1 \quad = \\ & (n+1) + (n+1) + \ldots + (n+1) + (n+1) = n(n+1) \end{aligned}$$



$$\sum_{i=1}^{5} i = \frac{5(5+1)}{2}$$

Per le potenze, c'è un'espressione esatta, ma anche una stima semplice

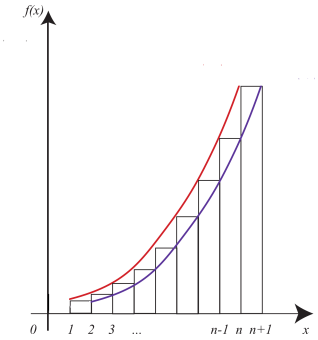
Figura 3.26: Approssimante lineare: somma aritmetica

Per le altre potenze, esistono espressioni esatte, ma dal punto di vista dell'analisi di complessità sono poco interessanti. Per queste funzioni, otterremo espressioni approssimate con un metodo che ha un certo interesse in sé, dato che è generale, cioè l'uso di integrali di funzioni continue (vedi Figura 3.27). Si tratta di passare dalla funzione $f(i)$, di variabile naturale e a valori naturali, alla funzione $f(x)$, di variabile reale e a valori reali. Se si ha un'espressione algebrica della complessità, questa estensione è banale. Nella Figura 3.27, il diagramma a gradini rappresenta la funzione $f(i)$, che assume un valore per ogni numero intero, e poi lo conserva fino all'intero successivo. Ci sono quindi n gradini. La stessa funzione $f(\cdot)$, vista come funzione di variabile reale ha invece il grafico in rosso, che in corrispondenza ai valori interi coincide col diagramma a gradini, ma nei valori successivi cresce in modo continuo fino a raggiungere il valore seguente. La funzione rappresentata in blu, infine, è semplicemente la funzione rossa traslata un passo in avanti: traslare una funzione significa sostanzialmente calcolarla in un valore modificato del suo argomento. In particolare, traslare avanti di 1 la funzione $f(x)$ significa calcolare $f(x-1)$: la funzione blu in un punto intero ha lo stesso valore della funzione rossa nel punto intero precedente. Per funzioni monotone non decrescenti (per quelle monotone non decrescenti vale una proprietà simile, con opportune modifiche minori), esistono interessanti relazioni fra le aree sottese dai grafici delle tre funzioni.

Per la funzione nera, $f(i)$, l'area sottesa dai gradini fra 1 e $n+1$ è semplicemente la somma di valori $f(i)$ per i crescente da 1 a n, dato che si tratta di rettangoli di base 1 e altezza $f(i)$. Questa è la somma che ci interessa valutare. Consideriamo ora la funzione rossa: essa è sempre sopra la funzione nera (nell'ipotesi che $f(x)$ sia monotona non decrescente). Arrivati a $x = n$, anziché proseguire, supponiamo di mantere costante il valore fino a $n+1$. L'area sottesa è pari all'integrale di $f(x)$ da 1 a n più l'ultimo gradino, che ha un'area pari a $f(n)$. Questa è una sovrastima dell'area $F(n)$ sottesa dalla funzione nera. Per la sottostima, procediamo analoga-

mente, considerando l'area sottesa dalla funzione blu fra 2 e $n+1$, più l'area del primo gradino, pari a $f(1)$, come se la funzione blu rimanesse costante fra 1 e 2 al valore $f(1)$. Quest'area è pari all'integrale di $f(x-1)$ da 2 a $n-1$ più $f(1)$. Con un semplice cambio di variabile, si verifica che i due integrali coincidono, e quindi le due stime differiscono solo per l'uso di $f(1)$ anziché $f(n)$. Se la funzione $f(x)$ è integrabile, si ottiene una stima per difetto e una per eccesso, che sono fin troppo precise per gli scopi dell'analisi algoritmica. Per funzioni non crescenti, vale lo stesso, semplicemente scambiando le due stime: quella per difetto diventa per eccesso, e viceversa.

Sia $f(x)$ una funzione non decrescente



La somma è pari all'area sottesa dal grafico a scaletta

$$F(n) = \sum_{i=1}^{n} f(i)$$

ed è compresa fra le aree sottese dai grafici di $f(x)$ e $f(x-1)$

$$f(1) + \int_2^{n+1} f(x-1)\,dx \leq F(n) \leq \int_1^n f(x)\,dx + f(n)$$

$$f(1) + \int_1^n f(x)\,dx \leq F(n) \leq \int_1^n f(x)\,dx + f(n)$$

Per le funzioni non crescenti, basta scambiare $f(1)$ e $f(n)$

Figura 3.27: Stima mediante integrali

Ora applichiamo questa stima alle potenze generiche i^r con $r \geq 1$ (vedi Figura 3.28). Il risultato mostra che entrambe le stime ricadono in $\Theta(nf(n))$, e quindi (basta pensare alla definizione dello spazio funzionale $\Theta(\cdot)$) anche l'incognita funzione sommatoria vi ricade. Di conseguenza, tutte le potenze, e tutti i polinomi, hanno sommatorie approssimabili con n volte l'ultimo termine, dato che crescono non troppo velocemente.

3.2.4 Sommatorie di polilogaritmi e stime mediante decomposizione

Restano da valutare le sommatorie di polilogaritmi, per le quali useremo un metodo valido anche per casi più generali quello di decomposizione. L'idea è dividere la sommatoria in due parti, associate a valori piccoli e a valori grandi dell'indice i, cioè nella complessità delle prime interazioni e in quella delle ultime. Si può poi approssimare separatamente le due parti.

La Figura 3.29 illustra l'applicazione a un generico polilogaritmo $f(i) = (\log i)^r$ con $r \geq 1$. Dividiamo la sommatoria complessiva nella sommatoria dei suoi termini

$$f(i) = i^r \qquad \text{e} \qquad F(n) = \sum_{i=1}^{n} i^r$$

La stima con l'integrale è

$$1^r + \int_1^n x^r dx \leq F(n) \leq \int_1^n x^r dx + n^r$$

da cui

$$1 + \frac{n^{r+1} - 1}{r+1} \leq F(n) \leq \frac{n^{r+1} - 1}{r+1} + n^r$$

e quindi $F(n) \in \Theta\left(n^{r+1}\right)$

Per le somme di potenze, $F(n) \in \Theta(nf(n))$ come nel caso lineare

Figura 3.28: Approssimante polinomiale

da 1 a $n/2$[8] e in quella da $n/2+1$ a n. Poi approssimiamo separatamente i due termini. Ci concentriamo sull'approssimazione per difetto, dato che già sappiamo che $nf(n)$ fornisce un'approssimazione per eccesso. Il nostro scopo è mostrare che questa è anche un'approssimazione per difetto, salvo costanti moltiplicative. Per farlo, ci limitiamo a osservare che il primo termine di ogni sommatoria è il minimo, e quindi l'intera sommatoria è almeno pari al numero di termini moltiplicato per il primo. Dalla prima sommatoria non otteniamo nulla di utile, dato che il primo termine è addirittura nullo. La seconda sommatoria, invece, è più interessante, e lo scopo della decomposizione era proprio quello di liberarsi dei termini meno utili per concentrarsi sui più utili. Ciascuno dei suoi termini è almeno pari al primo, che è $f(n/2+1) > f(n) = (\log(n/2))^r$. Il loro numero è $n/2$. Quindi, si ottiene una stima per difetto, che si può ulteriormente manipolare sottostimando $\log(n/2)$ con $(\log n)/2$ (disuguaglianza valida per n sufficientemente grande). Ne deriva un'espressione finalmente descrivibile come $nf(n)$ moltiplicata per una costante. La costante potrebbe essere molto piccola, ma è costante, indipendente da n, e quindi basta a dimostrare l'appartenenza allo spazio funzionale $\Omega(nf(n))$.

3.2.5 Prodotti di approssimanti

L'ultimo argomento da affrontare (Figura 3.30) riguarda le funzioni ottenute moltiplicando polilogaritmi, potenze ed esponenziali. Visto quanto sopra, in realtà è facile intuire il risultato. Qualunque prodotto contenga un'esponenziale sarà dominato da questa (gli altri termini non faranno che aumentare ulteriormente la velocità di crescita), e quindi il risultato ricadrà in $\Theta(f(n))$. Se invece non ci sono esponenziali, il meccanismo della decomposizione si può applicare come già visto, portando alla stessa conclusione: che la somma ricade in $\Theta(nf(n))$.

[8] Ignoriamo per semplicità gli arrotondamenti, che non portano a nessuna differenza nel risultato approssimato finale.

Le stime per difetto e per eccesso si possono raffinare

- decomponendo la somma
- approssimando separatamente le singole parti

Applichiamo la tecnica raffinata alle funzioni polilogaritmiche:

$$F(n) = \sum_{i=1}^{n} (\log i)^r = \sum_{i=1}^{n/2} (\log i)^r + \sum_{i=n/2+1}^{n} (\log i)^r \geq$$
$$\geq \frac{n}{2} (\log 1)^r + \frac{n}{2} \left(\log \frac{n}{2}\right)^r$$

Siccome per n abbastanza grande vale $\log \frac{n}{2} > \frac{\log n}{2}$:

$$F(n) \quad > \quad 0 \quad + \frac{n}{2} \frac{(\log n)^r}{2^r}$$

Quindi $F(n) \in \Omega\left(n (\log n)^r\right)$ e siccome in generale $F(n) \in O\left(n (\log n)^r\right)$

$F(n) \in \Theta\left(n (\log n)^r\right)$, cioè ancora $F(n) \in \Theta(nf(n))$

Figura 3.29: Stima mediante decomposizione

In generale

- se $f(i)$ contiene esponenziali, si maggiorano gli altri singoli termini

  $$F(n) = \sum_{i=1}^{n} \left(r^i i^a (\log i)^b\right) \leq \sum_{i=1}^{n} \left(r^i n^a (\log n)^b\right) = n^a (\log n)^b \sum_{i=1}^{n} r^i$$

  e si conclude che $F(n) \in \Theta(f(n))$

- se $f(i)$ non contiene esponenziali, si decompone la somma

  $$F(n) = \sum_{i=1}^{n} \left(i^a (\log i)^b\right) \geq \sum_{i=n/2+1}^{n} \left(i^a (\log i)^b\right) \geq \frac{n}{2} \left(\frac{n^a}{2^a} \left(\log \frac{n}{2}\right)^b\right)$$

  e si conclude che $F(n) \in \Theta(nf(n))$

Figura 3.30: Traccia per i prodotti di approssimanti

3.3 Esercizi sulla complessità asintotica

3.3.1 Principi generali

Le dimostrazioni di complessità asintotica si possono paragonare a un gioco, nel quale

- il primo giocatore decide i valori di tre "carte" c_1, c_2 e n_0 che sono numeri fissati una volta per tutte;
- il secondo giocatore decide il valore di una "carta" n, che è funzione dei primi tre, dato che viene scelto dopo.

Consideriamo l'appartenenza a Θ, che è il caso più complesso. Vince il primo giocatore se riesce a

- giocare le tre carte $c_1 = \bar{c}_1$, $c_2 = \bar{c}_2$ e $n_0 = \bar{n}_0$
- costruire una catena di implicazioni

$$\begin{cases} c_1 = \bar{c}_1 \\ c_2 = \bar{c}_2 \\ n_0 = \bar{n}_0 \\ n \geq n_0 \end{cases} \Rightarrow \ldots \Rightarrow \bar{c}_1 g(n) \leq f(n) \leq \bar{c}_2 g(n)$$

Vince il secondo giocatore se riesce a

- giocare la carta n
- costruire una catena di implicazioni

$$n = \bar{n}(c_1, c_2, n_0) \Rightarrow \ldots \Rightarrow \begin{cases} n \geq n_0 \\ e \\ \begin{cases} f(n) < c_1 g(n) \\ \textit{oppure} \\ f(n) > c_2 g(n) \end{cases} \end{cases}$$

Si notino gli *e* e *oppure*. La seconda e terza tesi sono in alternativa: bisogna sceglierne una. La prima tesi, invece, va congiunta con una delle altre due. Per farlo, è sufficiente fissare n come massimo fra n_0 e un valore che dimostri una delle altre due condizioni.

Esercizio 1

Dimostrare che

$$f(n) = 5n^2 + n \in O\left(n^2\right)$$

Svolgimento Occorre dimostrare che

$$\exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ : \ 5n^2 + n \leq cn^2 \ \forall n \geq n_0$$

In altre parole, le nostre ipotesi sono:

1. che n_0 abbia un valore scelto da noi a piacere
2. che c abbia un valore scelto da noi a piacere
3. che sia $n \geq n_0$

e la tesi cui dobbiamo arrivare è che $f(n) = 5n^2 + n \leq cn^2$.

Si procede "indovinando" il valore di c e n_0 per tentativi basati sull'esperienza. Per aiutarsi a indovinare, si può *sostituire la tesi con un'espressione equivalente o anche più forte*, ma più semplice.

Vediamo una sostituzione con un'espressione equivalente:

$$5n^2 + n \leq cn^2 \Leftrightarrow (c-5)\, n^2 \geq n$$

ed essendo $n \geq 0$

$$(c-5)\, n^2 \geq n \Leftrightarrow (c-5)\, n \geq 1$$

Ora la tesi da dimostrare è molto più semplice. Se poniamo $c = 6$, diventa

$$(c-5)\, n \geq 1 \Leftrightarrow n \geq 1$$

e dimostrarla è banale, perché l'ipotesi garantisce $n \geq n_0$ e possiamo fissare n_0 a piacere (per esempio, $n_0 = 1$).

Riassumendo:

$$\begin{cases} c = 6 \\ n_0 = 1 \\ n \geq n_0 \end{cases} \Rightarrow \begin{cases} c = 6 \\ n \geq 1 \end{cases} \Rightarrow (c-5)\, n \geq 1 \Rightarrow (c-5)\, n^2 \geq n \Rightarrow 5n^2 + n = f(n) \leq cn^2$$

L'esperienza aiuta facendo osservare che $f(n) = 5n^2 + n$ è un polinomio, e quindi il suo comportamento è dominato dal termine di grado massimo $5n^2$. Per approssimarla asintoticamente per eccesso, basta considerare una potenza di pari grado (secondo grado), ma con un coefficiente maggiore (un generico $5+\epsilon$ va bene, ma probabilmente 6 produce calcoli più semplici). Se fosse necessaria un'approssimazione per difetto (ad esempio, dovendo dimostrare un'appartenenza a Ω), basterebbe una potenza di pari grado (secondo), ma con un coefficiente minore (ad esempio, 4).

Esercizio 2

Dimostrare che

$$f(n) = 5n^2 + n \in \Omega\left(n^2\right)$$

Svolgimento Occorre dimostrare che

$$\exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ : \ 5n^2 + n \geq cn^2 \ \forall n \geq n_0$$

Le ipotesi sono che n_0 e c abbiano valori scelti a piacere e che $n \geq n_0$ e la tesi che $f(n) = 5n^2 + n \geq cn^2$.

Sostituiamo la tesi con un'espressione più semplice, ma più forte:

$$\phi(n) = 5n^2 \geq cn^2$$

Se dimostriamo questa, infatti, automaticamente dimostriamo la prima, perché $f(n) \geq \phi(n)$ per ogni $n \in \mathbb{N}$.

Quindi la nuova tesi viene sottoposta a qualche passaggio semplificativo:

$$5n^2 \geq cn^2 \Leftrightarrow 5 \geq c$$

Dimostrarla è banale: basta porre $c = 5$. Questa volta, non è neppure necessario scegliere un valore per n_0.

Riassumendo:

$$\begin{cases} c = 5 \\ n \in \mathbb{N} \end{cases} \Rightarrow 5n^2 \geq cn^2 \Rightarrow f(n) = 5n^2 + n \geq 5n^2 \geq cn^2$$

Esercizio 3

Dimostrare che

$$f(n) = 3n^4 \in O\left(n^5\right)$$

Svolgimento Occorre dimostrare che

$$\exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ : \ 3n^4 \leq cn^5 \ \forall n \geq n_0$$

Le ipotesi sono che n_0 e c abbiano un valore scelto da noi a piacere e che $n \geq n_0$. La tesi da dimostrare è che $3n^4 \leq cn^5$.

Supponiamo di avere un po' di intuizione e di "vedere" che un buon valore per c è 3 (in realtà, essendo una potenza di ordine superiore, qualsiasi valore positivo, anche piccolissimo, andrebbe bene, ma con $c = 3$ tutto diventa semplice). Posto $c = 3$, la tesi da dimostrare si semplifica

$$3n^4 \leq cn^5 \Leftrightarrow 3n^4 \leq 3n^5$$

e ancora, essendo $n \geq 0$

$$3n^4 \leq 3n^5 \Leftrightarrow 1 \leq n$$

Ora, poiché sappiamo per ipotesi che $n \geq n_0$ e possiamo fissare n_0 a piacere, il modo migliore di dimostrare la tesi è porre $n_0 = 1$

Riassumendo:

$$\begin{cases} c = 3 \\ n_0 = 1 \\ n \geq n_0 \end{cases} \Rightarrow \begin{cases} c = 3 \\ n \geq 1 \end{cases} \Rightarrow \begin{cases} c = 3 \\ 3n^5 \geq 3n^4 \end{cases} \Rightarrow 3n^4 = f(n) \leq cn^5$$

Esercizio 4

Dimostrare che

$$f(n) = n^2 \in \Omega\left(n^2 + 5n - 6\right)$$

Svolgimento Occorre dimostrare che

$$\exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ : \; n^2 \geq c\left(n^2 + 5n - 6\right) \; \forall n \geq n_0$$

Le ipotesi sono che n_0 e c abbiano un valore scelto da noi a piacere e che $n \geq n_0$. La tesi da dimostrare è che $n^2 \geq c\left(n^2 + 5n - 6\right)$.

L'esperienza e l'intuizione suggeriscono che occorre tenere il termine di grado massimo sulla destra più basso di quello sulla sinistra, ovvero $c < 1$. Per semplicità, proviamo con $c = 1/2$. La tesi da dimostrare diventa

$$n^2 \geq c\left(n^2 + 5n - 6\right) \Leftrightarrow \frac{1}{2}n^2 - \frac{5}{2}n + 3 \geq 0 \Leftrightarrow n^2 - 5n + 6 \geq 0$$

La disequazione si risolve facilmente

$$n^2 - 5n + 6 \geq 0 \Leftrightarrow \begin{cases} n \leq 2 \\ \textit{oppure} \\ n \geq 3 \end{cases}$$

Si noti l'*oppure*: le due condizioni non devono valere entrambe (anche perché sono contraddittorie): una basta a implicare la tesi. La prima è impossibile da dimostrare, perché *non scegliamo noi* n. Possiamo però fissare $n_0 = 3$, e quindi imporre $n \geq n_0 = 3$.

Riassumendo:

$$\begin{cases} c = 1/2 \\ n_0 = 3 \\ n \geq n_0 \end{cases} \Rightarrow \begin{cases} c = 1/2 \\ n \geq 3 \end{cases} \Rightarrow \begin{cases} c = 1/2 \\ n^2 - 5n + 6 \geq 0 \end{cases} \Rightarrow$$

$$\Rightarrow \begin{cases} c = 1/2 \\ n^2 \geq \dfrac{1}{2}\left(n^2 + 5n - 6\right) \end{cases} \Rightarrow f(n) \geq c\left(n^2 + 5n - 6\right)$$

Esercizio 5

Dimostrare che

$$f(n) = n^2 \in O\left(\frac{n^2}{4} - 2\right)$$

Svolgimento

$$\exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ : \; n^2 \leq c\left(\frac{n^2}{4} - 2\right) \; \forall n \geq n_0$$

Le ipotesi sono che n_0 e c abbiano un valore scelto da noi a piacere e che $n \geq n_0$. La tesi da dimostrare è che $n^2 \leq c\left(n^2/4 - 2\right)$.

Qualsiasi valore di $c > 4$ va bene. Per semplificare i conti, poniamo $c = 8$. La tesi diventa

$$n^2 \leq c\left(n^2/4 - 2\right) \Leftrightarrow n^2 \leq 2n^2 - 16 \Leftrightarrow n^2 \geq 16 \Leftrightarrow n \leq -4 \text{ oppure } n \geq 4$$

Ovviamente, la prima condizione non è dimostrabile, mentre la seconda lo è facilmente, ponendo $n_0 = 4$.

Esercizio 6

Dimostrare che[9]

$$f(n) = n \log_2 n \in O\left(n^2\right)$$

Svolgimento

$$\exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ : \; n \log_2 n \leq cn^2 \; \forall n \geq n_0$$

Qui è più complicato semplificare la tesi, perché logaritmi e polinomi non si combinano. Si può però osservare che

$$n \log_2 n \leq cn^2 \Leftrightarrow \log_2 n \leq cn$$

L'esperienza insegna che le funzioni logaritmiche crescono sempre meno fortemente dei polinomi, per cui qualsiasi valore di c sarebbe accettabile. Per semplicità poniamo $c = 1$, cosicché la tesi diventa

$$\phi(n) = n - \log_2 n \geq 0$$

Scegliamo un valore comodo per n_0 e poi dimostriamo la tesi per ogni $n \geq n_0$. Ad esempio, poniamo $n_0 = 2$. Qualora il tentativo fallisca, proveremo con un valore più alto.

Per dimostrare la tesi, ne dimostriamo una più forte, cioè che $\phi(x) = x - \log_2 x \geq 0$ per tutti i valori *reali* $x \geq 2$.

Per mostrare che una funzione è ≥ 0 da un certo punto in poi, basta mostrare

1. che in quel punto è ≥ 0;
2. che successivamente cresce, cioè ha *derivata strettamente positiva*.

$$\begin{cases} \phi(n_0) \geq 0 \\ \phi'(x) > 0 \; \forall x \geq n_0 \end{cases}$$

Procediamo:

$$\phi(2) = 2 - \log_2 2 = 2 - 1 = 1 > 0$$
$$\phi'(c) = 1 - 1/x \geq 0 \text{ per ogni } x \geq 1$$

Riassumendo

$$\begin{cases} c = 1 \\ n_0 = 2 \\ n \geq n_0 \\ \phi(2) > 0 \\ \phi'(x) \geq 0 \text{ per ogni } x \geq 1 \end{cases} \Rightarrow \begin{cases} c = 1 \\ n \geq 2 \\ \phi(x) \geq 0 \text{ per ogni } x \geq 2 \end{cases} \Rightarrow$$

$$\Rightarrow \begin{cases} c = 1 \\ \phi(n) \geq 0 \end{cases} \Rightarrow \begin{cases} c = 1 \\ n - \log n \geq 0 \end{cases} \Rightarrow cn \geq \log_2 n \Rightarrow n \log_2 n \leq cn^2$$

[9] Si può interpretare questo esercizio come la dimostrazione del fatto che l'algoritmo *InsertionSort* è asintoticamente peggiore dell'algoritmo *MergeSort*.

Esercizio 7

Dimostrare che

$$f(n) = n^2 \in O(2^n)$$

Svolgimento

$$\exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ : \; n^2 \leq c\, 2^n \; \forall n \geq n_0$$

Anche qui è impossibile semplificare la tesi, perché esponenziali e polinomi non si combinano. L'esperienza insegna che le funzioni esponenziali crescono sempre più fortemente dei polinomi, per cui qualsiasi valore di c sarebbe accettabile. Per semplicità poniamo $c = 1$ e rafforziamo la tesi passando ai numeri reali:

$$\phi(x) = 2^x - x^2 \geq 0$$

Per tentativi, ci si rende facilmente conto che questa tesi vale per $x = 1$ e $x = 2$, ma non per $x = 3$, e torna a valere per $x = 4$. Siccome 4 è una potenza di 2 e semplifica i calcoli, poniamo $n_0 = 4$.

Per dimostrare la tesi, mostriamo che $\phi(4) > 0$ e che $\phi'(x) \geq 0$ per ogni $x \geq 4$.

$$\begin{cases} \phi(4) = 2^4 - 4^2 = 0 \geq 0 \\ \phi'(x) = 2^x \ln 2 - 2x \end{cases}$$

Ora dobbiamo dimostrare che $\phi'(x) = 2^x \ln 2 - 2x$ per ogni $x \geq 4$. Procediamo come sopra

$$\begin{cases} \phi'(4) = 2^4 \ln 2 - 2 \cdot 4 = 16 \ln 2 - 8 > 0 \\ \phi''(x) = 2^x \ln 2 \ln 2 - 2 \end{cases}$$

La seconda tesi richiede

$$\phi''(x) = 2^x \ln 2 \ln 2 - 2 \geq 0 \Leftrightarrow 2^x \geq \frac{2}{\ln 2 \ln 2} \Leftrightarrow x \geq \log_2 \left(\frac{2}{\ln 2 \ln 2} \right) = 2.05\ldots$$

e quindi per $x \geq 4$ è dimostrata.

Secondo svolgimento

$$\exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ : \; n^2 \leq c\, 2^n \; \forall n \geq n_0$$

Un modo alternativo di dimostrare la tesi è osservare che la funzione logaritmo è monotona crescente in senso stretto, per cui

$$a > b \Leftrightarrow \log a > \log b$$

qualunque sia la base del logaritmo. Ne deriva che la tesi si può riformulare come

$$n^2 \leq c\, 2^n \Leftrightarrow \log_2 n^2 \leq \log_2 (c\, 2^n) \Leftrightarrow 2 \log_2 n \leq \log_2 c + n$$

Poniamo $c = 1$ per semplicità e dimostriamo che $n \geq 2 \log_2 n$. Si può dimostrare come nell'esercizio precedente che questo è vero per $n \geq 4$, e quindi si pone $n_0 = 4$.

Esercizio 8

Dimostrare che

$$f(n) = 4n^2 \notin \Theta\left(n^3\right)$$

Svolgimento

$$\nexists c_1, c_2 > 0, n_0 \in \mathbb{N} : c_1 n^3 \leq 4n^2 \leq c_2 n^3 \ \forall n \geq n_0$$

cioè

$$\exists n\,(c_1, c_2, n_0) \in \mathbb{N} : n \geq n_0 \text{ e } \begin{cases} c_1 n^3 > 4n^2 \\ \text{oppure} \\ 4n^2 > c_2 n^3 \end{cases} \qquad \forall c_1, c_2 > 0, n_0 \in \mathbb{N}$$

Cimentiamoci nella prima dimostrazione (la seconda è ovviamente falsa). Le ipotesi sono che $n_0 \in \mathbb{N}$, $c_1 > 0$ e $c_2 > 0$ e che n abbia un valore scelto da noi a piacere *dopo la scelta di* c_1, c_2 e n_0. La tesi da dimostrare è che $c_1 n^3 > 4n^2$.

Si noti l'asimmetria: siccome la definizione di Θ richiede che c_1, c_2 e n_0 siano scelti univocamente, mentre n deve assumere valori qualsiasi, e quindi che c_1, c_2 e n_0 siano scelti prima e n dopo, n è funzione di c_1, c_2 e n_0.

Al solito, semplifichiamo la tesi. Poiché sicuramente $n \geq 0$,

$$c_1 n^3 > 4n^2 \Leftrightarrow c_1 n > 4 \Leftrightarrow n > \frac{4}{c_1}$$

D'altra parte, deve anche essere $n \geq n_0$. Quindi, ponendo

$$n = \max\left(n_0, \left\lceil \frac{4}{c_1} \right\rceil + 1\right)$$

si dimostra la tesi. Si noti il $+1$, che deriva dalla disuguaglianza *stretta* nella tesi.

Esercizio 9

Dimostrare che

$$f(n) = \sqrt{n} \notin \Theta(n)$$

Svolgimento

$$\nexists c_1, c_2 > 0, n_0 \in \mathbb{N} : c_1 n \leq \sqrt{n} \leq c_2 n \; \forall n \geq n_0$$

cioè

$$\exists n\,(c_1, c_2, n_0) \in \mathbb{N} : n \geq n_0 \text{ e } \begin{cases} \sqrt{n} < c_1 n \\ \textit{oppure} \\ \sqrt{n} > c_2 n \end{cases} \quad \forall c_1, c_2 > 0, n_0 \in \mathbb{N}$$

Si tratta di scegliere una delle due tesi alternative (ovviamente la prima) e semplificarla

$$\sqrt{n} < c_1 n \Leftrightarrow 1 < c_1 \sqrt{n} \Leftrightarrow n > \frac{1}{c_1^2}$$

È sempre possibile trovare un valore di n che superi questo limite inferiore e insieme anche n_0:

$$n = \max\left(n_0, \left\lceil \frac{1}{c_1^2} \right\rceil + 1\right)$$

Esercizio 10

Dimostrare che

$$f(n) = 2^n \notin \Omega(3^n)$$

Svolgimento

$$\nexists c > 0, n_0 \in \mathbb{N} : 2^n \geq c3^n \ \forall n \geq n_0$$

cioè

$$\exists n\,(c, n_0) \in \mathbb{N} : n \geq n_0 \text{ e } 2^n < c3^n \quad \forall c > 0, n_0 \in \mathbb{N}$$

Si tratta, come sempre, di semplificare la tesi

$$2^n < c3^n \Leftrightarrow \left(\frac{3}{2}\right)^n > \frac{1}{c} \Leftrightarrow \log_2 \left(\frac{3}{2}\right)^n > \log_2 \frac{1}{c} \Leftrightarrow n \log_2 \left(\frac{3}{2}\right) > \log_2 \frac{1}{c}$$

Poiché $3/2 > 1$, il suo logaritmo è sempre positivo e

$$n \log_2 \left(\frac{3}{2}\right) > \log_2 \frac{1}{c} \Leftrightarrow n > \frac{\log_2 \frac{1}{c}}{\log_2 \left(\frac{3}{2}\right)}$$

È sempre possibile trovare un valore di n che superi questo limite inferiore e insieme anche n_0:

$$n = \max \left(n_0, \left\lceil \frac{\log_2 \frac{1}{c}}{\log_2 \left(\frac{3}{2}\right)} \right\rceil + 1 \right)$$

Esercizio 11

- $T_1(n) = 2^{n+1} \in O(2^n)$?
- $T_2(n) = 2^{2n} \in O(2^n)$?

Svolgimento

$$T_1(n) = 2^{n+1} = 2 \cdot 2^n$$

Quindi, se $c_1 = c_2 = 2$ risulta

$$T_1(n) = 2^{n+1} = c_1 2^n = c_2 2^n \Rightarrow c_1 2^n \leq T_1(n) = 2^{n+1} \leq c_2 2^n$$

da cui $T_1(n) \in O(2^n)$.

D'altra parte

$$T_2(n) = 2^{2n} = 2^n 2^n$$

per cui $T_2(n) \notin O\left(2^{2n}\right)$. Infatti, posto

$$n = \max(n_0, \lceil \log_2 c \rceil + 1)$$

risulta $n \geq n_0$ e

$$n > \log_2 c \Rightarrow 2^n > c \Rightarrow T_2(n) = 2^{2n} > c\, 2^n$$

Esercizio 12

Perché è insensata l'affermazione "l'algoritmo ha una complessità almeno pari a $O\left(n^2\right)$" ?

Svolgimento Perché l'espressione "almeno" suggerisce che si tratti di un limite inferiore, mentre $T\left(n\right) \in O\left(n^2\right)$ significa che, per un'opportuna scelta di c e n_0, $T\left(n\right) \leq c\, n^2$ per ogni $n \geq n_0$. Quindi la complessità non ha alcun limite inferiore (potrebbe persino essere nulla).

3.4 Esercizi sulle sommatorie asintotiche

Data un'istanza di dimensione n, un algoritmo iterativo si compone di

1. un passo iniziale, di complessità $T_{in}(n)$;
2. un passo iterativo ripetuto più volte (al massimo $i_{\max}$ volte, la cui complessità $T^{(i)}(n)$ dipende in genere da n, ma anche dall'iterazione i-esima;
3. un passo finale, di complessità $T_{fin}(n)$, che negli algoritmi più semplici spesso manca.

La sua complessità totale è quindi pari a

$$F(n) = T_{in}(n) + \sum_{i=1}^{i_{\max}} T^{(i)}(n) + T_{fin}(n)$$

e per valutarla occorre conoscere $T_{in}(n)$, $i_{\max}$, $T^{(i)}(n)$ e $T_{fin}(n)$ e saper calcolare la sommatoria. Nel seguito, per semplicità e per assumere una notazione più matematica e meno legata all'applicazione algoritmica, indicheremo $T^{(i)}(n)$ con $f(i)$, $i_{\max}$ con n e $\sum_{i=1}^{n} f(i)$ con $F(n)$. Inoltre, considereremo sommatorie estese da 0, 1 o 2 a n a seconda della funzione che viene sommata. Dato che ci interessa l'andamento asintotico della soluzione, il valore costante dal quale parte la sommatoria non influisce sul risultato.

Le funzioni $f(i)$ che occorre sommare studiando la complessità di un algoritmo sono tutte non negative. Da ciò derivano le seguenti banali approssimazioni per difetto e per eccesso:

- $\sum_{i=0}^{n} f(i) \geq n \cdot \min_{i=0,\ldots,n} f(n)$
- $\sum_{i=0}^{n} f(i) \geq \max_{i=0,\ldots,n} f(n)$
- $\sum_{i=0}^{n} f(i) \leq n \cdot \max_{i=0,\ldots,n} f(n)$

Inoltre, la stragrande maggioranza di tali funzioni è monotona, spesso crescente, talvolta decrescente. Quindi assume il valore minimo e massimo per $i = 0$ e in $i = n$. Se ne deduce che per le funzioni $f(i)$ crescenti

$$f(i) \in \Omega(n) \qquad f(i) \in \Omega(f(n)) \qquad f(i) \in O(n\, f(n))$$

e per le funzioni $f(i)$ decrescenti

$$f(i) \in \Omega(1) \qquad f(i) \in \Omega(n\, f(n)) \qquad f(i) \in O(n)$$

Ora si tratta di determinare un andamento più preciso, cioè possibilmente la classe Θ di appartenenza della funzione. Una buona regola empirica è la seguente:

1. per le funzioni esponenziali crescenti $F(n) \in \Theta(f(n))$
2. per le funzioni polinomiali e logaritmiche crescenti $F(n) \in \Theta(n\, f(n))$
3. per le funzioni decrescenti più lentamente di $1/n$, $F(n) \in \Theta(n\, f(n))$
4. per le funzioni decrescenti più rapidamente di $1/n$, $F(n) \in \Theta(1)$

Questa regola suggerisce che cosa convenga cercare di dimostrare. Se vogliamo però essere certi del risultato, occorre dimostrarlo.

Non tutte le funzioni cadono nelle quattro categorie su elencate. Un esempio notissimo è la somma armonica

$$F(n) = \sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log n)$$

Altri esempi sono le funzioni che decrescono più rapidamente o più lentamente di $1/i$, ma non in modo polinomialmente più rapido o più lento (vedi l'ultimo esempio).

Esempio 1

$$F(n) = \sum_{i=0}^{n} f(i) = \sum_{i=0}^{n} 2^i i \log i$$

La presenza dell'esponenziale fa intuire che $F(n) \in \Theta(f(n))$. Che sia $F(n) \in \Omega(f(n))$ è ovvio grazie alla stima per difetto banale; rimane da dimostrare la stima per eccesso.

È sufficiente osservare che $i \log i \leq n \log n$ per ogni $i \leq n$:

$$F(n) \leq \sum_{i=0}^{n} 2^i n \log n = n \log n \sum_{i=0}^{n} 2^i = \left(2^{n+1} - 1\right) n \log n$$

e trascurando i termini dominati e la costante moltiplicativa 2:

$$F(n) = \Theta(2^n n \log n)$$

Esempio 2

$$F(n) = \sum_{i=2}^{n} f(i) = \sum_{i=2}^{n} \frac{3^i}{i \log i}$$

La funzione è rapidamente crescente, a causa del termine esponenziale. Quindi, cercheremo sempre di dimostrare una stima per eccesso in $O(f(n))$. Tuttavia, questa volta il termine moltiplicativo polinomale è decrescente; la maggiorazione è leggermente più complicata e richiede una decomposizione.

$$F(n) = \sum_{i=2}^{n/2-1} \frac{3^i}{i \log i} + \sum_{i=n/2}^{n} \frac{3^i}{i \log i}$$

In ciascuna somma, sostituiamo il generico termine polinomiale e logaritmico con il primo:

$$F(n) < \sum_{i=2}^{n/2-1} \frac{3^i}{2 \log 2} + \sum_{i=n/2}^{n} \frac{3^i}{\frac{n}{2} \log \frac{n}{2}} = \frac{1}{2} \sum_{i=2}^{n/2-1} 3^i + \frac{1}{\frac{n}{2} \log \frac{n}{2}} \sum_{i=n/2}^{n} 3^i$$

La prima somma è in $\Theta\left(3^{n/2}\right)$, mentre la seconda si può maggiorare estendendo la somma, che attualmente è da $n/2$ a n, nuovamente da 1 a n: tale maggiorazione si può valutare in forma chiusa con la somma geometrica, ottenendo in complesso un termine in $\Theta(3^n / n \log n)$. Siccome il primo termine è dominato dal secondo, e tenendo conto della somma per difetto banale, si ottiene:

$$F(n) \in \Theta\left(\frac{3^n}{n \log n}\right)$$

Esempio 3

$$F(n) = \sum_{i=2}^{n} f(i) = \sum_{i=2}^{n} \frac{i}{\log i}$$

Essendo una funzione crescente polinomiale, cerchiamo di dimostrare che la somma sia in $\Theta(n\,f(n))$, e in particolare che sia in $\Omega(n\,f(n))$, dato che la stima per eccesso è banale.

Applichiamo la decomposizione in due sommatorie parziali:

$$F(n) = \sum_{i=2}^{n/2-1} \frac{i}{\log i} + \sum_{i=n/2}^{n} \frac{i}{\log i} > \frac{n}{2} \frac{n/2}{\log(n/2)}$$

Possiamo maggiorare questo valore sostituendo al denominatore $\log(n/2)$ un valore più grande, cioè $\log n$:

$$F(n) > \frac{n}{2} \frac{n}{2 \log n} = \frac{1}{4} \frac{n^2}{\log n}$$

da cui la tesi

$$F(n) \in \Theta\left(\frac{n^2}{\log n}\right)$$

Esempio 4

$$F(n) = \sum_{i=0}^{n} f(i) = \sum_{i=0}^{n} \sqrt[3]{i}$$

Trattandosi di un polinomio, è facile intuire che cresca lentamente. Applichiamo ancora la decomposizione in due sommatorie parziali:

$$F(n) = \sum_{i=0}^{n/2-1} \sqrt[3]{i} + \sum_{i=n/2}^{n} \sqrt[3]{i} > \frac{n}{2} \sqrt[3]{n/2}$$

da cui, tenendo conto della stima per eccesso banale

$$F(n) \in \Theta\left(n \sqrt[3]{n}\right)$$

In questo esempio sarebbe abbastanza semplice valutare la somma approssimata con l'integrale:

$$\sqrt[3]{0} + \int_0^n \sqrt[3]{x} dx \leq F(n) \leq \int_0^n \sqrt[3]{x} dx + \sqrt[3]{n}$$

e siccome l'integrale vale

$$\int_0^n \sqrt[3]{x} dx = \frac{3}{4}\left(n^{4/3} - 0^{4/3}\right) = \frac{3}{4} n^{4/3}$$

si deduce che

$$\frac{3}{4} n^{4/3} \leq F(n) \leq \frac{3}{4} n^{4/3} + \sqrt[3]{n}$$

confermando la tesi.

Esempio 5

$$F(n) = \sum_{i=1}^{n} f(i) = \sum_{i=1}^{n} \frac{1}{\sqrt[3]{i^2}}$$

Questa volta, ci troviamo di fronte a una funzione che decresce più lentamente di $1/i$. Infatti

$$f(i) = i^{-2/3} = \frac{i^{1/3}}{i}$$

Si può approssimare la sommatoria con l'integrale.

$$\frac{1}{\sqrt[3]{1}} + \int_1^n \frac{1}{\sqrt[3]{x^2}} dx \leq F(n) \leq \int_1^n \frac{1}{\sqrt[3]{x^2}} dx + \frac{1}{\sqrt[3]{n^2}}$$

e siccome l'integrale vale

$$\int_1^n \frac{1}{\sqrt[3]{x^2}} dx = 3\left(n^{1/3} - 1^{1/3}\right) = 3n^{1/3} - 3$$

si deduce che

$$3\sqrt[3]{n} - 2 \leq F(n) \leq 3\sqrt[3]{n} - 3 + \frac{1}{\sqrt[3]{n^2}}$$

che conferma la stima

$$F(n) \in \Theta\left(\sqrt[3]{n}\right)$$

Esempio 6

$$F(n) = \sum_{i=2}^{n} f(i) = \sum_{i=0}^{n} \frac{1}{i^2 \log i}$$

Questa funzione cala più rapidamente di $1/i$, per cui cercheremo di dimostrare che abbia somma asintoticamente costante. Per farlo, basta maggiorare il termine generico con $1/i^2$

$$F(n) = \sum_{i=2}^{n} \frac{1}{i^2 \log i} < \sum_{i=2}^{n} \frac{1}{i^2}$$

ottenendo una maggiorazione che già di per sé garantisce il risultato desiderato con il metodo dell'integrale:

$$\sum_{i=2}^{n} \frac{1}{i^2} \leq \int_2^n \frac{1}{x^2} dx + \frac{1}{n^2} = -\frac{1}{n} + \frac{1}{2} + \frac{1}{n^2} < \frac{1}{2}$$

da cui, tenendo conto anche della somma per difetto banale:

$$F(n) \in \Theta(1)$$

Esempio 7

$$F(n) = \sum_{i=0}^{n} f(i) = \sum_{i=0}^{n} \left(\frac{3}{4}\right)^i$$

Questa è una somma geometrica, il cui valore è noto in forma chiusa.

$$F(n) = \frac{\left(\frac{3}{4}\right)^{n+1} - 1}{\frac{3}{4} - 1} = 4 - 4\left(\frac{3}{4}\right)^{n+1} \in \Theta(1)$$

Esempio 8

$$F(n) = \sum_{i=2}^{n} f(i) = \sum_{i=2}^{n} \frac{1}{i \log i}$$

Questa funzione cala più rapidamente di $1/i$, ma meno rapidamente di qualsiasi potenza $i^{-1-\epsilon}$. Quindi non ricade in nessuna delle categorie su elencate. Si può procedere per integrazione:

$$\frac{1}{2 \log 2} + \int_2^n \frac{1}{x \log x} dx \leq \sum_{i=2}^{n} \frac{1}{i \log i} \leq \frac{1}{n \log n} + \int_2^n \frac{1}{x \log x} dx$$

Poniamo $y = \log x = \ln x / \ln 2$, da cui $dy = dx / (x \ln 2)$

$$\int_2^n \frac{1}{x \log x} dx = \int_1^{\log n} \frac{\ln 2}{y} dy = \ln 2 \int_1^{\log n} d \ln y = \ln 2 (\ln \log n - \ln \log 1)$$

da cui

$$F(n) \in \Theta(\log \log n)$$

Capitolo 4

Strutture dati astratte: vettori e record

Questo capitolo è dedicato a un concetto potenzialmente ostico per alcuni, del tutto naturale per altri: il concetto di *struttura dati astratta*. Questo concetto è in parte conflittuale rispetto alla principale raccomandazione dello scorso capitolo, cioè la massima efficienza nell'impiego del tempo e dello spazio. Le strutture dati astratte sono basate sul concetto di nascondere il più possibile all'utente, cioè al programmatore che implementa un algoritmo, l'effettiva realizzazione nel processore delle operazioni e della rappresentazione dei dati. Questo richiede necessariamente delle operazioni in più, in particolare un pesante utilizzo di procedure secondarie. Nel Capitolo X discuteremo il meccanismo del passaggio dei parametri alle funzioni, della gestione dello *stack*, della restituzione dei risultati. Questo meccanismo richiede necessariamente operazioni in più e un uso di memoria in più rispetto all'usare una sola funzione principale, e quindi è contraddittorio con l'obiettivo di risparmiare il più possibile il tempo e lo spazio. D'altra parte, questo impiego ulteriore è limitato (spesso costante rispetto alle dimensioni dell'istanza) e quindi va ad aggravare poco la complessità asintotica dell'algoritmo, e soprattutto ha degli indubbi vantaggi dal punto di vista della realizzazione pratica e della gestione del codice.

Un secondo possibile ostacolo alla comprensione è il fatto che i primi esempi del concetto di struttura dati astratta (i vettori e i *record*) sono talmente semplici da avere una sola implementazione naturale, cioè un solo modo ovvio di conservare i dati nel processore e di utilizzarli. Di conseguenza, l'uso effettivo della struttura dati astratta sembra quasi assurdo, mentre diventerà più naturale quando considereremo le liste, per non parlare degli alberi, dei grafi, e via dicendo. Gli esempi successivi diventeranno sempre più convincenti, mentre il primo potà lasciare perplessi.

Noi stiamo lavorando su un processore, cioè una macchina che esegue operazioni codificate in stringhe di bit su dati codificati in stringhe di bit per produrre soluzioni codificate in stringhe di bit. Lo fa in modo abbastanza implicito, grazie all'uso di un linguaggio ad alto livello. Nel processore, la memoria si può descrivere (sorvolando su mille difficoltà tecnologiche) come una sequenza di celle *elementari*. Con elementari si intende che le celle hanno una dimensione minima sotto la quale non si scende. Questa dimensione cambia secondo la macchina: tipicamente è una potenza di 2, come 8 bit (ovvero 1 byte), 16, 32, 64 bit, ma non è strettamente necessario che lo siano. Il punto fondamentale è che ci sono dati che occupano esattamente una cella perché sono molto semplici, e dati più grossi che ne occupano più d'una. Per esempio, un carattere tipicamente usa una sola cella da 1 byte, mentre un numero

reale potrebbe usarne 8 (ovvero 64 bit)[1]. Questi numeri sono significativi perché usati effettivamente in alcune macchine, ma vanno presi solo come esempi, non come regole. A questo punto, bisogna anche decidere se queste 8 celle sono consecutive o no, e come usarle per fare calcoli. Nel momento in cui un algoritmo opera non solo su un numero reale, ma su tanti, sapere come sono distribuiti nella memoria diventa importante. L'algoritmista che progetta e realizza un algoritmo in pratica non dovrebbe essere costretto a tener conto di tutti questi dettagli. Affinché non ne tenga conto, deve utilizzare strutture dati, che cercherà di rendere il più possibile astratte.

4.1 Strutture dati astratte

Una struttura dati è un modo in cui le celle della memoria del processore vengono organizzate per poter operare sui dati stessi in modo semplice ed efficiente, cioè con poche complicazioni pratiche, in un numero di passi piccolo e occupando un numero di celle piccolo. L'idea fondamentale è ora di distinguere tra due aspetti (vedi Figura 4.1):

1. la struttura dati astratta puramente logica e concettuale;
2. l'implementazione pratica.

La struttura dati indica che cosa sono i dati da un punto di vista Logico (se sono numeri reali, caratteri, interi, ecc...), come sono legati fra loro e che cosa si può fare su di loro. L'implementazione indica come effettivamente sono codificati i dati e come si eseguono le operazioni.

Una struttura dati astratta è definita da un insieme di dati elementari[2], o più insiemi legati fra loro (per esempio, una relazione fra due insiemi) e una lista di operazioni di operazioni che possono essere eseguite su questi insiemi di dati avendo determinati risultati. La *grammatica* delle operazioni (quali dati e quali risultati ammettono) definisce la struttura dati astratta. In questa fase, non ci riguarda assolutamente che cosa succede nel processore, che tecnologia si usa, come sono organizzati i dati, ecc... L'algoritmista vive il più possibile in questo mondo, che richiama quel che succede nella mente umana.

L'implementazione deve stabilire come distribuire i dati nelle celle in memoria, e una volta che sono stati distribuiti e si sa quante celle occupano (complessità spaziale) come si eseguono le operazioni che la struttura dati astratta deve rendere disponibili. Per esempio, si vuole poter determinare il quinto elemento di un insieme in base a un'opportuna indicizzazione. Ci saranno modi veloci e modi lenti di farlo, e di conseguenza diverse implementazioni.

Questa logica consente, quando si analizza un algoritmo, di dividere le analisi in due livelli: un livello superiore, nel quale l'algoritmo viene descritto parlando di dati, di risultati e di operazioni astratte; un secondo livello, nel quale le operazioni elementari vengono descritte in termini di operazioni specifiche sulle celle di memoria, senza arrivare al livello della specifica tecnologia, ma rispetto a un modello computazionale abbastanza generico che vede la memoria come sequenza di celle di una dimensione fissa. Per esempio, bisogna indicare in che modo l'estrazione dell'elemento di indice 5 di un insieme (oppure dell'elemento minimo) viene

[1] Ovviamente, i numeri reali sono infiniti, con la potenza del continuo, ma in realtà il processore rappresenta solo un campione finito dei numeri reali, per esempio 2^{64} numeri opportunamente scelti.

[2] Il caso particolare di un singolo dato è ovviamente possibile, anche se un po' degenere.

La memoria dei processori è una semplice sequenza di celle elementari

Una struttura dati è una organizzazione delle celle della memoria
che consenta di operare sui dati in modo semplice ed efficiente

Distingueremo tra

- struttura dati astratta (che cosa si fa e su quali dati), che consiste in
  - uno o più insiemi di dati elementari
  - una o più operazioni eseguibili su tali insiemi

  senza specificare l'organizzazione e la tecnologia adoperate
- implementazione (come si fa), che consiste nel modo specifico in cui
  - i dati vengono distribuiti nella memoria
  - le operazioni vengono eseguite sui dati

Questo consente di dividere l'analisi di un algoritmo in due fasi

1. descrizione dell'algoritmo in termini di dati, operazioni e risultati
2. realizzazione specifica di ciascuna operazione

La prima fase è più semplice, la seconda condivisa fra più algoritmi

Figura 4.1: Strutture dati astratte e implementazioni

compiuta. A diversi modi di compiere la stessa operazione corrispondono diverse implementazioni della stessa struttura dati.

Il livello superiore è per certi versi più semplice, perché più vicino alle operazioni della mente umana, mentre il secondo è più tecnologico. Però il secondo riguarda operazioni elementari, quindi si può definire più "semplice" in un'accezione diversa. Inoltre, riguarda operazioni che tipicamente sono riciclabili in algoritmi diversi: sia trovare l'elemento i-esimo di un insieme sia trovarne il minimo possono servire in moltissimi algoritmi. Nel caso di operazioni comuni come quelle citate, si arriva a costruire delle librerie di codice generiche, che permettono di gestire dati per qualsiasi scopo (vedi Figura 4.2). Per esempio, la conservazione di un insieme di oggetti con la capacità di determinarne il minimo e quella di eliminare elementi può essere usata per ordinare un insieme di oggetti: si tratta di trovare il minimo, estrarlo dall'insieme e metterlo da parte, poi trovare il minimo dell'insieme residuo, estrarlo e metterlo in coda al precedente, e così via, finché l'insieme residuo è vuoto. A quel punto, gli elementi accodati sono in ordine crescente. La disponibilità della struttura astratta ha consentito di progettare un algoritmo di ordinamento senza impegnarsi nell'implementazione su processore. la struttura dati astratta è semplicemente costituita da un insieme (o più d'uno) e da una lista di operazioni eseguibili ottenendo certi risultati. A questo livello, il costo delle operazioni in termini di tempo e spazio non è noto, ma va espresso simbolicamente.

Per operazioni molto comuni, si arriva al progetto di librerie generiche

Per esempio, conservare un insieme di oggetti potendo estrarne il minimo è utile in diverse situazioni

- ordinare l'insieme di oggetti
- costruire sottoinsiemi di peso minimo
- ...

Ogni diversa implementazione di una struttura corrisponde a:

- un costo spaziale, misurato dal numero di celle che richiede per rappresentare una struttura con n elementi
- un costo temporale per ogni operazione, misurato dal numero di passi che richiede per eseguirla su una struttura con n elementi

Figura 4.2: Strutture dati astratte e implementazioni

Qualche altro studioso nel frattempo si sarà dedicato al compito di creare una libreria per gestire dati in modo da individuare l'elemento minimo di un insieme ed estrarlo nella maniera più efficiente possibile dal punto di vista del tempo e dello spazio. Avendo definito un'implementazione specifica, il costo spaziale di conservare i dati (numero di celle elementari richieste) e quello temporale di eseguire ciascuna operazione (numeri di passi elementari) sarà misurato in dipendenza di una misura di dimensione. Come abbiamo visto, operativamente useremo come misura il numero di elementi della struttura stessa, se è sufficiente a caratterizzarla. Ne deriveranno le complessità effettive delle operazioni elementari, che prima avevamo indicato solo simbolicamente, e che permettono di completare l'analisi.

4.2 Vettori

La struttura dati astratta più semplice è il *vettore* (vedi Figura 4.3), che usiamo fin dalle scuole superiori (forse anche le scuole medie, nel caso della fisica). In genere, si vede un vettore come una n-upla di numeri reali, ma nulla vieta che siano numeri interi o oggetti di altro genere.

Definizione 5 *Definiamo* vettore di dimensione n su un insieme base U *una n-upla ordinata di elementi di un insieme base U.*

Un vettore V di dimensione n su un insieme U è definito come una n-upla ordinata $(v_1, \ldots, v_n)$ di elementi di U

V associa a ogni intero fra 1 e n un elemento di U

La struttura dati astratta "vettore di dimensione n su U" è definita come

- l'insieme $\mathcal{V}_{n,U}$ di tutti i possibili vettori di dimensione n su U

$$\mathcal{V}_{n,U} = U^n = U \times \ldots \times U$$

- le due operazioni fondamentali di

1. proiezione $\pi_i(V)$, che associa a un numero $i \in \{1, \ldots, n\}$ e un vettore V un elemento di U, generalmente indicato come v_i

$$\pi : \{1, \ldots, n\} \times U^n \to U$$

Restituisce l'elemento di indice dato del vettore dato

2. sostituzione $\sigma_i(V, u)$, che associa a un numero $i \in \{1, \ldots, n\}$, un vettore V e un elemento $u \in U$ il vettore ottenuto sostituendo v_i con u in V

$$\sigma : \{1, \ldots, n\} \times U^n \times U \to U^n$$

Cambia l'elemento d'indice dato del vettore dato con l'elemento dato

Figura 4.3: Vettori: struttura dati astratta

Un vettore associa a ciascun numero intero fra 1 e n un elemento dell'insieme base U. Il numero compreso fra 1 e n acquista il significato di *indice* di un elemento del vettore. La struttura dati astratta vettore di dimensione n su U combina l'insieme di tutti gli oggetti che ricadono in una descrizione data con le operazioni che su questi oggetti si possono significativamente eseguire. L'insieme $\mathcal{V}_{n,U}$ di tutti i possibili vettori di dimensione n su U non è altro che il prodotto cartesiano di U per sé stesso, fino ad avere n fattori, perché questo crea una n-upla ordinata di oggetti tratti dall'insieme U, eventualmente con ripetizione (è possibile anche che lo stesso oggetto compaia in tutte le posizioni del vettore). Dato uno di questi possibili vettori, che è il nostro insieme di definizione, abbiamo due operazioni fondamentali:

1. la *proiezione* $\pi_i(V)$ associa ad un numero i compreso fra 1 e n e a un vettore di $\mathcal{V}_{n,U}$ un elemento di U, tipicamente indicato con $V[i]$ e inteso come l'i-esimo

elemento del vettore;

$$\pi : \{1, \ldots, n\} \times U^n \to U$$

sostanzialmente, si tratta di andare a prendere l'elemento i-esimo di un vettore dato: potremmo tranquillamente definirla *lettura*;

2. *sostituzione* $\sigma_i\,(V, u)$ associa ad un numero i compreso fra 1 e n, un vettore di $\mathcal{V}_{n,U}$ un elemento di U, un altro vettore di $\mathcal{V}_{n,U}$:

$$\sigma : \{1, \ldots, n\} \times U^n \times U \to U^n$$

Il vettore risultato è quello che si ottiene da quello di partenza sostituendo il suo elemento i-esimo con l'elemento u passato dome dato; sostanzialmente, è la scrittura all'interno di un vettore.

Introdotte le definizioni astratte, bisogna realizzarne delle implementazioni concrete. l'implementazione che vedremo è praticamente ovvia, ma anche se è difficile pensare che ve ne possano essere altre ragionevoli, dobbiamo tenere aperta la mente all'idea che potrebbero.

In linguaggio C, un vettore non è altro che una sequenza di celle consecutive (vedi Figura 4.4). Gli n elementi sono tutti di un determinato tipo U. Ciascuno sarà costituito da una o più celle, consecutive per semplicità. Analogamente, è ragionevole che l'intero vettore sia fatto di n blocchi, ognuno dei quali contiene uno dei dati del vettore stesso. La figura illustra come si definisce un vettore di `N` interi, avendo avuto cura di introdurre la costante simbolica `N` di valore 10 con una direttiva `#define`. Il vettore si definisce specificando prima il tipo dei singoli elementi (`int`), poi il nome del vettore, e infine il numero degli elementi fra parentesi quadre (`[N]`), quindi:

```
int A[100];
int V[N];
int B[10*N+4];
```

dove il numero di elementi deve essere un valore costante, o esplicito (ad es., `100`), implicito (`N`), o infine implicitamente definito attraverso un'operazione aritmetica (`10*N+4`). Nello standard C89 che adottiamo non esistono vettori con dimensione variabile. Il motivo è ovvio per chi abbia capito il processo di compilazione descritto nel Capitolo **??**: quando il compilatore manipola la definizione del vettore, esegue l'operazione (come testimonia il punto e virgola finale) di annotare nella tabella dei simboli l'esistenza di un oggetto con un dato nome e riservargli un'area di memoria con lo spazio necessario a contenere un dato numero di numeri interi. Per poter eseguire questa operazione, il compilatore deve sapere quanto spazio occupa quel numero di numeri interi. Lo spazio richiesto da un numero intero è determinato dalla macchina, ma il numero di numeri interi è fissato dall'utente, e deve essere noto nel momento in cui l'operazione viene eseguita, cioè durante la compilazione. Questo è impossibile se si usa un'espressione variabile, il cui valore diventa noto solo nel momento in cui si esegue il programma, prima di quando si compila per trasformarlo in un eseguibile. Per motivi didattici, quindi, usare vettori di dimensione variabile è considerato un errore nel seguito del corso.

Va poi osservato che nel linguaggio C, una volta che un vettore di n elementi è stato definito, gli indici usati per scorrerlo saranno compresi fra 0 e $n-1$. Questo discorda con la definizione astratta, nella quale gli indici vanno da 1 a n. Il motivo è tecnologico, e avremo da discuterne in seguito.

Trattiamo ora i costi in spazio e tempo della gestione di un vettore di n elementi (vedi Figura 4.5). Il costo spaziale è costituito da n blocchi di celle, ognuno dei

In C vettore è realizzato con una sequenza di celle consecutive contenente un dato numero `N` di elementi dello stesso tipo

Il tipo in C determina l'insieme dei possibili valori (ad es., `int`)

L'indice d'un elemento è la sua posizione nella sequenza: va da 0 a `N`-1 (*non da* 1 *a* `N`*!*)

Si dichiara un vettore specificando

tipo variabile`[`*numero*`];`

- il tipo degli elementi: predefinito o definito da utente, semplice o "strutturato" (cioè vettore o record)
- il numero degli elementi: un'espressione costante positiva

Esempio:

```
#define N 10
int V[N], A[100];
int B[10*N+4];
```

Figura 4.4: Vettori: implementazione in C

quali è costituito da d_U celle elementari. Il valore di d_U è il numero di celle richiesto dal singolo dato elementare, e dipende dal suo tipo. Complessivamente, il vettore occuperà nd_U celle contigue. Quanto vale d_U per i tipi di dato più comuni? In generale, d_U dovrebbe essere il numero di bit strettamente necessario a rappresentare tutti gli oggetti del tipo, cioè dell'insieme U. Usando un alfabeto binario, questo numero è pari a $\log_2 |U|$. Per motivi di semplicità tecnologica, questo numero va standardizzato, in modo che ci sia solo un piccolo numero di possibili valori di d_U multipli fra loro. Siccome i caratteri (`char`) costituiscono un alfabeto (ASCII) di 256 simboli, la dimensione di un `char` tipicamente sarà di $\log_2 256 = 8$ bit, valore al quale si dà convenzionalmente il nome di *Byte* (in simboli, B). Nel seguito useremo alcuni valori convenzionali, che vanno intesi solo come indicativi e sono stati scelti in modo da essere abbastanza comuni e da differenziare tra loro tipi diversi, anche se in realtà questi valori cambiano da una macchina all'altra:

- per un carattere (`char`), la dimensione sarà 1B;
- per un numero intero (`int`), sarà 4B[3];
- per un numero reale (`double`) sarà 8B[4].

Concludendo, l'occupazione di memoria di un vettore è sempre lineare, $\Theta(n)$.

Passiamo al costo temporale di esecuzione delle operazioni associate alla struttura dati astratta *vettore*. Queste sono due: la proiezione e la sostituzione. La *proiezione* è, in termini più semplici, una lettura. Essa comporta di prendere un vettore e un indice numerico intero compreso fra 1 e n e restituire l'elemento del vettore nella posizione corrispondente all'indice. Da un punto di vista astratto, questo è quanto. Da un punto di vista concreto, il modo in cui si esegue questa operazione

[3] Ma gli interi `long` potrebbero anche occupare 8B.
[4] Ma i numeri reali di tipo `float` spesso occupano solo 4B.

ne determina il costo. Questo modo non è specificato a priori: compilatori diversi potrebbero farla in modo diverso. Tuttavia, qualsiasi implementazione sensata della proiezione fa la stessa cosa: siccome un vettore è formato da n blocchi consecutivi di celle, ciascuno di dimensione d_U, per recuperare il blocco i-esimo basta trovare la prima cella del vettore, Ind(V), (consultando la tabella dei simboli) e spostarsi in avanti nella memoria di id_U celle (sommando questo valore all'indirizzo della prima cella). Questa operazione aritmetica fornisce l'indirizzo della prima cella del blocco di indice i; le altre celle seguono di conseguenza. La proiezione comporta quindi di recuperare l'indirizzo del vettore, eseguire un prodotto e una somma, e consultare una sequenza finita di celle. Sono tutte operazioni che richiedono un tempo costante (dipendente da U , ma non da n). Quindi l'operazione di proiezione per qualsiasi vettore ha costo $\Theta(1)$. Per la *sostituzione*, che è poi la scrittura di un elemento in una data posizione di un vettore, valgono le stesse considerazioni, perché si tratta di trovare la prima cella del vettore, dedurne con un prodotto e una somma la prima cella in cui scrivere il dato e scrivere in quella cella e nelle successive il dato stesso. Il costo è ancora costante, cioè $\Theta(1)$.

Nelle dispense di teoria, si sottolinea la differenza tra le definizioni di *costo uniforme* e di *costo logaritmico* per la valutazione della complessità di un algoritmo. In laboratorio, adotteremo sempre il *criterio di corto uniforme*, che considera ciascun dato elementare come di dimensione costante diciamo, anche se la dimensione d_U non è la stessa per tutti gli insiemi U di dati. Questo perché in pratica, le dimensioni di tipi diversi hanno rapporti fissi fra loro: manipolare un `double` costa 8 volte un `char` in spazio e in tempo, ma non n volte, e questi rapporti spariscono nelle analisi asintotiche di complessità. Il criterio di costo logaritmico, invece, tiene conto del fatto che bisogna operare sui singoli bit della rappresentazione. Siccome le operazioni di lettura e scrittura comportano di scorrere d_U celle, la loro complessità diventa $\Theta(d_U)$, cioè $\Theta(\log |U|)$, che dipende dalla dimensione del dato stesso. Merita osservare che il criterio di costo uniforme comporta necessariamente che gli insiemi U su cui si opera siano tutti finiti, dato che deve esistere una dimensione d_U massima da considerare costante ai fini dell'analisi. Questo significa, in particolare, che i numeri interi e reali con cui si opera non sono l'intero insieme dei numeri interi e reali, ma solo opportuni sottoinsiemi finiti.

Un'osservazione utile per i progetti d'esame: i vettori sono strutture così semplici e fondamentali che molti studenti tendono a sottovalutarne i vantaggi, preferendo strutture dati più sofisticate. La verità è che, finché si tratta di conservare dati in memoria, leggerli e scriverli, i vettori sono sostanzialmente imbattibili. Per preferire altre strutture bisogna avere un buon motivo, cioè la necessità di fare altre operazioni, che per i vettori siano problematiche, come discuteremo in dettaglio per ciascuna delle strutture via via introdotte.

Nel linguaggio C, le operazioni di proiezione e di sostituzione si realizzano in modo molto semplice (vedi Figura 4.6), ma che nasconde il fatto che si tratta di operazioni su strutture dati astratte[5] (vedi Figura 4.6). Per leggere l'elemento di indice `i` di un vettore `V`, basta scrivere il nome del vettore seguito dall'indice fra parentesi quadre: `V[i]`. Per scrivere un elemento in un vettore, si fa esattamente la stessa cosa per accedere alla cella desiderata, e poi si riporta l'operatore di assegnamento `=` e il valore che si vuole scrivere nel vettore.

Lettura e scrittura non controllano in alcun modo che l'indice utilizzato per accedere al vettore sia lecito, come abbiamo visto quando abbiamo spiegato in che modo queste operazioni vengono realizzate tipicamente. Il compilatore e il processore non verificano né in compilazione (quando, del resto, probabilmente sarebbe impossibile

[5] In effetti, a rigore, non lo sono, dato che non è possibile in pratica modificarne l'implementazione, come si può per le altre strutture dati astratte. D'altra parte, come si diceva, è piuttosto improbabile che esistano implementazioni migliori di queste.

Adottiamo il criterio di costo uniforme

- ogni elemento di U è rappresentato da un numero $d_U = \log_{|\mathcal{A}|} |U|$ costante (superiormente limitato) di simboli di un alfabeto $\mathcal{A}$

Il costo spaziale per un vettore di dimensione n su U è lineare ($\Theta(n)$), dato che si usano $n \cdot d_U$ celle contigue (n blocchi da d_U)

Il costo temporale della proiezione $\pi_i(V)$ è costante ($O(1)$):

- determinazione della prima cella occupata dall'elemento v_i

$$\mathrm{Ind}(V) + i \cdot d_U$$

  dove $\mathrm{Ind}(V)$ è l'indirizzo della prima cella occupata dal vettore V

- lettura delle d_U celle che rappresentano v_i

Il costo temporale della sostituzione $\sigma_i(V, u)$ è costante ($O(1)$):

- determinazione della prima cella occupata da v_i
  dove $\mathrm{Ind}(V)$ è l'indirizzo della prima cella occupata dal vettore V
- copia delle d_U celle che contengono u in quelle che contengono v_i

Con il criterio di costo logaritmico, i costi sono diversi!

Figura 4.5: Vettori: costi

farlo, dato che l'indice può essere variabile, a differenza della dimensione) né in esecuzione, che l'indice abbia un valore valido. Questa è una delle principali fonti di errore nella programmazione C, anche se non richiede altro che un minimo di attenzione e precisione.

Un altro aspetto degno di nota è che in generale potrebbe essere utile avere vettori i cui indici non siano compresi fra 0 e $n-1$, se non altro perché la definizione astratta li indicizza da 1 a n, ma anche perché indici negativi potrebbero essere utili in pratica. Una soluzione parziale a questo problema è, volendo un vettore indicizzato da `S` a `D`, definirlo di `D+1` elementi. Questo crea un vettore indicizzabile da 0 a `D`, che è accettabile per qualsiasi valore non negativo di `S`. Ovviamente, questo comporta di sprecare (tenere occupata senza scopo) la memoria delle celle da 0 a `S-1`. Per esempio, supponiamo di voler conservare informazioni associate a numeri di matricola compresi fra 9 999 000 e 9 999, 999. Si tratta di 1 000 valori, ma il trucco richiede un'occupazione di 1 000 000 di blocchi di celle. Vedremo in seguito un trucco migliore, che consente anche di usare indici negativi. D'altra parte, finché si tratta di rappresentare vettori da 1 a n, semplicemente definendoli di dimensione $n+1$, il trucco è assai semplice e poco costoso. È un uso abbastanza comune fra gli studenti definire la dimensione minima necessaria e procedere a trasformare l'indice astratto in concreto e viceversa ogni volta che si legge o scrive un dato. Sembra un'idea intelligente, ma comporta un'operazione in più per ogni lettura e scrittura, cioè un costo temporale non indifferente (anche se nullo nell'analisi asintotica) al solo scopo di risparmiare un blocco di celle in memoria. Peggio ancora, le trasformazioni sono molto spesso sbagliate, e comunque faticose da ricordare mentre si scrive il codice e faticose da interpretare quando qualcuno lo legge. Nel complesso, non sembra un'idea così intelligente.

L'operazione di proiezione si rappresenta con il vettore,
seguito dall'indice dell'elemento, fra parentesi quadre

Esempio:
```
#define N 10
int V[N];
i = V[4];
```

Per definire vettori con estremi diversi (positivi), si allarga il vettore:
un vettore con estremi `S` e `D` tali che $0 \leq$ `S` $\leq$ `D` si dichiara con

```
int V[D+1];
```

cioè lasciando i primi `S` elementi inutilizzati

L'operazione di sostituzione combina parentesi quadre e assegnamento
`V[3] = 7;`

In C non c'è controllo che l'indice cada entro l'intervallo dichiarato:
si può scrivere in aree di memoria incontrollate (tipica causa di errori)

Figura 4.6: Indicizzazione: proiezione e sostituzione

Esistono anche vettori a più dimensioni, generalmente indicati col nome di *matrici* (vedi Figura 4.7). Il linguaggio C consente di definirle indicando il tipo dell'elemento singolo, seguito dal nome della matrice e dalle varie dimensioni in sequenza, ciascuna racchiusa fra parentesi quadre. Si tratta sempre di numeri costanti. Un aspetto tecnologico interessante è che, come in un vettore monodimensionale gli elementi sono contigui, così lo sono nei vettori multidimensionali, e sono continui

riga per riga. Cioè l'ultimo elemento di una riga precede immediatamente il primo della riga seguente. Nell'esempio in figura, un vettore di 5 righe (indicizzate da 0 a 4) e 10 colonne (indicizzate da 0 a 9) parte con l'elemento `V[0][0]`, seguito dalla riga `0` fino all'elemento `V[0][9]`, a cui segue subito l'elemento successivo `V[1][0]` e i suoi successivi fino a `V[1][9]`, ecc... Il cinquantesimo elemento della sequenza è `V[4][9]`. Il motivo è che l'operazione di prodotto e somma che abbiamo visto usare nella proiezione e sostituzione viene semplicemente estesa usando tutti gli indici e le dimensioni della matrice, a partire dall'ultimo indice e risalendo via via fino al primo. Da un punto di vista astratto, questo non ci interessa, perché non deve essere usato, ma è bene esserne coscienti per interpretare possibili comportamenti anomali dei propri algoritmi. Se nell'uso di una matrice, sbagliando, si eccedono i limiti di uno degli indici, il risultato sarà leggere o scrivere nelle righe successive. Questo può spiegare perché la matrice subisca modifiche o fornisca dati relativi a righe sbagliate: semplicemente, l'indice di colonna è eccede il valore massimo.

Un vettore può avere qualsiasi numero di dimensioni, cioè i suoi elementi possono essere identificati da qualsiasi numero di indici

Matrice è un vettore a due o più dimensioni

La dichiarazione specifica il numero di valori per ciascun indice

tipo variabile`[`*numero1*`][`*numero2*`][`*numero3*`];`

Gli elementi dei vettori multidimensionali sono in sequenza lessicografica: sono ordinati prima per righe, poi per colonne;
se l'indice di colonna eccede `N`, si accede alla riga seguente

Per una matrice di `M` righe, da 0 a `M-1`, e `N` colonne, da 0 a `N-1`:

```
#define M 5
#define N 10
int V[M][N];
```

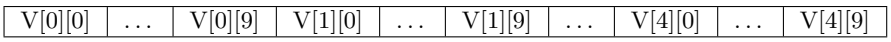


Figura 4.7: Vettori multidimensionali

Un'altra cosa da sapere è che in C non si può copiare un vettore in un altro (vedi Figura 4.8), cioè non si può semplicemente assegnare un vettore a un altro e aspettarsi che vengano copiati gli elementi del primo nei corrispondenti elementi del secondo. È importante ricordarlo perché la cosa diventa possibile quando si usano i *vettori dinamici* (vedi Capitolo **??**), ma con conseguenze molto sofisticate da gestire, e quindi è meglio pensare in prima approssimazione che non sia lecito farlo. Se si vuole un vettore o una matrice in un altra è necessario e sufficiente copiare elemento per elemento con un ciclo.

4.3 Record

Una seconda struttura dati astratta molto semplice è il così detto *record*. Si tratta, ancora una volta, di un insieme di oggetti indicizzato, come il vettore, ma con due fondamentali differenze:

I vettori non si possono copiare con il semplice assegnamento:
vanno copiati elemento per elemento

Sbagliato

```
#define M 5
#define N 10
int A[M+1][N+1];
int B[M+1][N+1];


B = A;
```

Corretto

```
#define M 5
#define N 10
int A[M+1][N+1];
int B[M+1][N+1];
int i, j;

for (i = 0; i <= M; i++)
  for (j = 0; j <= N; j++)
    B[i][j] = A[i][j];
```

Figura 4.8: Copia di vettori

1. gli oggetti non sono necessariamente dello stesso tipo (anche se possono esserlo);

2. gli indici non sono numeri interi, ma simboli tratti da un alfabeto finito.

Un esempio è l'insieme dei dati meteorologici raccolti da una stazione, per esempio pressione, temperatura e umidità. Si tratta di tre numeri reali, ma si potrebbero aggiungere altri dati di tipo diverso, come il nome della stazione, la sua posizione geografica, la data e l'ora delle misurazioni, ecc... È possibile indicarli convenzionalmente come "dato 1", "dato 2" e "dato 3", ma è un modo piuttosto goffo di procedere. Ha più senso che assumano dei nomi simbolici (appunto, `pressione`, `temperatura` e `umidita`). Se poi vi fossero dati di tipo diverso, conservarli in un vettore sarebbe impossibile. D'altra parte, si tratta di un insieme di dati logicamente collegati, che ha senso considerare unitariamente, ma accedendo a ciascuno con il suo nome simbolico.

La definizione astratta (vedi Figura 4.9 di un record R indicizzato su un insieme U_a di campi[6], dove gli indici a appartengono a un insieme finito, detto alfabeto A, è quella di un insieme finito di $|A|$-uple, in cui l'a-esimo elemento di ciascuna appartiene all'insieme U_a. Nell'esempio, l'alfabeto A è costituito dai tre elementi `pressione`, `temperatura` e `umidita`. Può sembrare strano sentirlo definire "alfabeto", ma questo è il nome generico di un insieme finito utilizzabile per indicizzarne un altro. Ogni simbolo dell'alfabeto identifica un *campo* del record, cioè uno degli elementi, che deve cadere nell'insieme di valori indicato dal tipo U_a, che può essere diverso per i vari campi: ci sarà un insieme di possibili valori per la pressione, uno per la temperatura e uno per l'umidità.

Per il resto, la situazione è molto simile a quella dei vettori: si ha l'insieme di tutti i possibili record $\mathcal{R}_{\{U_a\}}$, che è il prodotto cartesiano dei singoli U_a, e si hanno le due operazioni già viste per i vettori, cioè lettura (o *proiezione*) e scrittura (o *sostituzione*). La proiezione $\pi_a(R)$ associa a un dato record R e al simbolo a identificativo di un campo un elemento del corrispondente insieme base U_a, cioè in pratica determina nel record il contenuto del campo desiderato. La sostituzione $\sigma_a(R, u)$ associa a un record R, a un simbolo a e a un elemento dell'insieme U_a il

[6] Controllare sulle dispense di teoria: mi sembrerebbe più appropriato scrivere U_A visto come prodotto cartesiano dei singoli insiemi di definizione U_a.

record che si ottiene a partire da R sostituendo il contenuto del campo di indice a con il nuovo elemento fornito alla funzione.

Un record R sui campi U_a è definito come un insieme finito $\{x_a\}_{a \in A}$ che associa un elemento x_a dell'insieme U_a a ogni simbolo a di un alfabeto A

R associa a ogni simbolo $a \in A$ un elemento del campo associato U_a

La struttura dati astratta "record sui campi U_a" è definita come

- la collezione $\mathcal{R}_{\{U_a\}}$ degli insiemi di elementi singoli tratti dagli U_a
- le due operazioni fondamentali di
  1. proiezione $\pi_a(R)$, che associa a un record R e a un simbolo $a \in A$ un elemento di U_a, generalmente indicato come $R \cdot a$

     $$\pi : \mathcal{R}_{\{U_a\}} \times A \to U_a$$

     Restituisce il campo dato del record dato
  2. sostituzione $\sigma_a(R, u)$, che associa a un record R, un simbolo $a \in A$ e un elemento $u \in U_a$ il record ottenuto da R sostituendo $R \cdot a$ con u

     $$\sigma : \mathcal{R}_{\{U_a\}} \times A \times U_a \to \mathcal{R}_{\{U_a\}}$$

     Cambia il campo dato del record dato con l'elemento dato

Figura 4.9: Record: struttura dati astratta

Come si implementa tutto questo? Come nel caso dei vettori, si conserveranno gli elementi in un'area di memoria contigua, con un numero di blocchi pari al numero dei campi del record. Ogni blocco ha un numero di celle tale da poter contenere elementi dell'insieme base corrispondente, dunque diverso per ciascun blocco e determinata dal tipo corrispondente (vedi Figura 4.10). Invece di accedere alle singole celle combinando l'indirizzo della prima cella dell'intera struttura con il valore numerico dell'indice, attraverso un prodotto e una somma, si partirà dall'indirizzo del record, ma si dovrà aggiungere a tale indirizzo le dimensioni dei campi che precedono quello di interesse. Si tratta comunque di un'operazione in tempo costante, $O(1)$. Questo vale per la scrittura, come per la lettura.

La Figura 4.11 illustra alcuni aspetti della sintassi che il linguaggio C adopera per definire record (indicati come `struct`, o *strutture*). La definizione è molto più elaborata che per i vettori, perché occorre specificare i tipi e nomi dei singoli campi (essendo diversi e non essendo l'indice un semplice numero intero). La parola chiave `struct` è seguita da parentesi graffe, che raccolgono le coppie (tipo,nome) relative ai singoli campi e che specificano ciascun insieme base U_a e ciascun simbolo a. Sia la definizione delle variabili sia quelle dei singoli campi sono seguite da `;` dato che si tratta di istruzioni di aggiornamento della tabella dei simboli, per conservare le informazioni usate dalle operazioni di proiezione e sostituzione.

Va detto che in realtà nessuno definisce variabili di tipo record in questo modo perché è molto pesante da leggere, sia per l'utente umano sia per il compilatore (vedremo fra poco lo svantaggio tecnico che comporta). La definizione di uso comune è riportata in rosso sulla destra della Figura 4.11, e spezza la dichiarazione di una variabile `meteo_oggi` in due operazioni. La prima comunica al compilatore

In C record o struttura è una sequenza di celle consecutive contenente elementi eterogenei accessibili attraverso un nome simbolico (campo)

Una variabile di tipo struttura si dichiara specificando

- il tipo di ciascun campo
- il nome di ciascun campo
- il nome dell'intera variabile

```
struct {
  tipo1 campo1;
  tipo2 campo2;
  ...
} variabile;
```

La dichiarazione ha la solita struttura (*tipo variabile*;)
ma il *tipo* è composto da più parole: `struct {...}`

Figura 4.10: Record: implementazione in C

attraverso l'istruzione `typedef` l'esistenza, da quel punto in poi, di un nuovo tipo di dati `dati_meteo`, che rappresenta un record con tre campi di tipo `double` e di nome `pressione`, `temperatura` e `umidità`. La parola `dati_meteo` diventa un *alias*, un segnaposto, per l'intera definizione del record. La seconda operazione è la dichiarazione della variabile, nella quale si può usare `dati_meteo` anziché tutta la dichiarazione dettagliata. Questo non è solo comodo, ma spesso necessario, perché due record con la stessa struttura definiti in modo identico non sarebbero riconosciuti dal compilatore come dello stesso tipo (e quindi eventualmente assegnati l'uno all'altro). Non si è voluto caricare il compilatore del compito di confrontare definizione complesse per decidere se due variabili sono omogenee. Siccome il passaggio di dati a una funzione opera attraverso un'operazione di assegnamento (come vedremo nel Capitolo **??**, definire i record singolarmente ed esplicitamente vorrebbe dire non poterli passare come dati a funzioni. Se invece si introduce un tipo record con un nome semplice, si possono definire variabili di quel tipo in numero qualsiasi e si possono usare come se fossero variabili dei tipi elementari predefiniti nel linguaggio.

La Figura 4.12 illustra in che modo lettura e scrittura sono realizzate sui record in C. L'operatore `.` preceduto dal nome di un record e seguito dal nome di un campo consente di accedere a tale campo di tale record. Quindi, la proiezione π(`temperatura,meteo_oggi`) viene indicata come `meteo_oggi.temperatura` e restituisce l'oggetto di tipo `double` contenuto nel campo `temperatura` del record `meteo_oggi`. Lo stesso succede con la sostituzione, esattamente come abbiamo visto nel caso dei vettori.

Un aspetto interessante dei record è che si possono annidare (vedi Figura **??**), cioè possono contenere come elementi altri record. Questo valeva anche per i vettori: un vettore a più dimensioni non è altro che un vettore di vettori (ed eventualmente ancora di vettori, ecc...). Analogamente, una struttura `studente` può contenere una struttura `persona`, e così via. Lo scopo dell'annidamento è avere funzioni che operano su record, e quindi strutture dati astratte di tipo record che non si riducano alla sola proiezione e sostituzione. Per esempio, dato un record `persona`, si possono avere funzioni per la ricerca dell'identificatore di una persona, per stampare i dati di una persona in un opportuno formato, per ordinare insiemi di persone, e così via. Quando un record è annidato in un altro, le funzioni associate al secondo possono sfruttare le funzioni associate al primo. Per esempio, funzioni che gestiscono gli studenti possono usare le funzioni che gestiscono le persone.

Usando l'istruzione `typedef` è opportuno separare

- la dichiarazione della variabile (`meteo_oggi`)
- la dichiarazione del tipo (`dati_meteo`)

```
struct {
  double temperatura;
  double pressione;
  double umidita;
} meteo_oggi;
```

```
typedef struct {
  double temperatura;
  double pressione;
  double umidita;
} dati_meteo;

dati_meteo meteo_oggi;
```

In questo modo

- si evita di ripetere una lunga dichiarazione per ogni variabile
- si comunica al compilatore che due variabili sono dello stesso tipo (*il compilatore non saprebbe riconoscere l'uguaglianza!*)

Figura 4.11: Dichiarazione di strutture

La proiezione (accesso in lettura a un campo di una struttura) si rappresenta con la struttura seguita da un punto (`'.'`) e dal campo

Esempio: `t = meteo_oggi.temperatura;`
`strcpy(qui.nome,"Crema");`

La sostituzione (accesso in scrittura a un campo di una struttura) combina l'operatore punto con l'assegnamento

Esempio: `meteo_oggi.pressione = 1020.0;`

Figura 4.12: Accesso ai campi: proiezione e sostituzione

Strutture e vettori possono contenere strutture e vettori ricorsivamente, con qualsiasi numero di livelli

```
typedef struct {
  long id;
  char nome[LUNGHEZZA+1];
  char cognome[LUNGHEZZA+1];
} persona;

typedef struct {
  long matricola;
  persona identita;
} studente;
```

In questo modo è più facile modularizzare il codice

- costruire funzioni che operano su strutture
- combinando funzioni che operano su sottostrutture

Figura 4.13: Annidamento (1)

La cosa si può ripetere ricorsivamente (vedi Figura 4.14): un record può contenere al suo interno un record che contiene altri record (questo vuol dire applicare più volte l'operatore `.`), e si possono combinare record e vettori in qualsiasi modo: vettori che contengono strutture, strutture che contengono vettori, con una serie di combinazioni di operatori `.` e `[]` che sono piuttosto intuitive.

Per accedere a un campo di una sottostruttura, si specificano la struttura, la sottostruttura e il campo, separati da punti (`'.'`)

```
strcpy(studente1.identita.nome,"");
```

Per accedere a un elemento di un vettore che è campo di una struttura, si specifica la struttura, il campo vettore e la posizione

```
iniziale1 = studente1.identita.nome[0];
iniziale2 = studente1.identita.cognome[0];
```

Per accedere a un campo di una struttura elemento di un vettore, si specifica il vettore, la posizione e il campo

```
studente classe[100];
m = classe[12].matricola;
```

Figura 4.14: Annidamento (2)

Merita osservare che, mentre i vettori non si possono assegnare ad altri vettori, è possibile assegnare un record a un altro con un operatore di assegnamento, che consiste nel copiare campo per campo il primo nel secondo. Questo è molto comodo, e addirittura consente di copiare un vettore statico in un altro senza doverli scorrere con un ciclo, se sono contenuti in un campo di un record: un possibile trucco per consentire la copia di vettori, quindi, è di annidarli in record. È potenzialmente pericolosa se i record contengono campi di tipo puntatore (che discuteremo nel Capitolo **??**. Infatti, in tal caso, la copia riguarderà il semplice puntatore (o indirizzo), e non l'intera area di memoria contenente i dati puntati. Quindi, a seguito dell'assegnamento i due record non conterranno due copie degli stessi dati, ma due puntatori alla stessa area di memoria singola che li contiene. Se si presume falsamente che siano dati distinti, si avranno errori difficili da determinare.

4.4 Laboratorio

L'esercizio pratico che segue riguarda l'uso della struttura dati vettore (nel caso specifico bidimensionale). Non considereremo record perché ne vedremo diverse applicazioni nelle lezioni successive[7]

Lo scopo di questa esercizio è triplice, ed è praticamente impossibile tenere i tre obiettivi ben distinti:

1. ripassare il linguaggio di programmazione C;
2. progettare e realizzare in modalità *top-down* un algoritmo per un problema;
3. stendere una relazione sull'algoritmo realizzato.

[7] Verificare se ho aggiunto il record `mossa`.

Figura 4.15: Assegnamento
Si può applicare l'operatore di assegnamento (=) a intere strutture

```
dati_meteo meteo_ieri, meteo_oggi;

meteo_oggi = meteo_ieri;
```

equivale a

```
meteo_oggi.temperatura = meteo_ieri.temperatura;
meteo_oggi.pressione = meteo_ieri.pressione;
meteo_oggi.umidita = meteo_ieri.umidita;
```

Copia i campi della struttura a destra in quelli della struttura a sinistra

- copia i campi di tipo elementare
- copia i campi di tipo struttura, sottocampo per sottocampo
- copia i campi di tipo vettore statico (strano!)
- copia i campi di tipo puntatore, ma non duplica l'oggetto puntato (vettori dinamici!); questo può essere molto pericoloso

Dal primo punto di vista, la sezione contiene molte considerazioni di natura sintattica (su tipi, variabili, funzioni di libreria, ecc...). Dal secondo punto di vista, la sezione discute la riduzione del problema a sottoproblemi su più livelli, fino ad arrivare a problemi talmente semplici da poter essere risolti con le funzioni di base del linguaggio di programmazione. Dal terzo punto di vista, la sezione discute come descrivere l'algoritmo risolutivo e caratterizzarne la complessità spaziale e temporale, un compito che in pratica si rivela piuttosto arduo per gli studenti, come già discusso nella Sezione **??**. I tre punti di vista vanno separati nettamente, perché hanno scopi e destinatari del tutto diversi: il ripasso di programmazione è rivolto a studenti con competenze di programmazione da da rinverdire (dunque superfluo in teoria, ma in pratica utile); il progetto e la realizzazione dell'algoritmo sono fondamentali, e rivolti a studenti che conoscono la programmazione e stanno imparando gli algoritmi; la relazione è rivolta a lettori che conoscono programmazione e algoritmi e sono interessati allo specifico problema trattato e algoritmo realizzato.

Organizzazione del laboratorio Prima di affrontare la descrizione del problema, facciamo una nota metodologica: i materiali dell'esercizio consistono, come al solito, in una sequenza numerata di codici, che parte da `othello0` e termina con `othello7`: sono le fasi successive di implementazione dell'algoritmo risolutivo. In laboratorio, ogni fase consiste nella descrizione di che cosa va fatto e nell'assegnare del tempo agli studenti per farlo, rispondendo alle domande che questo provoca via via, e poi presentare e commentare la soluzione. Studiando in proprio, è comunque consigliabile sfruttare la divisione in fasi provando a realizzarle in sequenza, in modo da affrontare i dubbi che ciascuna comporta e confrontare le proprie soluzioni con quelle proposte nei materiali, invece di limitarsi a leggere passivamente le soluzioni commentate. Resta il principio che non c'è una soluzione corretta e tutte le altre sono sbagliate, ma vi sono tante possibili soluzioni, di cui è interessante analizzare vantaggi e svantaggi, a meno che siano perfettamente equivalenti.

Il problema riguarda un gioco da tavolo abbastanza noto: Othello. È noto anche con un altro nome, Reversi, che credo abbia regole simili, ma si concluda nel modo opposto (vince chi perde in Othello, e viceversa). Si gioca su una classica scacchiera da otto righe e otto colonne (vedi Figura **??**), sulla quale si posizionano pedine bianche e nere. Le pedine hanno due facce di colore opposto, una bianca e una nera, e ogni giocatore le posiziona sulla tabella con il proprio colore verso l'alto, ma nel corso del gioco possono essere catturate e allora cambiano faccia e passano all'altro giocatore. Partendo dalla posizione descritta nella Figura **??**, con due pedine bianche e due nere nelle caselle centrali, muove il nero mettendo una nuova pedina del suo colore sulla scacchiera, in maniera tale da poter fare delle prese. Per fare una presa, bisogna porre la nuova pedina in una casella che racchiuda con un'altra pedina dello stesso colore già esistente chiuda un filotto verticale, orizzontale o diagonale di pedine del colore avversario. Per esempio, se il giocatore nero mette una sua pedina in C4, fa una presa perché in E4 c'è un'altra pedina nera, che insieme a C4 racchiude a forcella un filotto di pedine bianche degenere, costituito dalla sola pedina in D4. Altro esempio: la nuova pedina nera in D3, insieme a una pedina nera preesistente in D5, racchiude a forcella la pedina bianca in D4, che quindi catturata e girata, diventando nera. A questo punto muove il bianco e le sue tre mosse lecite sono segnate con crocette nella Figura **??**. Sono tutte e sole le tre mosse che consentono di avere un filotto verticale di pedine nere catturate: E3 crea un filotto verticale costituito da E4, C3 crea un filotto diagonale che cattura la pedina nera in D4, e C5 crea un filotto orizzontale che cattura la pedina nera in D5

Il problema consiste nel ricevere una sequenza di mosse che descrive una partita e valutare se le mosse sono lecite. Se qualche mossa non è lecita, si deve abbandonare l'analisi spiegando che la mossa non è lecita; se si arriva in fondo alla partita, si deve indicare chi ha vinto, cioè quale giocatore ha più pedine dell'altro. Può capitare (abbastanza raramente), che il numero di pedine sia lo stesso, e quindi ci sia un pareggio. Può anche capitare (anche questo è raro) che uno dei due giocatori non abbia mosse disponibili. In tal caso, introduciamo per semplicità la regola che il giocatore passi semplicemente la mano all'altro senza muovere, e se entrambi i giocatori non hanno mosse disponibili, la partita termina e si valuta l'esito come quando la scacchiera è piena. È chiaro che il gioco deve terminare dopo al massimo 30 mosse nere e bianche, dato che si aggiungono 60 pedine che con le quattro iniziali riempiono l'intera scacchiera.

I dati sono una sequenza di mosse: ciascuna è rappresentata da una coppia, costituita da un carattere compreso fra `a` e `h`, che rappresenta la colonna in cui il giocatore che muove aggiunge una pedina, e una cifra compresa fra `1` e `8`, che rappresenta la riga. Quale giocatore stia muovendo è implicito. in base alle regole del gioco: le mosse dispari nella sequenza sono del nero, quelle pari sono del bianco. Quando il giocatore corrente non può muovere, la sequenza contiene due trattini consecutivi (`--`). Se questi trattini concludono la sequenza, significa che entrambi i giocatori sono privi di mosse lecite.

Questa sequenza deve essere validata, cioè il programma deve verificare che ciascuna delle mosse della sequenza è realizzabile nelle condizioni correnti. Se una mossa non lo è, il programma deve stampare un messaggio di errore indicando la mossa e il giocatore, e terminare. Anche il simbolo `--` deve essere validato, verificando che effettivamente non sia possibile al giocatore corrente muovere, e se sono in fondo, che anche l'altro non possa. Se vi sono mosse lecite, si stamperà un messaggio di errore e si terminerà. Se invece si arriva in fondo senza errori, bisogna stampare due blocchi di informazioni. Il primo è la scacchiera, in un formato abbastanza autoevidente, che ha una prima riga (che associeremo alla cifra `0`) che riporta gli indici carattere delle colonne, e altre otto righe numerate che rappresen-

tano le singole caselle, con una `N` per le caselle contenenti una pedina nera, una `B` per le caselle contenenti una pedina bianca e un punto (`.`) per le caselle vuote (per distinguerle dall'esterno della scacchiera. Il secondo blocco di informazioni riporta il numero di pedine nere e bianche e conclude indicando il vincitore.

Il formato in ingresso e il formato in uscita non sono parte del problema: sono la codifica in un file di testo dei dati (che sono oggetti matematici) e la codifica in una stampa a video dei risultati (che sono oggetti matematici). Quindi, non fanno parte del problema e non si riportano nella relazione. Ne parliamo qui perché ci stiamo accingendo a scrivere del codice, e sono informazioni necessarie a questo scopo.

Possiamo quindi partire con il file `othello0.c` e con il file di dati di esempio (`mosse.txt`). Passiamo al contesto "ripasso del linguaggio C". La struttura di `othello0.c` è la consueta struttura dei file di codice: parte con una sezione di direttive, `#include` e `#define` in particolare. Le prime includono librerie: `stdlib.h` gestisce i parametri del main e la terminazione del programma; `stdio.h` gestisce la lettura e scrittura da file e tastiera e su video; `string.h` gestisce le stringhe[8] Le direttive di definizione introducono costanti simboliche: `ROW_LENGTH` vale 256 e verrà usata per avere stringhe di caratteri abbastanza lunghe da contenere i dati e i risultati. I valori `FALSE` (0) e `TRUE` (1) consentono di verificare condizioni logiche. Altre costanti si riferiscono al problema specifico: servono a poter cambiare in qualsiasi momento con una sola operazione i caratteri che rappresentano a video le pedine bianche, quelle nere e le caselle vuote, senza doverli cercare nell'intero codice. Il precompilatore, infatti, si occupa di diffondere ovunque la modifica, dato che il suo compito è esattamente quello di fare modifiche testuali al codice. Esiste una costante simbolica `MOVE_LENGTH` che indica la lunghezza della codifica di una mossa: sono sempre due caratteri, che sia una mossa vera o una mossa di sospensione.

FIGURA CON LE DIRETTIVE

La seconda parte del codice contiene le dichiarazioni delle procedure secondarie (sorvoliamo sull'istruzione `typedef`, che discuteremo in seguito). Per ora, c'è solo la dichiarazione di `InterpretaLineaComando`. Questa funzione riceve gli argomenti del main e trasforma il secondo (quello di indice 1, che segue il nome del programma) in una stringa di caratteri contenente il nome del file delle mosse.

FIGURA CON I PROTOTIPI

La terza parte del codice è il programma principale, con la solita struttura: l'intestazione specifica il nome `main`, i dati (`argc` e `argv`) e il risultato; il corpo, fra parentesi graffe, comprende una parte dichiarativa (con il vettore di caratteri che conterrà il nome del file delle mosse già dichiarato) e una parte esecutiva che contiene già la chiamata alla funzione che interpreta la linea di comando e l'istruzione di uscita regolare (`return EXIT_SUCCESS;`) per comunicare al sistema operativo che tutto è andato bene. Ci sono anche dei commenti che preparano la divisione del problema in sottoproblemi.

FIGURA CON IL MAIN

La quarta parte del codice è composta dalla definizione delle procedure secondarie. Al momento c'è solo `InterpretaLineaComando`, con la solita struttura: intestazione con il nome, i dati e il tipo del risultato, e corpo contenente una parte dichiarativa (vuota) e una parte esecutiva con un paio di operazioni molto semplici.

FIGURA CON LA FUNZIONE SECONDARIA

Passiamo al contesto "progetto e implementazione dell'algoritmo". Dobbiamo dividere il problema in sottoproblemi. La decomposizione che mi pare ragionevole è:

[8] Controllare se serve; se no, toglierla e correggere.

1. costruire la configurazione iniziale della scacchiera;
2. eseguire la sequenza delle mosse indicata nel file, valutandone la validità: se sono scorrette, uscire con un messaggio di errore; se sono corrette, aggiornare via via la scacchiera e restituirla al termine;
3. stampare la scacchiera finale;
4. stampare l'esito della partita.

Si noti che per ogni sottoproblema abbiamo specificato i dati e i risultati, altrimenti non si tratterebbe di una vera decomposizione in problemi. Il terzo e il quarto sottoproblema potrebbero essere unificati, ma sono concettualmente abbastanza distinti, e il terzo (stampare la scacchiera) potrebbe essere utile da risolvere non solo alla fine della partita, ma in qualunque momento (per esempio, per controllare il codice). Questo suggerisce di tenerli separati.

Scrivere la decomposizione del problema in sottoproblemi sotto forma di commenti offre una serie di vantaggi. Per prima cosa, consente di rendersi conto se la decomposizione sia fatta bene e quali siano gli ingressi e le uscite di ciascuna "scatola" (i dati e i risultati di ciascun sottoproblema). Inoltre, vedremo che associare commenti ben scritti alle funzioni che risolvono i sottoproblemi consentono di ottenere una traccia utile per la stesura della relazione finale.

Come si fa a costruire la configurazione iniziale della scacchiera? Chiamiamola `InizializzaScacchiera` e definiamone dati e risultati. Non ci sono dati, e il risultato è una scacchiera, diciamo `S`. La scacchiera è però un dato composto (da molti caratteri) e il C non consente a una funzione di restituire un risultato composto, e bisogna quindi riportare il risultato stesso fra i dati (le ragioni saranno chiare nel Capitolo **??**. Di conseguenza, scriveremo:

```
Scacchiera S;

InizializzaScacchiera(S);
```

con un'istruzione di dichiarazione nella parte dichiarativa e un'istruzione operativa in quella esecutiva. Il codice è già predisposto con l'indicazione (attraverso l'istruzione `typedef`) che una scacchiera è un vettore bidimensionale di `BOARD_SIZE` righe e colonne (con `BOARD_SIZE` costante simbolica introdotta da una direttiva `#define`). Ovviamente, gli indici di riga e di colonna variano da 0 a 7. Perché non scrivere esplicitamente che si tratta di una matrice di caratteri? Perché stiamo perseguendo il concetto di struttura dati astratta, cioè di presentare i dati come oggetti astratti con funzioni che li manipolano o ne estraggono informazioni. Nel caso di una scacchiera, si può pensare a molte operazioni, come la stampa, l'esecuzione di una mossa, la valutazione di situazioni, e in particolare l'inizializzazione, che è quanto stiamo facendo ora. In linea di principio non vogliamo rendere esplicito che cosa è fisicamente una scacchiera in modo che, se si decidesse di cambiare la sua rappresentazione, non sarebbe necessario scorrere l'intero codice per trovare tutti i punti in cui compare una scacchiera, ma basterebbe cambiare le funzioni che la usano. Il programma principale ne risulterebbe intoccato.

È chiaro che è difficile pensare a un implementazione più semplice di una scacchiera rispetto a una matrice a due dimensioni, ma si tratta di un primo esempio didattico, e abbiamo già detto che gli esempi successivi saranno via via più convincenti.

Per raggiungere la correttezza sintattica, bisogna far seguire alla chiamata della funzione `InizializzaScacchiera` la sua dichiarazione fra i prototipi e la sua definizione nell'ultima parte del codice. La prima sarà

```
/* Costruisce la configurazione iniziale di una scacchiera */
void InizializzaScacchiera (Scacchiera S);
```

dove conviene cogliere l'occasione per sfruttare il commento alla chiamata copiandolo e rimaneggiandolo opportunamente in modo che diventi un commento generale alla funzione. Infine, la definizione sarà

```
/* Costruisce la configurazione iniziale di una scacchiera */
void InizializzaScacchiera (Scacchiera S)
{
}
```

cioè per il momento vuota. A questo punto, la comparsa nel programma principale di un'istruzione `InizializzaScacchiera(S);` non suscita preoccupazione perché sia `InizializzaScacchiera` sia `S` sono simboli noti, e si sa che il primo richiede come dato una `Scacchiera` e il secondo lo è proprio.

Il secondo sottoproblema è l'esecuzione delle mosse. Introdurremo una funzione `Othello` che richiede in ingresso una scacchiera e il nome di un file che contiene le mosse da valutare ed eseguire. Il risultato è la scacchiera nella sua configurazione finale. Come sappiamo, va tenuta nei dati e (come vedremo) non occorre (e in effetti non è possibile, almeno in modo facile) distinguere il suo ruolo come dato da quello come risultato. Come al solito, dalla chiamata e dal suo commento costruiamo la dichiarazione e la definizione e i relativi commenti.

```
/* Esegue sulla scacchiera S le mosse indicate nel file filemosse */
void Othello (Scacchiera S, char *filemosse);
```

e lo stesso vale per gli altri due sottoproblemi: sia `StampaScacchiera(S)` sia `StampaEsito(S)` hanno come dato la scacchiera e nessun risultato:

```
/* Stampa a video la scacchiera S */
void StampaScacchiera (Scacchiera S);

/* Stampa l'esito del gioco */
void StampaEsito(Scacchiera S);
```

A questo punto, il codice non fa nulla, ma abbiamo fatto un passo nella direzione della struttura finale ed è tutto sintatticamente corretto. L'unica operazione che viene compiuta è la verifica di correttezza della linea di comando, cioè che vi sia un parametro (in teoria, il nome del file delle mosse, ma questo non viene controllato).

Seconda fase Ora dobbiamo prendere i quattro sottoproblemi e risolverli o ulteriormente decomporli. Il programmatore inesperto tende a procedere in ordine, dal primo all'ultimo sottoproblema. Questo è ovviamente corretto nel descrivere il codice (nella relazione), ma non lo è nella stesura del codice. Questo perché scrivere migliaia di righe di codice senza alcuna stampa dei risultati, rende impossibile sapere se quello che si sta scrivendo abbia senso o no. Scoprirlo alla fine significa dover trovare gli errori in un codice estremamente lungo, e lasciare che errori diversi interagiscano e si nascondano a vicenda.

Nell'implementazione del codice ha senso procedere "a cipolla", per strati dall'esterno verso l'interno: prima tutto ciò che è interfaccia verso il mondo esterno, poi il corpo dell'algoritmo. Nel caso specifico, conviene realizzare prima l'inizializzazione della scacchiera e la stampa della scacchiera e dell'esito, poi le mosse. Infatti, anche

se non si eseguono le mosse, si può stampare la scacchiera inizializzata e l'"esito" che le corrisponde, e questo dà informazioni utili sull'eventuale presenza di errori.

Siccome le tre funzioni sono molto semplici, le realizzeremo tutte insieme. È consigliabile farlo indipendentemente e poi verificare la soluzione confrontandola con quella ufficiale. Si tenga conto che il codice già fornisce una serie di costanti simboliche che saranno certamente utili: `BIANCO` è il carattere `B`, `NERO` è il carattere `N` e `LIBERO` il carattere `.` che rappresenta le celle vuote; la dimensione della scacchiera si chiama `BOARD_SIZE` e una scacchiera è un vettore bidimensionale di `BOARD_SIZE` righe e colonne.

La funzione `InizializzaScacchiera` riempie la scacchiera di `.` in tutte le caselle, tranne le quattro caselle centrali, dove bisogna scrivere `B` e `N`.

Per scorrere tutte le righe e le colonne della matrice, occorrono due cursori (numeri interi) `r` e `c` e due cicli annidati. Contrariamente al resto del corso, useremo la classica indicizzazione C da 0 a `BOARD_SIZE-1`. Un primo motivo è che stiamo realizzando una funzione a basso livello su una struttura dati astratta. Un secondo motivo è che in realtà il problema nel mondo esterno non usa indici interi da 1 a 8 (il che comporterebbe un continuo fastidioso convertire gli indici interni in quelli esterni, e viceversa), ma usa caratteri da `a` ad `h` e cifre (cioè caratteri) da `1` ad `8`, e vedremo che per accedere a tali indici la convenzione del C è più comoda.

In ogni riga e colonna riga della scacchiera scriviamo `LIBERO`. Usare la costante simbolica, anziché il valore esplicito, garantisce la possibilità futura di cambiare la rappresentazione dello spazio libero con poca fatica.

Nelle celle centrali bisogna invece scrivere `NERO` e `BIANCO` Che indici hanno le celle centrali? Potremmo dire che gli indici sono 3 e 4, ma sprecheremo qualche operazione per ottenere un codice che possa funzionare anche su scacchiere di dimensione diversa da 8, calcolando i due indici come `BOARD_SIZE/2-1` e BOARD_SIZE/2.

INCLUDERE CODICE

Abbiamo prima riempito le celle centrali col simbolo di casella vuota, poi con i simboli definitivi. Avrebbe senso cercare di scriverli subito corretti? Introdurre una condizione nel ciclo, che ad ogni passo verifichi se la cella corrente è centrale o no costerebbe molto di più rispetto a quattro sovrascritture.

Per la stampa della scacchiera, si tratta di stampare prima l'intestazione nella prima riga, che si potrebbe scrivere esplicitamente, ma se vogliamo conservare la possibilità di cambiare la dimensione della scacchiera è opportuno usare il cursore `c` e farlo variare in modo da scorrere i caratteri associati alle colonne. A tale scopo, scorriamo da 0 a `BOARD_SIZE-1` e per ogni valore determiniamo il carattere corrispondente. In C è possibile farlo sfruttando l'*aritmetica dei caratteri*, cioè il fatto che si possono sommare numeri interi a caratteri, ottenendo dei caratteri che risultano successivi o precedenti al carattere iniziale nell'ordine alfabetico, spostati di tante posizioni quante ne indica il numero intero. Quindi per stampare i caratteri da `a` ad `h` è sufficiente scorrere con il cursore `c` i numeri da 0 a `BOARD_SIZE-1` e sommarli al carattere `a`. Questo rende conveniente l'uso di un cursore che parte da 0: non lo stiamo usando direttamente, ma solo per ottenere il codice ASCII che codifica il carattere effettivamente desiderato[9].

```
/* Stampa la cornice superiore */
printf(" ");
for (c = 0; c < BOARD_SIZE; c++)
  printf("%c",c+'a');
```

[9]Tutto questo si basa sull'ipotesi che le lettere dell'alfabeto e le cifre decimali seguano l'ordine naturale nella tabella utilizzata dal processore. È un'ipotesi tecnologica che lo standard C garantisce: non impone i valori specifici dei codici, ma impone che questi sottoinsiemi di caratteri (le lettere minuscole, le lettere maiuscole e le cifre) siano consecutive e seguano l'ordine naturale.

```
printf("\n");
```

Poi possiamo stampare le altre 8 righe con un ciclo sulle righe: per ciascuna stampiamo la cifra corrispondente (o come numero intero, sommando 1 all'indice, o ancora con l'aritmetica dei caratteri), il contenuto della riga, cioè i caratteri della scacchiera, e andiamo a capo. Infine, andiamo ancora a capo perché il testo dell'esercizio richiede una riga vuota dopo la scacchiera.

```
/* Stampa la cornice laterale e la scacchiera */
for (r = 0; r < BOARD_SIZE; r++)
{
  printf("%d",r+1);
  for (c = 0; c < BOARD_SIZE; c++)
    printf("%c",S[r][c]);
  printf("\n");
}
printf("\n");
```

Concludiamo con la stampa dell'esito, che richiede prima di contare le pedine nere e le pedine bianche, e poi di dedurne il vincitore (divisione del problema in due sottoproblemi). Queste tre informazioni si possono anche determinare nella funzione `Othello`. Non lo facciamo per non affrontare subito il problema di recuperare da una funzione risultati multipli, problema che tratteremo nel Capitolo **??**. Per calcolare il numero di caselle nere e di caselle bianche, basta scorrerle e incrementare opportuni contatori `nn` e `nb`, ovviamente inizializzati a 0, secondo il contenuto della casella corrente.

```
/* Conta le pedine bianche e nere */
nb = nn = 0;
for (r = 0; r < BOARD_SIZE; r++)
  for (c = 0; c < BOARD_SIZE; c++)
    if (S[r][c] == NERO)
      nn++;
    else if (S[r][c] == BIANCO)
      nb++;
```

Stampati i due numeri, la determinazione dell'esito si riduce a confrontarli per distinguere i tre casi possibili: che le caselle nere siano più di quelle bianche (vince il nero), che siano meno (vince il bianco) o che siano ugualmente numerose (pareggio).

```
/* Stampa il numero di pedine bianche e nere */
printf("%d pedine nere\n",nn);
printf("%d pedine bianche\n",nb);

/* Stampa il risultato */
if (nn > nb)
  printf("Vince il Nero\n");
else if (nb > nn)
  printf("Vince il Bianco\n");
else
  printf("Pareggio\n");
```

Eseguendo il codice, otteniamo la scacchiera iniziale con una valutazione dell'esito del gioco se la configurazione fosse finale. Questo suggerisce che il codice possa essere corretto (non una garanzia, ovviamente).

4.5 Terza fase

Rimane ora da risolvere il sottoproblema principale, cioè riempire la funzione `Othello`. Questa deve scorrere la sequenza di mosse e valutarle una per una: se si tratta di una sospensione, bisogna verificare che il giocatore corrente non abbia mosse lecite; se si tratta di una mossa vera e propria, bisogna verificare che il giocatore corrente possa effettivamente eseguirla ed eseguirla aggiornando la scacchiera. In caso di errori, si deve stampare un opportuno messaggio all'utente e terminare. Questa analisi, che corrisponde alla progettazione *top-down*, e che conviene fare scrivendo commenti completi e chiari che faranno da promemoria mentre si realizza il codice e da spiegazione dopo, suggerisce che per prima cosa si debba determinare il giocatore corrente, cosa che a prima vista non avevamo notato. Progettare il codice in questo modo aiuta a segnalare le operazioni che si potrebbe aver dimenticato e a esplicitare le ipotesi che si stanno facendo (e che potrebbero essere scorrette). Terminata la valutazione ed eventuale esecuzione della mossa corrente, bisogna passare la mano all'altro giocatore. Terminate tutte le mosse, si esce.

```
    for (c = 0; c < BOARD_SIZE; c++)
      printf("%c",S[r][c]);
    printf("\n");
  }
  printf("\n");
}


/* Stampa l'esito del gioco sulla scacchiera S */
void StampaEsito (Scacchiera S)
{
  int r, c;
  int nb, nn;


  /* Conta le pedine bianche e nere */
  nb = nn = 0;
  for (r = 0; r < BOARD_SIZE; r++)
    for (c = 0; c < BOARD_SIZE; c++)
      if (S[r][c] == NERO)
        nn++;
      else if (S[r][c] == BIANCO)
        nb++;

  /* Stampa il numero di pedine bianche e nere */
  printf("%d pedine nere\n",nn);
  printf("%d pedine bianche\n",nb);

  /* Stampa il risultato */
  if (nn > nb)
    printf("Vince il Nero\n");
  else if (nb > nn)
    printf("Vince il Bianco\n");
  else
    printf("Pareggio\n");
}
```

Ora non resta che trasformare i commenti in codice. Alcuni verranno realizzati effettivamente, i più sofisticati verranno trasformati in chiamate a opportune funzioni, che vanno dichiarate e definite al solito modo, per avere la correttezza sintattica. Per fare questo, bisogna aggiungere qualche dettaglio tecnologico, che a livello algoritmico non interessa: l'algoritmo presume di operare su una scacchiera e una sequenza di mosse; il codice opera su una `Scacchiera` e il nome di un file. Quindi, il codice deve anche aprire il file delle mosse all'inizio, reagire nel caso in

cui l'apertura fallisse, e chiudere il file alla fine. È consigliabile che a questo punto si proceda autonomamente prima di continuare e leggere la soluzione proposta.

Il procedimento "a cipolla" (prima l'interfaccia e poi il calcolo interno) suggerisce di cominciare aprendo, testando e chiudendo il file, dichiarando e usando un'opportuna variabile `fpMosse` di tipo `FILE*`. Abbiamo già discusso queste operazioni nel Capitolo **??** a cui rimandiamo per i dettagli.

Come si fa a determinare il giocatore corrente? In base alle regole del gioco, al principio il giocatore corrente è il nero. Conserviamo il corrispondente carattere in una variabile `giocatore`, così potremo usarlo direttamente anche per modificare la scacchiera quando eseguiremo le mosse.

La valutazione delle mosse corrisponde a un ciclo. Siccome a priori non sappiamo quante sono le mosse, il ciclo è a condizione, e siccome leggiamo le mosse una per una e il file potrebbe anche essere vuoto, potremmo impiegare un ciclo a condizione iniziale, usando come condizione il risultato dell'operazione di lettura. Dovendo leggere brevi stringhe di testo, l'operazione di lettura sarà una `fscanf` con specifica `%s`. Non è mai consigliabile leggere un carattere alla volta, se non altro perché poi ci si trova a dover gestire i caratteri separatori (spazi, a capi, tabulazioni), che invece le altre specifiche gestiscono autonomamente, e su cui non è mai bene fare ipotesi restrittive.

Quanto si può dare per scontato che il formato del file di ingresso sia rispettato? Nel caso specifico, possiamo ipotizzare che il file contenga solo una sequenza di stringhe di due caratteri, e che queste stringhe siano mosse lecite, cioè `--` oppure una lettera fra `a` e `h` seguita da una cifra fra `1` e `8`? Dipende dai casi, cioè dall'affidabilità della fonte dei dati. In questo corso e negli appelli d'esame, *la fonte va considerata totalmente affidabile per quanto è esplicitamente scritto nel testo* (quindi non sono ammesse ipotesi aggiuntive rispetto a quello che è scritto nel testo[10], e in particolare non ci saranno mai ipotesi semplificative sui caratteri separatori).

Visto quanto sopra, possiamo conservare le mosse in stringhe di due caratteri, cioè vettori di *tre* caratteri, che offrono lo spazio sufficiente per conservare anche il terminatore della stringa. È data una costante simbolica `MOVE_LENGTH` pari a 2 allo scopo di permettere l'eventuale estensione del gioco a matrici di più dimensioni. La lettura di una mossa ha successo quando `fscanf` restituisce 1. Se fallisce, il motivo più probabile è che il file sia terminato (non è probabile che in un file di testo ci sia qualcosa di diverso da una stringa).

È elementare riconoscere se la mossa è di sospensione: basta confrontarla con la stringa `--` con la funzione `strcmp`. L'elaborazione della singola mossa nei vari casi è invece abbastanza sofisticata da essere rimandata, cioè vista come un sottoproblema. Avremo quindi:

- una chiamata a `VerificaSospensione` i cui dati sono la scacchiera `S` e il `giocatore` corrente;
- una chiamata a `EsegueMossa` i cui dati sono la riga e colonna dove ha luogo la mossa, il `giocatore` corrente e la scacchiera `S`;

La funzione `EsegueMossa` fa sia la valutazione sia l'esecuzione. Avrebbe senso separarle, ma l'idea è che una mossa lecita viene anche subito eseguita e una non lecita porta a terminare l'algoritmo. Quindi le due cose sono strettamente legate. Inoltre, facendo un salto avanti e pensando a come funzionano valutazione ed esecuzione, ci si rende conto che la prima comporta in realtà di eseguire la mossa stessa, per cui tenerle separate è innaturale. Un altro aspetto degno di nota è che

[10] Questo andrebbe spostato nella sezione sul progetto, eventualmente lasciando solo un breve richiamo qui.

`EsegueMossa` potrebbe ricevere una mossa, intesa come stringa, anziché gli indici di riga e colonna[11]. Questo è assolutamente vero: si tratta di una questione di gusti, e semplicemente le operazioni che descriveremo fra poco per determinare tali indici andrebbero spostate dentro la funzione. Infine,

Determinare la riga e la colonna che corrispondono alla mossa corrente è un'applicazione inversa dell'aritmetica dei puntatori: la differenza di due caratteri è definita come il numero di passi che si devono compiere per raggiungere il primo a partire dal secondo scorrendo la tabella dei caratteri stessi. Se i passi sono verso destra, il numero è positivo; se sono verso sinistra, negativo. Quindi, `mossa[1]-'1'` determina l'indice di riga e `mossa[0]-'a'` quello di colonna.

```
/* Definisce il giocatore che muove per primo */
giocatore = NERO;

/* Finche' ci sono mosse, ne legge una */
while (fscanf(fp_filemosse,"%s",mossa) == 1)
{
  /* Se e' una sospensione, valuta se davvero non ci sono mosse
     lecite */
  if (strcmp(mossa,"--") == 0)
    VerificaSospensione(S,giocatore);
  else
  /* Se e' una mossa regolare, valuta se e' corretta e la esegue */
    EsegueMossa(mossa,giocatore,S);

  /* Passa la mano all'altro giocatore */
  if (giocatore == BIANCO)
    giocatore = NERO;
  else
    giocatore = BIANCO;
}
```

Infine, bisogna aggiornare il giocatore corrente, sostituendogli l'altro.

```
/* Passa la mano all'altro giocatore */
if (giocatore == BIANCO)
  giocatore = NERO;
else
  giocatore = BIANCO;
```

Per avere la correttezza sintattica, occorre naturalmente aggiungere la dichiarazione e la definizione delle due funzioni chiamate in `Othello`. Come già osservato, entrambe non hanno risultati (o hanno come risultato la scacchiera, che però figura come un dato). È interessante notare l'adattamento del commento alla situazione, che nella dichiarazione è più generale che in una specifica chiamata.

```
/* Verifica se la posizione corrente sulla scacchiera S e' davvero di
   sospensione per il giocatore */


/* Esegue, se possibile, la mossa per il giocatore sulla scacchiera */
void EsegueMossa (char *mossa, char giocatore, Scacchiera S);
```

Compilare ed eseguire questo codice suggerisce che (probabilmente) non abbiamo introdotto errori. Sembra non cambiare nulla, ma in realtà il nuovo codice controlla che il file delle mosse esista effettivamente e sia accessibile. Per sicurezza, potremmo

[11] Potrebbe essere anche un *record* contenente i due indici.

anche aggiungere istruzioni temporanee, per esempio stampando le mosse lette via via. Questo consente di verificare la correttezza del caricamento dei dati, che è sempre una buona norma prima di accingersi ad affrontare il vero e proprio calcolo dei risultati.

4.6 Quarta fase

Scegliamo, in modo piuttosto arbitrario, una delle due funzioni ancora non realizzate: `EsegueMossa`. Questa funzione riceve gli indici di riga e colonna di una casella, il giocatore corrente e la scacchiera. Deve valutare se il giocatore corrente può lecitamente aggiungere una propria pedina nella casella indicata, data la configurazione corrente della scacchiera. Come sempre, procediamo *top-down* scrivendo appunti che diventeranno i commenti del codice.

Per prima cosa, la mossa è lecita soltanto se (condizione necessaria) la casella `(r,c)` è libera. Inoltre, occorre che in una delle otto direzioni (due verticali, due orizzontali e quattro diagonali) ci sia una o più pedine del colore opposto in caselle contigue, seguite da una pedina del colore del giocatore corrente. È una condizione abbastanza sofisticata da costruire una funzione ad hoc e rimandare a dopo il sottoproblema di valutarla. Basta anche solo una delle otto direzioni per stabilire che la mossa è lecita. Quindi, ipotizzeremo che non lo sia e testeremo una per una le direzioni: per ciascuna, se si ha successo, si esegue direttamente la mossa e si appunta il fatto di aver avuto successo; altrimenti, non si fa nulla. È chiaramente una specie di ciclo che scorre le direzioni possibili, esegue le modifiche eventuali e annota la circostanza che la mossa è lecita. Al termine, si conoscerà l'esito della mossa e anche la configurazione finale della scacchiera, e si potrà reagire di conseguenza (proseguire, o terminare stampando un messaggio di errore). Scriviamolo a parole nel modo più semplice e chiaro possibile.

```
/* Verifica se la mossa e' lecita */
/* 1) la casella deve essere libera */
  /* 2) per ogni possibile direzione */
        /* valuta se la direzione e' lecita e, nel caso, esegue la
            mossa nella direzione */
/* Se la mossa e' lecita, aggiunge la nuova pedina alla scacchiera
    */
/* altrimenti, termina stampando un messaggio di errore */
```

Siccome valutare la mossa in ciascuna direzione porta spontaneamente ad eseguirla, in caso di successo l'unica cosa che rimane da fare è aggiungere la nuova pedina sulla scacchiera. Ci sono ovviamente molti modi leggermente diversi di realizzare questo algoritmo.

4.7 Quinta fase

Per prima cosa, occorre una variabile di tipo logico per conservare durante tutta l'elaborazione l'informazione se la mossa sia lecita oppure no. Non esistono in C89 variabili di tipo logico, ma il codice mette a disposizione sin dal principio un tipo `boolean` definito come `int` con un'istruzione `typedef` e i due valori logici `TRUE` e `FALSE` introdotti come costanti simboliche con direttive `#define`. Possiamo quindi direttamente introdurre una variabile apposita `mossa_lecita`.

Il punto più critico, che richiede un piccolo colpo di intuizione, è come realizzare un ciclo che scorra tutte le direzioni possibili. Il modo più banale di risolverlo è

sostituire il ciclo con una sequenza di 8 sequenze di operazioni, ciascuna dedicata a una direzione, simili fra loro, ma non identiche. Oppure un ciclo con un indice che cresce da 1 a 8 e chiama ogni volta una funzione che valuti se la mossa è lecita, passandole gli indici della casella di partenza e l'indice della direzione da valutare. Qualcosa di questo genere:

```
/* Verifica se la mossa e' lecita */
mossa_lecita = FALSE;
/* 1) la casella deve essere libera */
if (S[r][c] == LIBERO)
/* 2) per ogni possibile direzione */
for (dir = 1; dir <= 8; dir++)
{
/* valuta se la direzione e' lecita e, nel caso, esegue la mossa nella
direzione */
if (EsegueDirezione(r,c,dir,giocatore,S) == TRUE)
mossa_lecita = TRUE;
}
```

In realtà, un matematico dovrebbe notare che le mosse sono spostamenti e ricordare che posizioni e spostamenti in matematica sono descrivibili come vettori. In particolare, lo spostamento sarà un vettore elementare, con due componenti pari a -1, 0 oppure $+1$: combinando queste tre possibilità sulle righe e sulle colonne si ottengono 9 vettori elementari: uno è nullo e va scartato; gli altri descrivono le direzioni che ci interessa rappresentare. Quindi, possiamo generare le 8 direzioni componendo due cicli da -1 a $+1$ e ad ogni iterazione chiamare una funzione a cui passare i due indici della casella di partenza (riga e colonna) e le due componenti della direzione di spostamento (ancora riga e colonna). Per esempio, lo spostamento `(-1,0)` muove la casella `(r,c)` verso l'alto in `(r-1,c)`, lasciando immutata la colonna e riducendo la riga; lo spostamento `(+1,+1)` la muove in diagonale verso il basso e la destra in `(r+1,c+1)`, aumentando sia la riga sia la colonna corrente. La funzione restituirà un valore logico che indicherà se ci sono state catture di pedine oppure no, e in tal caso si aggiornerà la variabile `mossa_lecita`.

Concluso il ciclo, se almeno una delle direzioni ha dato luogo a catture la variabile lo rivela e basta aggiungere la pedina sulla scacchiera, cioè scrivere il carattere `giocatore` nella posizione `(r,c)`. Altrimenti, si esce con un messaggio di errore, nel quale sarà opportuno indicare di che mossa si tratti, trasformando di nuovo gli indici numerici di riga e colonna nella corrispondente stringa da due caratteri attraverso l'aritmetica dei caratteri, e indicare il giocatore corrente.

Si può discutere se sia più corretto dividere la valutazione della mossa dall'esecuzione anziché combinarle in una sola operazione a livello della singola direzione. L'unica situazione problematica è quella in cui esiste un filotto di pedine avversarie, ma al termine non c'è una pedina propria, ma una casella libera o il bordo della scacchiera. In tal caso, non devono aver luogo catture. Quindi, il filotto va scorso due volte: la prima per valutare dove comincia e dove finisce, la seconda per effetture le catture. A questo punto, è questione di gusti se si voglia dividere i due scorrimenti in funzioni separate (una per la ricerca degli estremi e una per l'esecuzione delle catture) oppure no. Nelle altre situazioni (quando non c'è filotto oppure c'è ed è lecito catturarlo) basterebbe addirittura uno scorrimento solo (salvo che non si può essere certi a priori di non essere nella prima situazione).[12] Ne deriva la seguente implementazione.

```
/* Verifica se la mossa e' lecita */
```

[12] Disquisizione poco chiara, forse, da cancellare, riscrivere o spostare.

```
mossa_lecita = FALSE;
/* 1) la casella deve essere libera */
if (S[r][c] == LIBERO)
  /* 2) per ogni possibile direzione */
  for (dr = -1; dr <= 1; dr++)
    for (dc = -1; dc <= 1; dc++)
      if ( (dr != 0) || (dc != 0) )
      {
        /* valuta se la direzione e' lecita e, nel caso, esegue la
           mossa nella direzione */
        if (EsegueDirezione(r,c,dr,dc,giocatore,S) == TRUE)
          mossa_lecita = TRUE;
      }

/* Se la mossa e' lecita, aggiunge la nuova pedina alla scacchiera
   */
if (mossa_lecita == TRUE)
  S[r][c] = giocatore;
else
/* altrimenti, termina stampando un messaggio di errore */
{
  fprintf(stderr,"La mossa %s non e' lecita per il giocatore
      %c!\n",mossa,giocatore);
  exit(EXIT_FAILURE);
}
```

che ovviamente richiede di dichiarare e definire (per il momento vuota) la funzione `EsegueDirezione` che valuta ed esegue la mossa in una specifica direzione. Ancora una volta, si noti il commento riscritto rispetto alla chiamata in modo da essere adatto al caso generale.

```
/* Se possibile, esegue la mossa (r,c) in direzione (dr,dc) per il
   giocatore sulla scacchiera;
   se non e' possibile, restituisce FALSE */
boolean EsegueDirezione (int r, int c, int dr, int dc, char
   giocatore, Scacchiera S);
```

Se si compila dopo aver dichiarato e definito questa funzione, si noterà un messaggio di avvertimento da parte del compilatore: la funzione è stata dichiarata come funzione che restituisce un intero, ma in effetti non lo fa, dato che non lo fa ancora. In linea di principio, conviene sempre correggere non solo gli errori, ma anche gli avvertimenti del compilatore, per non lasciarsi alle spalle questioni che nel seguito potrebbero combinarsi con altre in modo imprevedibile. Questa potrebbe essere un'eccezione: noi sappiamo, infatti, che la causa dell'avvertimento è la semplice mancanza di un corpo per la funzione, e che questa causa verrà rimossa al più presto naturalmente[13]

Eseguendo il programma a questo punto, il comportamento è purtroppo indeterminato, proprio a causa del fatto che la funzione `EsegueDirezione` non restituisce esplicitamente un valore, ma il processore usa il valore restituito per decidere che operazioni eseguire nel seguito. Vedremo nel Capitolo **??** che tale valore è il contenuto di un'opportuna cella, che a questo punto è del tutto casuale, da cui il comportamento indeterminato (cioè diverso a seconda del compilatore usato, del

[13]Per anni ho consigliato di eliminare l'avvertimento aggiungendo al corpo vuoto della funzione un'istruzione `return` seguita da un valore temporaneo di *default* (cioè valido in "difetto" di valori più appropriati), da sostituire poi durante la realizzazione della funzione. Mi sono reso conto negli anni che riempiendo il corpo della funzione tendevo a dimenticare di sostituire il valore temporaneo con quello definitivo, e questo provocava buona parte degli errori che commettevo in laboratorio. In effetti, non si trattava di correggere l'avvertimento, ma di nasconderlo, che è sempre errato in linea di principio.

computer, eventualmente anche del momento). Se il valore restituito è 0 (una cosa abbastanza frequente), il programma si comporta come se tutte le direzioni fossero dichiarate non lecite, e quindi già alla prima mossa termina stampando il messaggio di errore relativo alle mosse non lecite. Questa spiegazione ha lo scopo di mostrare che è opportuno conoscere i meccanismi di funzionamento della macchina per essere in grado di interpretarne le reazioni in situazioni di errore.

4.8 Quinta fase

Ora possiamo andare a realizzare la funzione `EsegueDirezione` che valuta ed esegue la mossa in una specifica direzione. Ovviamente, potremmo realizzare l'altra funzione in sospeso (`VerificaSospensione`), ma è più urgente arrivare ad avere qualche stampa di risultato per valutare la correttezza di quanto scritto finora. Questa funzione riceve il giocatore corrente, la casella in cui si vuole mettere una pedina e la direzione in cui si vuole valutare la possibilità di catturare pedine avversarie. La cattura richiede due condizioni, che devono valere entrambe:

1. che partendo dalla casella data e muovendosi nella direzione data si trovi una casella del colore opposto;
2. che le caselle avversarie terminino in una casella occupata da una pedina del colore del giocatore corrente, quindi non libera e non esterna alla scacchiera stessa (ovviamente, escludiamo anche le caselle dell'avversario, dato che il filotto altrimenti si limita a proseguire).

La prima cosa da fare è trovare il colore dell'avversario. Poi trovare la prima casella del filotto di pedine avversarie che ci si aspetta di catturare. A questo punto, si può scorrere il filotto stesso, muovendosi nella direzione indicata finché le caselle sono occupate da pedine avversarie. Si tratta ovviamente di un ciclo, una sequenza di spostamenti, che ha termine solo quando la casella corrente non è più avversaria, ma è libera p del giocatore corrente oppure esterna alla scacchiera. Lo spostamento avviene applicando (cioè sommando) lo spostamento elementare proprio della direzione alla posizione della casella corrente. Dal ciclo si esce quando si trova una casella non occupata da una pedina avversaria o quando si esce dai limiti della scacchiera, il che è illecito.

Terminato il ciclo, ci si può trovare in una casella occupata da una propria pedina, nel qual caso la mossa è lecita e si cattura il filotto di pedine avversarie, oppure in una casella libera o esterna alla scacchiera, nel qual caso la mossa è non lecita. È anche possibile che il ciclo sia vuoto, cioè che non si trovi un filotto di pedine avversarie. In questo caso, la mossa è illecita anche se la casella corrente è occupata da una pedina propria. Però si può identificare questo caso perché la casella finale è adiacente a quella di partenza. Per esempio, la mossa `(f,4)` nella configurazione di Figura 4.12.1 non è lecita, ma spostandosi verso sinistra, in direzione `(-1,0)` il ciclo termina immediatamente e la casella finale `(e,4)` è occupata da una pedina nera. Quindi anche questo caso va identificato e trattato autonomamente come caso illecito.

Se la mossa è lecita, bisogna scorrere nuovamente il filotto di pedine avversarie, assegnando loro il colore del giocatore corrente. Un modo semplice di procedere è tornare dalla casella corrente a quella iniziale applicando lo spostamento in direzione opposta (in questo modo si colorano inutilmente la casella corrente e quella iniziale, ma non è uno spreco degno di nota). I commenti costituiscono una buona traccia per la realizzazione del codice.

```
/* Trova il colore dell'avversario */
```

```
/* Trova la casella iniziale in direzione (dr,dc) */
/* Si sposta nella casella adiacente finche' e' nella scacchiera e
    occupata dall'avversario */
/* La mossa e' vietata se la casella finale e' fuori dalla
    scacchiera,
   o non e' occupata dal giocatore o infine e' adiacente a quella
       iniziale */
  /* Se la mossa e' lecita, torna dalla casella finale a quella
      iniziale marcando le celle */
```

Vediamo una possibile implementazione. Trovare il colore dell'avversario comporta un banale test sulla variabile `giocatore` e un assegnamento alla variabile `avversario`, come del resto avevamo già visto.

```
/* Trova il colore dell'avversario */
if (giocatore == BIANCO)
  avversario = NERO;
else
  avversario = BIANCO;
```

Un modo alternativo assegna alla variabile `avversario` il cosiddetto operatore condizionale a tre argomenti, che combina in un'espressione fra parentesi tonde una condizione logica, il simbolo `?`, un primo valore possibile (restituito quando la condizione logica è vera), il simbolo `:` e un secondo valore (restituito quando la condizione logica è falsa).

```
avversario = ( (giocatore == NERO) ? BIANCO : NERO );
```

che è certamente più compatto, ma forse meno chiaro. Quindi, si trova la casella iniziale del potenziale filotto avversario: questa casella è `(r+dr,c+dc))` (per esempio, partendo dalla casella $(2,3)$ e applicando lo spostamento $(1,0)$ si arriva alla casella $(3,3)$). Introdurremo due variabili intere e le useremo per indicare la casella corrente del filotto, che si sposterà via via sino a raggiungere la casella finale. Quindi chiameremo le due variabili `rf` e `cf`.

```
/* Trova la casella iniziale in direzione (dr,dc) */
rf = r+dr;
cf = c+dc;
```

Per eseguire lo spostamento scriveremo un ciclo `while`, in modo da poterlo terminare anche subito (come succede nel caso in cui non esista alcun filotto avversario). La condizione di permanenza nel ciclo sarà che la casella `(rf,cf)` sia interna alla scacchiera (e quindi `rf` e `cf` siano entrambe comprese fra 0 e `BOARD_SIZE-1`) e che sia occupata da pedine avversarie (quindi la matrice `S` deve contenere il colore avversario nella casella stessa). Finché queste condizioni sono valide, ci spostiamo sommando la direzione alla casella corrente.

```
/* Si sposta nella casella adiacente finche' e' nella scacchiera e
    occupata dall'avversario */
while ( (0 <= rf) && (rf < BOARD_SIZE) && (0 <= cf) && (cf <
    BOARD_SIZE) && (S[rf][cf] == avversario) )
{
  rf += dr;
  cf += dc;
}
```

Quando si esce dal ciclo, i casi sono tre: o si è usciti dalla scacchiera (e allora la mossa è illecita) o si è trovata una cella libera (e ancora la mossa è illecita) o si è trovata una cella col proprio colore (e la mossa è lecita). Nei primi due casi, si esce restituendo `FALSE`. Nel terzo caso, bisogna tornare alla casella iniziale, muovendosi da quella finale nella direzione `(-dr,-dc)`, ma ci sono ancora due casi: che la casella finale sia subito adiacente a quella iniziale (e dunque la mossa sia illecita e si restituisca `FALSE`) o che ci siano altre caselle intermedie, e allora bisogna marcare le caselle incontrate lungo il percorso col proprio colore e restituire `TRUE`.

```
/* La mossa e' vietata se la casella finale e' fuori dalla
    scacchiera,
   o non e' occupata dal giocatore o infine e' adiacente a quella
       iniziale */
if ( (rf < 0) || (rf >= BOARD_SIZE) || (cf < 0) || (cf >=
    BOARD_SIZE) ||
     (S[rf][cf] != giocatore) || ( (rf == r+dr) && (cf == c+dc) ) )
   return FALSE;
else
{
  /* Se la mossa e' lecita, torna dalla casella finale a quella
      iniziale marcando le celle */
  while ( (rf != r) || (cf != c) )
  {
    S[rf][cf] = giocatore;
    rf -= dr;
    cf -= dc;
  }

  return TRUE;
```

Terminato il ciclo, siamo nella cella `(r,c)` che è ancora libera: abbiamo lasciato alla funzione chiamante il compito di marcarla (anche perché la funzione viene ripetuta su ogni direzione, e quindi è poco sensato marcare la casella iniziale ogni volta.

4.9 Sesta fase

Rimane da riempire la funzione `VerificaSospensione`, che avevamo lasciato da parte perché è usata solo nella circostanza piuttosto rara in cui un giocatore è costretto a passare la mano dal fatto di non avere mosse lecite. Per verificare che la sospensione sia corretta dobbiamo verificare che il giocatore corrente non abbia alcuna mossa a disposizione. Il modo più semplice per farlo è scorrere tutte le caselle libere e per ciascuna scorrere le direzioni fino a trovarne una che permetta delle catture (nel qual caso la sospensione è scorretta) o mostrare che nessuna ne permette (nel qual caso, si passa alla casella seguente). Solo quando tutte le caselle libere e tutte le direzioni non consentono alcuna cattura si può terminare confermando la validità della sospensione. I commenti che descrivono quanto sopra mostrano chiaramente quanto queste operazioni hanno in comune con quelle della fase precedente.

QUI I COMMENTI MANCANO: BISOGNA AGGIORNARE IL CODICE (E QUINDI CORREGGERE TUTTI I LISTATI)

Una soluzione possibile è eseguire un doppio ciclo sulle coordinate delle caselle, verificare per ciascuna l'essere libera; in caso positivo, per ciascuna casella si esegue un doppio ciclo sulle direzioni possibili (escludendo lo spostamento nullo). Per ciascuna direzione, possiamo sfruttare la funzione `EsegueDirezione`, che effettivamente valuta la correttezza di una mossa rispetto a una specifica direzione.

Potrebbe essere un problema il fatto che la funzione, in caso positivo, esegue anche la mossa, mentre noi vogliamo semplicemente valutarla. Però il caso positivo implica che la sospensione è illecita, e quindi l'intero programma va terminato dopo aver stampato un messaggio di errore significativo. Possiamo quindi usare la funzione, anche se in teoria sarebbe più corretto dividere la valutazione dall'esecuzione della mossa per evitare di "sporcare" la scacchiera. Ne abbiamo già parlato in precedenza: a meno di voler stampare la scacchiera subito dopo la sospensione, non sembra necessario introdurre queste sofisticazioni.

```
for (r = 0; r < BOARD_SIZE; r++)
  for (c = 0; c < BOARD_SIZE; c++)
    if (S[r][c] == LIBERO)
      for (dr = -1; dr <= 1; dr++)
        for (dc = -1; dc <= 1; dc++)
          if ( (dr != 0) || (dc != 0) )
            if (EsegueDirezione(r,c,dr,dc,giocatore,S) == TRUE)
            {
              fprintf(stderr,"Il giocatore %c non e'
                  bloccato!\n",giocatore);
              exit(EXIT_FAILURE);
            }
```

Per verificare la correttezza del nuovo codice, si può applicare il programma aun file di mosse in cui compaia una mossa di sospensione spuria.[14]

4.10 Settima fase: compilazione condizionale

L'esercizio è sostanzialmente concluso, ma è possibile aggiungere una coda, nella quale si passa alla modalità *bottom-up*, ragionando sulla possibilità di sfruttare tutto il codice realizzato per consentire algoritmi più sofisticati (per esempio, un algoritmo che giochi a Othello). Si tratta anche di rendere la scacchiera una vera e propria struttura dati astratta, in modo che l'utente non abbia bisogno di sapere come è implementata (anche se la matrice bidimensionale statica è veramente semplice e naturale), ma possa usare scacchiere attraverso funzioni opportune, senza sapere in che cosa consistono. L'idea è costruire una *libreria*, cioè una coppia di file (intestazione e codice).

I file `scacchiera.h` e `scacchiera.c` fanno proprio questo. Il primo contiene la definizione della struttura dati, le direttive che specificano le costanti simboliche per i colori e la dimensione della scacchiera, i prototipi delle funzioni per manipolare scacchiere, ecc ... Il programmatore non ha bisogno di conoscere queste informazioni: gli bastano i nomi simbolici. Per farlo, basta spostare tutto ciò che è dichiarazione di tipi e di costanti in un file intestazione, da includere nel programma che si va stendendo. Tutto ciò che è codice, invece, cioè le definizioni delle funzioni, va nell'altro file. È abbastanza arbitrario scegliere quali funzioni meritino di entrare nella libreria. Certamente l'inizializzazione, che è fondamentale per poter usare una scacchiera, e ancora la stampa di una scacchiera e dell'esito di una partita. Probabilmente, le funzioni di analisi delle singole mosse e direzioni sono anch'esse di livello sufficientemente basso. La funzione `Othello`, invece, potrebbe essere troppo sofisticata e specifica per comprenderla nella libreria, cioè metterla a disposizione di altri. Quindi, potrebbe rimanere nell'algoritmo.

A questo punto, il programma è diviso in tre file che si devono "comunicare" l'informazione di cui hanno bisogno per poter superare la terza fase della compilazione, quella di collegamento, che fonde i diversi moduli binari ottenuti dalla

[14] Ne andrebbe aggiunto uno apposta.

compilazione dei singoli file di codice. Per prima cosa, il file `scacchiera.c` contiene brani di codice che danno per scontate la definizione di `Scacchiera` e parecchie costanti simboliche. Per poterlo fare, occorre adottare una direttiva `#include` che includa `scacchiera.h` in `scacchiera.c`. Diversamente dalle inclusioni di librerie standard, che ne racchiudono i nomi fra `<` e `>`, quelle di librerie fornite dall'utente ne racchiudono i nomi fra virgolette. Anche `othello7.c` ha lo stesso problema, per cui anch'esso deve includere `scacchiera.h`. La cosa non riguarda solo la libreria per la gestione della scacchiera, ma anche le librerie fondamentali, `stdio.h` e `stdlib.h`, di cui entrambi i file di codice hanno bisogno, e che quindi devono includere.

Ora il problema è che il contenuto dei file inclusi compare due volte per intero nel progetto, e questo produce una duplicazione delle dichiarazioni, che il compilatore non è in grado di riconoscere come perfettamente identiche, dunque coerenti. Si avrà quindi un messaggio di errore, e il compilatore non è in grado di produrre l'eseguibile nella terza fase del processo. D'altra parte, compilare un solo file fallisce la seconda fase di compilazione perché nessuno dei due contiene da solo tutte le informazioni necessarie[15]. Come evitare questo problema?

Occorre un meccanismo che consenta al compilatore di includere una volta sola il file `scacchiera.h`. Si potrebbe includerlo solo in `scacchiera.c` o solo in `othello7.c`, ma non è facile decidere quale dei due, procedere per tentativi diventa costoso quando i file di codice e di intestazione sono molti e in alcuni casi potrebbe anche essere impossibile o molto complicato ottenere un tutto coerente.

La soluzione standard usa un meccanismo di *compilazione condizionale* che consente di includere o non includere un file a seconda che sia verificata o no una data condizione. L'idea è racchiudere l'intero file di intestazione `scacchiera.h` (per le librerie di sistema, la cosa viene fatta da chi le scrive) fra due direttive, `#ifndef` e `#endif`, che nel complesso costituiscono una direttiva condizionale. La direttiva `#ifndef` deve essere seguita da un nome simbolico, che di solito è il nome del file di intestazione, preceduto da un doppio *underscore*, e con il punto che separa nome ed estensione da un altro *underscore*. Per esempio, `scacchiera.h` diventa `__scacchiera_h`. Questa è una pura convenzione, volta a creare nomi che probabilmente l'utente non sceglierà mai di usare per altri scopi nel resto del codice. Il significato del costrutto è il seguente: se il simbolo è non definito, allora si include ciò che segue fino a `#endif`; altrimenti, si evita di farlo. Ora, per garantire che il corpo del file venga incluso una volta sola, basta definire il simbolo stesso come prima cosa, cioè far seguire a `#ifndef __scacchiera_h` una direttiva `#define __scacchiera_h`. Notiamo che, al contrario delle definizioni che abbiamo visto sinora, non assegniamo alcun valore al nome simbolico, perché il nome stesso non verrà mai usato, tranne che come segnaposto. Da quel punto in poi, il nome risulta definito nella tabella dei simboli, e quindi tutte le inclusioni successive (qualunque sia il file che le contiene) si arrestano alla direttiva `#ifndef` e saltano il corpo del file fino all'`#endif` finale.

Riassumendo, la compilazione condizionale garantisce che alla prima inclusione si includa l'intero file e si definisca il nome simbolico, che in tutte le successive inclusioni garantisce di non includere più il file stesso. Quindi, il file viene incluso esattamente una volta, e non ci sono contenuti duplicati. Di tutto questo si fa carico il precompilatore: le fasi seguenti della compilazione non se ne avvedono.

```
#ifndef __SCACCHIERA_H
#define __SCACCHIERA_H

typedef int boolean;
#define TRUE  1
#define FALSE 0
```

[15] Bisognerebbe aggiungere i messaggi di errore e qualche commento.

```
#define BOARD_SIZE   8
#define BIANCO       'B'
#define NERO         'N'
#define LIBERO       '.'
#define MOVE_LENGTH  2


/* Prototipi delle procedure secondarie */

typedef char Scacchiera[BOARD_SIZE][BOARD_SIZE];

/* Costruisce la configurazione iniziale della scacchiera S */
void InizializzaScacchiera (Scacchiera S);

/* Stampa la scacchiera S */
void StampaScacchiera (Scacchiera S);

/* Stampa l'esito del gioco sulla scacchiera S */
void StampaEsito (Scacchiera S);

/* Esegue, se possibile, la mossa per il giocatore sulla scacchiera */
void EsegueMossa (char *mossa, char giocatore, Scacchiera S);

/* Verifica se la posizione corrente sulla scacchiera S e' davvero di
    sospensione per il giocatore */
void VerificaSospensione (Scacchiera S, char giocatore);

#endif
```

```
#ifndef __SCACCHIERA_H
#define __SCACCHIERA_H

typedef int boolean;
#define TRUE  1
#define FALSE 0

#define BOARD_SIZE   8
#define BIANCO       'B'
#define NERO         'N'
#define LIBERO       '.'
#define MOVE_LENGTH  2


/* Prototipi delle procedure secondarie */

typedef char Scacchiera[BOARD_SIZE][BOARD_SIZE];

/* Costruisce la configurazione iniziale della scacchiera S */
void InizializzaScacchiera (Scacchiera S);

/* Stampa la scacchiera S */
void StampaScacchiera (Scacchiera S);

/* Stampa l'esito del gioco sulla scacchiera S */
void StampaEsito (Scacchiera S);

/* Esegue, se possibile, la mossa per il giocatore sulla scacchiera */
void EsegueMossa (char *mossa, char giocatore, Scacchiera S);

/* Verifica se la posizione corrente sulla scacchiera S e' davvero di
    sospensione per il giocatore */
void VerificaSospensione (Scacchiera S, char giocatore);
```

```
#endif


/* othello.c */

/* Direttive */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "scacchiera.h"

#define ROW_LENGTH 256


/* Prototipi delle procedure secondarie */

void InterpretaLineaComando (int argc, char *argv[], char *filedati);

/* Esegue le mosse indicate nel file filemosse */
void Othello (Scacchiera S, char *filemosse);


/* Programma principale */
int main (int argc, char *argv[])
{
  /* Parte dichiarativa */
  char filemosse[ROW_LENGTH];
  Scacchiera S;

  /* Parte esecutiva */
  InterpretaLineaComando(argc,argv,filemosse);

  /* Costruisce la configurazione iniziale della scacchiera */
  InizializzaScacchiera(S);

  /* Esegue le mosse indicate nel file filemosse */
  Othello(S,filemosse);

  /* Stampa la scacchiera finale */
  StampaScacchiera(S);

  /* Stampa l'esito del gioco */
  StampaEsito(S);

  return EXIT_SUCCESS;
}


/* Definizione delle procedure secondarie */

void InterpretaLineaComando (int argc, char *argv[], char *filemosse)
{
  if (argc != 2)
  {
    fprintf(stderr,"Errore nella linea di comando!\n");
    exit(EXIT_FAILURE);
  }

  strcpy(filemosse,argv[1]);
}


/* Esegue le mosse indicate nel file filemosse */
void Othello (Scacchiera S, char *filemosse)
{
  FILE *fp_filemosse;
  char mossa[MOVE_LENGTH+1];
```

```
  char giocatore;


  fp_filemosse = fopen(filemosse,"r");
  if (fp_filemosse == NULL)
  {
    fprintf(stderr,"File %s non trovato!\n",filemosse);
    exit(EXIT_FAILURE);
  }

  /* Definisce il giocatore che muove per primo */
  giocatore = NERO;

  /* Finche' ci sono mosse, ne legge una */
  while (fscanf(fp_filemosse,"%s",mossa) == 1)
  {
    /* Se e' una sospensione, valuta se davvero non ci sono mosse
       lecite */
    if (strcmp(mossa,"--") == 0)
      VerificaSospensione(S,giocatore);
    else
    /* Se e' una mossa regolare, valuta se e' corretta e la esegue */
      EsegueMossa(mossa,giocatore,S);

    /* Passa la mano all'altro giocatore */
    if (giocatore == BIANCO)
      giocatore = NERO;
    else
      giocatore = BIANCO;
  }

  fclose(fp_filemosse);
}
```

4.11 Stendere una relazione su un algoritmo

INTRODUZIONE

Per risolvere il problema descritto nella prima sezione, l'algoritmo inizializza la scacchiera, poi esegue le mosse indicate nella sequenza (in termini matematici, è una sequenza: che stia in un file nel disco è un dettaglio irrilevante), poi stampa la scacchiera e infine stampa l'esito finale del gioco.

Dovè finita l'interpretazione della linea di comando? Non c'è, perché il fatto che l'algoritmo prenda i dati da un file di testo è irrilevante. Invece interessa che la scacchiera vada inizializzata, e così via. A questo punto, bisogna descrivere come si inizializza la scacchiera, come si eseguono le mosse, come si stampa la scacchiera e come si stampa l'esito finale del gioco.

Procederemo *top-down*, esattamente come nel progetto: avendo descritto brevemente che cosa le quattro funzioni che risolvono i sottoproblemi, possiamo ora dedicare loro una sottosezione a testa: una dedicata all'inizializzazione della scacchiera, una dedicata all'esecuzione delle mosse, una dedicata alla stampa della scacchiera e una dedicata alla stampa dell'esito. Il buon senso e il gusto personale potrà poi suggerirci che alcune di queste sottosezioni sono sproporzionate: se bastano poche righe a spiegare in dettaglio come si inizializza la scacchiera (e quanto costa), non occorre una sottosezione esplicita. Occorre menzionare esplicitamente cicli, assegnamenti e quant'altro? Ovviamente no: l'algoritmo non è la sua implementazione.

L'esecuzione delle mosse, invece, merita una sottosezione, perché descriverla in dettaglio porta via molto spazio e il lettore della relazione vuole avere in breve

(mezza pagina, diciamo) l'idea di come l'algoritmo risolve il problema, e per ottenere questo risultato non si può entrare subito nel dettaglio su tutto. Quindi nella descrizione generale si potrà entrare nel dettaglio conclusivo per i sottoproblemi semplici e si dovrà sintetizzare e rimandare a una sottosezione successiva per i sottoproblemi complessi.

Una volta spiegato a grandi linee l'algoritmo, dato che i sottoproblemi al livello superiore si scambiano l'un l'altro una scacchiera è opportuno spiegare che cosa sia una scacchiera. Questo perché la struttura dati concreta (l'implementazione) della scacchiera determina il costo spaziale e temporale delle operazioni.

Una scacchiera è un vettore bidimensionale di caratteri, indicizzato da 0 a 7 sia sulle righe sia sulle colonne (o, più semplicemente, di 8 righe e 8 colonne). Non avrebbe senso presentare la scacchiera prima di dire a che cosa serve. Spiegarlo dopo aver descritto a grandi linee l'algoritmo rende invece ovvio a che cosa serve e più naturale la scelta dell'implementazione: perché un vettore di caratteri? Perché occupa lo spazio minimo strettamente necessario, perché consente lettura e scrittura con il costo minimo e perché consente la stampa in modo semplice ed efficiente.

FINE: COME SI FA LA RELAZIONE

****

4.12 Laboratorio

La lezione è centrata su un esercizio per impratichirsi sui seguenti argomenti[16]:

- gestione di vettori statici multidimensionali

Chi notasse eventuali incoerenze o errori, oppure avesse dubbi sul contenuto di queste pagine e dei codici, è pregato di segnalarmeli per contribuire a migliorare la qualità dei materiali del corso.

4.12.1 Problema

Othello è un gioco da tavolo. Si gioca in due, su una scacchiera quadrata di 8 caselle per lato, le cui righe sono associate alle cifre da 1 a 8, mentre le colonne sono associate alle lettere da *a* ad *h*. I giocatori usano pedine con un lato nero e l'altro bianco: il giocatore Nero le posa sulla scacchiera col lato nero volto verso l'alto, il giocatore Bianco le posa col lato bianco verso l'alto. Si parte da una configurazione in cui le caselle *d4* ed *e5* sono occupate da pedine bianche, mentre le caselle *d5* ed *e4* sono occupate da pedine nere. Muove per primo il giocatore Nero. Ogni giocatore deve catturare almeno una pedina di colore opposto: per farlo, la nuova pedina e una pedina vecchia dello stesso colore devono racchiudere fra loro una sequenza ininterrotta (orizzontale, verticale o diagonale) di una o più pedine del colore opposto. Al termine della mossa, tutte le pedine della sequenza risultano catturate e vengono girate in modo da assumere il colore del giocatore che ha appena mosso. Con una sola mossa si possono catturare più sequenze di pedine contemporaneamente, se vi sono diverse pedine vecchie che formano sequenze utili con la nuova. Se un giocatore non ha mosse lecite, passa il turno. Se entrambi non hanno mosse lecite (per esempio, perché la scacchiera è interamente piena), il gioco si conclude con il conteggio delle pedine: vince il giocatore che ha più pedine del proprio colore. La figura 4.16 indica la configurazione iniziale della scacchiera e le mosse lecite per il giocatore Nero; la figura 4.17 indica la configurazione raggiunta dopo l'esecuzione della mossa *d3* e le mosse lecite per il giocatore Bianco.

Formato di ingresso e di uscita Si scriva un programma `othello.c` per valutare la correttezza di una sequenza di mosse di Othello indicata in un file di testo, eseguirle se corrette e indicare la situazione al termine della sequenza di mosse.

Il file di ingresso codifica le mosse secondo il seguente formato. Una mossa regolare corrisponde a una lettera compresa fra `a` e `h`, che indica la colonna, immediatamente seguita da una cifra compresa fra `1` e `8`, che indica la riga. Due trattini consecutivi indicano che un giocatore passa il turno perché non ha mosse lecite. Le mosse sono separate da spazi, tabulazioni o a capi. Ad esempio:

```
c4 e3 f2 b4 d3 c5
a4 e2 f1 e1 d1 g1
h1 d2 e6 c3 d6 f3
b2 f4 g4 g3 h3 f5
f6 g2 c1 c2 h2 --
```

La stampa a video segue invece questo formato. Se la sequenza di mosse non è corretta, si deve stampare a video la mossa non lecita e il giocatore a cui è

[16] Questa traccia va integrata con le sezioni precedenti, che trattano lo stesso argomento in modo prolisso.

stata attribuita. Altrimenti, si stampa la configurazione finale della scacchiera: la prima riga contiene i caratteri che corrispondono alle colonne; ciascuna riga seguente comincia con la cifra associata, seguita da 8 caratteri che descrivono lo stato di ogni casella. Se la casella contiene una pedina nera, si stampa il carattere `N`; se ne contiene una bianca, il carattere `B`; se è vuota, il carattere `.` (punto). Alla configurazione segue una riga vuota, una riga che fornisce il numero corrente di pedine nere seguito da `pedine nere` e una riga che fornisce il numero corrente di pedine bianche seguito da `pedine bianche`. Il tutto si conclude con l'indicazione dell'esito della partita, che può essere `Vince il Nero`, `Vince il Bianco` o `Pareggio`. La partita precedente è corretta e si conclude con la vittoria del Nero per 33 pedine a 0. Il programma dovrà quindi stampare a video:

```
 abcdefgh
1..NNNNNN
2.NNNNNNN
3..NNNNNN
4NNNNNNN.
5..NNNN..
6...NNN..
7........
8........

33 pedine nere
0 pedine bianche
Vince il Nero
```

Traccia della risoluzione

Prima di eseguire l'esercizio, conviene scorrere i lucidi della lezione 4 sulle strutture dati astratte, sulla definizione astratta e sulle implementazioni in C dei vettori e dei record.

Il punto di partenza dell'intero esercizio è il file `othello0.c`, che già contiene:

1. un `main` con la variabile `filemosse` che conserva il nome del file delle mosse;
2. la funzione `InterpretaLineaComando` che riceve il nome dalla linea di comando (`argv[1]`) e lo copia nella variabile;
3. le direttive per simulare il tipo `boolean`;
4. l'istruzione `typedef char Scacchiera[BOARD_SIZE][BOARD_SIZE];` e le costanti simboliche `BOARD_SIZE`, `BIANCO`, `NERO` e `LIBERO` per gestire la scacchiera;
5. la costante simbolica `MOVE_LENGTH` per gestire le mosse.

Tratteremo infatti la scacchiera come una struttura dati astratta, dotata di un suo nome, di funzioni con le quali operare su di essa (per esempio, per leggerla, stamparla a video, modificarla, valutare il vincitore, ecc...) e di costanti simboliche per poterne modificare le caratteristiche senza dover riscrivere l'intero codice (per esempio, per cambiare i simboli usati per i due giocatori, o le dimensioni della scacchiera). Contrariamente a quanto suggerito sui lucidi, indicizziamo righe e colonne da 0 a 7, anziché da 1 a 8, perché le mosse vengono rappresentate da

coppie di caratteri e risulta quindi comodo sfruttare l'"aritmetica dei puntatori" per convertire la lettera che indica una riga nell'indice corrispondente (sottraendole la `'a'`) e la cifra che rappresenta ogni colonna nell'indice corrispondente (sottraendole la cifra `'1'`), e viceversa.

Prima fase (`othello1.c`) Si avvia la realizzazione *top-down* dell'algoritmo scrivendo opportune funzioni e aggiungendo al codice la loro chiamata nel `main`, la dichiarazione fra i prototipi prima del `main`, la definizione dopo il `main`:

1. inizializzazione della scacchiera
2. esecuzione (e controllo) delle mosse indicate nel file
3. stampa della scacchiera finale
4. stampa dell'esito del gioco

Inoltre, si aggiunge al `main` la dichiarazione della scacchiera `S`, in modo da definire completamente i passaggi di dati e risultati da una procedura all'altro.

Seconda fase (`othello2.c`) Ora, vincendo la tentazione di scrivere il codice dall'inizio alla fine, procederemo invece "a cipolla", cioè dalle procedure più esterne a quelle più interne. In altre parole, cominceremo realizzando:

- l'inizializzazione della scacchiera, dunque l'operazione di sostituzione (scrittura): usare le costanti simboliche consente di scrivere codice valido per scacchiere di dimensione varia e rappresentate con simboli vari;
- stampa della scacchiera: richiede la banale operazione di proiezione (lettura), ma si può complicare la stampa aggiungendo una cornice con le lettere associate alle colonne e le cifre associate alle righe: per determinare la lettera e la cifra associata si può usare l'*aritmetica dei puntatori*;
- stampa dell'esito del gioco: richiede un conteggio delle caselle bianche e nere, e un confronto del risultato dei due conteggi.

Procedendo "a cipolla", ci si libera fin dal principio dei compiti più elementari, e soprattutto si comincia subito a verificare che le strutture dati svolgano il loro compito correttamente: infatti, una volta scritte queste funzioni si potrà constatare a video che la scacchiera è correttamente costruita e che la valutazione della configurazione (2 pedine nere e 2 pedine bianche, dunque un pareggio) è corretta.

Terza fase (`othello3.c`) Si affronta quindi il corpo principale dell'algoritmo, cioè la lettura delle mosse da file di testo, la valutazione della loro correttezza e la loro esecuzione sulla scacchiera. Procedendo *top-down*, aggiungiamo

- l'apertura e chisura del file (`fopen` e `fclose`);
- il ciclo di lettura, `while`, a condizione iniziale, perché il file potrebbe anche essere vuoto, basato sul *parsing* del file una parola alla volta con la specifica `%s` e terminato quando la funzione `fscanf` non riconosce più un valore nello *stream*;
- la valutazione della mossa deve distinguere due casi:

1. sospensione (mossa `--`): si deve valutare se il giocatore è veramente bloccato. Se lo è, non si deve fare nulla; altrimenti, si deve interrompere il programma con un messaggio di errore.
2. mossa regolare (lettera e cifra): si deve ricavare dalla mossa gli indici della riga e colonna dove inserire la nuova pedina e valutare se la mossa è lecita. Se lo è, si deve eseguire la mossa; altrimenti, si deve interrompere il programma con un messaggio di errore.

Ovviamente, fare tutto questo è troppo pesante, per cui si sostituisce ogni operazione complessa con la chiamata (e quindi la dichiarazione e definizione) di un'apposita funzione. Via via che le mosse vengono lette, le stamperemo (`printf("%s\n",mossa);`) per verificare che la lettura sia corretta (poi cancelleremo la stampa, che non è richiesta). Infine, siccome queste operazioni richiedono di conoscere il giocatore corrente, definiremo una variabile (di tipo `char`) per conservare e aggiornare questa informazione, inizializzandola a `NERO`, e assegnandole ad ogni mossa il valore complementare a quello corrente.

Quarta fase (`othello4.c`) La scelta tra le due funzioni vuote ancora da definire è abbastanza arbitraria. Si potrebbe cominciare dalla più semplice, che è forse la verifica della sospensione, ma partiremo con l'esecuzione delle mosse, perché ha il vantaggio di poterne immediatamente valutare la correttezza eseguendola sui dati disponibili. Per eseguire una mossa bisogna prima verificare se sia lecita. Questo comporta anzi tutto che la casella sia libera; quindi, che la mossa porti a catturare pedine avversarie. Per verificare questa seconda condizione bisogna considerare la casella della scacchiera indicata nella mossa e scorrere ciascuna direzione orizzontale, verticale e diagonale. Nelle direzioni in cui la verifica ha esito positivo, si può eseguire fisicamente la mossa cambiando il colore delle pedine intermedie catturate. Una possibilità è definire una funzione che verifichi la mossa in una specifica direzione e una che la esegua; la seconda viene chiamata solo quando la prima ha dato esito positivo. Però le due funzioni sono molto simili, e il gioco richiede di eseguire la mossa in ogni direzione che consenta delle catture. Di conseguenza, preferiremo realizzare una sola funzione, che valuta la direzione data, in caso positivo esegue la mossa, e restituisce un valore logico che segnala se si sono catturate pedine avversarie oppure no. Per il momento, lasciamo questa funzione vuota, facendole restituire il valore `FALSE` come valore di *default*, da correggere in seguito, in modo che il codice sia sintatticamente corretto. Per scorrere le direzioni, una buona idea è di definirle, come in fisica, come vettori di spostamento, le cui componenti (sulle righe e sulle colonne) possono essere pari a -1, 0 o 1. Combinando i tre valori sulle due dimensioni, si ottengono 9 vettori, di cui quello interamente nullo va ovviamente scartato. Questo consente di scorrere le direzioni con un doppio ciclo e di indicare alle funzioni di verifica e di esecuzione il vettore di spostamento con una coppia di componenti. Infine, bisogna tener traccia del fatto che almeno una direzione abbia dato esito positivo, per controllare che la mossa fosse effettivamente lecita.

Quinta fase (`othello5.c`) Nella fase seguente scendiamo al livello della verifica ed esecuzione delle mosse nelle singole direzioni. Dato il colore del giocatore che muove, si determina il colore dell'avversario. Si deve partire dalla casella immediatamente seguente a quella indicata dalla mossa, dunque passare da `(r,c)` a `(r+dr,c+dc)`. Da lì, ci si sposta, sempre nella direzione data verificando che esista una sequenza di caselle occupate dall'avversario. La sequenza termina quando:

1. si esce dalla scacchiera[17];
2. si trova una casella non occupata dall'avversario (dunque, libera o occupata dal giocatore).

La sequenza è lecita quando la casella finale:

1. è interna alla scacchiera;
2. è occupata dal giocatore;
3. non è immediatamente adiacente alla casella iniziale (in tal caso, la sequenza di caselle dell'avversario sarebbe vuota).

[17]Qui si potrebbe osservare che la verifica di appartenenza alla scacchiera potrebbe essere semplificata se si conoscesse la direzione in cui ci si sta muovendo: se si sta salendo verso l'alto, non occorre verificare che la riga sia minore di quella massima e non occorre verificare la colonna. Questo accelererebbe il codice, ma costringerebbe a scrivere otto versioni diverse della verifica. Anche se l'efficienza è in generale un criterio molto importante, in questo caso il programma è già abbastanza veloce da far preferire la semplicità. Nel progetto d'esame, sarà più probabilmente il contrario.

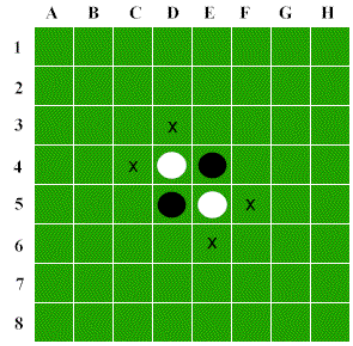


Figura 4.16: Configurazione iniziale: le crocette indicano le mosse lecite per il giocatore Nero, cioè *c4*, *d3*, *e6*, *f5*

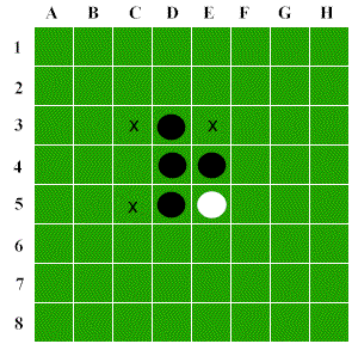


Figura 4.17: Configurazione della scacchiera dopo che il giocatore Nero ha eseguito la mossa *d3*: le crocette indicano le mosse lecite per il giocatore Bianco, cioè *c3*, *c5*, *e3*

4.13 Esercizi sull'esempio di laboratorio

Esercizio 1 Si ammetta che un giocatore possa non avere mosse a disposizione, che di conseguenza passi il turno e che questo sia rappresentato nel file di testo da due trattini consecutivi (`--`). L'algoritmo deve ora verificare che la sospensione sia corretta, cioè che davvero non ci siano mosse possibili. Lo farà scorrendo tutte le caselle libere della scacchiera e, per ciascuna, verificando che tutte le direzioni sono non lecite per il giocatore. Si tratta quindi di quattro cicli annidati, due per le caselle e due per le direzioni. Se si trova una direzione lecita, si deve stampare un messaggio di errore e uscire[18].

Esercizio 2 Si aggiunga ora anche una verifica finale che dopo aver eseguito tutte le mosse del file entrambi i giocatori siano privi di mosse disponibili. Ovviamente, la soluzione dell'esercizio precedente è particolarmente utile allo scopo.

Esercizio 3 Per approfondire e chiarire il concetto di struttura dati astratta, si estraggano dal codice realizzato in precedenza le dichiarazioni di tipo, le costanti simboliche e le funzioni dedicate a gestire la scacchiera, in modo da costruire una libreria che consenta a qualsiasi programma di gestire una scacchiera senza conoscere nulla della sua effettiva implementazione[19].

Esercizio 4 Si generi una procedura che per ciascun giocatore valuta la mossa migliore da eseguire in base al numero di pedine catturate e la esegue. In caso di parità si aggiunga un criterio di scelta qualsiasi (per esempio, l'ordine lessicografico). Si noti che questo criterio di scelta è scorretto, dato che in genere è più convienente avere poche pedine impossibili da catturare che tante soggette a cattura. Si possono utilizzare criteri più lungimiranti preferendo gli angoli alle caselle laterali, e queste a quelle interne.

[18] Una soluzione per l'esercizio è disponibile nel file `othello6.c`.

[19] Una soluzione per l'esercizio è disponibile nel file `othello7.c`, con la libreria `scacchiera.c` e l'intestazione `scacchiera.h`.

4.14 Esercizi sui vettori (statici)

Esercizio 1 Dato un numero intero `n`, si conti e si stampi a video il numero di volte che nella sua rappresentazione decimale compare ciascuna cifra da 0 a 9.

Suggerimento: Occorre un ciclo che generi le cifre calcolando i resti di successive divisioni del numero per 10 e un vettore di interi indicizzato con i numeri da 0 a 9 per contenere il numero di occorrenze di ciascuna cifra.

Esercizio 2 Si aggiunga al codice precedente la costruzione di un vettore di interi, che contenga le sole cifre ripetute più volte nel numero dato.

Suggerimento: Questo vettore ha un numero di elementi variabile e non noto a priori. Tale numero è sicuramente ≤ 10. Si definisca allora un vettore di 10 elementi, si usino solo i primi elementi per conservare le cifre e si usi una variabile intera per esprimere la lunghezza della porzione di vettore effettivamente usata.

Esercizio 3 Si scriva un programma `MATRICI.C` che costruisca due matrici quadrate di ordine 10 assegnando al generico elemento (i, j), rispettivamente, i valori $i + j$ e $i - j$. Quindi, il programma calcoli una terza matrice, prodotto delle prime due e due vettori che contengono, rispettivamente, la media aritmetica degli elementi di ciascuna riga e la media aritmetica degli elementi di ciascuna colonna. Si noti che ciò comporta una conversione da intero a reale.

Esercizio 4 Si scriva un codice che calcola e conserva in un vettore i primi `N` numeri di Fibonacci (con $N \leq 40$ per evitare l'*overflow*), per poi stamparli a video. Si stampi a video anche la sequenza dei rapporti fra due numeri consecutivi (attenzione alla conversione da intero a reale e a non eccedere i limiti del vettore).

Esercizio 5 Si scriva un codice che dichiara una scacchiera come matrice bidimensionale quadrata di caratteri composta da 8 righe e 8 colonne, e poi inizializza e stampa le scacchiere relative al gioco della dama e degli scacchi (codificando opportunamente le caselle vuote e quelle occupate dai vari pezzi).

4.15 Esercizi sui record

Esercizio 1 Si scriva un programma che definisce il tipo `rational` per rappresentare i numeri razionali, costituito da due campi `num` e `den` che conterranno il numeratore e il denominatore del numero stesso. Si definiscano funzioni che ricevono due numeri razionali e restituiscono la loro somma, differenza, prodotto, rapporto.

Nota: Questo esercizio si può complicare, chiedendo che, al termine delle operazioni, `num` e `den` siano semplificati sino a diventare primi fra loro. Per farlo, occorre calcolarne il massimo comun divisore, per esempio con l'algoritmo di Euclide.

Esercizio 2 Si scriva un programma che definisce il tipo `complex` per rappresentare i numeri complessi, costituito da due campi `r` e `i` che conterranno la parte reale e immaginaria del numero stesso. Si definiscano funzioni che ricevono due numeri complessi e restituiscono la loro somma, differenza, prodotto.

Esercizio 3 Si scriva un programma che definisce il tipo `data` per rappresentare le date, intese come terne giorno-mese-anno. Per il mese, si definisca un tipo enumerativo *ad hoc*. Si definiscano:

- una funzione che riceve una data e restituisce la posizione del giorno lungo l'anno (da 1 a 366)
- una funzione che riceve due date e restituisce -1 se la prima data precede la seconda, 0 se coincidono, $+1$ se la prima data segue la seconda

Esercizio 4 Si scriva un programma che definisce il tipo `time` contenente tre campi `ora`, `minuto` e `secondo`. Si definisca una funzione che riceve un valore `long secondi_totali`, lo interpreta come il numero di secondi trascorsi dalla mezzanotte e lo traduce in una struttura di tipo `time`.

Esercizio 5 Si scriva un programma che definisce il tipo `colore` contenente tre campi `R`, `G`, e `B`, corrispondenti all'intensità luminosa nel campo del rosso, del verde e del blu (valori compresi fra 0 e 255). Si definisca una funzione `Schiarisce()` che riceve un colore e restituisce un colore più chiaro, dividendo per 0.7 tutti i valori e arrotondandoli all'intero più vicino compreso fra 0 e 255.

Esercizio 6 Si scriva un programma che definisce il tipo `point` contenente due campi interi `x` e `y` per rappresentare un punto su un piano e il tipo `rectangle` che rappresenta un rettangolo attraverso due campi `nordovest` e `sudest` che rappresentano i due punti estremi in alto a sinistra e in basso a destra. Si definiscano funzioni che:

- dati due punti, restituisca il rettangolo da loro individuato
- dato un rettangolo, ne restituisca l'area
- dato un rettangolo e un punto, restituisca un valore `boolean` (tipo enumerativo da definire) che indichi se il punto sta nel rettangolo (bordi compresi) o fuori

Come cambierebbero tali funzioni se i punti avessero coordinate reali?

Esercizio 7 Si definiscano due tipi enumerativi: `pezzo`, con valori `RE, REGINA, TORRE, ALFIERE, CAVALLO, PEDONE, VUOTO` e `colore`, con valori `BIANCO, NERO, VUOTO`. Si definisca poi una struttura `casella`, che rappresenta una casella del gioco degli scacchi, componendo i due tipi precedenti nonché la riga e la colonna. Infine, si definisca un tipo `scacchiera` come matrice quadrata di caselle. Si scriva un programma con una funzione che inizializza le posizioni dei pezzi su una scacchiera e un'altra funzione che stampa a video la scacchiera con le posizioni correnti.

Capitolo 5

Gestione della memoria

Questo capitolo è l'ultimo dedicato al ripasso di argomenti di programmazione in linguaggio C. Si tratta degli argomenti tipicamente più ostici e causa di errori, cioè la gestione della memoria. Da un lato, si parla dell'area di memoria detta *stack*, e quindi del meccanismo con il quale si chiamano le funzioni, passando loro gli argomenti e recuperandone il risultato. Dall'altro, si parla dell'area di memoria detta *heap*, e quindi dell'allocazione di memoria dinamica, che, se non gestita bene, produce programmi errati o che si bloccano.

Per poter capire le cause di questi errori, bisogna conoscere un po' di tecnologia, in modo da poter intervenire e correggerli. Bisogna "aprire" il processore, che fino adesso abbiamo cercato di considerare come una scatola nera, che parla un linguaggio ad alto livello e trasforma dati in risultati, e considerare com'è costituita, in maniera da capire perché certe operazioni sbagliate danno comportamenti imprevisti. Vedremo un modello estremamente semplificato, perché questo non è un corso tecnologico e quindi descriveremo il tutto nel modo più astratto e semplice possibile.

La memoria di un processore può essere vista come una sequenza ordinata di N *celle* elementari, indicizzate da 0 a $N-1$ (vedi Figura **??**). Ciascuna cella ha la stessa dimensione, che viene spesso indicata come *byte*, e che spesso si suppone corrisponda a 8 *bit*, cioè 8 simboli dell'alfabeto binario, costituito da 0 e 1. Queste sono convenzioni comuni, che però non hanno nessuna necessità logica. Le celle sono ordinate in sequenza, e ognuna ha un indice, che è un numero naturale compreso fra 0 e N. Questo indice si chiama *indirizzo* della cella. Le celle della memoria contengono i dati, i risultati parziali e finali della computazione. Come avrete visto nelle lezioni di teoria, una macchina *RAM* contiene un insieme di *registri* ordinati, ciascuno dei quali è in grado di contenere un numero intero. La differenza fondamentale è che le celle il processore non sono sufficienti a contenere qualsiasi numero intero, ma hanno una dimensione ben determinata, sufficiente a contenere solo un pezzo di un dato. Se una cella corrisponde a 8 bit, può rappresentare al massimo $2^8 = 256$ diversi valori, e quindi può bastare a rappresentare un carattere con l'alfabeto ASCII. Ma non basta a rappresentare un numero intero, a meno che non siano degli `short int`, il cui ambito di estensione è limitato fra -127 e 128 o fra 0 e 255. Un numero intero occupa più di un byte. In generale, ogni variabile (che rappresenti un intero, un reale, o anche un dato composto da tanti caratteri, interi, reali, ecc. . . ed eventualmente, nel caso dei record, da dati eterogenei) occupa un certo insieme di celle. Per semplicità, le celle che contengono un dato sono consecutive, sia nel caso di un dato semplice (un singolo intero), sia nel caso di un dato composto (vettore o record). Questo semplifica la gestione, dato che conoscere la prima cella della variabile, automaticamente permette di conoscere tutte le altre

senza bisogno di avere informazioni ulteriori.

I processori organizzano la memoria in N celle

- di dimensione uniforme (tipicamente 1 byte = 8 bit)
- ordinate sequenzialmente
- dotate di un indirizzo numerico (da 0 a $N-1$)

Ogni variabile (o elemento di vettore, o campo di record)

- occupa un numero intero di celle consecutive
  che si può conoscere con la funzione `sizeof`
- ha come indirizzo quello della prima cella occupata

Figura 5.1: Modello della memoria di un processore

Nel seguito, adotteremo una serie di convenzioni tecnologiche piuttosto datate, che però servono solo per fissare le idee. Ipotizzeremo che una cella corrisponda a 1 byte composto da 8 bit, che un carattere occupi una singola cella, che un numero intero occupi 4 byte (dunque 32 bit, che consentono di rappresentare interi compresi fra circa -2 miliardi e rotti a circa 2 miliardi e rotti) e che un numero reale occupi 8 byte. In base a queste ipotesi, basta fornire l'indirizzo del primo byte che rappresenta un intero per sapere automaticamente che quel byte e i tre successivi complessivamente rappresentano tutto l'intero.

Quindi una variabile occupa un certo numero di celle consecutive, numero definito dal proprio tipo, e ha come indirizzo quello della prima cella occupata. Per conoscere il numero di celle occupate da una variabile, in C basta usare la funzione `sizeof`, che richiede come argomento fra parentesi tonde la variabile stessa, oppure il nome del tipo. Il risultato è il numero di byte occupati.

Variabile puntatore è una variabile che conserva l'indirizzo di un oggetto (variabile, elemento di vettore o campo di struttura)

Sono numeri interi, ma hanno un tipo specifico perché

- l'intervallo di definizione non è quello dei numeri interi
- le operazioni eseguibili sono diverse
- il formato di stampa è esadecimale anziché decimale

Una variabile puntatore si definisce indicando tipo e nome, come sempre, e il tipo puntatore si indica con *tipo_puntato* `*`

```
char *pc;
int *pi;
double *pd;
```

Figura 5.2: Tipi puntatore

Quindi, la memoria è una sequenza di celle. Ogni variabile occupa un insieme di celle consecutive e gli indirizzi sono gli indici numerici di queste celle. Nelle celle è possibile e utile conservare non solo dati, ma anche indirizzi di altre celle (vedi Figura 5.2).

Definizione 6 *Definiamo* variabile puntatore *una variabile che non contiene un dato, ma l'indirizzo di un dato, cioè l'indirizzo della prima cella di tale dato.*

L'indirizzo che viene puntato potrebbe essere quello di un dato semplice, di un dato composto (vettore o record) o quello di un elemento di un dato composto (ad esempio, la prima cella del terzo elemento di un vettore, la prima cella del campo `pressione` della variabile `meteo_oggi`, che contiene i dati meteo di oggi).

Abbiamo detto che gli indirizzi sono numeri interi. Tuttavia, nella definizione dei tipi di dato astratto, un tipo è definito dall'insieme dei valori rappresentabili dal tipo e dalle operazioni eseguibili su oggetti del tipo. Inoltre, un tipo di dato è concretamente definito anche da come viene implementato, cioè da quanta memoria occupa (quante celle consecutive), da come sono codificati gli oggetti di quel tipo come stringhe di bit nelle celle, e da come vengono fisicamente eseguite su tali stringhe di bit le operazioni che abbiamo stabilito di poter compiere. Questi aspetti determinano la complessità spaziale del dato e la complessità temporale di ciascuna operazione elementare sul dato. Allora, il tipo intero e il tipo puntatore, anche se sono numeri interi entrambi, non sono però lo stesso tipo, per molte ragioni. Anzi tutto, gli interi saranno positivi o negativi, compresi fra `MIN_INT` e `MAX_INT`, mentre i puntatori vanno da 0 a un valore massimo che non abbiamo ancora indicato, quindi formano un insieme completamente diverso. Inoltre, non ha senso fare sugli interi e sui puntatori le stesse operazioni: sugli interi si fanno somme, prodotti, differenze e divisioni; sui puntatori si può individuare l'indirizzo di una variabile, oppure dato un indirizzo individuare il valore della variabile. Vedremo che esiste una forma di aritmetica dei puntatori, ma che non è la stessa aritmetica dei numeri interi. Quindi, le operazioni sono diverse. Infine, anche operazioni analoghe producono risultati diversi: stampando un intero otterremo un'espressione decimale (con le cifre da 0 a 9), stampando un un indirizzo un'espressione esadecimale, cioè in base 16, che usa le cifre da 0 a 9, ma anche le lettere maiuscole da A ad F per indicare i numeri da 10 a 15. Quindi, indirizzi e interi si corrispondono almeno in parte, ma sono cose diverse.

Per dichiarare una variabile che contiene un indirizzo, si usa come sempre un'istruzione di dichiarazione che termina con un punto e virgola, e indica il nome della variabile, preceduto dal tipo dell'oggetto puntato seguito da un asterisco (vedi Figura 5.3). Per esempio, se `pc` è un puntatore a carattere, viene dichiarato scrivendo `char *pc;`, mentre i puntatori a numero intero `pi` e a numero reale con doppia precisione `pd` si dichiareranno con le istruzioni `int *pi;` e `double *pd;` Queste istruzioni di dichiarazione consistono nello scrivere in un'opportuna tabella che esiste (e verrà usato) un oggetto di nome `pc`, che occupa un certo insieme di celle di memoria e rappresenta l'indirizzo di un'altra cella di memoria, che è la prima cella di un altro oggetto. Siccome `pc` è un puntatore a carattere, l'oggetto puntato è un carattere e occupa solo la prima cella, mentre `pi` è un puntatore a intero e il suo contenuto è l'indirizzo di una cella che è la prima di quattro celle che rappresentano l'intero puntato. Nel caso di `pd`, invece, considereremo otto celle. Questo mostra l'importanza di conoscere il tipo dell'oggetto puntato: serve a sapere quali sono le celle che contano.

L'uso corretto di un puntatore è contenere l'indirizzo della prima cella di un oggetto. In realtà il compilatore C non controlla che l'uso sia effettivamente corretto: una variabile puntatore può contenere un indirizzo qualsiasi. La figura mostra

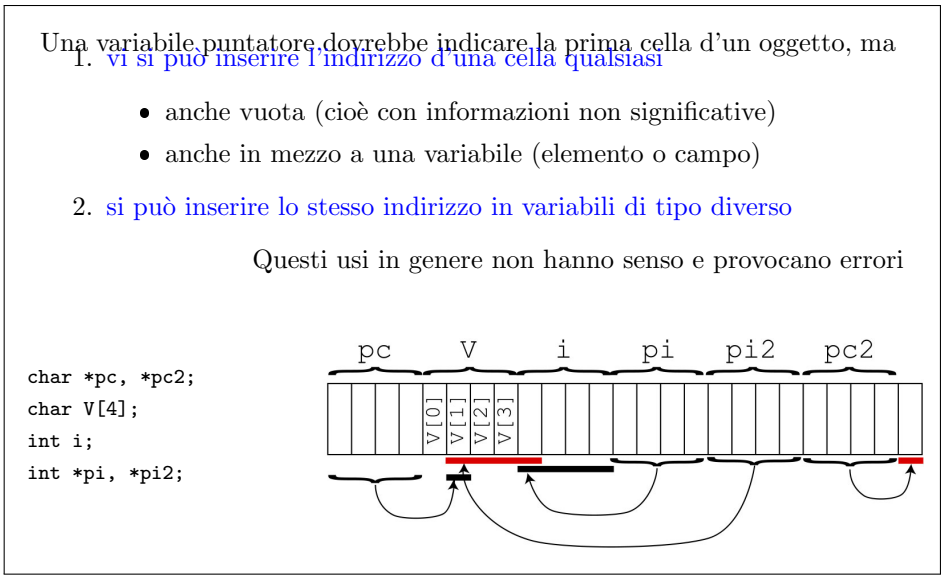


Figura 5.3: Esempi di puntatori corretti e scorretti

esempi di uso corretto e scorretto. Supponiamo di aver dichiarato alcune variabili e la figura mostra la sequenza di celle corrispondenti in memoria. Per prima cosa ci sono 4 celle occupate dalla variabile `pc`, che è un puntatore a carattere (come abbiamo detto, useremo 4 celle per i puntatori, come se il processore avesse una memoria a 32 bit, oggi se ne usano 64, ma questo renderebbe il disegno più complicato e meno leggibile. La variabile `pc` contiene nelle sue 4 celle una sequenza di 32 bit (zeri e uni), che, tradotta in un numero naturale con un'opportuna codifica, indica (e a questo corrisponde la freccina) un'opportuna cella della memoria. Questa cella contiene correttamente un carattere, precisamente il secondo carattere del vettore di caratteri `V`, che ne contiene quattro, cioè `V[0]`, `V[1]`, `V[2]` e `V[3]`, e il secondo è `V[1]`. Stiamo ipotizzando che ogni carattere occupi una singola cella. Qualche precedente operazione (che poi discuteremo) ha assegnato a `pc` l'indirizzo della cella `V[1]`. Abbiamo poi un intero `i`, che occupa 4 celle e un puntatore `i` che ne occupa altre 4. Il fatto che interi e puntatori occupino la stessa dimensione è casuale: è così solo in questo modello. La variabile `pi` punta la prima cella della variabile `i`, quindi è un indirizzo corretto: contiene l'indirizzo della prima delle quattro celle occupate da `i`. Poi c'è il primo errore: `pi2`, in quanto puntatore a intero, occupa 4 Celle il cui significato è l'indirizzo di un'altra cella. Seguendo la freccia, si scopre che questa cella è ancora `V[1]`. Non è illegale che diversi puntatori indichino la stessa cella, ma `V[1]` è un carattere, mentre `pi2` è un puntatore a intero, e questo è un errore. Lo è perché quando si prenderà l'oggetto puntato da `pi2`, automaticamente, trattandosi di puntatore a intero, si considereranno quattro celle: quella puntata e le tre successive. Quindi, si considereranno `V[1]`, `V[2]`, `V[3]` e la prima delle quattro celle di `i`. Queste quattro celle contengono una sequenza di zeri di uni che verranno interpretate come un numero intero, evidentemente sbagliando. Il compilatore non ha modo di sapere che questo è sbagliato: se qualcuno ha scritto in `pi2` un indirizzo sbagliato, l'esito può essere confuso. Infine, `pc2` è un puntatore, e quindi occupa ancora una volta quattro celle, ma essendo un puntatore a carattere, l'indirizzo che contiene viene interpretato come indirizzo di una sola cella. Si dà il caso che questa cella sia "vuota", cioè contenga zeri e uni che non hanno un significato, perché nessuno ha dato loro un significato. Sono valori casuali, dovuti alle attività precedenti

del processore. Questo significa che se cercando di usare l'oggetto puntato da `pc2` si otterrà un carattere casuale, con risultati imprevedibili.

L'operatore indirizzo (`&`) fornisce l'indirizzo di un oggetto

```
int i, j;
int *p;
p = &i;
```

L'operatore asterisco (`*`) fornisce l'oggetto puntato da un puntatore

```
j = *p;
```

Se un puntatore non è inizializzato, accedere all'oggetto puntato genera comportamenti indefiniti (valori errati, operazioni casuali, blocco)

Spesso il compilatore lo segnala come avvertimento, non come errore

Figura 5.4: Referenziazione e dereferenziazione

Vediamo ora le operazioni che si possono compiere su variabili puntatore (Figura 5.4): recuperare l'indirizzo di un oggetto e viceversa, dato un indirizzo, recuperare l'oggetto. La prima operazione si chiama *referenziazione*, cioè trovare un riferimento, e viene descritta in linguaggio con l'operatore `&`. Per esempio, data una variabile intera `i` e una variabile puntatore `pi`, è possibile assegnare alla seconda l'indirizzo della prima con l'istruzione `pi = &i;`. L'istruzione dice al processore di prendere la variabile `i`, ricavarne l'indirizzo consultando la tabella dei simboli e copiare questo indirizzo con l'assegnamento nelle celle occupate dalla variabile `pi`. La precedenza fra operatori è importante: prima si prende l'indirizzo, poi si fa l'assegnamento. Le precedenze sono regolate da regole di base che il compilatore segue. Non ripasseremo le precedenze fra operatori in questo corso, salvo sottolineare alcune precedenze che appaiono poco naturali.

L'operatore `*` descrive l'operazione opposta, cioè dato una variabile puntatore, spostarsi nella cella descritta dall'indirizzo che essa contiene, considerare il numero corretto di celle consecutive (in base al tipo dell'oggetto puntato) e tradurne il contenuto nel tipo stesso con l'operatore di assegnamento alla variabile nota. Anche qui c'è una regola di precedenza per cui la dereferenziazione precede l'assegnamento. Se la variabile puntatore è male inizializzata o non inizializzata, la dereferenziazione `j = *pi` non ha senso, e può succedere che il compilatore prenda le celle e faccia l'assegnamento sbagliato, o che il programma si blocchi. Il comportamento non è definito, perché lo standard del linguaggio C indica in modo preciso il comportamento di programmi corretti, ma non quello di programmi scorretti. Specificare il comportamento del compilatore lega le mani di chi li progetta e realizza. Aggiungere regole non necessarie impone legami inutili, perché in teoria si dovrebbe sempre scrivere codice corretto, e può impattare negativamente anche sul comportamento corretto. Per fare controlli e distinguere i comportamenti giusti da quelli sbagliati, il compilatore può diventare più lento o, peggio ancora, creare codice più lento. Chi definisce e aggiorna gli standard dei linguaggi di programmazione evita di imporre vincoli inutili e dannosi a chi progetta e realizza compilatori. Il massimo che ci si può aspettare è che il compilatore segnali con avvertimenti dei costrutti potenzialmente scorretti, se il compilatore è stato scritto in un modo sufficientemente preciso. Compilatori diversi daranno in generale avvertimenti diversi. Quindi, è possibile, e succede regolarmente nella consegna dei progetti, che codice che compila con un compilatore e funziona su una macchina, non compili su un compilatore diverso e

soprattutto non funzioni su un'altra macchina con un altro compilatore. Questo non succede nel caso di codice corretto, ma nel caso di codice non corretto. A questo problema si è già accennato nel Capitolo **??**.

5.1 Dettagli tecnici sui puntatori

Consideriamo ora una serie di dettagli di programmazione piuttosto tecnici. Da un punto di vista pratico questi dettagli non hanno un'applicazione, ma sono utili perché forniscono le basi per il fondamentale concetto dell'equivalenza fra vettori e puntatori, che è il principale concetto di questo capitolo.

La Figura 5.5 presenta una serie di considerazioni sui puntatori che vengono fatte anche nelle lezioni di teoria sulla macchina RAM, e che riprendiamo dal punto di vista della sintassi C. A differenza della figura precedente, che più correttamente rappresenta le celle tutte in sequenza, questa figura mostra per semplicità due celle a sinistra, corrispondenti a due variabili puntatore, `p` e `q`, e una sequenza di celle a destra, fra le quali due corrispondono a variabili intere `i` e `j`. Sempre per semplicità, le celle sembrano singole, anche se ciascuna variabile è in realtà un blocco opportuno di celle consecutive. Inizialmente, grazie a due operazioni di assegnamento le variabili intere `i` e `j` contengono i valori 2 e 3, mentre, grazie a due operazioni di referenziazione[1][2], la variabile puntatore `p` punta la variabile intera `i` e la variabile puntatore `q` punta la variabile intera `j`, cioè le celle del puntatore contengono l'indirizzo dell'intero.

5.1.1 Alias

In questa situazione, l'espressione `*p` significa "leggi l'indirizzo contenuto in `p`, vai nella cella corrispondente e leggine il contenuto" (vedi Figura 5.6). Da questo momento `p` diventa un *alias*, cioè un altro nome, della variabile `i`, e `q` un alias di `j`. A questo punto, se eseguiamo un assegnamento di un puntatore a un altro (`q = p;`), copiamo il contenuto della cella `p` nella cella `q`, sovrascrivendo il precedente contenuto, quindi entrambe le celle puntatore contengono l'indirizzo della variabile `i`, che ha valore 2. Quindi, automaticamente e implicitamente, dopo quell'operazione `*q` diventa anche'esso un alias di `i` e ci sono tre espressioni coincidenti. Solo `j` continua a valere 3. Questo è interessante, perché assegnare un puntatore a un altro fa perdere un riferimento alla cella precedentemente puntata. Nell'esempio, all'inizio possiamo leggere il valore di `j` sia direttamente sia attraverso `*q`; al termine, solo attraverso `j`. In certi casi, si può arrivare a commettere il classico errore di perdere tutti i riferimenti a una cella di memoria, che contiene informazioni utili. Il messaggio è fara attenzione agli assegnamenti di puntatori, per assicurarsi che non si perdano riferimenti utili.

Un caso differente è quello dell'espressione `*q = *p;`, che significa "leggi l'indirizzo contenuto in `p`, vai nella cella corrispondente e leggine il contenuto, leggi l'indirizzo contenuto in `q`, vai nella cella corrispondente e copia in essa il contenuto letto in precedenza" (vedi Figura 5.7). Nell'esempio, si copia il valore puntato da `p`, cioè `*p`, cioè 2, nella cella puntata da `q`, cioè `*q`, che è poi `j`. È un modo alternativo di scrivere `j = i;`.

[1] Va notato che le due operazioni sono `p = &i;` e `q = &j;`, mentre `int *` è un'altra operazione (a rigore due) di dichiarazione di variabile. Purtroppo, per motivi di spazio sono state combinate in modo da renderle meno leggibili.

[2] Qui sarebbe sensato spezzare le figure rispetto ai lucidi, e rendere più chiare le espressioni separandole.

Supponiamo ora che uno dei due valori venga modificato (vedi Figura 5.8). Per esempio, scriviamo 1 nella cella `*p` (oppure `i`). Entrambi assumono il valore 1, essendo alias l'uno dell'altro. D'altra parte, `j` rimane pari a 3. Diversa è la sorte di `*q`: dopo un assegnamento di puntatori, `*q` diventa alias di `i` e `*p`, e quindi il suo valore è modificato e diventa 1; dopo un assegnamento di oggetti puntati, invece, il valore di `*q` non cambia, cioè rimane pari a 2.

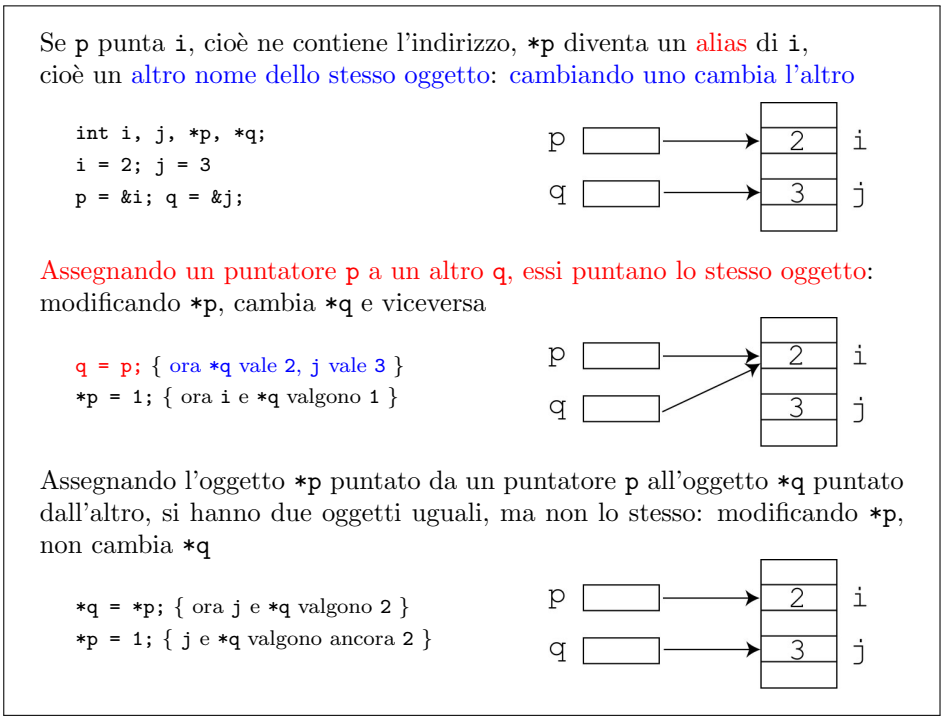


Figura 5.5: Assegnamento fra puntatori

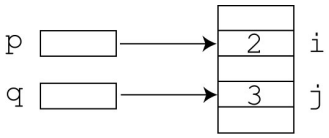
5.1.2 Aritmetica dei puntatori

Si è già osservato che i puntatori non sono numeri interi, ma si era anticipato che hanno una specie di loro aritmetica. La Figura 5.9 ne illustra gli aspetti principali.

Gli operatori di relazione binaria (uguaglianza e disuguaglianza, precedenza stretta o debole) sono applicabili anche ai puntatori. Con quale significato? Molto semplicemente, un puntatore è minore di un altro quando l'indirizzo numerico che contiene (quindi quello della cella puntata, non quello del puntatore stesso!) è minore dell'altro, cioè quando la cella puntata dal primo puntatore precede quella puntata dal secondo nella sequenza di celle che costituisce la memoria del processore. Se nella Figura **??** assumiamo che le celle siano ordinate dal basso verso l'alto, siccome la cella `i` precede la cella `j` possiamo dire che `p < q`. Nella Figura **??**, invece `p == q`. In pratica, servono solo uguaglianza e disugualianza, perché non abbiamo nessun controllo sul posizionamento dei dati nella memoria (il processore sceglie dove metterli quando esegue le operazioni di dichiarazione delle variabili). L'unico caso in cui può aver senso usare una disuguaglianza è quando si confrontano due elementi di un vettore, perché quelli sono ordinati per costruzione. Dato un vettore `V` di 10 elementi, l'indirizzo di `V[1]` e minore dell'indirizzo di `V[6]`, ovvero `&V[1] < &V[6]`. Va detto che, ovviamente, scrivere che `1 < 6` è molto più semplice e intuitivo.

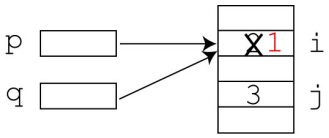
Se `p` punta `i`, cioè ne contiene l'indirizzo, `*p` diventa un alias di `i`, cioè un altro nome dello stesso oggetto: cambiando uno cambia l'altro

```
int i, j, *p, *q;
i = 2; j = 3
p = &i; q = &j;
```



Assegnando un puntatore `p` a un altro `q`, essi puntano lo stesso oggetto: modificando `*p`, cambia `*q` e viceversa

```
q = p; { ora *q vale 2, j vale 3 }
*p = 1; { ora i e *q valgono 1 }
```



Assegnando l'oggetto `*p` puntato da un puntatore `p` all'oggetto `*q` puntato dall'altro, si hanno due oggetti uguali, ma non lo stesso: modificando `*p`, non cambia `*q`

```
*q = *p; { ora j e *q valgono 2 }
*p = 1; { j e *q valgono ancora 2 }
```

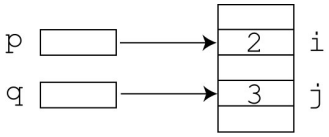
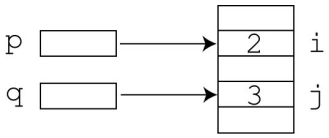


Figura 5.6: Assegnamento fra puntatori

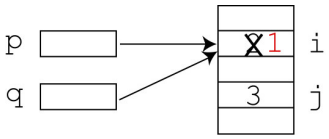
Se `p` punta `i`, cioè ne contiene l'indirizzo, `*p` diventa un alias di `i`, cioè un altro nome dello stesso oggetto: cambiando uno cambia l'altro

```
int i, j, *p, *q;
i = 2; j = 3
p = &i; q = &j;
```



Assegnando un puntatore `p` a un altro `q`, essi puntano lo stesso oggetto: modificando `*p`, cambia `*q` e viceversa

```
q = p; { ora *q vale 2, j vale 3 }
*p = 1; { ora i e *q valgono 1 }
```



Assegnando l'oggetto `*p` puntato da un puntatore `p` all'oggetto `*q` puntato dall'altro, si hanno due oggetti uguali, ma non lo stesso: modificando `*p`, non cambia `*q`

```
*q = *p; { ora j e *q valgono 2 }
*p = 1; { j e *q valgono ancora 2 }
```

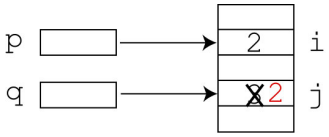


Figura 5.7: Assegnamento fra puntatori

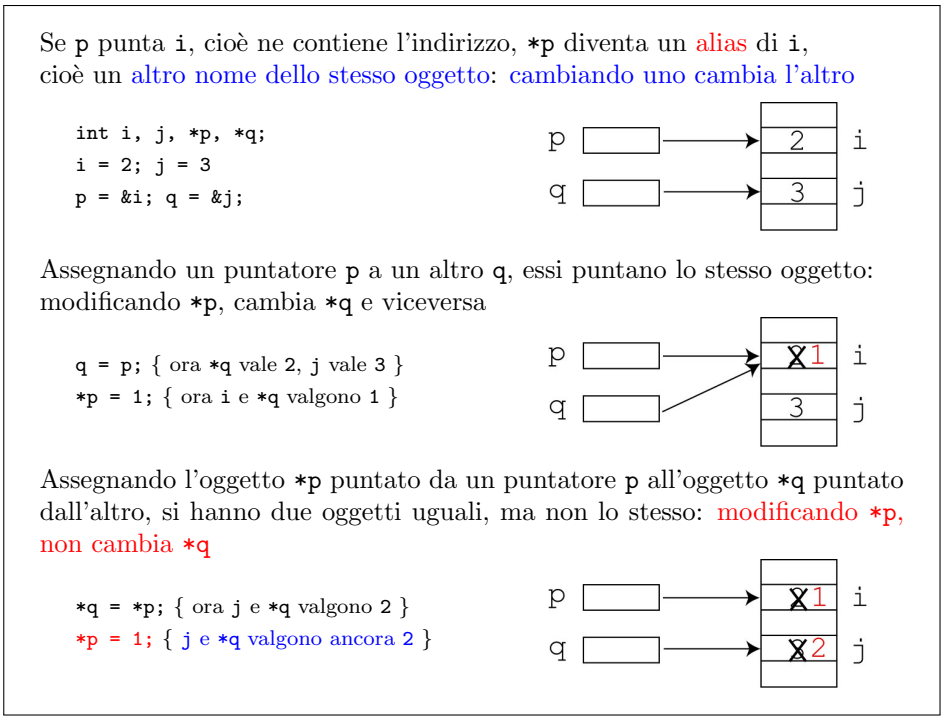


Figura 5.8: Assegnamento fra puntatori

L'aritmetica dei puntatori non ammette prodotti e divisioni, ma consente le somme, sotto certe condizioni e con un significato diverso da quello classico. Non si possono sommare puntatori, ma solo un puntatore e un intero (positivo o negativo). Il risultato della somma è un altro puntatore, ma non quello che ci si aspetterebbe. Una logica spicciola suggerirebbe che, siccome ogni puntatore è un intero, incrementarlo di 1 corrisponda a l'indirizzo immediatamente successivo, quello della cella immediatamente seguente. Per esempio, nella Figura 5.3, siccome il valore di `pi` è l'indirizzo della prima cella di `i`, il valore di `pi+1` parrebbe dover essere l'indirizzo della seconda cella di `i`. Invece, si ragiona per blocchi di celle: essendo `pi` un puntatore a intero, anche se il suo valore è l'indirizzo della prima cella, da un punto di vista concettuale punta l'intero blocco di 4 celle di `i`. Ma allora `pi+1` punta il blocco seguente, cioè il suo valore è l'indirizzo della prima cella subito dopo `i`, ovvero la quarta cella dopo la prima di `i`. Operativamente, dato `p` e la dimensione dell'oggetto puntato da `p`, il puntatore `p+1` vale l'indirizzo contenuto in `p` più la dimensione. E in generale il puntatore `p+j` si sposta avanti di `j` blocchi di celle, ciascuno della dimensione determinata dal tipo dell'oggetto puntato da `p`. Lo scopo è procedere per blocchi, con l'unità di misura data dalla dimensione dell'oggetto puntato.

Anche questo non ha vera applicazione pratica, sempre perché non c'è controllo sul posizionamento dei dati in memoria, salvo che per i vettori. Dato un vettore `V` e assegnato a un puntatore `p` l'indirizzo della cella `i` del vettore (`p = &V[i]`), l'espressione `p+j` è l'indirizzo dell'elemento di indice `i+1`, cioè `p+j = &V[i+j]`. Considerando i valori interi degli indirizzi il valore intero di `p+j` è il valore intero di `p` più `j` volte la dimensione dell'oggetto puntato, `sizeof(*p)`.

Siccome l'intero `j` può essere positivo o negativo, possiamo scorrere un vettore in avanti o all'indietro sommando un indice intero all'indirizzo di un vettore. In pratica, non si fa perché è molto più chiaro e semplice scorrere i vettori attraverso

i loro indici.

Siccome i puntatori sono in effetti numeri interi, è lecito

- applicare ai puntatori gli operatori di relazione (`== != < <= > >=`)
- sommare qualsiasi intero `j` a un puntatore `p`: si ottiene un puntatore all'indirizzo `p` spostato di `j` volte la dimensione degli oggetti puntati

`(int) (p + j)` vale `(int) p + sizeof(*p) * j`

Non è una banale somma di interi!

Questa operazione ha senso solo per puntatori a elementi di un vettore: se `p` è l'indirizzo di `V[i]`, `p + j` è l'indirizzo di `V[i+j]`

Figura 5.9: Aritmetica dei puntatori

5.2 Equivalenza fra vettori e puntatori

Lo scopo fondamentale dell'aritmetica dei puntatori è introdurre un'equivalenza fra i vettori e i puntatori (vedi Figura 5.10). Ricordiamo che un vettore in C è una sequenza di celle, organizzate in blocchi il cui numero è dato dalla lunghezza del vettore, e la cui dimensione è determinata dal tipo degli elementi del vettore: l'istruzione `int V[10];` determina la costruzione di una sequenza di 10 blocchi di 4 celle ciascuno (sempre nell'ipotesi di 4 celle per ogni `int`) e l'associazione nella tabella dei simboli dell'indirizzo della prima cella al simbolo `V`. Se si dichiara un puntatore a intero `p` e gli si assegna l'indirizzo del vettore (`p = &V;`), oppure l'indirizzo del primo blocco del vettore (`p = &V[0];`), la tabella dei simboli associa al simbolo `p` la stessa informazione associata al simbolo `V`, cioè l'indirizzo della sua prima cella.

In C è lecito scrivere addirittura `p = V;`, sempre con lo stesso significato. La cosa interessante è che da questo momento in poi può il simbolo `p` diventa un *alias* del vettore `V`, cioè può essere usato per scorrere il vettore, usando la stessa sintassi: `p[5]` indica l'elemento di indice 5 del vettore `V`. Questo è consentito dall'aritmetica dei puntatori. Si veda la Figura 5.9: l'espressione `p[5]` viene interpretata dal compilatore come se fosse `p+5`, cioè la somma del puntatore `p` (indirizzo della prima cella del vettore) e dell'intero 5, da cui si ottiene l'indirizzo `p + 5 * sizeof(int)`, cioè l'indirizzo della prima cella dell'elemento di indice 5 del vettore. Ricordiamo che anche i vettori realizzano proiezione e sostituzione allo stesso modo (vedi Sezione **??**), quindi l'equivalenza è profondamente naturale. La caratteristica specifica del linguaggio C è che questo sia accettato a livello di sintassi, cioè che l'operatore `[]` sia lecito anche per i puntatori, con lo stesso significato che vale per i vettori.

Quindi puntatori e vettori sono la stessa cosa? No, sono equivalenti, ma non uguali. Un *vettore statico* `V` ha la propria memoria dichiarata e allocata dal processore *staticamente*, cioè nel momento in cui si compila l'istruzione di dichiarazione `int V[10];`, eventualmente anni prima di eseguire il codice. Un puntatore `p` ha solo una cella di memoria in cui è contenuto un indirizzo. Se in questa cella si scrive un indirizzo corretto, assegnato a strutture dichiarate altrove e contenenti informazioni

sensate, bene; se ci si scrive un indirizzo sbagliato (o nessun indirizzo, dunque se il contenuto è casuale), invece si avranno problemi.

Questo rende anche evidente il motivo per cui è lecito scrivere `p = V;`, ma non è lecito scrivere `V = p;`: la prima espressione contiene una conversione implicita per cui nella cella `p` si scrive l'indirizzo della prima cella di `V`, ma la seconda richiede di assegnare a una sequenza di interi un singolo indirizzo, e non si vede che significato dare a questa operazione.

L'aritmetica dei puntatori e la struttura ordinata dei vettori implicano che un vettore e il puntatore alla cella di indice 0 sono equivalenti

`V` equivale a `&V[0]`

Si può assegnare un vettore a un puntatore di tipo compatibile

```
int i, V[10];
int *p;
p = V;
```

e da lì in poi usare il puntatore come se fosse un vettore

```
i = p[5];
```

È una conversione implicita e non è lecita la conversione contraria: scrivendo `V = p;` si ottiene un errore

Figura 5.10: Equivalenza vettori-puntatori

Ci sono tre motivi principali per cui è utile l'equivalenza fra vettori e puntatori (vedi Figura 5.12). Il primo è che consente di trattare un sottovettore come se fosse un vettore a sé stante. Supponiamo di avere un vettore `V` di `N` elementi e una funzione che riceve `V` e `N` e restituisce la somma degli elementi di `V` da `V[0]` a `V[N-1` compresi. Questa funzione può essere usata anche per sommare solo i primi elementi del vettore, per esempio da `V[0]` a `V[N/2`: basta chiamarla passandole `V` e `N/2` come argomenti. Sarebbe interessante poter sommare gli elementi di qualsiasi sottovettore, per esempio quelli della seconda metà. In questo caso, bisognerebbe riscrivere la funzione in modo che riceva l'indice iniziale e quello finale, oppure copiare la seconda metà in un vettore ausiliario e chiamare la funzione sul vettore ausiliario. In alternativa, però, possiamo assegnare a un puntatore `p` l'indirizzo del primo elemento da sommare (`p = V[N/2];`) e usare `p` come se fosse un vettore, di cui vogliamo sommare i primi `N/2` elementi.

Nell'esempio seguente, l'istruzione `int V[6];` dichiara un vettore `V` di 6 elementi, rappresentati da asterischi e indicizzati da `V[0]` a `V[5]`.

```
   V
[  *     *     *     *     *     *  ]
 V[0]  V[1]  V[2]  V[3]  V[4]  V[5]
```

Volendo sommare gli ultimi 3 elementi, anziché i primi 3 basta dichiarare un puntatore con l'istruzione `int *p;` e assegnargli l'indirizzo del primo elemento da sommare (`p = &V[3];`), ottenendo la situazione seguente:

Figura 5.11: Vettori e puntatori

- Si può trattare un sottovettore come se fosse un vettore

```
int V[N], *p, s;
s = Somma(V,N);                          somma da V[0] a V[N-1]
p = &V[N/2];
s = Somma(p,N/2);              somma da V[N/2] a V[N/2+N/2-1]
```

- Si può definire un vettore con indice iniziale diverso da 0

```
int V[D-S+1], *p;
p = &V[-S];
```

Il vettore `p` va da `p[S]` a `p[D]`

(`S` *e* `D` *possono anche essere negativi!*)

- Si può definire un vettore dinamico, cioè la cui dimensione è determinata durante l'esecuzione (*vedremo poi come*)

Figura 5.12: Applicazioni dell'equivalenza vettori-puntatori

```
   V                    p
[  *     *     *     *     *     *  ]
 V[0]  V[1]  V[2]  V[3]  V[4]  V[5]
                   p[0]  p[1]  p[2]
```

dove `p` può essere visto come un vettore e l'istruzione `Somma(p,3);` somma i suoi elementi `p[0]`, `p[1]` e `p[2]`, che coincidono con gli elementi `V[3]`, `V[4]` e `V[5]`. Questo spiega anche perché finora, passando delle stringhe a funzioni, abbiamo scritto che le funzioni avevano come argomento dei puntatori a carattere (`char *`) anziché dei vettori di caratteri. (`char []`). Le stringhe sono vettori di caratteri, ma quando vengono passate a una funzione (vedremo nel Capitolo **??**) vengono assegnate a puntatori, sfruttando l'equivalenza. Di conseguenza, all'interno della funzione non usiamo il vettore, ma un puntatore che funziona da *alias*.

Un secondo motivo per cui l'equivalenza fra vettori e puntatori è utile è che consente di gestire vettori con estremi diversi da 0 e $n-1$. Supponiamo di volere un vettore di 3 interi con indici estremi `S` e `D` (per esempio, 3 e 5). Già sappiamo che si può definire un vettore con `D+1` elementi e usare solo quelli che partono dall'indice `S`, ammesso che questo sia non negativo. Se `S` è un numero piccolo, come 3 o come il classico caso del vettore che comincia con l'indice 1, questa è la soluzione più semplice. Se però `S` è un numero molto alto (e quindi lo spreco è elevato) oppure è negativo, bisogna procedere altrimenti. Definiamo un vettore `V` della lunghezza corretta, cioè `D-S+1` (`int V[3];`) e introduciamo un puntatore (`int *p;`) a cui assegniamo l'indirizzo `&V[-S]`. Questo indirizzo può precedere o seguire quello di `V`, a seconda del segno di `S`. Nel nostro esempio, `V` ha 3 elementi, `V[0]`, `V[1]` e `V[2]`, e `q` punta un indirizzo che precede quello di `V` dello spazio occupato da 3 interi. Siccome il linguaggio C non controlla che gli indici dei vettori rispettino i limiti,

è lecito pensare che dopo `V[2]` ci siano gli elementi `V[4]`, `V[5]`, ecc... e prima di `V[0]` ci siano gli elementi `V[-1]`, `V[-2]`, ecc... Questi elementi non vanno usati, pena commettere errori in lettura e scrittura (e in esecuzione, ma questo dipende dal compilatore e dalla macchina usata), ma sintatticamente non è un errore riferirsi ad essi in un programma. Ora è facile notare che il puntatore `q` si può correttamente usare come vettore se si rimane nei limiti della memoria assegnata al vettore `V`, cioè da `p[3]` fino a `p[5]`.

```
      p                   V
      *     *     *   [   *     *     *   ]
... V[-3] V[-2] V[-1]  V[0]  V[1]  V[2]  V[3]  V[4] ...
    p[0]  p[1]  p[2]   p[3]  p[4]  p[5]  p[6]  p[7]
```

In breve, questo meccanismo consente di assegnare un'area di memoria della dimensione giusta, ma di indicizzarla come si vuole.

Vediamo un caso in cui `S` è negativo (eventualmente, anche `D` potrebbe esserlo), cioè costruiamo un vettore indicizzato da -2 a 2. Serve un'area di 5 interi, che costruiamo con la solita istruzione di dichiarazione statica `int V[5];`, e un puntatore `int *p;` al quale assegniamo l'indirizzo `&V[-S]`, cioè `&V[2]`. Come si può vedere nell'esempio, il vettore `V` fornisce esattamente l'area di memoria per un "vettore" `p` che va da `p[-2]` a `p[2]`.

```
    V           p
[   *     *     *     *     *   ]
  V[0]  V[1]  V[2]  V[3]  V[4]
  p[-1] p[-1] p[0]  p[1]  p[2]
```

Infine, abbiamo il terzo motivo per cui l'equivalenza fra vettori e puntatori è utile, cioè che consente di definire vettori la cui dimensione non è nota a priori. Finora, abbiamo simulato con puntatori dei vettori che sfruttavano aree di memoria allocate staticamente attraverso dei veri e propri vettori. In realtà è anche possibile allocare sequenze di celle in modo *dinamico*, non con istruzioni che dichiarano variabili e vengono eseguite durante la compilazione, ma con istruzioni che vengono eseguite durante il vero e proprio calcolo. Molto spesso capita che non si sappia a priori quanti elementi debba avere un vettore, perché il numero necessario dipenderà dai dati e cambierà di volta in volta. Per esempio, gli studenti iscritti a un esame non sono prevedibili quando si prepara il programma che gestisce le iscrizioni, ma solo pochi giorni prima dell'esame. E non ha senso pensare di scrivere un programma che possa gestire qualunque numero di studenti. Di conseguenza, i linguaggi di programmazione mettono a disposizione istruzioni per gestire la memoria in maniera dinamica. Le aree di memoria gestite dinamicamente funzionano esattamente come i puntatori, e possono quindi ospitare dei "vettori" simulati, detti *vettori dinamici*, che sono in realtà puntatori ai quali viene assegnato l'indirizzo della prima cella di un'area che non è condivisa con un vettore statico, ma viene assegnata per l'occasione dal processore.

Queste aree di memoria stanno in una zona, detta *heap*, completamente separata dall'area di memoria in cui stanno tutte le altre variabili, che è detta *stack* (vedi Figura 5.13). Il motivo dei due nomi è che lo *heap* (ovvero "mucchio") è organizzato in modo molto più semplice e disordinato rispetto allo *stack* (ovvero "pila"), che invece funziona in un modo estremamente rigido e ordinato che descriveremo nel Capitolo **??**.

La dimensione di queste aree viene decisa non dal programmatore mentre scrive il codice, ma dal programma mentre viene eseguito sulla base dei dati o dei risultati

parziali noti al momento. Addirittura, la dimensione può anche variare in modo totalmente libero nel corso dell'esecuzione, a seconda di come organizziamo queste aree. I capitoli seguenti vedranno altri esempi di strutture dinamiche, oltre ai vettori dinamici: liste, pile, code, alberi, ecc...

Per allocare questa memoria si usa la funzione `malloc`, che sta per "memory alloc" e ha un unico parametro che indica la dimensione, cioè il numero di celle da allocare. In genere, non le si passa direttamente tale numero (che dipende dal tipo degli oggetti che si vogliono conservare nelle celle), ma si costruisce questo numero scrivendo esplicitamente il prodotto del numero di blocchi del vettore moltiplicato per la dimensione di ciascun blocco, la quale viene determinata dalla funzione `sizeof`. Per esempio, per allocare lo spazio per contenere un vettore di 10 interi non si scrive `malloc(40);`, ma `malloc(10 * sizeof(int));`, per semplice motivo che macchine diverse potrebbero assegnare un numero di celle diverso a un intero. In questo modo, ogni processore allocherà la dimensione corretta per il proprio specifico funzionamento.

È costume far precedere la chiamata a `malloc` da un'istruzione detta di *cast*, costituita da una coppia di parentesi tonde che racchiudono il tipo del puntatore di destinazione a cui va assegnata l'area di memoria. Lo scopo dell'istruzione è convertire esplicitamente il risultato della funzione `malloc`, che è un `void *`, cioè un puntatore generico in un puntatore specifico a un determinato tipo (per esempio, `(int *)`). In realtà, tutte le variabili puntatori occupano la stessa memoria, e quindi possono ricevere il risultato di `malloc`. L'operazione di conversione sembra quindi ridondante. Nei forum ci sono discussioni infinite sul fatto che questo uso comune sia giusto o che sia sbagliato, in base a diverse motivazioni tecniche. Non ho mai ricavato una chiara sensazione dalla lettura di queste disquisizioni, per cui aderisco alla convenzione. Un motivo secondario per cui lo faccio è che scrivere esplicitamente il tipo di puntatore dovrebbe richiamare alla memoria il tipo corretto da indicare nella chiamata alla funzione `sizeof`: se si vuole ottenere un `int *`, significa che nella `malloc` avremo un `sizeof(int)`. Indicare il tipo corretto in `sizeof` è fondamentale, perché indicando un tipo sbagliato si rischia (e la cosa dipende dalla macchina) di allocare una quantità scorretta di memoria, e quindi di avere errori molto elusivi.

Esiste un'altra funzione, che si chiama `calloc` (dove la `c` sta per "clear"), la quale fa esattamente la stessa cosa, ma divide l'argomento unico in due: bisogna indicare separatamente il numero dei blocchi di memoria e il tipo di ciascuno (vedi Figura ??. Invece di scrivere `malloc(10*sizeof(int));` si scriverà `calloc(10,sizeof(int));`. Una seconda differenza fondamentale è che la funzione `calloc` non si limita a riservare un'area di memoria, ma la riempie di bit nulli. L'effetto di questo assegnamento dipende dal tipo di dato contenuto nelle celle stesse, ma in genere è piuttosto naturale: le celle intere o reali conterranno la codifica del numero 0, quelle di tipo carattere conterranno il terminatore delle stringhe (`\0`). Questo può essere un motivo per preferire `calloc` a `malloc`, ma va considerato che la complessità è ovviamente superiore: `calloc` ha un costo proporzionale alla dimensione allocata, `malloc` un costo costante (si tratta solo di indicare nella tabella dei simboli che la memoria compresa fra due indirizzi dati è allocata).

In entrambe le funzioni, il numero di blocchi può essere un'espressione variabile anziché costante. È proprio questo che consente di usarle per gestire vettori dinamici. In effetti, allocare dinamicamente un vettore con dimensione costante è del tutto inutile e inelegante[3] Volendo quindi allocare un vettore indicizzato da 1 a n si potrà chiamare `calloc(n+1,sizeof(int));`, sempre con la convenzione di sprecare una cella per avere la semplicità di usare gli stessi indici del mondo esterno e senza introdurre puntatori ausiliari con cui scalare il vettore perché sarebbe inu-

[3] E viene penalizzato durante la valutazione dei progetti d'esame.

A una variabile puntatore si può assegnare un'area di memoria allocata dinamicamente, cioè un intervallo di celle consecutive nuovo prese in un'area della memoria detta *heap*

Questo consente di gestire strutture dati più flessibili di quelle statiche

- strutture di dimensione costante, ma incognita prima dell'esecuzione oppure variabile, ma con un massimo determinato al principio
- strutture di dimensione completamente libera durante l'esecuzione

La funzione `malloc`, dichiarata nella libreria `stdlib.h`

- richiede il numero totale di celle assegnate che può essere variabile e che conviene calcolare con `sizeof`
- rende un puntatore da convertire nel tipo giusto

`p = (`*tipo* `*) malloc(`*dimensione*`);`

Esempio: allochiamo spazio per 10 numeri interi consecutivi

```
int *pi;
pi = (int *) malloc(10*sizeof(int));
```

Figura 5.13: Allocazione dinamica della memoria

tilmente barocco e sprecherebbe una cella di memoria per il puntatore al solo scopo di risparmiare una cella di memoria per l'elemento di indice 0. A questo punto, il puntatore `p` funge come alias di un vettore a tutti gli effetti, sia in lettura sia in scrittura.

Potrebbe succedere che non vi sia memoria sufficiente nell'*heap*, o che si stia chiedendo di allocare un numero di celle o di blocchi assurdo (per esempio non positivo). In tali casi, le funzioni di allocazione falliscono. Quando ciò avviene, esse restituiscono un valore significativo, che per convenzione si chiama `NULL`. Questo valore (che corrisponde a una sequenza di bit tutti nulli, dunque allo zero) non è l'indirizzo di alcuna cella, e quindi garantisce di distinguere le allocazioni sbagliate da quelle giuste.

I vettori dinamici sono sequenze di oggetti omogenei il cui numero è fissato al momento dell'allocazione, ma varia da un'esecuzione all'altra

Si dichiarano con la funzione `calloc`, che alloca il blocco e lo azzera

```
int *p = (int *) calloc(n+1,sizeof(int));
```

dove `n` è una variabile intera

Il blocco di memoria è trattabile come un vettore statico:
si accede agli elementi in lettura e scrittura con gli indici numerici

```
i = p[4];
p[5] = 7;
```

Quando l'allocazione fallisce, viene restituito il puntatore nullo (`NULL`)
prima di usare l'area occorre verificare che il puntatore non sia nullo

```
if (p == NULL) printf("Errore di allocazione!\n");
```

Figura 5.14: Vettori dinamici

L'operazione complementare all'allocazione è la *deallocazione* (vedi Figura 5.15), che consiste nel liberare la memoria, ovvero renderla disponibile per altre allocazioni successive. La deallocazione consente, nel caso vi siano allocazioni successive, di non incrementare il consumo di memoria, e quindi di non aumentare la complessità spaziale dell'algoritmo. Si ricordi che la complessità spaziale di un algoritmo è il numero di celle che l'algoritmo stesso richiede per essere eseguito. Se si liberano le celle non usate, questo coincide con il massimo numero di celle contemporaneamente occupate durante l'esecuzione. Se non le si libera, invece, l'occupazione continua a crescere e con essa la complessità. Per questo è buona norma liberare la memoria allocata dinamicamente e non più necessaria con la funzione `free`, alla quale si passa il puntatore con l'indirizzo della prima cella occupata.

È in parte vero che tutta la memoria allocata dinamicamente da un programma viene automaticamente deallocata dal sistema operativo alla terminazione, ma l'aumento di complessità spaziale può rendere ineseguibile su macchine dotate di poca memoria un algoritmo che in realtà potrebbe tranquillamente funzionare, semplicemente perché la memoria disponibile viene consumata tutta e quella non usata risulta scorrettamente ancora necessaria. Il fenomeno si chiama *memory leak*, cioè "sgocciolamento" di memoria (come avviene per le perdite di acqua dalle tubazioni idrauliche con scarsa manutenzione) e ha riguardato anche programmi di una certa fama, in versioni messe frettolosamente sul mercato. C'è però un secondo motivo

per liberare esplicitamente tutta la memoria allocata: la deallocazione è un'operazione che tipicamente porta a problemi (blocchi del programma) se le memoria è stata gestita male. Il problema non è il fallimento della deallocazione, ma il fatto che quando un algoritmo usa in lettura o in scrittura celle esterne alle aree allocate, questo in genere non ha un effetto immediato (salvo che i valori casuali contenuti in quelle celle entrano nei calcoli e influiscono sul risultato in modo imprevedibile), ma nel momento in cui si chiama la funzione di liberazione questi problemi si manifestano, con un *crash* o con stampe incomplete (dovute al fatto che il programma termina senza svuotare il *buffer* di stampa: per motivi di efficienza, essendo l'accesso al video molto più lento di quello alla memoria, quando si stampa qualcosa a video, la stampa non avviene subito, ma viene prima eseguita su un'area di memoria, e solo quando questa si riempie avviene la stampa vera e propria; se il programma termina in anticipo, la stampa non ha luogo). Come principio generale, quindi, libereremo sempre tutta la memoria che è stata allocata[4]

Il meccanismo dell'allocazione e deallocazione è una fonte potenziale di errori gravi ed elusivi. Sottolineiamo il fatto che si parla di memoria allocata dinamicamente. La memoria statica viene gestita automaticamente dal processore, col meccanismo descritto nella Sezione **??** Se un puntatore contiene l'indirizzo di un'area di memoria allocata staticamente (per esempio, i puntatori usati per cambiare l'indicizzazione dei vettori) cercare di liberare quella memoria è un grave errore: quell'area di memoria non sta nemmeno nell'*heap*.

Non si deallocano i puntatori che identificano sottovettori di vettori statici

Non si deallocano i puntatori che scalano i vettori statici

Se si alloca dinamicamente un vettore `p` e poi si crea un *alias* assegnando il puntatore a un altro (`q = p;`), è un errore deallocarli entrambi, dato che l'area è una sola. Ricordiamo che i *record* si possono copiare con una semplice operazione di assegnamento, e che questa operazione consente di copiare interamente i vettori statici contenuti nei *record* stessi. Ma se un *record* contiene un vettore dinamico, la copia riguarda solo la cella del puntatore e l'area allocata rimane una sola, puntata due volte. Quindi un'eventuale deallocazione deve tener conto del fatto che solo in uno dei due *record* bisogna deallocare la memoria, mentre il puntatore contenuto nell'altro *record* andrà azzerato per evitare che continui a puntare un'area di memoria deallocata.

Gli alias di vettori deallocati non si deallocano, si azzerano

In breve, bisogna stare molto attenti e capire che cosa si sta facendo.

Si noti che la deallocazione consiste semplicemente nel rendere nuovamente disponibile la memoria per future allocazioni, ma il puntatore continua a contenere l'indirizzo della prima cella dell'area di memoria: questo rende l'operazione efficiente (tempo costante), ma impedisce di sapere dalla semplice ispezione di un puntatore se esso sia corretto oppure no, cioè se punti già (e punti ancora) a un'area dedicata, se non sia mai stato allocato o se sia stato allocato e poi deallocato. Per ridurre i rischi, è una buona norma assegnare ai puntatori deallocati l'indirizzo convenzionale `NULL` e usarlo come marcatore per i puntatori attualmente non utilizzati. Azzerare i

[4] Alcuni studenti evitano di farlo allo scopo di nascondere gli errori di gestione della memoria, ma l'errore viene scoperto comunque in fase di valutazione e la mancanza della deallocazione viene valutata come inefficienza. Un altro trucco controproducente è assegnare `NULL` ai puntatori prima di deallocarli. Questo chiaramente corrisponde a lasciare la memoria allocata, renderla irraggiungibile (perché si cancella l'indirizzo per raggiungerla) e poi deallocare un puntatore nullo, il che corrisponde a non fare niente. Anche in questo caso l'errore di gestione della memoria viene individuato e la deallocazione mancante conta come inefficienza.

L'area allocata è marcata così che successive allocazioni non la usino

Quando non occorre più, l'area allocata va deallocata (resa disponibile) per evitare il memory leakage (consumo progressivo della memoria)

`free(`*puntatore*`);`

dove *puntatore* è l'indirizzo della prima cella dell'area

- un'area va deallocata una volta sola, anche se ha più puntatori
- si deallocano solo puntatori a oggetti dinamici, non statici
- dopo la deallocazione, conviene fissare il puntatore a `NULL`, altrimenti continua a indicare l'area, ora libera
- il puntatore deve indicare la prima cella dell'area da liberare (potrebbe essere stato scalato per ottenere vettori da `S` a `D`)

Figura 5.15: Deallocazione

puntatori deallocati è ovviamente lecito e non costituisce inefficienza, ma una forma di pulizia (ovviamente utile se l'algoritmo prosegue, del tutto non necessaria se sta terminando).

Oltre ai vettori dinamici, è possibile allocare matrici dinamiche multidimensionali, sempre a patto di sapere che cosa si sta facendo. La Figura 5.16 illustra il caso delle matrici bidimensionali. Queste matrici hanno un numero di righe e di colonne non noto a priori (altrimenti, si usano le matrici statiche, magari specificamente su una delle dimensioni, tenendo le altre dinamiche: basta definire un tipo ausiliario di dato). Supponiamo che occorrano `m` righe e `n` colonne, per esempio indicizzate a partire da 1. È sufficiente usare un doppio puntatore a intero, cioè un puntatore a un puntatore a intero (`int **A`). Le celle della variabile `A`, in altre parole, contengono l'indirizzo della prima cella di una zona di memoria. L'istruzione `(int **) calloc(m+1,sizeof(int *));` alloca un vettore dinamico di `m+1` blocchi di celle della dimensione opportuna per contenere dei puntatori a intero. Queste celle vengono inizializzate con un ciclo assegnando a ciascuna l'indirizzo della prima cella di una zona di memoria, organizzata come un vettore dinamico di `n+1` interi (`(int *) calloc(n+1,sizeof(int));`). Il risultato finale è che l'*heap* contiene `m+1` vettori dinamici, il primo dei quali fa da "costola" alla matrice, mentre gli altri ne costituiscono le righe. L'equivalenza fra vettori e puntatori consente di accedere a queste aree di memoria in proiezione e in sostituzione esattamente come se si trattasse di una matrice statica. Si ricordi che nella Sezione **??** avevamo visto che la matrice statica era costituita da celle tutte contigue fra loro, mentre qui ciascun vettore è formato da celle contigue, ma i vettori possono essere sparpagliati. Inoltre, l'occupazione di memoria è maggiore, dato che la matrice dinamica richiede il vettore "costola" in più. Il vantaggio è poter scoprire il numero di righe e colonne durante l'esecuzione, anziché scrivendo il programma. Un secondo vantaggio è che il numero di colonne potrebbe anche essere diverso per ciascuna riga, cioè la matrice non deve necessariamente essere rettangolare, ma potrebbe avere una forma "sfrangiata".

Infine, la memoria occupata sta nell'*heap* (salvo la variabile puntatore `A`) anziché nello *stack*. Questo non porta alcuna differenza nell'uso, ma una differenza fondamentale al termine dell'uso: la matrice va ovviamente deallocata. Non si può

semplicemente deallocare il puntatore `A`, perché questo renderebbe disponibile solo il vettore "costola", lasciando occupata (e soprattutto irraggiungibile, avendo distrutto i relativi indirizzi) tutte le righe della matrice. Quindi, bisogna deallocare sia ciascuna delle righe sia la "costola" e l'ordine è importante: prima si deallocano le righe `A[i]`, poi `A`.

Matrice dinamica è un vettore dinamico di puntatori a vettori dinamici

Serve a ottenere matrici di dimensione incognita a priori
ed eventualmente matrici con righe non uniformi

Allocazione

```
int **A;
int m, n, i, j;
A = (int **) calloc(m+1,sizeof(int *));
for (i = 1; i <= m; i++)
  A[i] = (int *) calloc(n+1,sizeof(int));
```

Deallocazione

```
for (i = 1; i <= m; i++)
  free(A[i]);
free(A);
```

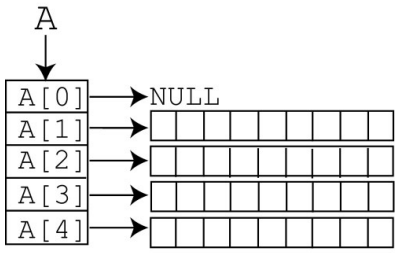


Figura 5.16: Matrici dinamiche

5.3 Gestione dello *stack*

Possiamo ora considerare la gestione dell'altra grande componente della memoria, cioè quella che contiene le variabili allocate staticamente, ovvero dalle operazioni di dichiarazione, nel programma principale oppure nelle definizioni delle procedure secondarie. Questa componente è gestita in maniera molto più rigida. *Stack* sta per "pila", come una pila di fogli su un tavolo.

Abbiamo già usato spesso procedure che chiamano altre procedure, su più livelli. Che cos'è una procedura o funzione (vedi Figura 5.17)?

È un insieme di istruzioni abbastanza unite logicamente e abbastanza importanti da meritare l'uso di un nome e di una interfaccia che ne specifichi i dati (racchiusi fra parentesi tonde) e il risultato. L'uso delle funzioni non è strettamente necessario, ma corrisponde esattamente alla modalità di progetto *top-down* degli algoritmi. Il criterio di struttura del codice è uno dei tre criteri di valutazione del progetto d'esame e riguarda l'uso di buone funzioni, cioè una buona scelta dei nomi, dei dati e dei risultati.

Partiamo con un esempio: vogliamo calcolare i quadrati e i cubi dei primi 10 numeri positivi. La Figura 5.18 illustra un esempio di codice che risolve il problema. Si vedono le solite quattro sezioni: direttive, prototipi di funzioni secondarie,

Funzioni o procedure sono brani di codice abbastanza importanti da avere un nome, dei dati e dei risultati, come i programmi

L'uso delle funzioni consente di

- dividere un programma in brani di codice
  - più facili da capire
  - più facili da modificare
- non ripetere brani di codice identici o molto simili
- riutilizzare brani di codice in programmi diversi

Figura 5.17: Funzioni

programma principale e definizioni di funzioni secondarie. In particolare, ci sono due funzioni, `quadrato` e `cubo`.

Calcolo dei quadrati e dei cubi per i primi dieci numeri positivi

```
#include <stdio.h>
#include <stdlib.h>

int quadrato (int y);
int cubo (int y);

int main (int argc, char *argv[])
{
  int x;
  int Q[11];
  int C[11];

  for (x = 1; x <= 10; x++)
    Q[x] = quadrato(x);

  for (x = 1; x <= 10; x++)
    C[x] = cubo(x);

  return EXIT_SUCCESS;
}
```

```
int quadrato (int y)
{
  return y * y;
}

int cubo (int y)
{
  int q;

  q = quadrato(y);
  return q * y;
}
```

Figura 5.18: Esempio

Ogni funzione compare in tre punti (vedi Figura 5.19):

1. una volta nella seconda sezione del codice come prototipo, o dichiarazione, cioè come intestazione con il nome, i nomi e tipi dei dati e il tipo del risultato;

2. una o più volte nella terza e quarta sezione del codice, come chiamata (per esempio, `quadrato` compare nel programma principale, ma anche nella funzione `cubo`), con dati diversi nelle varie chiamate e senza specificare il tipo: la chiamata applica la funzione a espressioni costanti, variabili, o anche a combi-

nazioni delle une e delle altre con operazioni varie (per esempio, `quadrato(x)`, ma anche `quadrato(y)`);

3. una volta nella quarta sezione del codice, come definizione della funzione stessa, con un'intestazione identica al prototipo, salvo il punto e virgola, e un corpo che specifica che cosa effettivamente faccia la funzione.

Il prototipo (dichiarazione) specifica

- il tipo del risultato della funzione (`void` se non dà risultati)
- il nome della funzione
- i dati richiesti dalla funzione

La definizione della funzione riprende il prototipo e specifica le variabili locali e le operazioni eseguite dalla funzione

Chiamata di una funzione (invocazione) è
ciascun uso della funzione in espressioni del `main` o di altre funzioni e
specifica i dati sui quali effettivamente la funzione opera nel caso specifico

Figura 5.19: Prototipo, chiamate e definizione

Già sappiamo che il prototipo serve a informare il compilatore dell'esistenza e delle possibili future chiamate della funzione, con dei dati e un risultato di un certo tipo (vedi Figura 5.20). È un'istruzione, ma impatta solo sulla tabella dei simboli. Le chiamate sono invece vere e proprie istruzioni di calcolo. È importante quindi distinguere tra:

- *parametri formali*, che sono i dati forniti al prototipo e alla definizione, ovvero "pro forma", come segnaposto simbolico;
- *parametri attuali*, che sono i dati forniti al momento della chiamata, ovvero "in atto" (per esempio, la variabile `x` che va da 1 a 10.

In generale, parametri formali e attuali avranno nomi diversi, e i parametri attuali potrebbero anche non avere nomi, ma essere costanti esplicite o espressioni composte.

Della definizione già sappiamo (vedi Figura 5.21) che ha la stessa struttura del programma principale, con una parte dichiarativa e una esecutiva, conclusa da un'istruzione `return` se la funzione restituisce un valore, assente se invece è una funzione di tipo `void`, che non restituisce alcun valore.

Le chiamate, invece, usano i parametri attuali, detti anche *argomenti*, nel corpo della funzione chiamante (vedi Figura 5.22).

L'aspetto importante della chiamate è il fatto che esse costituiscono una struttura ad albero ordinato. La Figura 5.23 illustra questa struttura per l'esempio della funzione che calcola quadrati e cubi dei primi 10 numeri. Un primo ciclo chiama per 10 volte la funzione `quadrato` sull'argomento `x` che cresce via via da 1 a 10. Corrispondentemente, l'albero delle chiamate mostra un nodo etichettato `main` con i primi 10 nodi figli etichettati `quadrato(1)`, `quadrato(2)` ... `quadrato(10)`. Quindi, si ha un secondo ciclo che chiama per 10 volte la funzione `cubo` sull'argomento `x` che cresce da 1 a 10. Corrispondentemente, l'albero delle chiamate ha altri 10 figli

tipo funzione`(`*tipo1 nome1*`,`*tipo2 nome2*`,...);`

Il prototipo specifica

- il *tipo* del risultato della funzione (`void` se non dà risultati)
- il *nome* della funzione
- il *tipo* e il *nome* di ciascun dato

I parametri formali sono i nomi dei dati indicati nel prototipo
I parametri attuali saranno i valori dei dati indicati nelle chiamate
Il prototipo mette il nome della funzione nella tabella dei simboli
(essendo un'istruzione, termina con `;`)

Figura 5.20: Prototipo

tipo funzione`(`*tipo1 nome1*`,`*tipo2 nome2*`,...)`
`{`
parte dichiarativa
parte esecutiva
[`return` *espressione*`;`]
`}`

- l'intestazione riprende il prototipo senza il `;` (*non è un'istruzione*)
- il corpo fra parentesi graffe specifica variabili locali e istruzioni e termina con l'istruzione `return` seguita da un'*espressione*: risultato della funzione è il valore dell'espressione

Una procedura non restituisce risultati: è di tipo `void` e
non occorre che termini con `return`

Figura 5.21: Definizione

Ogni chiamata di una funzione (invocazione) è
il suo uso in espressioni del `main` o di altre funzioni

oggetto = *funzione*`(`*parametri attuali*`);`
procedura`(`*parametri attuali*`);`

Parametri attuali sono le espressioni i cui valori forniscono i dati
Si chiamano anche argomenti

Per ogni funzione si possono avere molte chiamate su dati diversi

- da parte della stessa funzione
- da parte di diverse funzioni

Figura 5.22: Chiamata

del nodo `main`, etichettati `cubo(1)`, `cubo(2)` ... `cubo(10)`. All'interno della funzione `cubo`, però, compare una chiamata alla funzione `quadrato`, dato che il quadrato dell'argomento viene moltiplicato per l'argomento stesso in modo da calcolare il cubo. Corrispondentemente, ogni nodo etichettato `cubo` ha un nodo figlio etichettato `quadrato` con lo stesso argomento. Una funzione può contenere chiamate ad altre funzioni. Nell'albero delle chiamate, questo corrisponde a un ulteriore livello. In linea di principio, la cosa non ha un limite. L'albero è ordinato totalmente, nel senso che ogni nodo precede immediatamente il suo primo figlio e ogni nodo precede (in genere non immediatamente) il nodo fratello seguente. Questo ordine corrisponde all'ordine cronologico di esecuzione delle funzioni. Quando si torna al `main` e ne termina l'esecuzione, l'intero programma termina.

L'idea alla base di questa struttura ad albero è che ogni funzione chiamante conosce soltanto i propri dati e i propri risultati, passa gli argomenti (parametri attuali) alle funzioni chiamate, e ciascuna di queste fa quel che deve senza che la funzione chiamante conosca per ciò che che succede al loro interno. Le funzioni interagiscono solo attraverso i dati e i risultati.

Come si fa a realizzare in pratica questo sistema ad albero su un processore? Concretamente, non c'è alcun albero, ma viene simulato il meccanismo di discesa e di risalita relativo ai livelli dell'albero stesso. La cosa è facile attraverso una struttura a pila (vedi Figura 5.24). Metaforicamente, stiamo lavorando su foglio etichettato `main`. Dopo alcune operazioni, ci si rende conto che occorre chiamare la funzione `quadrato`. Allora, si prende un altro foglio etichettato `quadrato` e lo si sovrappone al primo, copiando sul foglio superiore le informazioni minime necessarie e proseguire il calcolo. Si esegue l'intera funzione `quadrato` su questo foglio e al termine si salva a parte il risultato, si toglie il foglio superiore, tornando al foglio etichettato `main` e si riprende a lavorare usando il risultato stesso nella funzione `main`. Questa metafora si può svolgere su un numero qualsiasi di livelli, a patto di avere abbastanza fogli: se chiamiamo la funzione `cubo`, che richiede la funzione `quadrato`, a un certo punto sovrapporremo un terzo foglio al secondo, come abbiamo sovrapposto il secondo al primo, e quando avremo finito torneremo al secondo e infine al primo.

Ora descriviamo il meccanismo in modo più dettagliato, sempre sul solito esempio. All'avvio del programma, il sistema operativo pone automaticamente sul "tavolo" un "foglio" che chiameremo *scope* ("ambito", ovvero "orizzonte") del programma

Le chiamate a funzione sono organizzate ad albero, radicato nel `main`

- la funzione chiamante indica quali funzioni chiamate devono operare e in che ordine e fornisce a ciascuna i dati su cui operare
- le funzioni chiamate restituiscono alla funzione chiamata i risultati del loro lavoro
- ogni funzione chiamata può avere sottoprocedure, con le quali organizza il lavoro allo stesso modo (da cui la struttura ad albero)
- la funzione chiamante conosce solo dati e risultati; non sa quali operazioni compiono e che mezzi usano le funzioni chiamate

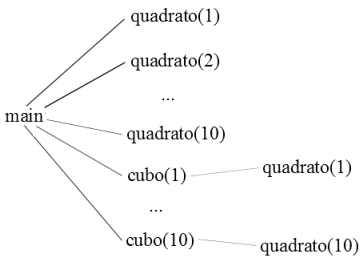


Figura 5.23: Struttura modulare

Il processore riserva alle funzioni un'area di memoria detta *stack* che viene gestita come una pila e che è completamente separata dallo *heap*

- alla chiamata, alloca in cima alla pila lo spazio necessario a
  - parametri formali (dati)
  - variabili locali (risultati parziali)
  - risultato della funzione
- al termine della funzione, dealloca lo spazio e rende il risultato

A causa della deallocazione, parametri e variabili locali esistono solo fra la chiamata e il termine della funzione

Le strutture dinamiche stanno in un'altra area di memoria detta *heap*: vengono deallocate solo da `free` o al termine dell'intero programma

Vediamo un esempio in dettaglio

Figura 5.24: Gestione dello *stack*

principale. Questa è un'area di memoria dimensionata e strutturata in modo da contenere:

- gli argomenti del programma principale (cioè le variabili `argc` e `argv`);
- il risultato del programma principale (cioè un intero senza un nome simbolico);
- le variabili dichiarate nel corpo del programma principale (cioè `x`, `Q` e `V`).

Grazie alle dichiarazioni di tipo nell'intestazione e nella parte dichiarativa, il compilatore ha specificato esattamente il numero di celle necessario per ciascuno dei quattro oggetti (vedi Figura 5.25)[5]

Supponiamo di aver già eseguito tutte le iterazioni del primo ciclo e la prima iterazione del secondo ciclo (perché sono meno significative dal punto di vista dei valori). In questo momento, quindi `x` vale 2 e ci accingiamo ad eseguire la chiamata `cubo(x)`. Le chiamate a funzione consistono nelle seguenti tre operazioni, che è importante capire per poter usare le funzioni in C correttamente:

1. vengono valutati i valori dei parametri attuali;
2. si alloca sullo *stack* lo spazio necessario ad eseguire la funzione;
3. si copiano i valori dei parametri attuali nei parametri formali.

I parametri attuali, si è detto, sono in genere espressioni composte, per cui possono comportare calcoli anche sofisticati. Nel caso specifico, il parametro è `x` e quindi si tratta banalmente di andare a leggere il valore della variabile `x`. In generale, si sarebbe potuto avere `x+1`, `3*x` o qualsiasi espressione basata sulle costanti e variabili note al compilatore[6]. L'allocazione dello spazio necessario sullo *stack* richiede di calcolarlo. Questo è possibile in base alla definizione della funzione. La funzione `cubo` richiede spazio per un dato intero (`y`) e un risultato intero, più lo spazio per una *variabile locale* `q`, che conterrà un risultato intermedio, utile per i calcoli della funzione stessa. Quindi si tratta solo di tre interi, molto più dello spazio richiesto dal programma principale. Questo mostra che i "fogli" della metafora non sono tutti uguali: ciascuno ha la dimensione strettamente necessaria a fare quel che deve. L'area di memoria è organizzata in modo da contenere i dati necessari. In concreto, "allocare in cima allo *stack*" significa che si prende l'indirizzo della prima cella non utilizzata e lo si incrementa della dimensione totale richiesta dal nuovo *scope*. In questo modo, un'intera zona di memoria diventa utilizzabile, e viene organizzata scrivendo nella tabella dei simboli gli indirizzi relativi ai nuovi simboli che si possono usare. Ovviamente, bisogna annotarsi da qualche parte il vecchio indirizzo della prima cella non utilizzata, in modo da poter regredire alla situazione iniziale quando la funzione avrà terminato il proprio compito. Il terzo passo è prendere i parametri attuali (che stanno in registri dedicati alla gestione di questo meccanismo) e copiarli nelle aree di memoria dei parametri formali che il secondo passo ha preparato. Nell'esempio, il valore 2 dell'unico parametro attuale viene copiato nelle celle di memoria del parametro formale `y`. A voler essere molto tecnici, un quarto passo consiste nell'annotare in un registro dedicato la linea di codice della chiamata, in modo che al termine dell'esecuzione della funzione, si possa tornare al punto di partenza (dato che la stessa funzione può essere chiamata

[5] La figura non riporta per compattezza i vettori `Q` e `V`, e non riporta i registri dedicati a gestire il funzionamento del meccanismo, fra cui quello che contiene i valori restituiti dalle funzioni terminate e quello che contiene la posizione corrente nel corso dell'esecuzione, in modo che il processore sappia da dove ricominciare l'esecuzione quando una funzione termina.

[6] Oltre che a chiamate di ulteriori funzioni, cosa su cui sorvoliamo per semplicità, finché non terminiamo la descrizione di questo argomento.

partendo da moltissime posizioni diverse nel codice). Però questo è abbastanza ininfluente sull'uso delle funzioni in C.

BISOGNA DISTRIBUIRE SENSATAMENTE I RIFERIMENTI ALLE FIGURE 5.26, 5.28, 5.29, 5.30, 5.31.

Ogni funzione ha accesso (spesso in gergo si dice che "vede") soltamente il contenuto del proprio *scope* di memoria, cioè solo il proprio foglio. Nel nostro caso, la funzione `cubo` chiama per prima cosa la funzione `quadrato`. Questo provoca una nuova esecuzione del meccanismo che abbiamo descritto.

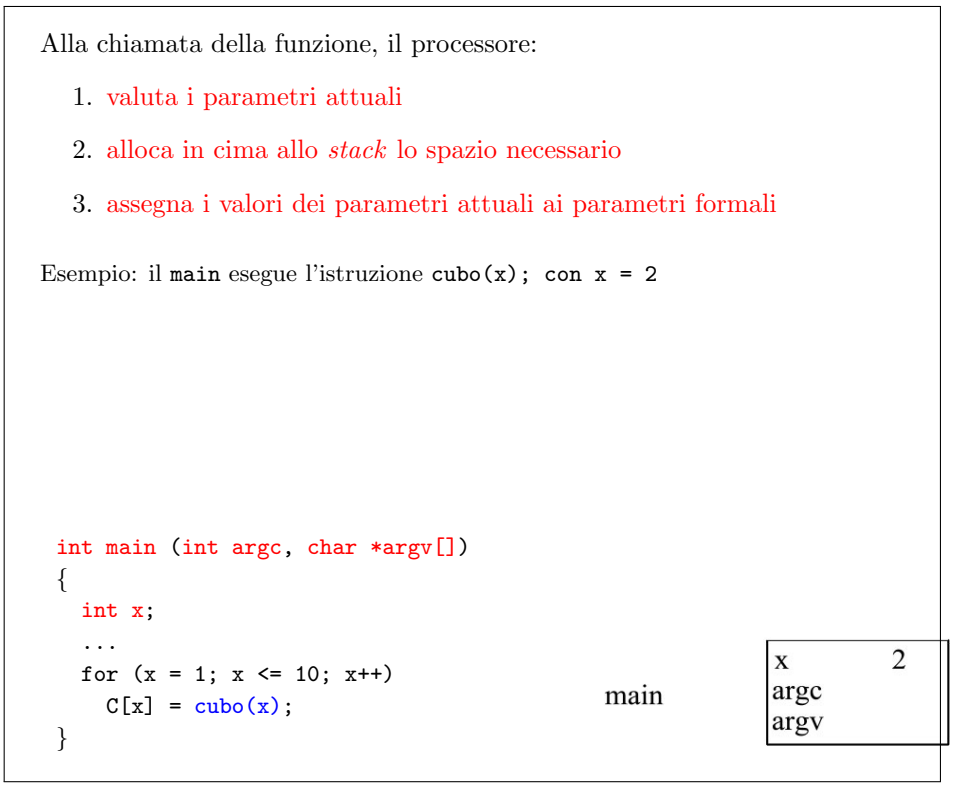


Figura 5.25: Gestione dello *stack*: esempio

La valutazione dei parametri attuali consiste in questo caso nel leggere il valore della variabile locale `y`, cioè 2. Quindi, bisogna allocare in cima allo *stack* lo spazio necessario per la funzione `quadrato`, che consiste solo in due interi, il dato e il risultato, non essendoci variabili locali. Quindi, sulla pila di sistema compare un terzo "foglio", ancora più piccolo dei primi due. Infine, si assegna il valore del parametro attuale al parametro formale, così che `y` ora vale 2. Si noti che nella memoria del processore lo stesso valore 2 compare ben tre volte: nello *scope* del programma principale viene indicato con `x`, nello *scope* della chiamata `cubo(2)` viene indicato con `y` e nello *scope* della chiamata `quadrato(3)` viene indicato con `y`. Si noti che queste occorrenze dello stesso dato, essendo in celle diverse, possono avere nomi diversi come possono avere lo stesso nome. Anche se hanno lo stesso nome, le celle sono distinte. Il motivo per cui possono avere lo stesso nome è che gli *scope* sono distinti, cioè le tabelle dei simboli sono diverse, e quindi non si ha sovrapposizione e confusione. È importante aver capito questo meccanismo perché nel seguito vedremo che modificare il valore di `y` in `quadrato` non modifica il valore di `y` in `cubo`, e questo schema mostra graficamente il perché.

Alla chiamata della funzione, il processore:

1. valuta i parametri attuali
2. alloca in cima allo *stack* lo spazio necessario
3. assegna i valori dei parametri attuali ai parametri formali

Esempio: il `main` esegue l'istruzione `cubo(x); con x = 2`

1. il parametro attuale è `x` e vale 2
2. si alloca sullo *stack* spazio per `y`, per `q` e per il risultato di `cubo`
3. si assegna il valore 2 a `y`

```
int cubo (int y)
{
  int q;

  q = quadrato(y);
  return q * y;
}
```

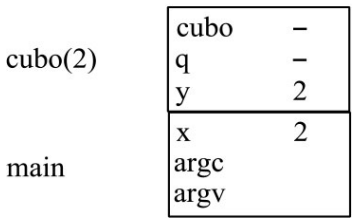


Figura 5.26: Gestione dello *stack*: esempio

A questo punto, si esegue il corpo della funzione `quadrato`, che consiste in una singola operazione `y*y` e nella restituzione del risultato attraverso l'istruzione `return`. Questa esegue tre passaggi:

1. scrive il valore del risultato (2 per 2, cioè 4) nel registro dedicato;
2. dealloca lo *scope* corrente, ovvero butta via il "foglio" in cima alla pila;
3. riattiva l'esecuzione della funzione chiamante, recuperando l'istruzione dal registro dedicato, cioè procede assegnando il valore 4 alla variabile `q` della funzione `cubo` (`q = quadrato(y);`).

Si prosegue quindi regolarmente con la funzione `cubo`, la quale semplicemente moltiplica `q` per `y` e restituisce il risultato. Ancora una volta, l'istruzione `return` esegue i tre passaggi sopra descritti: valuta l'espressione `q*y` (che vale 8) e ne salva il valore nel registro dedicato, dealloca la pila e torna all'istruzione rimasta in sospeso nella funzione chiamante `main`.

Quando si torna al `main`, l'operazione che si esegue assegna il valore 8 all'elemento `C[2]` (che nella Figura 5.32 non compare perché abbiamo ridotto la rappresentazione del *record* di attivazione del `main`)

5.4 Passaggio dei parametri per valore e per indirizzo

Torniamo al meccanismo della chiamata. La prima fase valuta i parametri attuali e la seconda li copia nei parametri formali. Questo meccanismo, che è quello usato nel

Figura 5.27: Gestione dello *stack*: esempio

Alla chiamata della funzione, il processore:

1. valuta i parametri attuali
2. alloca in cima allo *stack* lo spazio necessario
3. assegna i valori dei parametri attuali ai parametri formali

Esempio: la funzione `cubo` esegue l'istruzione `quadrato(y);` con `y = 2`

```
int cubo (int y)
{
  int q;

  q = quadrato(y);
  return q * y;
}
```

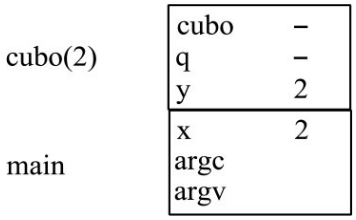


Figura 5.28: Gestione dello *stack*: esempio

Alla chiamata della funzione, il processore:

1. valuta i parametri attuali
2. alloca in cima allo *stack* lo spazio necessario
3. assegna i valori dei parametri attuali ai parametri formali

Esempio: la funzione `cubo` esegue l'istruzione `quadrato(y);`

1. il parametro attuale è `y` e vale 2
2. si alloca sullo *stack* spazio per `y` e per il risultato di `quadrato`
3. si assegna il valore 2 a `y`

```
int quadrato (int y)
{
  return y * y;
}
```

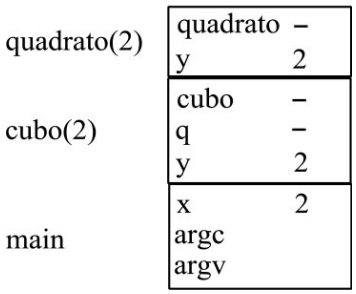


Figura 5.29: Gestione dello *stack*: esempio

Al termine della funzione, il processore:

1. valuta l'espressione che segue l'istruzione `return` come risultato
2. dealloca dalla cima allo *stack* lo spazio dell'ultima chiamata
3. restituisce il risultato alla funzione chiamante

Esempio: la funzione `quadrato` esegue l'istruzione `return y * y;`

```
int quadrato (int y)
{
  return y * y;
}
```

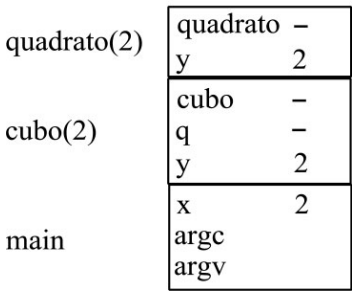


Figura 5.30: Gestione dello *stack*: esempio

Al termine della funzione, il processore:

1. valuta l'espressione che segue l'istruzione `return` come risultato
2. dealloca dalla cima allo *stack* lo spazio dell'ultima chiamata
3. restituisce il risultato alla funzione chiamante

Esempio: la funzione `quadrato` esegue l'istruzione `return y * y;`

1. il risultato `y * y` vale 4
2. si dealloca lo spazio per `y` e per il risultato (salvando questo a parte)
3. si restituisce 4 alla funzione `cubo`, che lo assegna a `q`

```
int cubo (int y)
{
  int q;

  q = quadrato(y);
  return q * y;
}
```

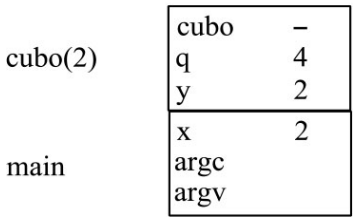


Figura 5.31: Gestione dello *stack*: esempio

Al termine della funzione, il processore:

1. valuta l'espressione che segue l'istruzione `return` come risultato
2. dealloca dalla cima allo *stack* lo spazio dell'ultima chiamata
3. restituisce il risultato alla funzione chiamante

Esempio: la funzione `cubo` esegue l'istruzione `return q * y;`

```
int cubo (int y)
{
  int q;

  q = quadrato(y);
  return q * y;
}
```

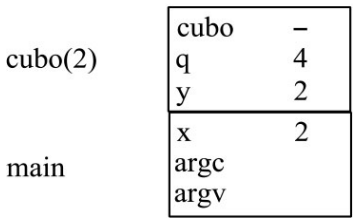


Figura 5.32: Gestione dello *stack*: esempio

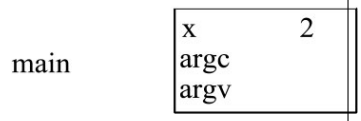
Al termine della funzione, il processore:

1. valuta l'espressione che segue l'istruzione `return` come risultato
2. dealloca dalla cima allo *stack* lo spazio dell'ultima chiamata
3. restituisce il risultato alla funzione chiamante

Esempio: la funzione `cubo` esegue l'istruzione `return q * y;`

1. il risultato `q * y` vale 8
2. si dealloca lo spazio per `y`, `q` e per il risultato (salvando questo a parte)
3. si restituisce 8 al `main`, che lo assegna a `C[x]`

```
int main (int argc, char *argv[])
{
  int x;
  ...
  for (x = 1; x <= 10; x++)
    C[x] = cubo(x);
}
```



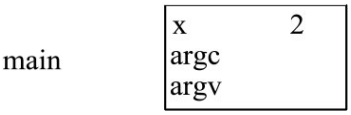


Figura 5.33: Gestione dello *stack*: esempio

linguaggio C e in molti altri, è detto *passaggio per valore*. In questo meccanismo, il valore 2 viene copiato dalla variabile locale `x` del `main` all'argomento `y` della chiamata `cubo(2)`, e ancora da questo all'argomento `y` della chiamata `quadrato(2)`. Questo meccanismo moltiplica l'occupazione di spazio copiando le informazioni necessarie. Esiste un altro importante meccanismo (vedi Figura 5.34), che viene usato in diversi altri linguaggi di programmazione, fra cui il Fortran, oppure il linguaggio astratto della macchina *RAM* usato nelle lezioni di teoria. È il *passaggio per indirizzo*. In questo meccanismo, una funzione accede in scrittura non solo al proprio *scope*, ma anche a quei dati originali che le vengono passati come argomenti, e quindi li può modificare.

Ad ogni chiamata i parametri attuali vengono valutati e sostituiscono i parametri formali nella funzione chiamata

In generale vi sono due modi di passare i parametri

- per valore: la funzione accede a copie dei dati e le eventuali modifiche interne vanno perse
- per indirizzo: la funzione accede ai dati originali e le eventuali modifiche interne si conservano

Figura 5.34: Passaggio dei parametri

Consideriamo l'esempio della Figura 5.35. Eseguiamo nel programma principale l'istruzione `quadrato(10);` e modifichiamo la funzione `quadrato` in modo che assegni `y*y` all'argomento `y` e poi lo restituisca, anziché restituire direttamente la prima espressione. In questo caso, l'istruzione scrive il valore 100 nelle celle dell'argomento `y`. Con un passaggio dei parametri per valore, quando l'esecuzione della funzione termina queste celle escono semplicemente dallo *scope* e il loro contenuto è perso[7]. Se invece si usa un passaggio per indirizzo, il valore 100 viene assegnato alle celle del parametro attuale `x`, e quindi rimane disponibile anche nel `main`.

Il passaggio dei parametri per valore non ha effetti collaterali

Si possono usare i parametri formali come variabili locali, modificandoli per produrre il risultato senza sporcare i parametri attuali

```
int main (int argc, char *argv[])
{
  int x, q;

  x = 10;
  q = quadrato(x);
  /* x vale ancora 10 */
  return EXIT_SUCCESS;
}
```

```
int quadrato (int y)
{
  y = y * y;
  /* ora y vale 100 */
  return y;
}
```

Figura 5.35: Passaggio per valore

[7] Ovviamente, il valore 100 restituito dalla funzione no.

Un esempio classico di questa differenza è quello della funzione che scambia i valori di due variabili dello stesso tipo (vedi Figura 5.36). Supponiamo di avere due variabili intere `a` e `b`, rispettivamente di valore 1 e 2. La funzione `scambia` assegna a ciascuna il valore dell'altra, usando come passaggio intermedio una variabile locale `temp`, per conservare uno dei due valori, che viene sovrascritto dall'altro, che poi viene a sua volta sovrascritto dal primo. È un'operazione che riutilizzeremo pesantemente negli algoritmi di ordinamento, che sono per lo più basati sulle operazioni di confronto e scambio di valori. Eseguite le tre operazioni, `a` vale 2 e `b` vale 1, ma quando la funzione `scambia` termina, si pone il problema se si siano effettivamente scambiati i due valori. Infatti, se il passaggio è avvenuto per valore, quelli che si sono scambiati sono le copie dei valori originali contenute nello *scope* della funzione, che vanno perduti all'uscita dalla funzione stessa: i valori originali sono rimasti invariati. Al principio si ha:

```
****** main ******
main
int a 1
int b 2
int argc
char *argv[]
******************
```

che dopo la chiamata diventa

```
**** scambia *****
int a 1
int b 2
int temp
****** main ******
main
int a 1
int b 2
int argc
char *argv[]
******************
```

e dopo le tre istruzioni di scambio

```
**** scambia *****
int a 2
int b 1
int temp 1
****** main ******
main
int a 1
int b 2
int argc
char *argv[]
******************
```

ma (non essendo cambiato nulla nello *scope* del `main`), quando la funzione termina si torna alla situazione iniziale:

```
****** main ******
main
int a 1
```

```
int b 2
int argc
char *argv[]
******************
```

È positivo o negativo che questo avvenga? È positivo in tutti i casi in cui è utile usare i dati modificandoli senza distruggerli, come quando abbiamo usato `y` per calcolare il suo quadrato (un esempio abbastanza poco significativo). Nel caso della funzione `scambia`, invece, è chiaramente negativo, dato che fa fallire lo scopo della funzione stessa. Se il passaggio dei parametri avvenisse per indirizzo, come in altri linguaggi, i due valori sarebbero stati scambiati.

Il passaggio di parametri per indirizzo consente effetti collaterali

Quindi consente alla funzione di accedere ai dati originali e modificarli

Ma in C il passaggio per indirizzo non esiste!

```
void scambia (int a, int b);

int main (int argc, char argv[])
{
  int a, b;

  a = 1;
  b = 2;
  scambia(a,b);
  /* ora a vale ancora 1
       e b vale ancora 2 */
  return EXIT_SUCCESS;
}
```

```
void scambia (int a, int b)
{
  int temp;

  temp = a;
  a = b;
  b = temp;
  /* ora temp vale 1,
     a vale 2 e b vale 1 */
}
```

Figura 5.36: Passaggio per indirizzo

Il linguaggio C è basato sul concetto che gli *scope* sono reciprocamente inaccessibili, ma lascia aperta una scappatoia per ottenere in pratica il passaggio per indirizzo. La scappatoia si basa sull'uso dei puntatori, che già sappiamo possono contenere indirizzi qualsiasi. Si veda la Figura 5.37. L'idea è che la funzione `scambia` non usi come parametri due interi, come nella versione ideale, ma due puntatori a intero. Per sottolineare questo aspetto, i due puntatori sono indicati come `pa` e `pb`. Nel seguito, useremo sempre la convenzione di premettere una `p` al nome di una variabile per indicare un'altra variabile di tipo puntatore che ne contiene l'indirizzo. È semplicemente un artificio mnemonico. Molti usano chiamare le due variabili allo stesso modo, sfruttando il fatto che vivono in *scope* diversi e affrontando la leggera confusione che questo provoca.

Quando si chiama la funzione `scambia`, non si passano `a` e `b`, ma due puntatori alle relative celle, cioè i loro indirizzi, che si ottengono l'operatore di referenziazione `&`, come evidenziato in rosso nella Figura 5.37. La situazione sopra descritta cambia come segue. Si parte esattamente come nell'altro caso:

```
****** main ******
main
int a 1
int b 2
int argc
```

```
char *argv[]
******************
```

e dopo la chiamata si ottiene

```
**** scambia *****
int *pa ----------+
int *pb ----------)-+
int temp | |
****** main ****** | |
main | |
int a 1 <-------+ |
int b 2 <---------+
int argc
char *argv[]
******************
```

dove il compilatore ha per prima cosa valutato gli indirizzi delle variabili intere `a` e `b`, per seconda cosa allocato lo *scope* di `scambia`, che include due puntatori a intero e un intero, anziché tre interi, e infine ha copiato i due indirizzi nelle celle dei due puntatori. Questo è rappresentato dalle frecce, che sottolineano graficamente come il processore sia ora in grado di "vedere fuori" dallo *scope*, dunque di leggere e scrivere su parte di un "foglio" che non sta in cima alla pila.

Ora le tre istruzioni di scambio devono tenere conto del fatto che non abbiamo direttamente i valori da scambiare, ma dobbiamo raggiungerli attraverso i loro indirizzi. Questo richiede le operazioni di dereferenziazione con l'operatore `*` evidenziate in rosso nella Figura 5.37, al termine delle quali lo scambio è avvenuto al livello delle variabili originali:

```
**** scambia *****
int *pa ----------+
int *pb ----------)-+
int temp 2 | |
****** main ****** | |
main | |
int a 2 <-------+ |
int b 1 <---------+
int argc
char *argv[]
******************
```

e questo significa che la terminazione della funzione `scambia` porta nella situazione desiderata:

```
****** main ******
main
int a 2
int b 1
int argc
char *argv[]
******************
```

Grazie ai puntatori, si è potuto modificare le celle della funzione chiamante partendo da quelle della funzione chiamata.

Abbiamo detto che il linguaggio astratto della macchina *RAM* ha il passaggio per indirizzo, mentre il C ha il passaggio per valore e simula quello per indirizzo

attraverso i puntatori. Questo ha diversi impatti nel seguito del corso: le dispense di laboratorio cercano di avvicinarsi il più possibile a quelle di teoria, ma non sempre questo è possibile. Anche quando lo è, occorrerà adottare la scrittura attraverso i puntatori, che è quindi bene assicurarsi di aver capito completamente. L'uso errato dei puntatori è ovviamente causa di errori nel codice, ma anche il passaggio per indirizzo di strutture che si possono passare per valore senza perdite di efficienza[8] va a danneggiare la struttura del codice nella valutazione dei progetti.

Per simulare il passaggio per indirizzo in C si usa un trucco:
si passa per valore l'indirizzo del parametro

```
void scambia (int *pa, int *pb);

int main (int argc, char argv[])
{
  int a, b;

  a = 1;
  b = 2;
  scambia(&a,&b);
  /* ora a vale 2 e b vale 1 */
  return EXIT_SUCCESS;
}
```

```
void scambia (int *pa, int *pb)
{
  int temp;

  temp = *pa;
  *pa = *pb;
  *pb = temp;
  /* ora temp vale 1,
     a vale 2 e b vale 1 */
}
```

Figura 5.37: Passaggio per indirizzo in C

5.4.1 Risultati multipli

Il passaggio per indirizzo ha un'altra applicazione importante in C, oltre alla possibilità di scrivere funzioni che modifichino i propri dati. Si tratta di superare un limite che abbiamo già incontrato, cioè il fatto che una funzione C restituisce solo un risultato alla funzione chiamante. In C non si possono essere risultati multipli, e in particolare non si possono restituire vettori statici (già sappiamo che un vettore dinamico è in realtà un puntatore, e quindi sfugge a questa limitazione: fra poco vedremo ulteriori dettagli su questo aspetto). Un'applicazione del passaggio per indirizzo simulato è la possibilità di ottenere risultati multipli o vettoriali (vedi Figura 5.38).

Abbiamo già visto un trucco per sfuggire a questa limitazione: definire un tipo *record* con un'istruzione `typedef`, contenente campi dello stesso tipo dei risultati che si vogliono ottenere, usarlo come tipo del risultato di una funzione e riempire i singoli campi all'interno della funzione. Sappiamo che in questo modo è addirittura possibile restituire vettori statici. È un trucco un po' macchinoso, dato che per ogni combinazione di risultati desiderati richiede di definire un *record* specificamente strutturato per contenerli[9].

Il trucco standard è di definire dei finti dati che in realtà sono risultati e passarli per indirizzo, cioè passare dei puntatori ad essi. Lo abbiamo già visto fare per le stringhe: che fossero dati o risultati, venivano inseriti nella lista degli argomenti (`AcquisisceSaluto` nel Capitolo **??**). Rimandiamo per il momento il caso delle

[8] Di questo aspetto parleremo in seguito, a partire dal Capitolo **??**.

[9] Fra l'altro, mi risulta (da non esperto) che questo trucco si risolva in pratica nel fatto che il compilatore traduce il codice in modo tale da adottare il meccanismo che andiamo a descrivere nel seguito, quindi da un punto di vista pratico il trucco è uno solo.

Un altro limite del passaggio per valore è che

- si può restituire come risultato un solo valore
- il risultato non può essere un vettore statico
  (dato che non si può usare un vettore statico in un assegnamento)

Ma se occorre restituire più valori o un vettore di valori?

- risultati multipli:
  - si restituisce un record contenente i valori come campi
  - si passano dei finti "dati" che vengono modificati sfruttando gli effetti collaterali del passaggio per indirizzo
- vettore:
  - si restituisce un "vettore" dinamico (puntatore);
    l'area di memoria va allocata nella funzione e deallocata fuori
  - si passa l'intero vettore come dato fittizio, ma per indirizzo (dunque non il vettore, ma un puntatore al vettore)

Figura 5.38: Risultati multipli o vettoriali

stringhe (e dei vettori), che richiedono ulteriori considerazioni e concentriamoci sul caso dei risultati multipli. La Figura 5.39 fornisce un esempio: realizziamo una funzione `decompone`, che decompone un numero reale (`double d`) nella sua parte intera e nella sua parte frazionaria: la parte intera è un `long l`, cioè un intero di dimensioni potenzialmente grandi, la parte frazionaria è un numero reale, `double f`. Ci piacerebbe scrivere:

```
(l,f) = decompone(3.5);
```

ma non è permesso. Allora, mettiamo `l` e `f` nella lista degli argomenti, come se fossero dati, passandoli per indirizzo per poterli modificare, in quanto sono dei risultati. Semplicemente, la funzione non li usa come dati: ignora il loro valore iniziale. Quindi, la funzione `decompone` ha un vero dato `double d` e due puntatori a risultati, `long *pl` e `double *pf`. Il corpo della funzione non fa altro che prendere il numero reale, trovarne la parte intera e assegnarla all'oggetto puntato da `pl` (non a `pl`, che è un puntatore); lo stesso fa per la parte frazionaria. Questo consente alla funzione di avere un impatto sul mondo esterno, di modificare un numero qualsiasi di valori, e quindi di avere risultati multipli.

5.4.2 Parametri di tipo vettore o stringa

Abbiamo lasciato aperta la questione delle stringhe. Quando abbiamo passato stringhe a funzioni, non le abbiamo mai referenziate applicando l'operatore `&`, ma le abbiamo sempre trattate direttamente. Il motivo è molto semplice ed è che la stringa è un vettore e i vettori in C non vengono mai passati alle funzioni completamente per copia, ma viene sempre passato solo l'indirizzo alla loro prima cella. L'esercizio di laboratorio è dedicato proprio a questo punto.

Decomposizione di un numero `double` nella parte intera e frazionaria

```
void decompone (double d, long *pl, double *pf);
int main (int argc, char argv[])
{
  double d, f;
  long l;

  d = 3.5;
  decompone(d,&l,&f);
  /* ora l vale 3L e f vale 0.5 */
  return EXIT_SUCCESS;
}

               void decompone (double d, long *pl, double *pf)
               {
                 *pl = (long) d;
                 *pf = d - *pl;
               }
```

Figura 5.39: Risultati multipli: esempio

Supponiamo di avere un file di testo contenente una sequenza di numeri interi. L'esempio in rete (`sort.txt`) contiene 100 numeri interi, che intendiamo caricare e poi stampare a video. Dobbiamo leggere il file di numeri interi, copiarli in un vettore e stampare quest'ultimo. Cominciamo con un'ipotesi semplificativa, necessaria finché ragioniamo in termini di vettori statici, cioè che il file non conterrà mai più di un dato numero `N` di numeri. Questa costante simbolica è già definita con una direttiva nel file iniziale `stampa_numeri0.c`, che contiene anche le solite direttive per gestire ingresso e uscita e per definire la solita lunghezza massimale per i nomi dei file e le righe lette. Anche l'interpretazione della linea di comando è già definita (si limita a controllare il numero dei parametri e copiare il primo, che dovrebbe essere il nome del file, in una stringa, come nel capitolo precedente). Infine, c'è un programma principale che dichiara la stringa con il nome del file e un vettore statico `V` di `N+1` elementi, in modo da poterlo indicizzare da 1 a `N`. Quindi, il programma principale chiama la funzione che interpreta la linea di comando. Una serie di commenti guidano le operazioni che l'esercizio richiede di aggiungere:

1. leggere gli elementi del file in un vettore;
2. stampare il vettore.

Al solito, procedendo *top-down*, facciamo corrispondere ad ogni sottoproblema una funzione che lo risolva.

5.4.3 Prima fase

La prima funzione `CaricaVettoreInteri` ha come dato il nome del file dei dati e dovrebbe avere come risultato un vettore di interi. Siccome il vettore è un dato multiplo e non si può assegnare un vettore a un altro, la scrittura naturale

```
V = CaricaVettoreInteri(filedati)
```

non è corretta. Quindi, come già fatto varie volte per le stringhe (la più recente in `InterpretaLineaComando`), trasformiamo il risultato in un dato fittizio. Inoltre, per essere un po' più generali, aggiungeremo un ulteriore parametro (attuale e

formale, nella chiamata e nella dichiarazione e definizione), che indichi la lunghezza del vettore stesso, o meglio il suo indice massimo (dato per scontato che quello minimo sia 1). Al momento, questo non serve a nulla (anzi, ci si può chiedere, giustamente, se non debba essere anch'esso un risultato della funzione), ma ne parleremo in seguito.

`CaricaVettoreInteri(filedati,V,N)`

A questo punto, dobbiamo costruire il prototipo della funzione

```
/* Carica dal file filedati una sequenza di interi nel vettore V indicizzato
da 1 a n */ void CaricaVettoreInteri (char *filedati, int *V, int n);
```

a cui corrisponderà la solita dichiarazione con un corpo lasciato temporaneamente vuoto.

Ora possiamo porci la domanda importante: perché i due dati, che sono vettori statici (di `char` e `int`), vengono tradotti in puntatori, rispettivamente `char*` e `int*`?

Il motivo è chiaro a valle di questo capitolo e del precedente: nel programma principale ci sono effettivamente due vettori, nella funzione `CaricaVettoreInteri` ci sono semplicemente due puntatori; il passaggio dei parametri assegna ai puntatori gli indirizzi delle celle iniziali dei due vettori grazie all'equivalenza fra vettori e puntatori, come se si trattasse di operazioni esplicite di assegnamento. Andremo nel dettaglio fra poco.

L'altra funzione avrà prototipo

```
/* Stampa a video il vettore di interi V indicizzato da 1 a n */ void
StampaVettoreInteri (int *V, int n);
```

e la corrispondente definizione, con il corpo ancora vuoto.

5.4.4 Seconda fase

Ora bisogna caricare e stampare il vettore di interi. Per caricarlo, la prima cosa da fare sarà aprire il file, con le consuete operazioni e controlli e la dichiarazione della consueta variabile di tipo puntatore a file, nonché la chiusura alla fine.

```
FILE *fp;
int i;


fp = fopen(filedati,"r");
if (fp == NULL)
{
  fprintf(stderr,"Errore nell'apertura del file %s!\n",filedati);
  exit(EXIT_FAILURE);
}
```

A questo punto, possiamo dedicarci alla lettura del file, di cui ipotizziamo di conoscere già esattamente il contenuto e il numero di elementi. Ci libereremo poi di questa forte ipotesi.

Possiamo fare un ciclo a conteggio con un cursore `i` che va da 1 a `n` e ad ogni iterazione legge un numero intero. I vari numeri sono separati da a capi, e quindi la funzione `fscanf` gestisce tranquillamente il passaggio dall'uno all'altro; lo farebbe anche senza modifiche nel caso di spazi bianchi e tabulazioni in numero qualsiasi. I numeri letti vanno inseriti nella cella di indice `i` del vettore `V`.

```
for (i = 1; i <= n; i++)
  if (fscanf(fp,"%d",&V[i]) != 1)
  {
    fprintf(stderr,"Errore nella lettura del file %s!\n",filedati);
    exit(EXIT_FAILURE);
  }
```

Ne approfittiamo per osservare che `V[i]` non è un dato per la funzione `fscanf`, ma un risultato. La funzione `fscanf`, però, ha un risultato standard, che è il numero degli oggetti riconosciuti, interpretati e assegnati a celle di memoria, e quindi gli altri risultati, che sono i singoli valori letti, devono passare attraverso il meccanismo del passaggio per indirizzo e figurare come dati fittizii. Ci si potrebbe chiedere perché quando si leggono stringhe, queste non hanno l'operatore di referenziazione `&`. La ragione è ovvia, ora: la stringa è vettore, quindi passandola per valore si passa il puntatore alla prima cella del vettore stesso. Quindi, passare direttamente la stringa significa già passare un puntatore ai caratteri che costituiscono la stringa. Nel caso di `V[i]`, invece, si passerebbe un numero intero, che non ha senso. Ogni volta che si passa un puntatore, esplicito o implicito (vettore o stringa) a una funzione per ricavarne dei dati l'operatore `&` non va usato. Quando invece si passano dei dati da modificare, l'operatore va usato.

La stampa del vettore è più semplice, perché non comporta di aprire file: si tratta semplicemente di scorrere gli elementi del vettore passato e stamparli a video.

```
int i;

for (i = 1; i <= n; i++)
  printf("%d ",V[i]);
```

Notiamo che in questo caso `printf` usa `V[i]` senza referenziazione, perché si tratta di un dato: ciascun numero intero viene letto dalla funzione per essere stamparto, e non modificato. In generale, non ha senso passare indirizzi a `printf`, a meno che proprio si vogliano stampare degli indirizzi.

A questo punto, il codice è completo e funziona correttamente, ma possiamo usarlo per sviluppare qualche discorso più sofisticato. In particolare, possiamo descrivere in dettaglio che cosa avviene alla chiamata di `StampaVettoreInteri(V,N);`

```
****** main ******
int V[] [ ?  61 74 ...  ]
char filedati[] ['s' 'o' 'r' 't' '.'  't' 'x' 't' '
0']
int argc 2
char *argv[] ["stampa_numeri0" "sort.txt"
*****************
```

dove `?` sta per un contenuto casuale.

Alla chiamata, si valutano i parametri attuali, che sono `V` e `N`, dove il secondo in realtà è `100`, dato che il precompilatore ha sostituito la costante simbolica col suo valore. Quindi, si alloca sullo *stack* lo spazio necessario alla funzione `StampaVettoreInteri`, che è costituito dai parametri formali (un puntatore a intero `int *V` e un intero `int n`) e dalla variabile locale (un intero `int i`), non essendovi risultati. Infine, si copiano i parametri attuali in quelli formali corrispondenti, cioè si copia il vettore statico `V` del programma principale nel puntatore a intero `V` della

funzione (usando l'equivalenza fra vettori e puntatori, cioè copiando l'indirizzo della prima cella del vettore) e si copia la costante `100` nell'intero `n`.

```
** StampaVettoreInteri **
int i
int *V ----+
int n 100 |
****** main ****** v
int V[] [ ?  61 74 ...  ]
char filedati[] ['s' 'o' 'r' 't' '.'  't' 'x' 't' '
0']
int argc 2
char *argv[] ["stampa_numeri0" "sort.txt"
******************
```

Grazie all'aritmetica dei puntatori, l'espressione `V[i]` ha nel corpo della funzione `StampaVettoreInteri` lo stesso significato che ha nel corpo del programma principale, dato che consiste nel sommare `i` volte la dimensione di un intero all'indirizzo puntato da `V` e recuperare il contenuto delle celle così ottenute, ma tali celle sono quelle del vettore `V[i]` originale. Per esempio, `V[1]` vale 61 in entrambi i casi. Al variare di `i` da 1 a `n`, si scorre il vettore e si può stamparlo. Al termine del ciclo, si eseguono le solite fasi per la terminazione della funzione. La valutazione del risultato non ha luogo perché la funzione è di tipo `void`, l'area di memoria sullo *stack* viene deallocata e si ritorna all'istruzione nella funzione chiamante, che può finalmente usare l'eventuale risultato, qui non presente.

5.4.5 Terza fase

Ora vogliamo leggere un file che non contenga necessariamente 100 elementi. Questo esclude completamente la possibilità di usare un vettore statico, perché mentre si scrive il programma non si sa quanto dovrà essere lungo il vettore. Si potrebbe pensare a una lunghezza massima, con due effetti negativi: che se la lunghezza viene superata, il programma non può far altro che terminare con un errore; che se la lunghezza viene rispettata, ma è molto alta, il programma è inefficiente in spazio.

Bisogna quindi usare un vettore dinamico, cioè un puntatore che contiene l'indirizzo non di un'area di memoria statica (come `int *V` nella sezione precedente), ma un'area di memoria allocata dinamicamente nello *heap* di sistema. Questo comporta una serie di modifiche. La prima è che nel programma principale `V` è dichiarato come puntatore a intero (`int *V`), anziché come vettore statico di interi (`int V[N+1]`). La seconda è che la lunghezza sarà una variabile intera `int n` dichiarata nel programma principale. La terza e la quarta sono che da qualche parte si dovrà allocare la memoria per `V` e che al termine bisognerà deallocarla. Infatti, la memoria statica è gestita automaticamente dal meccanismo di gestione dello *stack*, mentre la memoria dinamica va gestita esplicitamente dall'utente. Questo è il motivo per cui non si alloca mai dinamicamente una quantità di memoria nota a priori, che si potrebbe gestire staticamente. Allocare pone il problema di sapere quanti elementi avrà il vettore, cosa ignota prima di aver aperto e scorso il file.

Una possibilità è introdurre una funzione che apra il file, ne cont gli elementi e restituisca il numero. Non è molto elegante, se non altro perché il file andrà aperto una seconda volta per leggere gli elementi stessi, e le operazioni di accesso al disco fisso sono particolarmente lente (anche se sempre considerate di tempo costante).

Una possibilità migliore è assegnare alla funzione di lettura il compito di allocare il vettore di interi, eseguendola dopo l'apertura del file e prima del ciclo di lettura dei

suoi elementi. Rimane comunque il problema di conoscere il numero degli elementi. Per farlo, basta scorrere il file leggendo i numeri in esso contenuti con `fscanf`, assegnandoli non al vettore (che ancora non esiste), ma a una generica variabile intera. Scorrendoli, si possono contare, incrementando `n`, dopo averlo inizializzato a 0. Trovato il valore di `n`, si può allocare il vettore, e finalmente leggere i suoi elementi dal file. Resta il problema che non siamo più all'inizio del file, ma al suo termine, ma questo problema si risolve facilmente con l'istruzione `rewind(fp);`. Il codice che ne risulta è sintatticamente corretto e sembra affrontare tutti i punti del problema.

```
/* Carica dal file filedati nel vettore V di dimensione n i numeri da
    ordinare */
void CaricaVettoreInteri (char *filedati, int *V, int n)
{
  FILE *fp;
  int i;


  fp = fopen(filedati,"r");
  if (fp == NULL)
  {
    fprintf(stderr,"File %s non apribile!\n",filedati);
    exit(EXIT_FAILURE);
  }

  n = 0;
  while (fscanf(fp,"%d",&i) == 1)
    n++;

  V = (int *) calloc(n+1,sizeof(int));
  if (V == NULL)
  {
    fprintf(stderr,"Allocazione fallita!\n");
    exit(EXIT_FAILURE);
  }

  rewind(fp);

  for (i = 1; i <= n; i++)
    fscanf(fp,"%d",&V[i]);

  fclose(fp);
}
```

In realtà il compilatore fornisce due avvertimenti simili fra loro: `V` e `n` risultano `used uninitialized in this function`, riferendosi alla funzione `CaricaVettoreInteri(filedati,V,n)`. Il messaggio è abbastanza criptico, perché `V` e `n` vengono inizializzate in `CaricaVettoreInteri`, ma il messaggio si riferisce alle due variabili che hanno sono dichiarate con lo stesso nome nel programma principale e vengono passate alla funzione, come dati, senza essere state inizializzate. Potrebbe sembrare un messaggio inutilmente pignolo, dato che verranno inizializzate nella funzione, ma dobbiamo ricordare che in essa si lavora su copie, e quindi le due variabili restano non inizializzate anche dopo, quando verranno usate (e qui si avrà un errore!) in `StampaVettoreInteri`. Questo è un esempio di come i messaggi di avvertimento del compilatore possano essere difficili da interpretare, ma allo stesso tempo importanti come segnali di errore. Un programmatore inesperto potrebbe furbescamente pensare di rimuovere questi messaggi inizializzando `V` e `n` in qualche modo (per esempio con i valori `NULL` e `0`), ma questo sarebbe alquanto stupido, perché non solo non correggerebbe l'errore, ma lo nasconderebbe anche.

Rivediamo lo schema di esecuzione, considerando il momento in cui si chiama `CaricaVettoreInteri(filedati,V,n)`.

```
****** main ******
int n ?
int *V ?
char filedati[] ['s' 'o' 'r' 't' '.'  't' 'x' 't' '
0']
int argc 2
char *argv[] ["stampa_numeri0" "sort.txt"
******************
```

Il primo passaggio è la valutazione dei tre parametri formali, che sono la stringa `filedati`, il puntatore `V` e l'intero `n`. Gli ultimi due hanno valori non inizializzati esplicitamente, dunque casuali (cioè contengono i bit che le relative celle contenevano prima di eseguire il programma). Il secondo passaggio è allocare lo spazio richiesto dalla funzione, che include tre argomenti, cioè un puntatore a carattere (`char *filedati`), un puntatore a intero (`int *V`) e un intero (`int n`) e due variabili locali, cioè un puntatore a file (`FILE *fp_dati`) e un intero (`i`). Non c'è un risultato. Il terzo passaggio della chiamata copia i parametri attuali in quelli formali.

```
** CaricaVettoreInteri **
int n ?
int *V ?
char *filedati ---+
int i ?  |
FILE *fp_dati ?  |
****** main ****** |
int n ?  |
int *V ? v
char filedati[] ['s' 'o' 'r' 't' '.'  't' 'x' 't' '
0']
int argc 2
char *argv[] ["stampa_numeri2" "sort.txt"
******************
```

dove si intende che i valori casuali delle due variabili `V` coincidono fra loro, e lo stesso avviene per i valori casuali delle due variabili `n`. Questo comincia a suggerire che stia succedendo qualcosa di strano.

Ora l'operazione di apertura del file assegna a `fp_dati` l'indirizzo di un'area di memoria che contiene tutte le informazioni necessarie a gestire il file. Il ciclo che conta gli elementi del file prima azzera e poi incrementa `n` e modifica via via `i`, terminando con `n` pari a 100 e `i` pari all'ultimo valore letto (56). Quindi, allochiamo il vettore dinamico di lunghezza `n+1`, cioè 101. Questo riserva nello *heap* un'area di memoria formata da 101 blocchi di `sizeof(int)` celle consecutive, e scrive l'indirizzo della prima cella di tale area in `V`. Ritornati all'inizio del file, lo scorriamo ancora leggendo via via i numeri e scrivendoli in `V[i]`, cioè nelle celle allocate nell'*heap*. Alla fine, chiudiamo il file.

```
 STACK HEAP

** CaricaVettoreInteri **
int n 100
int *V ------------------------------------------>[ 0 61 74 ...  ]
```

```
char *filedati ---+
int i 56 |
FILE *fp_dati ---)--------> FILE
****** main ****** |
int n ?  |
int *V ? v
char filedati[] ['s' 'o' 'r' 't' '.'  't' 'x' 't' '
0']
int argc 2
char *argv[] ["stampa_numeri2" "sort.txt"
******************
```

Fin qui, tutto bene. Il problema è che quando la funzione termina, non essendovi alcun risultato, si ha semplicemente la deallocazione dello *scope* della funzione, con una serie di pessimi effetti:

- il valore di `n` nel programma principale è ancora casuale;
- il valore di `V` nel programma principale è ancora casuale;
- l'area di memoria allocata dinamicamente è ancora occupata, ma irraggiungibile;
- i dati letti sono andati persi.

```
 STACK HEAP
[ 0 61 74 ...  ]
****** main ******
int n ?
int *V ?
char filedati[] ['s' 'o' 'r' 't' '.'  't' 'x' 't' '
0']
int argc 2
char *argv[] ["stampa_numeri2" "sort.txt"
******************
```

Finché restiamo in `CaricaVettoreInteri`, possiamo manipolare il vettore correttamente (per esempio, si può stamparlo chiamando `StampaVettoreInteri` dall'interno). Appena usciamo, lo abbiamo perso. Se proviamo a stampare il vettore, possiamo ottenere nessun risultato (in particolare, se per caso `n` ha valore casuale 0) o il programma può bloccarsi o interrompersi.

5.4.6 Quarta fase

Avendo capito l'errore, possiamo procedere a risolverlo. Non dobbiamo usare variabili `V` e `n` interne alla funzione, ma quelle del programma principale. Quindi, dobbiamo passarle entrambe per indirizzo, come abbiamo già fatto per la funzione `Scambia`.

```
CaricaVettoreInteri(filedati,&V,&n);
```

La modifica dei parametri formali richiede una serie di modifiche nel corpo della funzione. Per `n`, conviene rinominare il parametro formale `pn` per ricordare che è un puntatore e bisogna usare `*pn` nel corpo della funzione, per poter accedere alla variabile vera e propria. C'è qualche complicazione dovuta alle precedenze fra

operatori: per esempio, si deve scrivere `(*pn)++;`, perché `*pn++;` eseguirebbe prima l'incremento sul puntatore `pn` e poi andrebbe a leggere il blocco successivo a quello puntato da `pn`, anziché modificare fisicamente il contenuto del blocco puntato da `pn`.

Per `V` succede in effetti lo stesso, ma la scrittura si fa più complicata. Per prima cosa, se `V` è un puntatore a intero, `pV` è un puntatore a un puntatore a intero, ovvero un doppio puntatore a intero (`int **pV`). Quindi, l'area di memoria dinamica viene assegnata non a `V`, ma a `*pV` (che è un *alias* della variabile `V` del programma principale). La complicazione più grossa è accedere in scrittura all'elemento `i`-esimo del vettore `V` usando il suo *alias* `*pV`. Siccome l'operatore di proiezione (`[]`) ha una precedenza superiore a quello di dereferenziazione, bisogna racchiudere l'*alias* fra parentesi (dunque, `(*pV)[i]`) e poi applicare l'operatore di referenziazione per passare alla funzione `scanf` l'indirizzo dell'elemento del vettore. La scrittura è decisamente barocca, ma perfettamente sensata, se si comprende la situazione e le regole del linguaggio C.

```
/* Carica dal file filedati nel vettore *pV di dimensione *pn i
    numeri da ordinare */
void CaricaVettoreInteri (char *filedati, vint *pV, int *pn)
{
  FILE *fp;
  int i;


  fp = fopen(filedati,"r");
  if (fp == NULL)
  {
    fprintf(stderr,"File %s non apribile!\n",filedati);
    exit(EXIT_FAILURE);
  }

  *pn = 0;
  while (fscanf(fp,"%d",&i) == 1)
    (*pn)++;

  *pV = (int *) calloc(*pn+1,sizeof(int));
  if (*pV == NULL)
  {
    fprintf(stderr,"Allocazione fallita!\n");
    exit(EXIT_FAILURE);
  }

  rewind(fp);

  for (i = 1; i <= *pn; i++)
    fscanf(fp,"%d",&(*pV)[i]);

  fclose(fp);
}
```

Dal punto di vista dell'esecuzione, può essere utile dare un'occhiata alla situazione in cui ci si trova al termine dell'esecuzione della funzione.

```
 STACK HEAP

** CaricaVettoreInteri **
int *pn -------+
int **pV -+ |
char *filedati | ---+ |
int i | 56 | |
```

```
FILE *fp_dati | ---)---)----> FILE
****** main *****| | v
int n v | 100
int *V ----v----------------------------------------->[ 0 61 74 ...  ]
char filedati[] ['s' 'o' 'r' 't' '.'  't' 'x' 't' '
0']
int argc 2
char *argv[] ["stampa_numeri2" "sort.txt"
******************
```

dove le frecce sottolineano il fatto che le informazioni che vengono aggiornate si trovano tutte o nello *scope* del programma principale o in aree dello *heap* puntate da variabili che stanno in quello *scope*.

Bisogna fare lo stesso per `StampaVettoreInteri`? Assolutamente no, per il banale motivo che per questa funzione `V` e `n` sono effettivamente dati, e non risultati.

In sintesi, il punto sostanziale è la differenza sostanziale tra dato e risultato: il dato si passa per copia, il risultato per indirizzo. Nel caso dei vettori, bisogna capire se

- vettore ed elementi sono dati da leggere: si passa il vettore per copia;
- il vettore e' un dato, gli elementi sono risultati: si passa il vettore per copia (e quindi gli elementi per indirizzo);
- il vettore nella sua totalità è un risultato (da allocare, deallocare, riallocare con dimensione diversa, scalare spostando il puntatore alla cella iniziale): si passa il vettore per indirizzo.

L'ultima osservazione è che un doppio puntatore `int **V` può essere:

- un puntatore a un puntatore a un singolo intero;
- un puntatore a un vettore dinamico di interi (come nell'esercizio appena svolto);
- un vettore dinamico di puntatori a interi;
- una matrice dinamica di interi.

Siccome non c'è modo di sapere quale sia il caso dalla dichiarazione in sé, la notazione è estremamente ambigua. È però possibile chiarificarla. Nel seguito, useremo il simbolo `*` solo per i puntatori veri e propri, mentre sostituiremo i vettori e le matrici con tipi definiti da utente. Per esempio, i vettori dinamici saranno denotati come `vint` attraverso l'istruzione:

```
typedef int* vint;
```

il che consentirà di definire il parametro formale `pV` come `vint *`, chiarendone la natura di puntatore a un vettore dinamico di interi. Corrispondentemente, nel programma principale avremo `vint V;` anziché `int *V;`.

Non cambia assolutamente nulla dal punto di vista tecnico, per il processore, ma il programmatore avrà più chiaro che cosa stia succedendo.

FIGURE DA INSERIRE NEL TESTO

Alla chiamata, i vettori vengono convertiti in puntatori e passati

- Non si deve specificarne la lunghezza, anche se sono statici
  Non `int funzione(int V[LUNGHEZZA]);`
  ma `int funzione(int *V);`
  Se occorre conoscerla, la lunghezza si passa come parametro
- gli elementi di un vettore sono passati per indirizzo, il vettore per valore: corrispondentemente, le modifiche agli elementi si conservano, quelle al vettore no dato che il puntatore non cambia

Figura 5.40: Dati vettoriali

Vi sono due modi per far scrivere una funzione che restituisce un vettore

1. la funzione alloca un vettore dinamico e
   restituisce il puntatore alla funzione chiamante
2. la funzione chiamante passa per indirizzo un puntatore alla funzione,
   la funzione alloca un vettore dinamico e
   modifica il puntatore assegnandogli il vettore dinamico

```
int main (int argc, char argv[])
{
  int *V;
  int n;

  n = 10;
  V = creavettore(n);
  /* ora V e' un vettore dinamico */
  free(V);
  return EXIT_SUCCESS;
}

int *creavettore (int n)
{
  int *V;
  V = (int *) calloc(n+1,sizeof(int));
  return V;
}
```

```
int main (int argc, char argv[])
{
  int *V;
  int n;

  n = 10;
  creavettore(&V,n);
  /* ora V e' un vettore dinamico */
  free(V);
  return EXIT_SUCCESS;
}

void creavettore (int **pV, int n);
{
  *pV = (int *) calloc(n+1,sizeof(int));
}
```

Figura 5.41: Risultati vettoriali

Le strutture si passano per valore copiando campo per campo, ma

- se contengono campi puntatore, ci si limita a copiare gli indirizzi:
  gli oggetti puntati risultano passati per indirizzo

Si usa passare per indirizzo le strutture di grandi dimensioni, anche quando non si vuole modificarle, per rendere più efficiente il passaggio

Si copia un semplice indirizzo anziché molte celle di dati

Figura 5.42: Passaggio delle strutture

5.5 Laboratorio

L'esercizio illustra:

- la gestione dei vettori statici e dinamici (puntatori con memoria allocata nello *heap*);
- l'equivalenza fra vettori e puntatori;
- la differenza tra il passaggio dei parametri alle funzioni per valore e per indirizzo.

L'esercizio consiste nel caricare un elenco di numeri interi da un file di testo `sort.txt` in un vettore della lunghezza corretta e nello stampare il vettore stesso, interamente o in parte. Prelude a una lezione successiva, nella quale il vettore di numeri verrà ordinato.

Chi notasse eventuali incoerenze o errori, oppure avesse dubbi sul contenuto di queste pagine e dei codici, è pregato di segnalarmeli per contribuire a migliorare la qualità dei materiali del corso.

Problema

Si scriva un programma `stampa_numeri.c` che apre un file di testo, carica nelle posizioni di 1 a n di un vettore dinamico della dimensione corretta i numeri in esso contenuti e stampa gli elementi del vettore nell'ordine.

Traccia della risoluzione

Per cominciare, faremo l'ipotesi (piuttosto forte) di sapere che il file di testo contiene esattamente `N` = 100 numeri interi. Il file `stampa_numeri0.c` contiene le solite inclusioni di librerie, le costanti simboliche `ROW_LENGTH` per la lunghezza del nome del file e `N` per il numero di interi da leggere, il `main` e la dichiarazione, chiamata e definizione della procedure per interpretare la linea di comando e ottenere il nome del file di testo. Queste ultime sono identiche a quelle realizzate nella lezione precedente.

Prima fase (`stampa_numeri1.c`) Impostiamo la risoluzione del problema in maniera *top-down*, dividendolo in due passi fondamentali:

1. caricamento dei numeri in un vettore statico;
2. stampa del vettore;

Entrambi i passi sono piuttosto semplici, e non richiedono nessuna osservazione particolare, salvo il fatto di osservare che entrambe le procedure ricevono non un vettore di interi, ma un puntatore a intero, come anticipato nei lucidi della lezione.

Seconda fase (`stampa_numeri2.c`) L'aspetto principale dell'esercizio sta nel fatto che molto spesso non conosceremo in anticipo (cioé mentre scriviamo il codice) il numero di elementi contenuti nel vettore, anche perché probabilmente vorremo utilizzare il programma per leggere file di lunghezza diversa. Questo richiede necessariamente di sostituire il vettore statico con un vettore dinamico, cioè con un puntatore a intero al quale venga assegnata un'area di memoria opportunamente dimensionata, creata con la funzione `calloc`, e deallocata al termine dell'esecuzione.

Sarebbe spontaneo ora dividere il problema in quattro passi, aggiungendo prima del caricamento dei numeri l'allocazione del vettore dinamico e dopo la stampa la deallocazione. Tuttavia, mentre la deallocazione va effettivamente aggiunta in coda al `main`, l'allocazione non può precedere il caricamento, dato che la lunghezza del vettore si può conoscere solo aprendo il file e contandone gli elementi[10]. Procederemo quindi ad aprire il file e a scorrerne gli elementi dal primo all'ultimo contandoli. Detto n il loro numero, allocheremo un vettore di $n + 1$ interi, per caricarli nelle posizioni da 1 a n secondo la convenzione di questo corso[11]. Quindi, torneremo al principio del file (`rewind`) e rileggeremo gli elementi caricandoli nel vettore. In questa seconda lettura, possiamo dare per scontato che sappiamo quanti sono e che abbiano il formato corretto.

Un altro aspetto importante di questo passo è che vettore e numero di elementi devono essere restituiti dalla procedura di caricamento come risultati. Siccome il risultato è multiplo (sono esattamente $n + 1$ interi), non si può usare l'istruzione `return` (in una lezione successiva, dedicata alle *tabelle*, vedremo un modo per farlo, il cui uso non è però molto comune). Bisogna invece passare i risultati come se fossero dei dati fittizii. La soluzione di questo passo fornita in rete é sbagliata: la lunghezza `n` e il vettore `V` vengono costruiti correttamente (si possono stampare da dentro la procedura), ma non vengono restituiti correttamente al `main`.

Terza fase (`stampa_numeri3.c`) L'errore della fase precedente è che i risultati vanno passati per indirizzo, e non per valore. Per la cardinalità n questo è facile: basta definire un parametro `int *pn`. Per il vettore si potrebbe obiettare che `int *V` é già un puntatore, come si è osservato nel primo passo dell'esercizio. Invece, non basta. Se l'allocazione del vettore V avviene dentro la funzione, il parametro formale `int *V` è solo una copia dell'omonimo puntatore `int *V` definito nel `main`:

- quando si alloca il nuovo vettore, la cella `V` della funzione di caricamento contiene l'indirizzo della nuova area allocata dinamicamente sullo *heap*, ma la cella `V` del `main` continua a contenere l'originale indirizzo casuale che aveva al principio del programma;

- quando si leggono i numeri contenuti nel file, essi vengono correttamente scritti nell'area allocata dinamicamente e puntata dal parametro `V` della funzione;

[10] Se fossimo fortunati, il file comincerebbe indicando il numero degli elementi, ma non siamo in questa situazione. Del resto, anche in questo caso, dovremmo comunque almeno aprire il file per leggere questo numero.

[11] In questo esercizio andrebbe bene anche indicizzarli da 0 a $n - 1$, ma se fosse necessario stampare oltre ai numeri anche degli indici da 1 a n la convenzione standard del C ci costringerebbe a ricordare ad ogni passo se per ottenere l'indice da stampare bisogna incrementare o decrementare quello effettivo nel vettore: ho visto intere generazioni di studenti consumare energie preziose in questo compito stupido, talvolta fallendo. C'è anche la possibilità di allocare un vettore dinamico con indici da 0 a $n-1$, spostarlo un passo indietro con l'*aritmetica dei puntatori* e usare il puntatore modificato. Questo andrebbe bene, ma richiederebbe di spostare nuovamente il puntatore un passo avanti prima di deallocarlo. Sforzo piuttosto ingiustificato per risparmiare una cella di memoria.

- quando si esce dalla funzione, la cella contenente il parametro `V` viene deallocata dallo *stack* e rimane solo la cella `V` del `main`, che punta un indirizzo a caso: l'area allocata dinamicamente è persa e irraggiungibile.

Per evitare questo errore, bisogna passare per indirizzo anche il puntatore `V`, cioè scrivere `CaricaVettoreInteri(char *filedati, int **pV, int *pn)`. Il nome `pV` del parametro cerca di ricordare che si tratta non di un vettore, ma di un puntatore alla cella `V` del `main` che è destinata a contenere l'indirizzo della prima cella del vettore dinamico. Per facilitare un po' l'interpretazione, si può introdurre un tipo ausiliario `vint` con l'istruzione `typedef int* vint;`. Questo nuovo tipo rappresenta i vettori dinamici di interi e coincide con i puntatori, ma si chiama in modo diverso per chiarire che ha un significato diverso. La dichiarazione diventa `CaricaVettoreInteri(char *filedati, vint *pV, int *pn)`.

A questo punto, la scrittura dell'allocazione del vettore e del caricamente dei dati diventa piuttosto barocca, perché il vettore da allocare e in cui scrivere non è `pV`, che è un puntatore a vettore, ma `*pV`, che è un vettore, e precisamente è un *alias* della variabile `V` del `main`. Quindi, dovremo scrivere ovunque `*pV`. Dovremo inoltre racchiuderlo spesso fra parentesi tonde per garantire la corretta precedenza degli operatori. Ad esempio, per leggere e scrivere i singoli elementi del vettore useremo l'espressione `(*pV)[i]`.

A questo punto, può venire il dubbio che anche la stampa del vettore richieda un puntatore al vettore stesso e uno alla sua lunghezza. Non lo richiede, per il semplice motivo che qui i due oggetti sono dati, non risultati, e la stampa non li modifica. Quindi si passano per valore, e il prototipo della funzione di stampa rimane `StampaVettoreInteri(vint V, int n)`. Anche funzioni che dovessero modificare gli elementi del vettore continuerebbero a richiedere di passare i parametri in questo modo. Infatti, pur essendo il vettore passato per valore, i suoi elementi sono implicitamente sempre passati per indirizzo. Il passaggio di vettori per indirizzo serve solo quando si vuole modificare il vettore in quanto oggetto globale (per esempio per riallocarlo, deallocarlo o spostarlo in altre aree di memoria, per esempio con l'aritmetica dei puntatori). Solo in questi casi occorre la complicata procedura sopra descritta.

A questo punto, è possibile stampare sottovettori del vettore sfruttando l'equivalenza con i puntatori,, cioè passando alla funzione di stampa non `V` e `n`, ma l'indirizzo della cella 0 del sottovettore e la sua lunghezza.

Del tutto banale, infine, é la deallocazione del vettore (`free`). Se fosse seguita da altre istruzioni, potrebbe valere la pena di annullare il puntatore (`V = NULL;`) per evitare di usarlo in maniera scorretta in seguito. Ma qui non ci sono altre operazioni. Si potrebbe anche pensare che non valga la pena di deallocare il vettore esplicitamente, perché viene fatto in automatico al termine del programma. In realtà, conviene farlo perché la deallocazione tipicamente produce situazioni di errore nei casi in cui il vettore è stato usato in modo scorretto durante l'esecuzione (per esempio, scrivendo in celle esterne ad esso). Questi errori non verrebbero scoperti lasciando la deallocazione al sistema operativo.

5.6 Esercizi

Esercizio Sfruttando la funzione StampaVettoreInteri senza modificarla e la possibilità di passarle un puntatore anziché un vettore si realizzi un programma che carica da un file di testo una sequenza di numeri interi, li stampa, e stampa una sottosequenza compresa fra due indici dati da linea di comando.

Esercizio Si realizzi una funzione IncrementaVettoreInteri, che carica da un file di testo una sequenza di numeri interi, li stampa, li incrementa di un valore passato dal linea di comando e li ristampa.

Esercizio Si realizzi una funzione ScalaVettoreInteri, che carica da un file di testo una sequenza di numeri interi, li stampa, sposta di un dato numero di passi avanti o indietro passato da linea di comando e ristampa quelli con indici sensati per entrambi i vettori (originale e riscalato).

5.7 Lezione 5

Esercizi sui puntatori[12]

Esercizio 1 Date le dichiarazioni `int i;` e `int *p;`, quali delle seguenti espressioni sono lecite?

- `*p`
- `&p`
- `*&p`
- `&*p`
- `*i`
- `&i`
- `*&i`
- `&*i`

Quali sono fra loro equivalenti?

Esercizio 2 Date le dichiarazioni `int i;` e `int *p, *q;`, quali dei seguenti assegnamenti sono leciti? Quali sono sicuri?

- `p = 1;`
- `p = &i;`
- `&p = q;`
- `p = &q;`
- `p = *&q;`
- `p = q;`
- `p = *q;`
- `*p = q;`
- `*p = *q;`

Esercizio 3 Dato il vettore `int V[8]` di valore `[ 5 15 34 54 14 2 52 72 ]` e i due puntatori `int *p, *q`, inizializzati rispettivamente con

```
p = &V[1];
q = &V[5];
```

- quanto vale `*(p+3)`?
- quanto vale `*(q-3)`?
- quanto vale `q-p`?
- è vero o falso che `p < q`?
- è vero o falso che `*p < *q`?

[12] tratti o ispirati dal testo di K.N. King

Esercizio 4 Supponendo che `s`, `d` e `m` siano puntatori a elementi di un vettore, si vuole scrivere un'istruzione che faccia puntare `m` all'elemento intermedio fra `s` e `d` (nel caso vi siano due elementi intermedi, si consideri quello di sinistra; ad esempio, per `s = &V[3]` e `d = &V[6]` sia `m = &V[4]`). L'istruzione `m = (s + d)/2;` è scorretta. Perché? Come ottenere un'istruzione corretta, usando l'aritmetica dei puntatori?

Esercizio 5 Sia `int V[10]; int *p;` e `p = V;`
Indicare se le seguenti espressioni sono lecite o no, e per quelle lecite se sono vere o false:

- `p == V[0]`
- `p == &V[0]`
- `*p == V[0]`
- `p[0] == V[0]`

Esercizi sull'allocazione

Esercizio 1 Si considerino le seguenti dichiarazioni:

```
struct punto { int x, y; };
struct rettangolo { struct punto no, se; };
struct rettangolo *pr;
```

Si scriva un programma che alloca e assegna al puntatore `pr` un rettangolo con vertice superiore sinistro in $(10, 25)$ e vertice inferiore destro in $(20, 15)$.

Esercizio 2 Se `x` è una struttura e `a` un suo membro, è vero o falso che l'espressione `(&x)->a` equivale all'espressione `x.a`?

Esercizi sulle funzioni

Esercizio 1 Scrivere un programma che contenga una funzione `int range(int x, int y, int n)`, la quale restituisca `1` se `x` e `y` sono compresi fra `0` e `n-1` inclusi, e `0` in ogni altro caso.

Esercizio 2 Data la funzione `int f(int a, int b)` e date le variabili `int i` e `double d`, quali delle seguenti istruzioni sono ammissibili?

- `i = f(83,12);`
- `d = f(83,12);`
- `i = f(3.15,9.28);`
- `d = f(3.15,9.28);`
- `f(83,12);`

Esercizio 3 Scrivere un programma che contenga tre funzioni con parametri `int *V` e `n` (dove `V` è un vettore di lunghezza `n`), le quali calcolino rispettivamente:

- il massimo elemento del vettore `V`
- il valor medio degli elementi del vettore `V`
- il numero di elementi positivi del vettore `V`

Esercizio 4 Scrivere un programma che contenga una funzione la quale calcoli il prodotto scalare di due vettori di ugual lunghezza, contenenti numeri reali.

Esercizio 5 La seguente funzione dovrebbe restituire `TRUE` se il vettore `V` contenesse almeno un elemento nullo, `FALSE` altrimenti. Però è sbagliata. Come si può correggerla?

```
boolean HaZeri (int *V, int n)
{
  int i;

  for (i = 1; i <= n; i++)
    if (V[i] == 0)
      return TRUE;
    else
      return FALSE;
}
```

Capitolo 6

Tabelle e algoritmi di ordinamento quadratici

Questo capitolo tratta due argomenti collegati, ma abbastanza distinti tra loro. Il primo è una struttura dati astratta, detta *tabella*, che ha lo scopo di allentare alcune delle rigidità del vettore, al costo di un aggravio temporale per l'esecuzione di alcune operazioni. Il secondo è il problema dell'ordinamento e alcuni algoritmi per risolverlo. In breve, si tratta di trasformare un insieme di oggetti, sui quali sia definita una relazione d'ordine, in una permutazione tale che gli elementi seguano lo stesso ordine imposto dalla relazione. Si tratta di algoritmi semplici, con una complessità quadratica, dunque non ecezionale, ma accettabile per istanze di dimensione non troppo grande. Uno dei due algoritmi (*InsertionSort*) viene trattato anche nelle dispense di teoria, ma lo riconsidereremo, cercando di mettere in luce alcuni aspetti e soprattutto tratteremo l'implementazione in linguaggio C di entrambi.

6.1 Tabelle

I vettori, trattati nel Capitolo 4, hanno una dimensione fissata una volta per tutte. Se sono vettori statici, questa dimensione è addirittura fissata da chi scrive il programma (per esempio, il gioco dell'Othello si svolge in una matrice $8 * 8$, e non si può cambiare senza riscrivere il codice). Se sono vettori dinamici, la dimensione viene calcolata in qualche modo durante l'esecuzione, si crea il vettore in un'area dello *heap*, e poi la dimensione non cambia nel resto dell'esecuzione[1]. Potrebbe però capitare di voler raccogliere delle informazioni la cui dimensione non solo non è nota a priori, ma è anche variabile durante l'elaborazione. L'elenco dei clienti di un servizio *online*, per esempio, vede arrivare nuovi nominativi durante il servizio e sparire quelli via via serviti. Le applicazioni sono molteplici. Per cominciare faremo un'ipotesi semplificativa, che poi cercheremo di rilassare, cioè che si conosca una stima per eccesso nel numero di elementi possibili.

Come nel caso dei vettori, partiamo da una definizione astratta. Una tabella è una n-upla ordinata di oggetti (potenzialmente con ripetizioni) appartenenti a un insieme base U (vedi Figura 6.1), esattamente come un vettore. Diversamente da un vettore, il numero n dei suoi elementi non è fissato a priori, ma può variare nel corso del tempo, fra 0 e un valore massimo k. Questo valore è ciò che definiremo *dimensione* della tabella. L'insieme di tutte le possibili tabelle è facilmente definibile

[1] Accantoniamo per il momento la possibilità di cambiarla con l'istruzione `realloc`: qui stiamo trattando la struttura dati *astratta* di nome vettore.

come l'insieme di tutti i vettori di dimensione k, unito all'insieme dei vettori di dimensione $k-1$, $k-2$, e così via, fino ai vettori di dimensione 1 (singoli elementi dell'insieme base U) e persino il vettore di dimensione 0, che interpretiamo come l'insieme vuoto. L'insieme di tutte le tabelle è quindi $\cup_{n=0}^{k} U^n$

I vettori hanno una dimensione fissata una volta per tutte

Spesso occorre raccogliere un numero di informazioni

- non noto a priori
- variabile durante l'elaborazione

ma di cui si conosce una stima per eccesso

Una tabella T di dimensione k su un insieme U è definita come una n-upla ordinata $(v_1, \ldots, v_n)$ di elementi di U con $n \in \{0, \ldots, k\}$

La tabella ha una cardinalità n scelta a piacere e può anche essere vuota

La struttura dati astratta è definita come

- l'insieme $\mathcal{T}_{k,U}$ di tutte le possibili tabelle di dimensione k su U

$$\mathcal{T}_{k,U} = \bigcup_{n=0}^{k} U^n$$

Figura 6.1: Tabelle: struttura dati astratta

Dato l'insieme delle tabelle, bisogna introdurre le operazioni che si possono eseguire su di loro (vedi Figura 6.2). Data la somiglianza con i vettori, possiamo aspettarci che abbiamo le stesse operazioni, cioè la proiezione e la sostituzione, la lettura e la scrittura. Cambia un aspetto fondamentale: nel caso della lettura dell'elemento di indice i di una tabella T, distinguere fra indici leciti e indici illeciti non è ovvio a priori. Anche se i è un intero positivo $\leq k$, la tabella T potrebbe attualmente non contenere un oggetto di indice i (per esempio, potrebbe essere vuota). Quindi l'operazione di proiezione non fornisce necessariamente un risultato appartenente a U: il risultato potrebbe essere indeterminato. In termini matematici, una tabella è una funzione parziale con dominio $\{1, \ldots, k\}$. Dobbiamo scegliere un comportamento da tenere quando il risultato è indeterminato. Si potrebbe abbandonare il programma, oppure restituire un valore speciale, il cui significato è segnalare l'errore. Si può anche scegliere di non gestire esplicitamente la cosa, caricando l'utente del compito di non superare i limiti prefissati. In questo caso, il comportamento del programma sarà effettivamente indeterminato, e in genere diverso da macchina a macchina e da compilatore a compilatore. È una scelta pericolosa, che può essere motivata e compensata da una superiore efficienza.

Per quanto riguarda la sostituzione, cioè la scrittura, vale lo stesso: l'operazione riceve una tabella T, un elemento dell'insieme base u e un indice numerico i compreso fra 1 e k e restituisce una tabella identica a T, ma contenente l'elemento u nella posizione di indice i, al posto dell'elemento originale. Come nel caso della lettura, questo ha senso se l'indice i è attualmente lecito, cosa che può valere o non valere in momenti diversi dell'esecuzione.

La distinzione fra indici leciti e non leciti si basa essenzialmente sul fatto che i sia compreso fra 1 e n oppure no. Indicheremo il numero n corrente di elementi

come *cardinalità* della tabella. Siccome questo valore gioca un ruolo fondamentale, è spontaneo aggiungere un'operazione alla struttura dati astratta, che consenta di conoscere questo numero. Questa è l'operazione di *cardinalità*, appunto, che riceve una tabella T e restituisce un numero naturale compreso fra 0 e k, che coincide con il numero degli elementi della tabella T. È il fatto di poter eseguire questa operazione che consente di lasciare indeterminato il comportamento in lettura e scrittura per indici illeciti: l'idea, infatti, è che sia compito dell'utente usare l'operazione di cardinalità per verificare che le operazioni di lettura e scrittura siano lecite in tutti i casi in cui questo non è matematicamente dimostrato per altra via.

La dinamicità delle tabelle, inoltre, richiede di poter aggiungere e togliere elementi. A questo servono l'operazione di *inserimento*, che riceve una tabella T e un elemento u dell'insieme base e restituisce una nuova tabella uguale a T, salvo che contiene u come ultimo elemento, incrementando quindi la cardinalità. Il fatto di aggiungere l'elemento in fondo è ovviamente arbitrario, ma è legato alle possibili implementazioni, che discuteremo in seguito. Può sembrare che questa funzione non comporti problemi particolari, ma è possibile che si cerchi di inserire un elemento in una tabella piena, cioè di cardinalità n pari alla dimensione k. Riprenderemo poi l'argomento. Poi avremo un'operazione di *cancellazione*, che riceve una tabella T e un indice numerico i (supposto lecito) e restituisce una tabella contenente tutti gli elementi di T tranne quello di indice i. Si tratta ancora una volta di una funzione definita parzialmente, perché l'elemento di indice i potrebbe non esistere.

Le tabelle

- ammettono le operazioni di proiezione $\pi_i(T)$ e sostituzione $\sigma_i(T, u)$ ma occorre verificare che l'indice i sia in $\{1, \ldots, n\}$
- possono ammettere altre operazioni:
  - cardinalità $\mathrm{card}(T)$ che associa a una tabella il numero degli elementi
    $$\mathrm{card} : \mathcal{T}_{k,U} \to \{0, \ldots, k\}$$
  - inserimento $\mathrm{ins}(T, u)$ che associa a una tabella e a un elemento la tabella ottenuta aggiungendo l'elemento in posizione terminale
    $$\mathrm{ins} : \mathcal{T}_{k,U} \times U \to \mathcal{T}_{k,U}$$
    Occorre verificare che la dimensione non ecceda la soglia k
  - cancellazione $\mathrm{canc}(T, i)$ che associa a una tabella e a un indice la tabella ottenuta cancellando l'elemento associato all'indice
    $$\mathrm{canc} : \mathcal{T}_{k,U} \times \{1, \ldots, k\} \to \mathcal{T}_{k,U}$$
    Occorre verificare che l'indice non ecceda la cardinalità n

Figura 6.2: Tabelle: operazioni

Consideriamo una prima implementazione pratica in C, la più spontanea e naturale. Vedremo che è possibile introdurne altre, e ne discuteremo vantaggi e svantaggi. Una tabella può essere rappresentata con una struttura, che contenga un vettore `V` di k elementi del tipo U, in modo da poter gestire tutte le tabelle possibili. Il vettore è dinamico, perché in questo modo possiamo usare il codice per gestire tabelle

di qualsiasi dimensione; in caso contrario, bisognerebbe riscrivere tutto per ogni singola dimensione. È poi necessario includere nella struttura il valore intero della dimensione `k` e quello della cardinalità corrente `n`. Il primo numero resterà fisso durante l'uso della tabella (dalla sua creazione alla distruzione), il secondo varierà in base alle operazioni di inserimento e cancellazione. L'istruzione `typedef` permette di sostituire alla dichiarazione composta di tipo `struct _tabella` un nome simbolico semplice `tabella`, in modo che le dichiarazioni delle variabili di tipo `tabella` siano più chiare e semplici. Questo consente di avvicinare il codice il più possibile alla struttura dati astratta, cioè di fingere che si stiano manipolando oggetti matematici (tabelle) senza dover tenere conto esplicitamente del fatto che in realtà sono delle `struct` con le loro particolarità tecniche. Nel resto del corso cercheremo di fare questo il più possibile, anche se a volte non sarà tecnicamente possibile e altre volte, rinunceremo a farlo per questioni di efficienza.

In C una tabella si può realizzare con una struttura contenente

- un vettore di k elementi di tipo U
- il valore intero k, che rappresenta la dimensione allocata, costante
- il valore intero n, che rappresenta la cardinalità, variabile

Una tabella T di oggetti di tipo U si dichiara come segue:

```
typedef struct _tabella tabella;
struct _tabella {
   U *V;
   int k;
   int n;
};
tabella T;
```

Per poterla usare, non bisogna dimenticare le procedure per la

- creazione, cioè per l'allocazione del campo `V`
- distruzione, cioè per la deallocazione del campo `V`

In realtà spesso si tengono i tre dati separati senza accorparli in un record

Figura 6.3: Tabelle: implementazione in C

6.1.1 Costruzione e distruzione di strutture dati

Inoltre, per gestire le strutture che simulano strutture dati astratte tipicamente dovremo introdurre operazioni esplicite di costruzione e distruzione. Siccome il vettore `V` è dinamico, prima di poter usare una tabella dovremo allocare la memoria corrispondente, e al termine dovremo deallocarla. In una struttura dati astratta questo è ovviamente non necessario: basta nominare la struttura ed essa compare dal nulla, per sparire da sola quando non occorre più. I vettori statici sono le sole strutture dati astratte composte (ignoriamo i dati semplici come numeri interi, reali, caratteri, ecc...) per i quali la creazione e la distruzione avvengono in modo praticamente implicito (a rigore, la dichiarazione è necessaria, e svolge proprio questo ruolo, ma è facile dimenticarlo). Ogni altra struttura dati astratta richiede

operazioni ad hoc, che in genere si accorpano in una funzione di creazione e in una funzione di distruzione.

Va anche osservato che spesso chi usa una tabella non si prende la briga di adottare un approccio basato sulle strutture dati astratte, ma si limita a gestire esplicitamente un vettore dinamico e due numeri interi, tenendoli separati, anche perché `k` spesso svolge qualche altro ruolo nell'algoritmo, e quindi è già presente sotto altra forma nel codice. Come si è già detto, i vantaggi delle strutture dati astratte sono di leggibilità e progettazione, non di efficienza, e le strutture più semplici tendono ad avere compromessi dubbii fra i vantaggi e gli svantaggi. Solo con le liste, nel Capitolo **??** si comincerà ad avere un chiaro senso dell'utilità che le strutture dati astratte portano con sé.

6.1.2 Implementazione: programma principale

A questo punto, si tratta di determinare il costo spaziale della struttura e i costi temporali delle singole operazioni. Lo faremo riprendendo l'esempio del capitolo precedente (la stampa di un insieme di numeri caricati da file) e realizzandolo con una tabella, anziché un vettore. È un po' eccessivo rispetto alla banalità del problema. Per renderlo un po' più ragionevole, aggiungeremo un paio di semplici operazioni a mo' di esempio: la cancellazione del primo elemento (seguita da una stampa per vedere che effettivamente l'elemento è sparito) e l'aggiunta di un numero intero in coda alla tabella (ancora seguita da una stampa). Si tratta solo di un programma dimostrativo.

Partiamo quindi dal codice `stampa_numeri0.c` usato nel capitolo scorso, il cui programma principale contiene sotto forma di commenti la decomposizione del problema in sottoproblemi, e dal file intestazione `tab_int.h`, che contiene la dichiarazione di una tabella di interi in base all'implementazione appena discussa. Rispetto alla discussione, è cambiato il nome della struttura, perché stiamo specificando il tipo dell'insieme base, e il nome della dimensione (da `k` a `dim`) perché è più difficile fare confusione con `n`[2].

La prima fase dell'esercizio consiste, al solito secondo la modalità *top-down*, nello scrivere le chiamate a funzione nel programma principale che realizzano le operazioni indicate nei commenti. Siccome queste chiamate suggeriscono la necessità di disporre di una tabella di interi, passeremo poi (se vogliamo, in modalità *bottom-up*) a realizzare l'intera libreria di gestione delle tabelle, riempiendo le definizioni delle sei funzioni nel file `tab_int0.c`: creazione, distruzione, cardinalità, inserimento, cancellazione e stampa. Seguiremo questo approccio (*top-down* al principio, fino a scoprire la necessità di una libreria, poi *bottom-up* per realizzare la libreria, poi ancora *top-down* per concludere la soluzione del problema) molto spesso nel resto del corso. Lo scopo è didattico, dato che molte lezioni riguardano strutture dati astratte e cominciano con un richiamo alla loro definizione, è naturale che l'esercizio richieda di realizzare una libreria, ed è ragionevole farlo quasi al principio, quando si è ancora freschi del ripasso della definizione. In una situazione più professionale (per esempio, già nel realizzare il progetto d'esame), ci si può aspettare che qualche libreria sia già disponibile e che basti includerla per proseguire, evitando questo salto di modalità. Oppure si può rimandare l'implementazione della libreria fino al momento in cui si arriva ai sottoproblemi che la usano davvero. O, ancora, si può costruire la libreria un pezzo alla volta, via via che emerge la necessità di nuove operazioni, anche se una struttura dati astratta andrebbe definita in modo il più

[2] In linea di principio, adottiamo la notazione delle dispense di teoria, tranne quando ci sono buoni motivi per discostarcene.

possibile organico. Questo esercizio terminerà appena completata la libreria, perché a quel punto il problema, che è molto semplice, è risolto.

Siccome abbiamo già risolto il problema usando un vettore dinamico, anziché una tabella, può essere utile considerare le differenze introdotte dall'uso di una struttura diversa. Ovviamente, anziché un vettore dinamico `vint V`, si dichiara nel programma principale una tabella `tab_int T`. Questo comporta di includere la corrispondente libreria con la direttiva `#include "tab_int.h"`. Ciò fatto, il precompilatore copia interamente il file nel codice, in modo che il compilatore legga la dichiarazione della struttura e i prototipi di tutte le funzioni prima che nel programma principale o nelle procedure secondarie vengano dichiarate e usate variabili di tipo `tab_int`.

A questo punto, per ogni commento si può aggiungere una chiamata a funzione che risolve il corrispondente sottoproblema. Il primo sottoproblema richiede di caricare da file in una tabella i numeri interi da stampare.

```
/* Carica i numeri da filedati in una tabella */
CaricaVettoreInteri(filedati,&T);
```

senza dimenticare il prototipo e il corpo, lasciato temporaneamente vuoto, della nuova procedura.

```
/* Carica dal file filedati nella tabella T i numeri da stampare */
void CaricaVettoreInteri (char *filedati, tabint *pT);
```

Siccome la tabella viene modificata, si passa per indirizzo. Abbiamo un solo parametro contro i due usati dal vettore dinamico (vettore e dimensione).

La stampa viene effettuata da una procedura interna alla libreria (`print_tab_int`). Questo consente di evitare l'aggiunta del prototipo e della definizione, per il semplice motivo che il prototipo è contenuto in `tab_int.h` (e quindi è già incluso al principio del file precompilato) e la definizione è contenuta in `tab_int.c` e l'operazione di *linking* la renderà disponibile al compilatore (ovviamente, se la linea di comando della compilazione contiene anche quel file).

```
/* Stampa i numeri */
stampatabint(&T);
```

dove la tabella viene ancora passata per indirizzo. Questo è strano, perché non si modifica la tabella, ma ci si limita a stamparla. Il motivo è tecnologico: la tabella si può passare per copia, ma in tal caso il meccanismo di chiamata della procedura andrebbe ad allocare sullo *stack* un *record* di attivazione con un'area di memoria sufficiente a contenere una tabella di interi (cioè un puntatore a intero e due numeri interi) e poi vi copierebbe i valori contenuti in `T`. Passando la tabella per indirizzo, l'argomento che viene passato (e quindi occupa memoria e richiede una copia) è un semplice puntatore. C'è un risparmio di memoria e di tempo (un oggetto anziché tre). Il risparmio è risibile, ma è un'usanza piuttosto diffusa passare le `struct` per indirizzo anziché per valore, perché nel caso di strutture contenenti molti oggetti il risparmio diventa significativo. Le chiamate alla procedura di stampa sono in effetti tre, e si possono aggiungere tutte subito.

Cancellare il primo elemento e aggiungere un elemento nuovo richiedono ancora funzioni di libreria, per cui si tratta semplicemente di aggiungere le relative chiamate, senza prototipi e definizioni.

```
/* Cancella il primo elemento */
canctabint(&T,1);
printf("card = %d\n",card_tab_int(&T));



/* Aggiunge il numero 47 */
instabint(&T,47);
printf("card = %d\n",card_tab_int(&T));
```

dove abbiamo aggiunto anche delle stampe di spiegazione. In entrambi i casi, è sensato che la tabella sia passata per indirizzo, dato che viene modificata. In particolare, viene modificata la cardinalità `n`; a rigore, il vettore `V` non viene modificato: vengono modificati i suoi elementi, che però verrebbero passati per indirizzo anche se la tabella fosse passata per copia, dato che `V` è un semplice puntatore, come tutti i vettori dinamici.

Ovviamente, se si compila solo `stampa_numeri0.c`, il compilatore segnala errori nella fase di collegamento, indicando che cinque funzioni a cui si è fatto riferimento sono indefinite. Si tratta delle cinque funzioni di libreria utilizzate (tutte tranne la cardinalità). Compilando entrambi i file, `stampa_numeri0.c` e `tab_int0.c`, si ha invece un codice sintatticamente corretto e con la struttura richiesta per risolvere il problema, salvo il fatto che le funzioni di libreria sono tutte vuote, e quindi il programma non fa nulla, salvo controllare che da linea di comando gli si passi anche un argomento (il nome del file da cui caricare i dati).

Avendo una sola procedura secondaria da realizzare, viene abbastanza naturale procedere prima di passare alla libreria, anche perché la procedura sarà praticamente identica a quella che usava il vettore dinamico: aprirà il file e ne conterà gli elementi.

```
fp = fopen(filedati,"r");
if (fp == NULL)
{
  fprintf(stderr,"File %s non apribile!\n",filedati);
  exit(EXIT_FAILURE);
}

n = 0;
while (fscanf(fp,"%d",&u) == 1)
  n++;
```

Quindi, anziché allocare il vettore dinamico, creerà la tabella con la dimensione indicata dal numero di elementi con la funzione `create_tab_int`, che ovviamente passa la tabella per indirizzo e la dimensione per valore, dato che il primo è un risultato e il secondo è un dato. È interessante notare che l'indirizzo della tabella è esattamente l'argomento `pT` che viene passato dalla funzione chiamante, per cui qui non avrebbe senso un operatore `&` di referenziazione (non ci interessa l'indirizzo della cella che contiene l'indirizzo della tabella).

```
/* Crea la tabella T di dimensione n */
creatabint(pT,n);
```

Tornata al principio del file, lo scorrerà leggendo i numeri, ma invece di leggerli direttamente nel vettore dovrà leggerli in una variabile intera `j` di appoggio, e

inserirli nella tabella con la procedura di libreria `ins_tab`, che ancora una volta richiede come argomento l'indirizzo `pT` della tabella. L'inserimento avviene in coda alla tabella per definizione. Questo è coerente con la richiesta di stampare i numeri nello stesso ordine con cui sono riportati nel file.

6.1.3 Implementazione: la libreria

Passiamo a realizzare la libreria, partendo dalla funzione di creazione `crea_tab_int`. Una tabella è costituita da tre campi, che vanno inizializzati: il puntatore `vint V` che fa da vettore dinamico, l'intero `int dim` che contiene la dimensione allocata e l'intero `n` che contiene la cardinalità corrente. La dimensione viene fornita dall'esterno: è il dato della funzione di creazione.

```
/* Crea la tabella T di dim elementi */
void creatabint (tabint *pT, int dim)
{
  pT->V = (int *) calloc(dim+1,sizeof(int));
  if (pT->V == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione della tabella!\n");
    exit(EXIT_FAILURE);
  }

  pT->dim = dim;
  pT->n = 0;
}
```

dove l'operatore `->` semplifica la scrittura `(*pT).dim`, che esprime la ricerca del campo `dim` della tabella puntata da `pT` (le parentesi tonde sono richieste per imporre la precedenza della dereferenziazione (operatore `*`) sull'accesso a un campo della struttura (operatore `.`). Le due scritture sono equivalenti, ma la prima è molto più chiara. La cardinalità corrente è nulla, perché la tabella appena creata non ha alcun elemento. Infine, il campo `V` viene allocato con un vettore dinamico di `dim+1` numeri interi, controllando quindi che l'allocazione sia andata a buon fine, perché altrimenti usare il vettore stesso non è consigliabile.

La distruzione `distrugge_tab_int` comporta fondamentalmente di liberare la memoria allocata e assegnata a `V`. La struttura rimanente, però (il puntatore e i due interi) sono dichiarati nel programma principale, e quindi vengono allocati automaticamente dal processore sullo *stack*, e non spariscono sino a che il programma principale stesso non termina. In questo esempio la funzione di distruzione sta alla fine di tutto il programma, e quindi questo passaggio avviene praticamente subito, ma in generale è buona norma, quando si distrugge una struttura dinamica, sia liberare la memoria dinamica stessa dallo *heap* sia lasciare le celle dello *stack* riferite ad essa in uno stato tale da denunciare chiaramente il fatto che non costituiscono più una struttura dinamica funzionante. Come si è già detto, spesso la deallocazione è l'occasione per far emergere problemi nella gestione delle aree di memoria dinamiche (tipicamente dovuti a sforamenti dei limiti). Scrivendo fin dal principio le operazioni di creazione e distruzione si può verificare se le righe di codice via via aggiunte provocano problemi, e quindi trovare più facilmente gli errori, che diventa invece oltre modo complicato localizzare quando il codice è scritto completamente (si finisce per dover cancellare pezzi di codice fino a regredire al momento nel quale l'errore è stato compiuto: ovviamente conviene evitare questo regresso e fare la verifica quanto prima sia possibile). Per lasciare i tre campi in uno stato che segnali chiaramente una tabella "distrutta", converrà assegnare a dimensione e cardinalità

il valore nullo, e al vettore dinamico il puntatore `NULL` (la funzione `free` non modifica il puntatore, ma si limita a rendere nuovamente disponibile l'area di memoria puntata).

```
/* Distrugge la tabella T */
void distruggetabint (tabint *pT)
{
  free(pT->V);
  pT->V = NULL;
  pT->dim = 0;
  pT->n = 0;
}
```

In questo modo, sia `V` sia `dim` possono essere usati per distinguere una tabella vuota (che ha `n` nullo) da una "distrutta".

La funzione cardinalità restituisce un intero (infatti, finché il corpo della funzione resta vuoto, in compilazione compare un avvertimento del fatto che manca il risultato promesso dalla dichiarazione). Siccome questo intero è il valore del campo `n`, l'implementazione è banale.

```
/* Restituisce la cardinalita' corrente della tabella T */
int cardtabint (tabint *pT)
{
  return pT->n;
}
```

È talmente banale da sollevare la questione di quanto sia sensato sprecare il tempo e lo spazio richiesti da una chiamata di funzione per ottenere quello per cui basterebbe una singola istruzione. Dal punto di vista dell'efficienza, è insensato: anziché accedere a un campo di un *record*, si valuta l'indirizzo della tabella, si alloca sullo *stack* lo spazio per tale indirizzo, lo si copia nello spazio allocato, si accede al campo (unica istruzione veramente necessaria) e si restituisce il suo valore come risultato, deallocando lo spazio dallo *stack*. Stiamo però applicando l'approccio basato sulle strutture dati astratte, per cui non si vuole sapere come sia implementata la tabella, ma solo avere a disposizione un'operazione che ne fornisce la cardinalità. Come vedremo poi, questa operazione potrebbe essere realizzata in un altro modo.

Passiamo all'inserimento di un nuovo elemento in fondo alla tabella (`ins_tab_int`). La funzione è definita solo se la tabella ha spazio per il nuovo elemento. Se invece la cardinalità attuale è non inferiore alla dimensione (non sarà mai superiore, ma gestire anche questo caso non costa nulla), c'è un problema. Come si è già detto, si può scegliere se:

- optare per l'efficienza, e non verificare che l'operazione sia lecita, lasciando all'utente della libreria il compito di controllare a priori che lo sia nei casi dubbi;
- valutare la condizione di errore (`pT->n >= pT->dim`) e restituire un codice di errore nel caso in cui sia soddisfatta; ancora una volta, sarà l'utente a dover controllare il codice e reagire di conseguenza: il problema viene trasferito da questa funzione di libreria alle funzioni che la usano (quindi potenzialmente scaricato da un programmatore a un altro);
- valutare la condizione di errore e, nel caso sia soddisfatta, agire in modo conservativo, per esempio non facendo nulla, ma senza avvisare l'utente, che a quel punto si trova di fronte a un errore non segnalato in alcun modo;

- valutare la condizione di errore e, nel caso sia soddisfatta, limitarsi a stampare a video (`stdout`) o sullo *stream* di errore (`stderr`) un messaggio di avvertimento (ammesso che il video sia disponibile e che abbia senso lasciar proseguire l'algoritmo in una situazione di errore);

- valutare la condizione di errore e, nel caso sia soddisfatta, terminare l'elaborazione, con un messaggio che spieghi la situazione.

A seconda dei casi, una o l'altra di queste strategie (o altre ancora) può essere la più adeguata. Opteremo per l'ultima, che è la più drastica, ma anche sicura e semplice al tempo stesso. Se la tabella non è piena, basta incrementarne la cardinalità e scrivere il nuovo elemento nella cella del vettore dinamico che ha l'indice uguale alla nuova cardinalità.

```
/* Inserisce in fondo alla tabella T l'elemento u */
void instabint (tabint *pT, int u)
{
  if (pT->n >= pT->dim)
  {
    fprintf(stderr,"Tabella piena (%d elementi)!\n",pT->dim);
    exit(EXIT_FAILURE);
  }

  pT->n++;
  pT->V[pT->n] = u;
}
```

A questo punto, la funzione di inserimento consente di riempire effettivamente il vettore. Se realizziamo anche la stampa, possiamo cominciare a valutare la correttezza semantica del codice con qualche prova pratica. È sempre una buona norma dotarsi della possibilità di vedere il risultato di quello che si sta facendo. Abbiamo già detto, del resto, che procedere "a cipolla" è meglio che procedere dal principio alla fine. Cambiamo quindi l'ordine di realizzazione rispetto a quello di presentazione delle funzioni e procediamo con la stampa.

La stampa è molto simile alla stampa del vettore di interi: un ciclo (da 1 a `pT->n`) consente di accedere ai singoli elementi del vettore `pT->V`, che possono essere stampati a video come al solito, aggiungendo un a capo finale per separare le stampe successive da quella corrente.

```
/* Stampa la tabella T */
void stampatabint (tabint *pT)
{
  int i;

  for (i = 1; i <= pT->n; i++)
    printf("%d ",pT->V[i]);
  printf("\n");
}
```

Compilando ed eseguendo il programma, si può notare che vengono stampati i numeri caricati da file, e poi si esce con un messaggio di errore legato all'inserimento. Il motivo è, a valle del caricamento e della prima stampa, si ha un'operazione di cancellazione ancora vuota, che non ha effetti, e poi un'operazione di inserimento che fallisce perché abbiamo assegnato alla tabella la stessa dimensione dei dati originali, e quindi l'inserimento non è possibile. Quindi, è un errore che dovrebbe scomparire realizzando la funzione di cancellazione.

Terminiamo con la procedura di cancellazione `canc_tab_int`. Questa richiede di togliere dalla tabella l'elemento di indice i, ovviamente solo se $1 \leq i \leq n$. Nel caso l'indice sia illecito, dobbiamo decidere la stessa cosa discussa per l'inserimento, e le soluzioni possibili sono più o meno le stesse. Risolveremo la questione allo stesso modo cioè segnalando l'errore e terminando il programma.

Se l'indice è lecito, rimane però il problema di come cancellare l'elemento. La questione fondamentale è che gli elementi in una tabella per definizione occupano tutti gli indici fra 1 e n, ed è spontaneo pensare che a questi indici corrispondano le posizioni fisicamente occupate nel vettore dinamico V[3]. A questo punto, bisogna chiedersi se gli elementi nella tabella abbiano un ordine significativo oppure no. Se non ce l'hanno, il modo più semplice di togliere un elemento da una tabella è sovrascriverlo con l'ultimo elemento e cancellare questo (visto che ora compare due volte) decrementando la cardinalità n in modo da cancellare la seconda occorrenza. L'ultimo elemento è l'unico cancellabile senza violare la proprietà fondamentale che gli elementi occupino le posizioni con indice compreso fra 1 e n: basta ridurre n.

```
/* Cancella dalla tabella T l'elemento di indice i */
void canctabint (tabint *pT, int i)
{
  if ( (i <= 0) || (i > pT->n) ) exit(EXIT_FAILURE);
  pT->V[i] = pT->V[pT->n];
  (pT->n)--;
}
```

Se invece l'ordine degli elementi fosse significativo, dovremmo scalare all'indietro, uno per uno, tutti gli elementi successivi a quello cancellato, cioè cancellare l'elemento di indice i sovrascrivendolo con quello di indice $i+1$, poi cancellare questo sovrascrivendolo con quello di indice $i+2$, e così via. Ovviamente, alla fine bisogna comunque ridurre n. Questo ha il vantaggio di conservare l'ordine, ma siccome nel problema che ci riguarda i numeri non hanno un ordine rilevante, non adotteremo questa implementazione.

Ora si può verificare che vengono stampati tre insiemi di numeri: il primo coincide con i dati su file, il secondo contiene un numero in meno e si può vedere che in esso il numero inizialmente riportato per ultimo ha sostituito il primo ed è scomparso dalla sua posizione iniziale. Infine, il terzo elenco ha la stessa cardinalità del primo, e contiene gli stessi elementi del secondo più l'elemento aggiuntivo in posizione finale.

FIGURA CON LA STAMPA

Volendo verificare la correttezza della funzione `card_tab_int`, che è l'unica rimasta inutilizzata, potremmo semplicemente chiamarla dopo ogni operazione e stampare a video il risultato.

```
/* Programma principale */
int main (int argc, char *argv[])
{
  char filedati[ROW_LENGTH];
  tabint T;


  /* Legge da linea di comando il file che contiene i dati */
  InterpretaLineaComando(argc,argv,filedati);

  /* Carica i numeri da filedati in una tabella */
  CaricaVettoreInteri(filedati,&T);

  /* Stampa i numeri */
```

[3] Non è obbligatorio: si potrebbe pensare a implementazioni in cui non sia così.

```
  stampatabint(&T);

  /* Cancella il primo elemento */
  canctabint(&T,1);
  printf("card = %d\n",card_tab_int(&T));

  /* Stampa i numeri */
  stampatabint(&T);

  /* Aggiunge il numero 47 */
  instabint(&T,47);
  printf("card = %d\n",card_tab_int(&T));

  /* Stampa i numeri */
  stampatabint(&T);

  distrugge_tab_int(&T);

  return EXIT_SUCCESS;
}
```

Implementando la funzione di cancellazione in modo che conservi l'ordine degli elementi, ovviamente il risultato sarebbe diverso, anche se equivalente per il problema che ci eravamo posti.

6.1.4 Costi

Avendo implementato in dettaglio la struttura dati astratta, possiamo discutere la complessità spaziale che essa comporta e la complessità temporale richiesta da ciascuna delle operazioni (vedi Figura 6.4). Il parametro di dimensione più naturale da usare è il numero n degli elementi contenuti nel file.

Dal punto di vista spaziale, la tabella occupa $n+1$ interi per il vettore dinamico e due interi per la dimensione e la cardinalità. Quindi c'è un leggero eccesso (in gergo, *overhead*) di memoria rispetto allo stretto necessario, ma la complessità spaziale è chiaramente in $\Theta(n)$. Anche senza provarlo formalmente, è piuttosto chiaro che non potrebbe essere inferiore, dunque è asintoticamente ottimale nel caso pessimo[4]

Rimandiamo per il momento le funzioni di creazione e distruzione, che richiedono funzioni della libreria standard, su cui bisogna fare alcune considerazioni tecniche. La funzione di cardinalità consuma chiaramente tempo costante, $\Theta(1)$. Anche l'inserimento consiste in un numero costante di operazioni elementari (confronti, incrementi, accessi indiretti, assegnamenti), e ha quindi complessità $\Theta(1)$. Per lo stesso motivo, la cancellazione eseguita sovrascrivendo l'elemento con l'ultimo occupa ancora tempo costante, $\Theta(1)$ (vedi Figura 6.5). Infine, la stampa ha costo lineare, $\Theta(n)$, ed è ragionevole pensare che non si possa fare meglio, volendo stampare tutti gli elementi, nel caso in cui gli elementi siano numeri interi del tutto generici.

Se invece si implementasse la cancellazione scalando gli elementi successivi a quello cancellato, avremmo ancora tempo costante nel caso ottimo (cancellazione dell'ultimo elemento), ma tempo lineare nel caso pessimo (cancellazione del primo elemento). Questa implementazione quindi è decisamente più inefficiente, ma offre il vantaggio di conservare l'ordine degli elementi. Bisogna soppesare vantaggi e svantaggi in base alle richieste del problema.

[4] "Asintoticamente" perché i 3 interi aggiuntivi contano sempre meno al crescere di n, "nel caso pessimo" perché se gli n numeri fossero tutti uguali si potrebbero rappresentare con il loro valore e la loro cardinalità, ma in generale questo non è sufficiente.

Il costo spaziale della tabella è ovviamente lineare in k ($\Theta(k)$)

I costi temporali delle operazioni sono tutti costanti

- per la cardinalità, si restituisce il valore di n

```
int card (tabella *pT)
{
  return pT->n;
}
```

- per l'inserimento, si incrementa n e si assegna l'elemento

```
void ins (tabella *pT, U u)
{
  if (pT->n >= pT->k) exit(EXIT_FAILURE);
  (pT->n)++;
  pT->V[pT->n] = u;
}
```

Passiamo la tabella per indirizzo solo per efficienza (non è necessario)

Figura 6.4: Tabelle: costi delle operazioni

I costi temporali delle operazioni sono tutti costanti

- per la cancellazione, si sovrascrive l'elemento indicato con l'ultimo e si decrementa n

```
void canc (tabella *pT, int i)
{
  if ( (i <= 0) || (i > pT->n) ) exit(EXIT_FAILURE);
  pT->V[i] = pT->V[pT->n];
  (pT->n)--;
}
```

L'implementazione assume che l'ordine degli elementi non sia fissato

Se l'ordine va conservato, la cancellazione passa da $\Theta(1)$ a $\Theta(k)$ perché si scalano un passo indietro gli elementi che seguono quello cancellato

Figura 6.5: Tabelle: costi delle operazioni

Passiamo alla funzione di creazione di una tabella di dimensione k. Si noti che il parametro rilevante qui non è n, ma k, e in generale $k > n$. La complessità di questa funzione dipende essenzialmente dalla funzione usata per allocare la memoria sullo *heap*. Questa è a rigore indeterminata, perché lo standard C non prescrive in che modo la memoria debba essere gestita. Si possono però fare dei ragionamenti per dedurne un'espressione ragionevole. Ad ogni modo, in rete è possibile trovare discussioni molto approfondite e molto tecniche, che concordano solo in parte con le convenzioni che adotteremo nel seguito. L'allocazione del vettore `pT->` può essere effettuata con la funzione `malloc` o con la funzione `calloc`. Nel primo caso, semplicemente si riserva al vettore un'area di memoria di dimensione opportuna. Si può pensare che un'implementazione efficiente riesca a farlo in tempo costante, $\Theta(1)$. Nel secondo caso, invece, oltre a riservare la memoria, essa viene anche interamente riempita di bit uguali a 0. È ragionevole pensare che questo abbia un costo proporzionale al numero di celle scritte, dunque un tempo lineare, $O(k)$. La funzione di distruzione ha una simile dipendenza dalla funzione `free`, la quale lascia semplicemente libera la memoria. Si può assumere che un'implementazione efficiente abbia costo costante, $\Theta(1)$.

6.1.5 Rendere pienamente dinamiche le tabelle

Abbiamo implementato le tabelle ipotizzando che abbiano una dimensione massima non superabile. Avevamo accennato al fatto che questo limite è in qualche modo aggirabile. Vediamo come.

Il punto fondamentale è che il linguaggio C offre la possibilità di riallocare i vettori dinamici, cioè di modificare la dimensione dell'area di memoria nell'*heap* allocata al puntatore che simula il vettore stesso. Ovviamente, se l'area viene ridotta, non si ha alcun problema: si tratta solamente di aggiornare la dimensione allocata, rendendo disponibile per altri scopi le celle non più richieste. Il problema nasce quando si vuole aumentare l'area, dato che essa deve essere contigua e limitarsi ad allargarla può portare a includere celle già allocate ad altri puntatori.

La funzione `realloc` procede in questo modo, con gli stessi argomenti di `calloc`:

```
pT->V = (int *) realloc(n2+1,sizeof(int));
```

Se il valore della nuova dimensione (`n2+1`) è inferiore (o uguale) a quello corrente, semplicemente aggiorna la dimensione allocata, liberando le eventuali celle sovrannumerarie. Per esempio, si può passare da `n` uguale a 7

```
STACK HEAP
0 1 2 3 4 5 6 7 8 9 10
pT->V ---------------------->[.  5 7 9 2 4 6 8].  .  'x'
```

a `n2` uguale a 3:

```
STACK HEAP
0 1 2 3 4 5 6 7 8 9 10
pT->V ---------------------->[.  5 7 9]2 4 6 8 .  .  'x'
```

senza cambiare il valore del puntatore `pT->V`, con un aggiornamento che presumibilmente (al solito, lo standard non specifica come questo vada fatto) richiede tempo costante, $\Theta(1)$.

Se invece il valore della nuova dimensione (`n2+1`) è superiore a quello corrente e le celle di memoria successive all'area inizialmente allocata sono libere, semplicemente

si aggiorna la dimensione allocata, occupando le nuove celle. Per esempio, si può passare da `n` uguale a 7

```
STACK HEAP
0 1 2 3 4 5 6 7 8 9 10
pT->V ---------------------->[.  5 7 9 2 4 6 8].  .  'x'
```

a `n2` uguale a 9:

```
STACK HEAP
0 1 2 3 4 5 6 7 8 9 10
pT->V ---------------------->[.  5 7 9 2 4 6 8 .  .]'x'
```

ancora una volta conservando il valore del puntatore `pT->V` e richiedendo un tempo presumibilmente, $\Theta(1)$.

Ma nel momento in cui una cella è occupata da altri dati (per esempio il carattere `'x'`), non è più possibile limitarsi ad allargare il vettore, pena l'includervi dati posseduti da altre variabili e creare potenziali errori. Bisogna invece cercare un'area di memoria di dimensione sufficiente in un'altra zona dello *heap*. Non è tutto: il nuovo spazio è vuoto (o meglio, pieno di informazioni casuali); bisogna quindi copiare le informazioni iniziali nel nuovo spazio. Per esempio, si può passare da `n` uguale a 7

```
STACK HEAP
0 1 2 3 4 5 6 7 8 9 10
pT->V ---------------------->[.  5 7 9 2 4 6 8].  .  'x'
```

a `n2` uguale a 10:

```
STACK HEAP
0 1 2 3 4 5 6 7 8 9 10
pT->V -------------------+ .  5 7 9 2 4 6 8 .  .  'x' .  .[.  5 7 9 2
4 6 8 .  .  .].
+------------------------------^
```

modificando il valore del puntatore `pT->V`, copiando `n+1` valori dalle celle originali alle nuove e aggiornando la dimensione allocata. Tutto questo richiede presumibilmente un tempo, $\Theta(n)$, e costituisce il caso pessimo al quale la definizione di complessità fa riferimento.

È un caso pessimo che può essere rarissimo o frequentissimo a seconda di come viene gestita la memoria dello specifico processore durante l'esecuzione di questo algoritmo. Non si può prevedere a priori e a parità di algoritmo e codice dipende dalla macchina usata.

Ci sono due aspetti importanti da considerare, a questo punto. Il primo è una complicazione tecnica che può portare a errori di programmazione. Se una procedura (come per esempio `ins_tab_int` rialloca un campo puntatore di una struttura passata per indirizzo, la riallocazione modifica il campo e la funzione chiamante riceve la struttura aggiornata correttamente. Se invece la procedura riceve separatamente i campi per copia (o se la struttura è passata per copia), la modifica del puntatore che avviene nella procedura riguarda solo una copia, e quindi non si propaga alla procedura chiamante, che continua a puntare l'area originale dello *heap*, che ora figura non allocata e disponibile. Allo stesso modo, anche gli altri campi (dimensione e cardinalità non vengono modificati). Questo porta ovviamente a problemi (se non altro, al termine, quando si cerca di deallocarla). È esattamente il motivo per cui nel Capitolo **??** si passano per indirizzo i vettori dinamici alle funzioni che li allocano.

Il secondo aspetto è quale dimensione indicare quando si rialloca un vettore. Sembra naturale indicare la dimensione minima necessaria, per un principio di risparmio dello spazio occupato. Nel caso dell'inserimento di un nuovo elemento in una tabella, però, questa dimensione supera la precedente solo di 1, e spesso l'operazione viene ripetuta molte volte in sequenza. L'errore tipico che si riscontra nei progetti è il ciclo di lettura di una sequenza di numeri con riallocazione della tabella ad ogni passo. Perché è un errore? Perché se ogni passo costa $\Theta(n)$, ancorché in un caso pessimo molto raro, l'intero ciclo costa $\Theta(n^2)$ (vedi Sezione **??**). Una prima soluzione al problema è, se possibile, *valutare a priori il numero di elementi massimo* (o una sua stima per eccesso ragionevole) e allocare subito l'intera memoria necessaria.

Un'altra soluzione è riallocare il vettore con una dimensione superiore a quella strettamente necessaria, in modo da rendere meno frequente la riallocazione stessa, con l'eventuale spostamento integrale della memoria. Esiste una regola empirica, che consiste nel *raddoppiare la dimensione corrente*

```
pT->V = (int *) realloc(2*pT->dim+1,sizeof(int));
```

dove abbiamo conservato il solito `+1` per indicizzare il vettore da 1 alla dimensione. Si tratta di un compromesso fra efficienza temporale e spaziale. Supponiamo infatti di eseguire un ciclo di n inserimenti. Nel caso pessimo, l'ultimo inserimento fa raddoppiare la dimensione del vettore inutilmente: si è sprecata metà della memoria allocata, che rimane comunque $\Theta(n)$. Dal punto di vista temporale, la riallocazione non avviene n volte, ma solo il numero di volte necessario a raggiungere $2n$ raddoppiando la dimensione stessa a partire da una dimensione iniziale che potrebbe essere 1 (nel caso pessimo). Dopo ogni riallocazione, c'è una serie di inserimenti via via più lunga in cui si sfruttano le celle allocate senza dover riallocare tutto. Il numero di riallocazioni è pari a $\log_2 n$ e ciascuna riguarda una dimensione che il più delle volte è molto inferiore a n e che va crescendo esponenzialmente (raddoppia ogni volta). La valutazione esatta del costo è discussa nelle dispense di teoria[5], e porta a un costo temporale in $\Theta(n)$, dato che la crescita esponenziale del tempo per ogni allocazione rende l'ultima dominante su tutte le precedenti.

6.1.6 Implementazione di tabelle con terminatore

Consideriamo ora una seconda possibile implementazione delle tabelle (vedi Figura 6.6), in modo da sviluppare un po' l'idea che la stessa struttura dati astratta possa ammettere implementazioni diverse, con vantaggi e svantaggi sotto diversi punti di vista. L'implementazione che consideriamo si basa sull'idea di non indicare esplicitamente la cardinalità della tabella, ma di farlo implicitamente attraverso un *terminatore*. In concreto, si tratta di rappresentare un insieme di n elementi come un vettore contenente $n+1$ elementi: i primi n sono quelli della tabella, e l'ultimo è un elemento particolare che fa parte dell'insieme base, ma non è un elemento utilizzabile dall'algoritmo, cioè dotato di un contenuto informativo proprio, bensì viene utilizzato solo per rappresentare il fatto che la tabella termina con l'elemento precedente e la posizione corrente è la prima posizione esterna alla tabella.

Ovviamente, questo comporta che l'insieme base abbia un elemento generalmente non usato nel calcolo (altrimenti dovrebbe giocare un doppio ruolo, il che non è possibile). Nel caso dei numeri interi o reali, può essere facile o molto complicato trovarne uno (dipende dal problema specifico: se i numeri sono positivi, si può usare lo zero o un numero negativo; altrimenti si possono usare costanti simboliche come `MAX_INT`, sperando con una certa dose di rischio che non si presenti mai un dato

[5] AGGIUNGERE IL RIFERIMENTO O COPIARE LA DIMOSTRAZIONE

Un'implementazione alternativa (poco usata) impiega

- un vettore di $k+1$ elementi di tipo U
- un valore intero k, che rappresenta la dimensione allocata, costante
- un terminatore, cioè un elemento esterno ad U che non rappresenta un'informazione effettiva, ma indica il termine della tabella

Si risparmia l'intero n, ma si spende lo spazio occupato dal terminatore

Gli svantaggi sono:

- non si può usare il terminatore come informazione effettiva
- cardinalità, inserimento e cancellazione richiedono tempo lineare, perché richiedono di individuare il terminatore scorrendo la tabella

E allora perché ne parliamo?

Figura 6.6: Tabelle: implementazione come vettori con terminatore

di quel valore). Bisogna inventarsi un oggetto con la stessa rappresentazione di un dato, ma che non sia un dato. Nel caso dei puntatori, abbiamo visto che `NULL` svolge spesso questo ruolo.

A questo punto, non occorre più un campo per la cardinalità. Quindi ci si può limitare ai campi dimensione e vettore dinamico. Il costo spaziale è lo stesso. Dal punto di vista del costo temporale, tutte le operazioni risultano meno efficienti. Infatti, tutte richiedono un tempo lineare, $\Theta(n)$. La cardinalità richiede di scorrere il vettore dall'indice 1 fino a trovare il terminatore, incrementando via via un contatore che alla fine fornirà il valore cercato. L'inserimento in posizione terminale richiede ancora una volta uno scorrimento fino a trovare il terminatore, che viene poi sovrascritto con l'elemento aggiuntivo, e un nuovo terminatore viene aggiunto nella posizione immediatamente successiva. Il tutto richiede $\Theta(n)$. La cancellazione richiede ancora una volta di trovare l'elemento finale (subito prima del terminatore), copiarlo al posto dell'elemento da cancellare, e poi annullare l'elemento finale scrivendovi sopra un terminatore: ancora tempo $\Theta(n)$. Non cambiano sensibilmente le complessità della stampa, creazione e distruzione.

La domanda spontanea è che vantaggi porti questa implementazione, che per ora ha solo mostrato svantaggi in termini di efficienza. Il vantaggio è l'estrema semplicità della rappresentazione, che consente (se si rimuove anche la dimensione allocata `k`) di rappresentare una tabella con un semplice puntatore. Il punto è che conosciamo già questo genere di tabelle: sono le stringhe C (vedi Figura 6.7).

Le stringhe C sono tabelle di caratteri implementate come vettori (statici o dinamici) dotati di terminatore. Ogni stringa, infatti, termina con un carattere speciale (`'\0'`) che non è un vero carattere (non può essere stampato, per esempio), ma serve solo a indicare il termine di una stringa. Questo carattere in genere ha una rappresentazione formata di bit tutti nulli. Il vantaggio sta nel fatto che alle funzioni di manipolazione di stringhe non occorre passare una struttura composta da vari campi, ma semplicemente un puntatore. Siccome le stringhe per lo più si stampano e la stampa ha costo lineare, le inefficienze (pur presenti) hanno un

impatto relativamente ridotto.

In C, le stringhe sono rappresentate come

- vettori di caratteri (`char s[N+1];`)
- terminati dal carattere *null* (`'\0'`), detto terminatore, il quale ha codifica binaria interamente nulla

Se la stringa `s` vale "`pro`", significa che contiene 4 caratteri:

- `s[0]` vale `'p'`
- `s[1]` vale `'r'`
- `s[2]` vale `'o'`
- `s[3]` vale `'\0'`

anche se lo spazio allocato è più lungo:

`'p'`	`'r'`	`'o'`	`'\0'`	`'v'`	`'a'`	`'\0'`	vale `''pro''`
0	1	2	3	4	5	6	

Non occorre specificare la dimensione di una stringa: un vettore di `N+1` caratteri può rappresentare stringhe di qualsiasi dimensione da 0 a `N`
Però non c'è controllo che una stringa contenga il carattere `'\0'`

Figura 6.7: Stringhe: implementazione in C

6.2 Il problema dell'ordinamento

Questa sezione considera uno dei problemi fondamentali nei corsi di base di algoritmi, cioè il problema dell'ordinamento. Partiremo con alcuni concetti di base, che torneranno utili anche per argomenti successivi. Riprenderemo l'algoritmo *InsertionSort* descritto anche nelle dispense di teoria, lo progetteremo in modalità top-down e lo realizzeremo in linguaggio C. Poi considereremo l'algoritmo di *SelectionSort*

6.2.1 Relazioni d'ordine

Supponiamo di avere un insieme base U (vedi Figura 6.8) e consideriamo alcune definizioni fondamentali sulle relazioni d'ordine.

Definizione 7 *Un* preordine *su un insieme U è una relazione binaria (nel seguito indicata con il simbolo $\preceq$ e l'espressione "precede") che gode di due proprietà: la* riflessività *e la* transitività.

Con *riflessività* si intende che qualunque elemento dell'insieme U è in relazione con se stesso: si "precede" (la relazione non è stretta: sono ammessi elementi che occupano lo stesso posto nell'ordine).

Con *transitività* notoriamente si intende che se un elemento ne precede un altro e questo precede un terzo, allora il primo precede il terzo, qualunque sia la terna di elementi considerati (coincidenti o meno).

Questa è la base di tutte le relazioni che ci interessano. Introducendo altre proprietà, specifichiamo meglio queste relazioni e otteniamo relazioni più forti.

In particolare con *antisimmetria* intendiamo il fatto che due elementi vicendevolmente in relazione (ciascuno dei due precede l'altro) devono essere lo stesso elemento (cosa ammessa dalla riflessività) e non possono essere diversi (condizione aggiuntiva). Un esempio classico è quello dell'inclusione fra insiemi: se due insiemi sono contenuti l'uno nell'altro, sono lo stesso insieme. Oppure i numeri: se due numeri sono ciascuno non superiore all'altro, allora coincidono. Molte relazioni, però, non rispettano queste condizioni: la relazione "figurare non più in basso nella classifica di un campionato sportivo a squadre" è riflessiva e transitiva, ma non antisimmetrica, dato che due squadre possono precedersi a vicenda nella classifica anche senza essere la stessa squadra (da cui la necessità di spareggi in circostanze particolari). Per la relazione "essere non più vecchio" è riflessiva e transitiva, ma non antisimmetrica, dato che due persone possono essere ciascuna non più vecchia dell'altra, cioè avere la stessa età, senza essere la stessa persona. L'antisimmetria è quindi indipendente dalle prime due.

Definizione 8 *Una* relazione d'ordine parziale *su un insieme U è una relazione binaria riflessiva, transitiva e antisimmetrica.*

Con *completezza* si intende invece il fatto che qualsiasi coppia di elementi possa essere confrontata concludendo che almeno uno dei due precede l'altro (eventualmente, precedendosi a vicenda). La definizione formale consiste nel richiedere che quando un elemento non precede l'altro, l'altro deve precedere il primo, il che non impedisce che i due si precedano. La completezza vale ovviamente per i numeri, dato che se uno non è minore o uguale all'altro, cioè è maggiore, l'altro è certamente minore o uguale al primo (in effetti, è strettamente minore). Vale anche per le squadre in una classifica di campionato e per le persone rispetto alla loro età. Ma non vale per l'inclusione fra insiemi, dato che due insiemi possono tranquillamente non includersi a vicenda. E non vale per la relazione "essere progenitore". Questa relazione è riflessiva, se si intende che una persona sia progenitrice di sé stessa, è transitiva e antisimmetrica (nel caso di progeniture strettamente biologiche e naturali), ma non completa (date due persone a caso, in genere nessuna delle due è progenitrice dell'altra).

Definizione 9 *Una* relazione d'ordine debole *su un insieme U è una relazione binaria riflessiva, transitiva e completa.*

Questa relazione garantisce la possibilità di mettere gli oggetti in un ordine lineare che ammetta degli ex aequo. Per questo, è la relazione che interessa negli algoritmi di ordinamento.

Per concludere, le quattro proprietà imposte simultaneamente garantiscono la relazione tipica dei numeri (interi o reali).

Definizione 10 *Una* relazione d'ordine debole *su un insieme U è una relazione binaria riflessiva, transitiva e completa.*

Dato un insieme U sul quale sia definito un ordine debole, il problema dell'ordinamento (vedi Figura 6.9) ha come dato un sottoinsieme di elementi di U, eventualmente con ripetizioni, e come risultato una sequenza, o permutazione, di tali elementi tale che ogni elemento di indice inferiore preceda ogni elemento di indice superiore[6]. Quindi, si parte da un sottoinsieme e si arriva a una sequenza, un sottoinsieme ordinato. Per esempio, se prendiamo il sottoinsieme di numeri $S =$

[6] A rigore, non vale l'inverso: un elemento che ne precede un altro può avere indice superiore, se in realtà i due elementi si precedono a vicenda.

Un preordine su un insieme U è una relazione binaria $\preceq$ su U che gode delle proprietà

1. riflessiva: $u \preceq u$ per ogni $u \in U$
2. transitiva: se $u_1 \preceq u_2$ e $u_2 \preceq u_3$, allora $u_1 \preceq u_3$ per ogni $u_1, u_2, u_3 \in U$

Una relazione d'ordine parziale è un preordine che gode della proprietà

- antisimmetrica: se $u_1 \preceq u_2$ e $u_2 \preceq u_1$, allora $u_1 = u_2$ per ogni $u_1, u_2 \in U$

Una relazione d'ordine debole è un preordine che gode della proprietà

- di completezza: se $u_1 \not\preceq u_2$, allora $u_2 \preceq u_1$ per ogni $u_1, u_2 \in U$

Una relazione d'ordine totale è un preordine che gode di ambo le proprietà

Figura 6.8: Relazioni d'ordine

$\{5, 2, 8, 4, 7, 1, 3, 6\}$, dobbiamo ottenere la permutazione $\Pi = (1, 2, 3, 4, 5, 6, 7, 8)$. In caso di ex aequo, i numeri uguali saranno adiacenti, ma il loro ordine nel risultato è libero.

Il problema dell'ordinamento riguarda insiemi e permutazioni. Da un punto di vista pratico, questi possono essere rappresentati con molte strutture dati diverse. Nei testi di algoritmi, il problema viene generalmente discusso parlando di vettori, cioè ipotizzando che i dati siano un vettore e i risultati un altro. Questo perché i vettori sono la struttura più ragionevole per rappresentare insiemi e per rappresentare permutazioni. Come vedremo, gli algoritmi tipicamente usano la stessa struttura vettore fisica per rappresentare dati e risultati, cioè sono algoritmi *distruttivi*, che cancellano i dati per costruire i risultati: l'area di memoria occupata dai risultati è la stessa inizialmente occupata dai dati e i dati vanno persi, a meno che non se ne faccia una copia in partenza.

In linea di principio, si possono realizzare algoritmi di ordinamento anche su altre strutture dati. Di solito, non si fa perché ne derivano implementazioni inefficienti. Conviene tenere presente questo punto durante i progetti d'esame o le applicazioni pratiche: se si deve ordinare un insieme non rappresentato da un vettore, si tratta di valutare se convertire l'insieme in un vettore (e soprattutto in che modo) oppure cercare di realizzare l'algoritmo direttamente sulla struttura iniziale. Ne parleremo, ma per ora ci concentriamo sul caso dei vettori.

Inoltre, per semplicità, come in tutti i testi di algoritmi, considereremo come insieme base U quello dei numeri interi, per cui "precede" ($\preceq$) significa "è minore o uguale" ($\leq$). Tutti i concetti che tratteremo sono estendibili a qualunque insiemi base debolmente ordinato, a patto di sostituire l'operazione di confronto fra numeri interi con l'analoga operazione che valuta la relazione d'ordine. C'è una sola modifica da tenere presente: che l'operazione di confronto fra numeri interi ha costo unitario ($\Theta(1)$), mentre quella fra oggetti più sofisticati potrebbe avere un costo diverso, che influirà sulla complessità dell'intero algoritmo. più piccolo o uguale

Sia U un insieme dotato di un ordine debole $\preceq$ *(si ammettono ex-aequo)*

Il problema dell'ordinamento ha come

- istanza: qualsiasi vettore V su U
- soluzione: un vettore V' permutazione di V tale che

$$V[i] \preceq V[j] \text{ per ogni } i \leq j$$

Esempio:

$V =$

5	2	8	4	7	1	3	6

$V' =$

1	2	3	4	5	6	7	8

Figura 6.9: Il problema dell'ordinamento

6.3 Ordinamento per inserimento (*InsertionSort*)

L'algoritmo di *InsertionSort*, ovvero *ordinamento per inserimento*, è basato sull'uso di tabelle (vedi Figura 6.10). Da qui la combinazione dei due argomenti in questo capitolo. Introduciamo una convenzione: dato un vettore V, indichiamo con $V[s, d]$ il suo sottovettore che contiene gli elementi con indici consecutivi compresi fra s e d, estremi compresi (ovviamente s sta per "sinistra" e d sta per "destra").

L'algoritmo di *InsertionSort* costruisce la soluzione un passo alla volta gestendola come una tabella, in particolare una tabella ordinata. In essa, cioè l'elemento di indice 1 è non superiore a quello di indice 2, e così via, fino all'indice n. Inizialmente, la tabella ordinata conterrà solo il primo elemento del vettore di partenza. Quindi, passo per passo l'algoritmo inserisce gli altri elementi nella tabella, seguendo il loro ordine iniziale, dal secondo all'n-esimo. L'inserimento, però, non può applicare la funzione `ins_tab_int` che abbiamo visto in precedenza: tale inserimento avverrebbe in fondo, e la tabella alla fine sarebbe identica al vettore iniziale. Andremo invece a inserire ogni nuovo elemento in modo che la tabella rimanga ordinata.

Abbiamo visto come cancellare un elemento da una tabella in modo da mantenerla ordinata: si fanno scalare all'indietro tutti gli elementi successivi e si riduce la cardinalità della tabella. Questo meccanismo è più lento di quello standard, ma consente di conservare l'ordine degli elementi. Adotteremo quindi un meccanismo simile per lo scopo opposto, cioè per allargare la tabella insererendovi un elemento in modo da mantenerla ordinata. Procederemo quindi aumentando la cardinalità e scalando in avanti tutti gli elementi maggiori dell'elemento da aggiungere. Più in dettaglio, al passo j (con j crescente da 2 a n) prendiamo l'elemento $V[j]$ e scaliamo in avanti nella tabella che contiene gli elementi del sottovettore $V[1, j-1]$ tutti quelli che sono maggiori di $V[j]$. Ciò fatto, la tabella ha un buco (quindi, non è una tabella), nel quale inseriamo l'elemento $V[j]$.

A questo punto, applichiamo un trucco, a rigore non strettamente necessario: la tabella T non è separata dal vettore V, ma ne condivide una parte, precisamente il sottovettore $V[1, j-1]$. Quindi, al principio, j vale 2 e T è il sottovettore $V[1, 1]$. Questa tabella è ordinata, dato che contiene un solo elemento. Di volta in volta, si prendono gli elementi successivi; per inserirli, bisogna scalare in avanti gli elementi compresi fra 1 e $j-1$ e che sono maggiori di quello che stiamo cercando di inserire.

Questi elementi sono nella seconda parte della tabella, dato che sono i più grandi. Per scalarli, bisogna partire dall'ultimo e retrocedere, altrimenti ciascuno andrebbe a cancellare il successivo[7]. Alla fine, otteniamo un elemento vuoto, e in questo spazio bianco andiamo a scrivere l'elemento da inserire nella tabella. Questo elemento occupava la posizione $V[j]$, che però viene immediatamente sovrascritta scalando in avanti $V[j-1]$. Di conseguenze, prima di scalare gli elementi, bisogna salvare da parte l'elemento da inserire.

Convenzione: dato un vettore V, indichiamo con
$V[s,d]$ il sottovettore degli elementi di V con indici compresi fra s e d

InsertionSort gestisce la soluzione come una tabella ordinata T

- inizialmente T contiene solo il primo elemento di V
- ogni elemento $V[j]$ (con $j = 2, \ldots, n$) viene inserito in T in ordine:
  - scalando gli elementi $> V[j]$ nella posizione di indice successivo
  - inserendo $V[j]$ nella posizione liberata

La tabella T viene rappresentata con il sottovettore $V[1, j-1]$

- gli elementi vanno scalati partendo da $V[j-1]$ per j decrescenti
  (altrimenti ognuno cancellerebbe il successivo)
- bisogna salvare $V[j]$ a parte per prima cosa
  (altrimenti $V[j-1]$ lo cancellerebbe)

Figura 6.10: Ordinamento per inserimento

La Figura 6.11 illustra uno pseudocodice, apparentemente diverso da quelli comunemente adottati nei testi di algoritmi (comprese le dispense di teoria). La differenza sta solo nel fatto che per sottolineare la struttura concettuale del codice e la relazione con le tabelle, anziché riportare esplicitamente tutte le operazioni si sono usate delle procedure secondarie, creando una struttura a due livelli. Questo rende il tutto più chiaro e più facile da ricordare, anche se ovviamente meno efficiente, a causa della gestione del meccanismo di chiamata di funzione. Come si diceva, per `j` crescente da 2 a `n` si salva l'elemento `j`-esimo del vettore in una variabile `x` e poi chiamiamo la funzione `InserisciOrdinato`, che si occupa di inserire `x` nella tabella, cioè nel sottovettore $V[1, j-1]$, che rappresentiamo attraverso il vettore `V` e l'indice `j-1`, dato che l'indice 1 è costante.

A sua volta, `InserisciOrdinato` riceve un elemento `x`, un vettore `V` e la sua cardinalità `n` (che è il parametro formale che rappresenta la dimensione del vettore di interesse, variabile ad ogni chiamata). Per scalare in avanti gli elementi corretti, parte dall'ultimo e retrocede, fermandosi o quando arriva a 0 (nel qual caso, si sono considerati tutti gli elementi) o quando trova un elemento non superiore a `x` (la condizione di permanenza nel ciclo è che sia `V[i] > x`). Ad ogni passo, si scala l'elemento corrente in avanti, cioè si scrivere `V[i]` in posizione `i+1`. Questa operazione sarebbe pericolosa, dato che sfora il vettore `V` alla prima iterazione, ma ci siamo assicurati nella funzione chiamante che in `V[n+1]` ci sia un elemento lecito e lo abbiamo salvato in `x` (ovviamente, chiamare la funzione `InserisceOrdinato` senza

[7] Ovviamente, si potrebbero copiare a parte, ma questo richiederebbe memoria aggiuntiva, mentre noi stiamo addirittura distruggendo i dati per evitare consumo di memoria aggiuntiva.

aver fatto questi controlli creerebbe problemi). Fatta la scalatura, in posizione `i+1` è rimasto un buco, dato che la posizione `i` o è nulla oppure contiene un elemento `V[i] <= x`. In tale posizione scriviamo `x`.

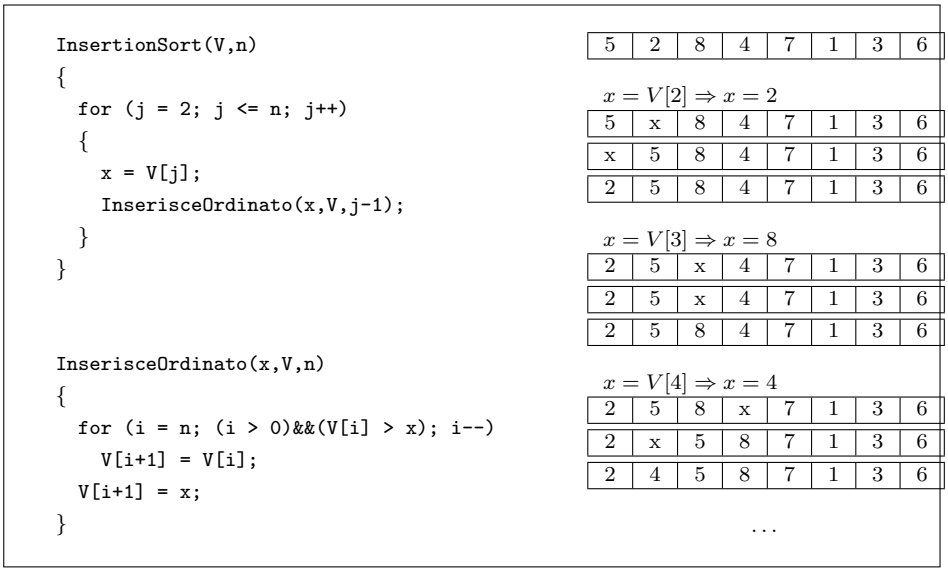


Figura 6.11: InsertionSort: pseudocodice ed esempio

Le Figure 6.11, 6.12 e 6.13 riportano anche un esempio numerico di esecuzione passo per passo della procedura stessa.

Implementazione

I materiali di supporto forniscono un esempio di istanza del problema dell'ordinamento (`sort.txt`), che coincide con l'esempio usato per il cariamento di vettori nel capitolo precedente e un codice iniziale (`sort0.c`). Quest'ultimo contiene un programma principale, le procedure di interpretazione della linea di comando (che controlla il numero di parametri e carica il nome del file da aprire) e di caricamento da file del vettore di interi da ordinare, nonché la procedura di stampa del vettore e l'istruzione per la sua deallocazione. Contiene anche un commento che invita ad aggiungere la procedura di ordinamento.

Procediamo a implementare l'algoritmo di `InsertionSort` come al solito, aggiungendo la chiamata

```
/* Ordina il vettore */
InsertionSort(V,n);
```

dove il vettore `V` e la dimensione `n` sono i dati, mentre gli elementi di `V` sono il risultato. Siccome il risultato non è il vettore, ma i suoi elementi, che sono già passati per indirizzo, è corretto che il vettore sia passato per copia. Aggiungeremo anche la dichiarazione e la definizione inizialmente vuota.[8]

```
void InsertionSort (vint V, int n);
```

[8]NEL CODICE MANCANO I COMMENTI

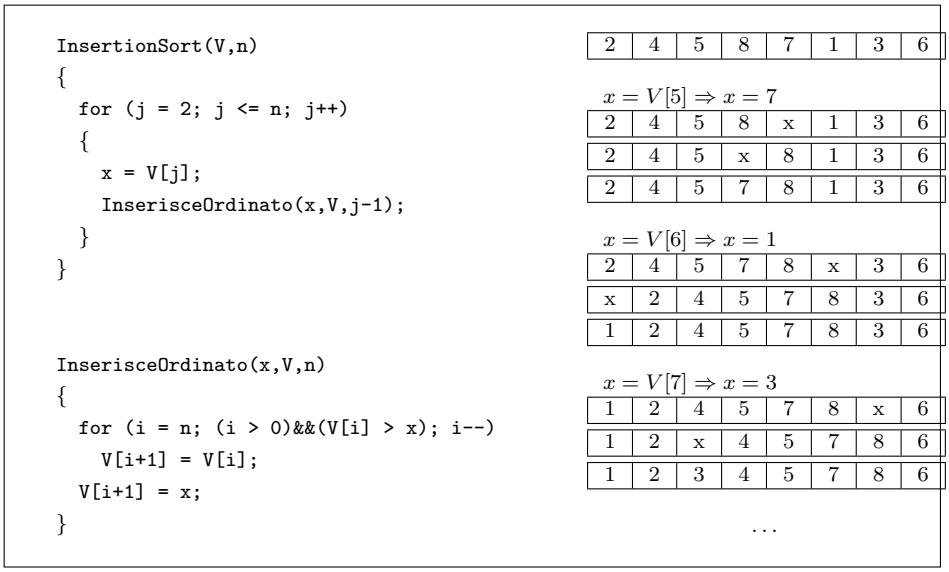


Figura 6.12: InsertionSort: pseudocodice ed esempio

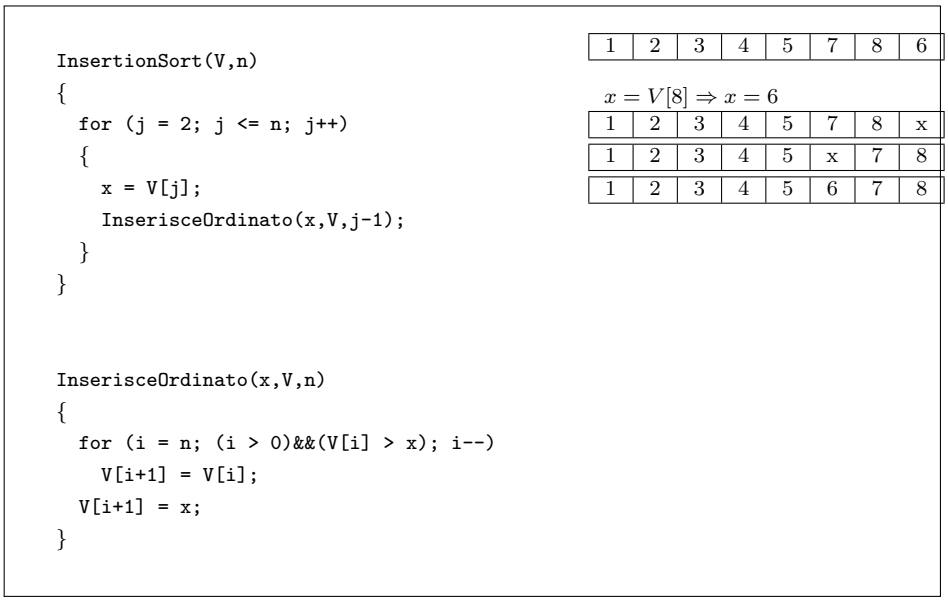


Figura 6.13: InsertionSort: pseudocodice ed esempio

con il solito tipo `vint` definito da noi per sottolineare che si tratta non di un generico puntatore, ma di un vettore dinamico. Compilazione ed esecuzione confermano la correttezza sintattica, anche se il vettore stampato rimane quello originale, dato che la procedura di ordinamento non fa nulla.

A questo punto, possiamo riempire la procedura di ordinamento, dichiarando una variabile locale intera `j` che scorra come cursore gli elementi del vettore dal secondo all'ultimo, salvando tali elementi in una variabile locale `x`, poi inserendoli nel vettore con una procedura secondaria `InserisceOrdinato` (che opera sul vettore `V` fino alla posizione `j-1`)[9].

```
/* Ordina il vettore V di estremi 1 e n */
void InsertionSort (vint V, int n)
{
  int j, x;

  for (j = 2; j <= n; j++)
  {
    x = V[j];
    /* Inserisce x nel sottovettore V[1,j-1] in maniera ordinata */
    InserisceOrdinato(x,V,j-1);
  }
}
```

A questo segue, sempre in modalità *top-down*, la dichiarazione e definizione della funzione `InserisceOrdinato`

```
void InserisceOrdinato (int x, vint V, int n);
```

Ancora una volta, il codice compila correttamente, ma non fa nulla. Bisogna riempire la funzione `InserisceOrdinato` con lo scorrimento retrogrado del vettore (dunque una variabile intera cursore `i`), la scalatura in avanti dei suoi elementi maggiori di `x` e al termine la copia di `x` nella posizione `i+1` liberata dalla scalatura.

```
void InserisceOrdinato (int x, vint V, int n)
{
  int i;

  for (i = n; (i > 0) && (V[i] > x); i--)
    V[i+1] = V[i];
  V[i+1] = x;
}
```

Compilato ed eseguito il codice, si può notare che il risultato è ordinato, e che il vettore contiene parecchi doppioni, cosa perfettamente accettabile, dato che si tratta di un ordine debole.

6.3.1 Correttezza dell'algoritmo di *InsertionSort*

Perché l'algoritmo di *InsertionSort* funziona? La garanzia di correttezza di questo algoritmo, come di molti altri, è un principio base della matematica, cioè il *principio*

[9]Cercherò di essere chiaro: questo non è un esempio di relazione, ma solo una penosissima descrizione passo per passo del codice a scopi didattici, carica di dettagli informatici inutili, che in una relazione sono dannosi per la comprensione e il buon umore del lettore.

di induzione (vedi Figura 6.14). L'idea è che se si riesce a dimostrare che una certa proposizione è corretta per un dato numero naturale (di solito 0 o 1) e che la sua correttezza per un generico numero naturale garantisce la correttezza anche per il numero seguente, allora la proposizione è corretta per tutti gli infiniti numeri naturali maggiori o uguali a quello di partenza.

Nel caso dell'algoritmo di *InsertionSort*, la proposizione che intendiamo dimostrare è che la tabella $T = V[1, j-1]$ contenga elementi in corrispondenza biunivoca con alcuni un sottoinsieme di V e che sia ordinata. Partiremo col numero $j = 2$. La proposizione è banalmente vera, perché $T = V[1, 1]$ contiene un elemento di V ed è ordinata, in quanto qualsiasi tabella con un solo elemento è necessariamente possibile. Ora, supponendo che la proposizione sia vera per un valore j, proviamo ad incrementare j. Questo significa che la tabella T cresce di un elemento. Così facendo, include un nuovo elemento del vettore V (a patto che j non superi n, cosa garantita dalla condizione di permanenza). Quindi la prima parte della proposizione resta valida. D'altra parte, la funzione `InserisciOrdinato`, ammesso che sia corretta, non lascia questo elemento dov'è, ma lo sposta in una posizione tale che la tabella T rimanga ordinata. Quindi anche la seconda parte della proposizione rimane valida. Ne deriva che la proposizione rimane valida per qualsiasi valore di j sino al termine del ciclo. Al termine del ciclo, la condizione di permanenza è certamente falsa, dunque $j = n + 1$[10]. Ne deriva che la proposizione ora afferma che gli elementi di T sono in corrispondenza biunivoca con quelli di $V[1, j-1] = V[1, n]$, cioè V, e che T è ordinata. Quindi, T è una permutazione ordinata del vettore V, cioè una soluzione del problema di ordinamento. Ne deriva che `InsertionSort` è un algoritmo corretto.

Per completezza, aggiungiamo che l'algoritmo è corretto anche su vettori vuoti, dato che è facile verificare che non fa nulla, e un vettore vuoto è per definizione automaticamente già ordinato.

6.3.2 Complessità dell'algoritmo di *InsertionSort*

Come abbiamo visto nel Capitolo **??**, la complessità di un algoritmo iterativo si può ridurre a una sommatoria della complessità delle operazioni compiute all'interno del ciclo, più quelle precedenti e successive, che in genere sono trascurabili e in questo caso praticamente non ci sono.

La Figura 6.15 introduce lo studio per il nostro caso specifico. Il ciclo è a conteggio, per cui la sommatoria è esplicita, con indice j crescente da 2 a n. Le operazioni interne al ciclo sono un assegnamento, in tempo costante $\Theta(1)$ (che sia un'operazione, due o altro non importa), e una chiamata a funzione, che va discussa a parte, attribuendole una complessità $f(j)$, dipendente dall'indice della sommatoria. Si noti che anche nell'analisi di complessità stiamo procedendo in modalità *top-down*, per cui l'analisi può tranquillamente essere compiuta subito dopo aver descritto la procedura ad alto livello, e senza dover aspettare di descrivere (o persino progettare) la procedura a basso livello. Al limite, la funzione $f(j)$ potrebbe dipendere da altri parametri, ma vedremo che non è così e non è grave introdurre questa ipotesi fin da subito nella descrizione.

Entriamo nel dettaglio della funzione `InserisceOrdinato`. Le sue operazioni dipendono dalla dimensione del sottovettore, definita internamente `n`, ma coincidente con `j-1`. Questo valore non è uguale ad ogni iterazione del ciclo. Per semplicità, usiamo anche noi n e poi sostituiamolo con $j-1$. Dopo il ciclo c'è un assegnamento, che richiede tempo costante, $\Theta(n)$. Il ciclo ha un indice i decrescente. Questo non crea alcun problema, dato che le complessità si sommano e le somme finite godono

[10] A rigore, $j > n$, ma incrementando j di un passo alla volta è chiaro che $j = n + 1$.

L'algoritmo funziona per induzione matematica

- Al principio, $j = 2$ e la tabella T
  1. è in corrispondenza biunivoca
     con un sottoinsieme di $j - 1$ elementi di V
  2. è ordinata
- Ad ogni iterazione, j cresce di 1 e la tabella T
  1. include un nuovo elemento di V
  2. lo inserisce in posizione ordinata

Dunque conserva le due proprietà

Al termine, $j = n + 1$ e la tabella T

- è in corrispondenza biunivoca
  con un sottoinsieme di $j - 1 = n$ elementi di V (cioè tutti)
- è ordinata

In altre parole, per qualsiasi n e per qualsiasi vettore V,
al termine dell'algoritmo la tabella T è una permutazione ordinata di V

Figura 6.14: InsertionSort: correttezza

della proprietà commutativa. Però il ciclo non è a conteggio, e quindi richiede di fare una stima del numero di iterazioni. Indichiamo con p_x l'ultimo valore dell'indice i in cui la condizione di permanenza rimane vera. Se prevale la prima parte della condizione (`i > 0`), vale $p_x = 1$; se invece prevale la seconda parte (`V[i] > x`), p_x è la posizione nella quale si trova l'ultimo elemento scalato in avanti, e quindi la posizione che ospiterà il nuovo elemento `x` della tabella ordinata. Infatti, si chiama p_x proprio perché dipende da `x`. Quindi, otteniamo una sommatoria sull'indice i dall'indice incognito p_x a n. Le operazioni interne al ciclo richiedono tempo costante $\Theta(1)$.

```
InsertionSort(V,n)                                  
{                                                   
  for (j = 2; j <= n; j++)                          sum_{j=2}^{n} (...)
  {                                                 
    x = V[j];                                       Θ(1)
    InserisceOrdinato(x,V,j-1);                     f(j)
  }                                                 
}                                                   


InserisceOrdinato(x,V,n)                            f(j) = ...   (con n = j − 1)
{                                                   
  for (i = n; (i > 0)&&(V[i] > x); i--)             sum_{i=p_x}^{n} (...)
    V[i+1] = V[i];                                  Θ(1)
  V[i+1] = x;                                       Θ(1)
}                                                   con p_x = indice finale di x in V
```

Figura 6.15: InsertionSort: complessità

Ora possiamo procedere risolvendo la sommatoria approssimata (vedi Figura 6.16). Sappiamo che possiamo scambiare il segno di sommatoria con il simbolo dello spazio funzionale Θ, e ci ricordiamo di sostituire n con $j-1$ nella sommatoria interna, relativa alla funzione `InserisceOrdinato`. Di conseguenza, ci riconduciamo a sommatorie su funzioni elementari. Rimane da risolvere il problema dell'indice incognito p_x, che compare nella sommatoria interna, e che non è costante, ma varia da iterazione a iterazione, dato che x cambia in ogni iterazione (quindi p_x dipende da j). Per procedere, dobbiamo fare delle ipotesi aggiuntive. Se siamo molto sfortunati (ipotesi del caso pessimo), p_x è sempre piccolo, cioè vale 1. In tal caso, si ottiene praticamente la somma aritmetica (con $j-1$ anziché j), che ricade in $\Theta\left(n^2\right)$, per cui l'algoritmo `InsertionSort` è quadratico nel caso pessimo.

Volendo discutere un caso medio, dovremmo ipotizzare una distribuzione sulla permutazione iniziale degli elementi del vettore V. Per esempio, si potrebbe assumere che le $n!$ permutazioni siano equiprobabili. Ne deriverebbe che p_x assume tutte le posizioni fra 1 e j con ugual probabilità[11], e quindi mediamente p_x vale $j/2$. Rispetto alla sommatoria precedente, compare solo un fattore $1/2$, ma il risultato ricade ancora in $\Theta\left(n^2\right)$. L'algoritmo `InsertionSort` è quadratico nel caso medio.

È interessante il caso ottimo (che in generale abbiamo denigrato in quanto poco probabile ed eccessivamente ottimistico). Questo si verifica quando $p_x = j$ sempre,

[11] Potrebbe anche essere che nel ciclo non si entri mai, perché `x` è maggiore degli elementi della tabella.

cioè quando il nuovo elemento `x` è non inferiore agli elementi della tabella, ovvero quando il vettore V iniziale è già ordinato. In questo caso, non si entra mai nel corpo del ciclo interno. In questo caso, rimangono le altre sommatorie e complessivamente il risultato ricade in $\Theta(n)$. L'algoritmo `InsertionSort` è lineare nel caso ottimo.

Perché è interessante il caso ottimo (che riguarda una sola permutazione su $n!$)? Perché se il vettore è quasi ordinato, il risultato non è molto diverso, e può capitare che si sappia a priori che i vettori che si vogliono ordinare siano in effetti già quasi ordinati. Questo può succedere quando un vettore viene alternativamente ordinato e perturbato, modificando di poco i valori correnti dei suoi elementi.

Un altro aspetto interessante di `InsertionSort` sono le costanti moltiplicative. Per quanto sia corretto ignorarle in un'analisi a grandi linee, va osservato che le costanti moltiplicative di `InsertionSort` sono piuttosto basse: ci sono poche operazioni, semplici e veloci: confronti, assegnamenti e accesso a vettori.

È vero che in generale si adotta il criterio di costo uniforme, quello del caso pessimo e la valutazione asintotica di complessità, e che altri algoritmi che analizzeremo in seguito sono strettamente migliori di `InsertionSort` da questi punti di vista. D'altra parte, se si hanno informazioni ulteriori sul problema che si sta affrontando, è possibile tenerle in considerazione e scegliere un algoritmo che nel caso specifico funziona meglio. Se l'istanza è quasi ordinata, l'algoritmo che ha un costo lineare, anche se su altre istanze è quadratico, è migliore di un algoritmo di costo $\Theta(n \log n)$. Se n è sufficientemente piccolo, una algoritmo di costo quadratico $c_2 n^2$ con c_2 piccolo costa meno di un algoritmo di costo lineare $c_1 n$ con c_1 grande. Sperimentalmente, si osserva che `InsertionSort` è più veloce di algoritmi più blasonati per vettori fino a una decina di elementi.

Riassumendo la precedente analisi dettagliata

$$T(n) = \sum_{j=2}^{n} (\Theta(1) + f(j)) = \sum_{j=2}^{n} \left(\Theta(1) + \sum_{i=p_x(j)}^{j-1} \Theta(1) + \Theta(1) \right)$$

da cui

$$T(n) \in \Theta\left(\sum_{j=2}^{n} 1 + \sum_{j=2}^{n} (j - p_x(j)) \right)$$

Ha costanti asintotiche piccole: è l'algoritmo migliore per istanze piccole

Per istanze grandi, la complessità dipende dal valore (incognito) di p_x

- caso pessimo, cioè $p_x(j) = 1$ sempre: $T(n) \in \Theta\left(n^2\right)$
- caso medio (per opportune distribuzioni): $T(n) \in \Theta\left(n^2\right)$
- caso ottimo, cioè $p_x(j) = j$ sempre: $T(n) \in \Theta(n)$

Il caso ottimo è interessante: corrisponde a vettori già ordinati (*o quasi*)

Figura 6.16: InsertionSort: complessità

6.4 Ordinamento per selezione (*SelectionSort*)

Il secondo algoritmo che consideriamo è noto come algoritmo di ordinamento per selezione, o *SelectionSort* (vedi Figura 6.17). Anche questo algoritmo lavora con tabelle implicitamente contenute nel vettore dei dati (e quindi è distruttivo) e ha complessità quadratica.

Questo algoritmo gestisce non una, ma due tabelle: una tabella ordinata contiene gli elementi della soluzione e si va gradualmente riempiendo come in *InsertionSort*, una non ordinata contiene i dati non ancora considerati e si va gradualmente svuotando. Al principio, la tabella non ordinata è piena e contiene tutti i dati e quella ordinata per la soluzione è vuota. Questo è diverso da *InsertionSort*, dove la tabella ordinata partiva con il primo elemento del vettore. Elemento per elemento, l'algoritmo sceglie nella seconda tabella il dato di valore massimo fra quelli non ancora considerati (da qui il nome "selection"), lo estrae e lo mette nella prima tabella mantenendola ordinata. La scelta dell'elemento massimo è cruciale, perché garantisce che ad ogni passo questo elemento sia rigorosamente non maggiore di quelli della soluzione parziale (altrimenti, sarebbe stato selezionato prima). Quindi, può essere inserito in cima alla tabella ordinata. Questo è diverso dall'inserimento classico per le tabelle (che avviene in fondo), ma la differenza è trascurabile: per costruire una tabella con inserimenti in cima anziché sul fondo, basta considerare ancora una volta un vettore dinamico, ma inserirvi gli elementi in posizioni segnalate da un indice che va decrescendo anziché crescere. Al posto della cardinalità (che era l'indice dell'ultimo elemento corrente), si conserva l'indice del primo elemento corrente.

Ovviamente, la tabella non ordinata che contiene i dati residui non è altro che la parte iniziale del vettore dato, mentre la tabella ordinata che contiene la soluzione sarà la parte finale, mentre nella *InsertionSort* era quella iniziale. Entrambe le tabelle, quindi condividono la memoria col vettore dei dati V, il che rende l'algoritmo distruttivo, nel senso che al termine il vettore dei dati iniziali non è più disponibile. D'altra parte, il consumo di memoria è ridotto al minimo essenziale.

L'algoritmo parte con la tabella ordinata vuota. Siccome rappresentiamo le tabelle come sottovettori $V[s, d]$, per indicare che una tabella è vuota, basta fissare $s > d$. Nel caso presente, siccome la tabella ordinata è la parte finale del vettore, per imporre che inizialmente sia vuota basta definire $T = V[j + 1, n]$ e porre $j = n$. Siccome la tabella dei dati non ancora processati è il sottovettore $V[1, j]$, inizialmente, essa contiene tutti i dati. Con il vettore dei dati V e un solo indice, j, quindi rappresentiamo entrambe le tabelle. Di passo in passo, decrementando j si trasferisce automaticamente l'elemento $V[j]$ dalla tabella dei dati a quella della soluzione. Il problema è che questo elemento non deve essere uno qualsiasi, ma l'elemento massimo fra quelli non processati. Quindi, prima di decrementare j, occorre trovare l'elemento massimo e scambiarlo con l'elemento finale $V[j]$, dopo di che il decremento opera correttamente.

La Figura 6.18 illustra uno pseudocodice, ancora una volta organizzato su due livelli per sottolineare la struttura dell'algoritmo. Si scorre il vettore all'indietro dall'indice `n` fino al principio (vedremo poi perché ci fermiamo all'indice 2 anziché 1). Ad ogni passo, si trova l'indice dell'elemento massimo nella tabella non ordinata che occupa la parte iniziale del vettore `V`, quella compresa fra 1 e `j`. Contrariamente a *InsertionSort*, non è necessario salvare l'elemento: basta scambiarlo con l'elemento finale, di indice `j`. Quindi, si decrementa `j`. Questa procedure richiede due sottoprocedure: quella che cerca l'indice dell'elemento massimo e quella che scambia due elementi. La funzione `Scambia` l'abbiamo discussa nella Sezione **??** trattando le chiamate di funzione, e quindi non la riprendiamo, salvo far osservare il passaggio

SelectionSort gestisce due tabelle

- i dati non ordinati come una tabella che si svuota progressivamente
- la soluzione come una tabella T ordinata che si riempie via via

Si procede in questo modo:

- inizialmente T è vuota
- ogni passo estrae l'elemento massimo da V e lo inserisce in cima a T

Per inserire in cima, la tabella è un vettore con indice iniziale decrescente

Rappresentiamo la tabella T con il sottovettore $V[j+1, n]$ e
la tabella dei dati residui con il sottovettore $V[1, j]$

- decrementando j, si sposta l'elemento $V[j]$ da V a T
- per spostare l'elemento massimo, basta prima scambiarlo con $V[j]$

Figura 6.17: Ordinamento per selezione

per indirizzo dei due elementi, che ovviamente devono essere modificati. La ricerca dell'indice dell'elemento massimo è semplice, dato che consiste nello scorrere gli elementi da 1 a n (il valore corrente della dimensione della tabella) e aggiornare l'indice `iMax` ogni volta che si trova un indice `i` il cui elemento `V[i]` supera quello considerato sinora il massimo (`V[iMax]`). Si inizializza `iMax` a 1 e si fa partire il ciclo da 2 sotto l'ipotesi che il vettore non sia vuoto.

Perché il ciclo esterno si interrompe alla posizione di indice 2 anziché 1? In realtà non è vietato proseguire fino a 1, ma tale iterazione vede un solo elemento nei dati, per cui l'indice dell'elemento massimo è chiaramente 1, cioè coincide con l'indice finale, e scambiare l'elemento massimo con quello finale significa non fare niente (o meglio, ricordando che cosa fa la funzione `Scambia`, eseguire operazioni inutili).

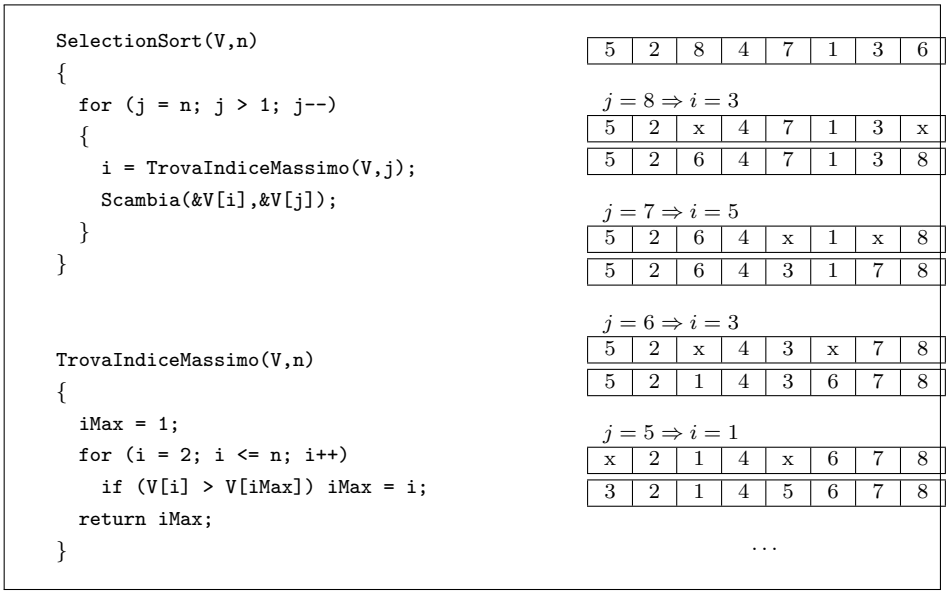
Le Figure 6.18 e 6.19 riportano un esempio numerico di esecuzione passo per passo della procedura stessa.

6.4.1 Implementazione

Possiamo implementare l'algoritmo `SelectionSort` sfruttando gli stessi materiali di supporto usati per `InsertionSort`. Basta sostituire la chiamata a quest'ultima con la chiamata a una nuova procedura, che ha esattamente gli stessi parametri:

```
/* Ordina il vettore */
SelectionSort(V,n);
```

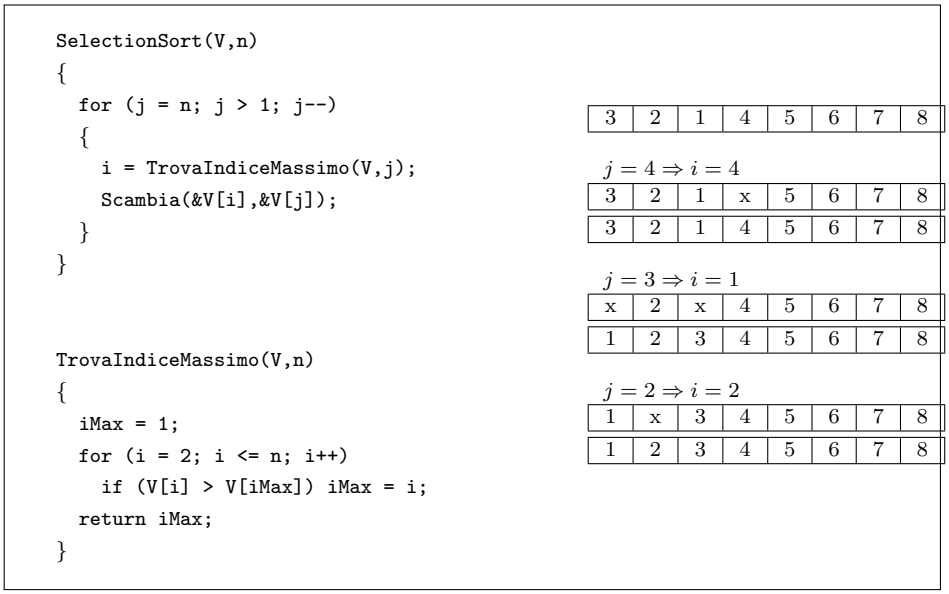
e quindi anche una dichiarazione e definizione praticamente identiche:



```
SelectionSort(V,n)
{
  for (j = n; j > 1; j--)
  {
    i = TrovaIndiceMassimo(V,j);
    Scambia(&V[i],&V[j]);
  }
}

TrovaIndiceMassimo(V,n)
{
  iMax = 1;
  for (i = 2; i <= n; i++)
    if (V[i] > V[iMax]) iMax = i;
  return iMax;
}
```


Figura 6.18: SelectionSort: pseudocodice ed esempio



```
SelectionSort(V,n)
{
  for (j = n; j > 1; j--)
  {
    i = TrovaIndiceMassimo(V,j);
    Scambia(&V[i],&V[j]);
  }
}

TrovaIndiceMassimo(V,n)
{
  iMax = 1;
  for (i = 2; i <= n; i++)
    if (V[i] > V[iMax]) iMax = i;
  return iMax;
}
```


Figura 6.19: SelectionSort: pseudocodice ed esempio

```
void SelectionSort (vint V, int n);
```

Si tratta ora di realizzare l'algoritmo, almeno al livello superiore, quindi dichiarando una variabile locale intera `j` che fa da cursore, e facendola calare progressivamente dalla cardinalità `n` a 2, mentre ad ogni passo si trova l'indice dell'elemento massimo del sottovettore $V[1, j]$, salvato in una variabile locale intera `i`, e si scambia l'elemento corrispondente con l'ultimo (`V[i]` con `V[j]`).

```
void SelectionSort (vint V, int n)
{
  int j, i;

  for (j = n; j >= 1; j--)
  {
    i = TrovaIndiceElementoMassimo(V,j);
    Scambia(&V[i],&V[j]);
  }
}
```

Questo richiede due funzioni ausiliarie, che andremo al solito a dichiarare e definire:

```
int TrovaIndiceElementoMassimo (vint V, int n);

void Scambia (int *pa, int *pb);
```

osservando che `TrovaIndiceElementoMassimo` restituisce un numero intero e riceve un vettore di interi e una dimensione intera, mentre `Scambia` non ha risultati, apparentemente, ma riceve due numeri interi passati per indirizzo, dato che fanno sia da dati sia da risultati.

Compilare ed eseguire il codice a questo punto produce la stampa del vettore iniziale, dato che l'algoritmo di ordinamento in effetti non fa nulla, ma permette di verificare se si siano commessi errori nella stesura di queste poche righe, prima che diventino troppe e che gli errori comincino a interagire fra loro in modi complicati ed elusivi.

Ora possiamo implementare `TrovaIndiceElementoMassimo`, dichiarando una variabile locale intera `iMax` per l'indice cercato, ipotizzando per cominciare che esso valga 1 e scorrendo il vettore con una variabile locale intera `i`, eventualmente aggiornando `iMax` nei passi in cui questo si rivela necessario. Come si è detto, la funzione ipotizza che il vettore non sia vuoto: in tal caso, restituirebbe l'indice 1, che sarebbe un risultato scorretto (ma interpretabile correttamente, dato che il vettore avrebbe cardinalità $n = 0$ e avremmo $i_{\max} > n$). Al solito, bisogna decidere a priori quanto e come le procedure sono tenute a gestire gli errori. Noi tendiamo verso la responsabilizzazione dell'utente, che deve sapere (dal commento alla dichiarazione) sotto quali condizioni la funzione può essere impiegata in modo sicuro.

```
int TrovaIndiceElementoMassimo (vint V, int n)
{
  int i, iMax;

  iMax = 1;
  for (i = 2; i <= n; i++)
    if (V[i] > V[iMax]) iMax = i;
```

```
  return iMax;
}
```

Per concludere, possiamo dichiarare e definire la funzione `Scambia`:

```
void Scambia (int *pa, int *pb)
{
  int temp;

  temp = *pa;
  *pa = *pb;
  *pb = temp;
}
```

che, come noto, salva uno dei due numeri interi `*pi` (ottenuto a partire dal suo indirizzo) in un *buffer*, ovvero variabile locale di sponda, `temp`; quindi, sovrascrive il numero salvato con l'altro (sempre ottenuto a partire dal suo indirizzo) e infine recupera il numero dal *buffer* per assegnarlo alla cella prima occupata dall'altro numero.

Compilazione ed esecuzione dovrebbero condurre al risultato richiesto.

6.4.2 Correttezza dell'algoritmo di *SelectionSort*

Per mostrare la correttezza dell'algoritmo, si procede per induzione matematica, come per *InsertionSort*. Si introduce cioè una proposizione, di cui si dimostra la correttezza per un numero naturale piccolo, e poi l'estensione da un numero generico al numero successivo (vedi Figura 6.20).

L'induzione è ancora sulla dimensione della soluzione parziale, che in questo caso non corrisponde a una variabile esplicita, ma alla differenza $n - j$, che va crescendo di passo in passo. La proposizione riguarda la tabella $T = V[j + 1, n]$ e combina tre parti: come per *InsertionSort*, la tabella è ordinata e i suoi elementi sono in corrispondenza biunivoca con un sottoinsieme del vettore originale; in aggiunta, i suoi elementi sono tutti non inferiori a quelli di $V \setminus T$.

Inizialmente, $j = n$, per cui la tabella T è vuota. Questo soddisfa automaticamente tutte e tre le condizioni, dato che degli elementi di un insieme vuoto si può dire quel che si vuole. Ad ogni iterazione, si decrementa j, cioè si incrementa $n - j$, e quindi la tabella include un nuovo elemento. Questo elemento fa parte di V (è $V[j]$, con $j \geq 1$) e non viola l'ordinamento di T, dato che tutti gli elementi di $V \setminus T$ sono non superiori a quelli di T, e quindi possono esservi inseriti in cima. Affinché, però, questa terza proprietà si conservi, bisogna che il nuovo elemento sia anch'esso non inferiore a quelli che restano in T, e quindi che sia di valore massimo. Questo è garantito dalle due funzioni.

Volendo essere rigorosi, e dimostrare la correttezza di `TrovaIndiceElementoMassimo`, basterebbe osservare che la proposizione "`iMax` è l'indice dell'elemento di valore massimo in $V[1, i - 1]$" è inizialmente vera (per `i = 2`) e rimane vera via via che `i` cresce; al termine, quando `i = n+1`, si ottiene la tesi. Ancora una volta, si tratta di introdurre una proposizione vera all'inizio e conservarne la validità ad ogni passo della procedura.

Quando il ciclo termina (supponiamo per semplicità che la condizione di permanenza sia `j > 0`, anziché `j > 1`), la proposizione è ancora vera, ma j vale 0. Di conseguenza, la tabella è ordinata e contiene tutti gli elementi del vettore iniziale, dunque è una permutazione ordinata del vettore, cioè una soluzione corretta.

L'algoritmo funziona per induzione matematica

- Al principio, $j = n$ e la tabella T
  1. è in corrispondenza biunivoca
     con un sottoinsieme di $n - j = 0$ elementi di V (*vuoto!*)
  2. gli elementi di T sono tutti $\geq$ agli elementi residui di V
  3. è ordinata
- Ad ogni iterazione, j cala di 1 e la tabella T
  1. include l'elemento massimo di V
  2. lo inserisce in posizione iniziale
  3. tale elemento è $\leq$ a tutti gli altri elementi di T

  Dunque conserva le tre proprietà

Al termine, $j = 0$ e la tabella T

- è in corrispondenza biunivoca con un sottoinsieme di n elementi di V
- è ordinata

In altre parole, per qualsiasi n e per qualsiasi vettore V,
al termine dell'algoritmo la tabella T è una permutazione ordinata di V

Figura 6.20: SelectionSort: correttezza

6.4.3 Complessità dell'algoritmo di *SelectionSort*

Passando alla complessità, il procedimento è il solito (vedi Figura 6.21): il ciclo corrisponde a una sommatoria, con un indice che cala da n a 2. Per proprietà commutativa, la sommatoria può salire da 2 a n. Le operazioni interne corrispondono al costo delle due funzioni, ignorando una serie di termini costanti che sono dominati. Diamo già per costante ($\Theta(1)$) la complessità di `Scambia`, che consiste in tre assegnamenti, e indichiamo con $f(j)$ la complessità di `TrovaIndiceElementoMassimo`, che opera sul sottovettore $V[1, j]$, e quindi probabilmente dipende da j.

La complessità $f(j)$ deriva da un assegnamento e un ciclo, quindi una seconda sommatoria, che va da 2 a n (il valore locale di n, pari a j) e riguarda un corpo costituito da poche operazioni elementari, in tempo costante.

```
SelectionSort(V,n)                                  
{                                                   
  for (j = n; j > 1; j--)                           sum_{j=2}^{n} (...)
  {                                                 
    i = TrovaIndiceMassimo(V,j);                    f(j)
    Scambia(&V[i],&V[j]);                           Θ(1)
  }                                                 
}                                                   


TrovaIndiceMassimo(V,n)                             f(j) = ...      (con n = j)
{                                                   
  iMax = 1;                                         Θ(1)
  for (i = 2; i <= n; i++)                          sum_{i=2}^{n} (...)
    if (V[i] > V[iMax]) iMax = i;                   Θ(1)
  return iMax;                                      
}                                                   
```

Figura 6.21: SelectionSort: complessità

Svolgendo i passaggi algebrici (vedi Figura 6.23), si ottiene una sommatoria esterna sull'indice j da 2 a n, contenente una sommatoria interna sull'indice i da 2 a j. La soluzione è abbastanza semplice, ed è in $\Theta\left(n^2\right)$. L'algoritmo di *SelectionSort* è quadratico.

Questa complessità non prevede casi fortunati e sfortunati: in qualsiasi situazione l'algoritmo esegue lo stesso numero di operazioni. *SelectionSort*, quindi, è un algoritmo poco efficiente in qualsiasi situazione. Lo discutiamo perché nel Capitolo **??** vedremo che l'introduzione di opportune strutture dati per gestire meglio la ricerca dell'elemento massimo consente di abbatterne la complessità da $\Theta\left(n^2\right)$ a $\Theta(n \log n)$, ottenendo un algoritmo che ha un nome diverso (*HeapSort*), anche se concettualmente esegue le stesse operazioni.

Volendo essere completi, ma eccedendo i limiti di questo corso, si può menzionare una circostanza a favore di *SelectionSort*. Le operazioni tipiche degli algoritmi di ordinamento che discutiamo in questo scorso sono sostanzialmente di due tipi (escludendo operazioni aritmetiche come gli incrementi e decrementi di indici): operazioni di confronto fra elementi dell'insieme base e operazioni di scambio fra posizioni del vettore (scambi veri e propri, come in *SelectionSort*, e spostamenti, come in *InsertionSort*). Dal punto di vista tecnologico, se andiamo oltre l'ipotesi

Figura 6.22: SelectionSort: complessità

Riassumendo la precedente analisi dettagliata

$$T(n) = \sum_{j=2}^{n} (f(j) + \Theta(1)) = \sum_{j=2}^{n} \left(\Theta(1) + \sum_{i=2}^{j} \Theta(1) + \Theta(1) \right)$$

da cui

$$T(n) \in \Theta\left(\sum_{j=2}^{n} 1 + \sum_{j=2}^{n} (j-1) \right) \Rightarrow T(n) \in \Theta\left(n^2\right)$$

La complessità è sempre quadratica, senza casi fortunati e sfortunati

Vedremo che una variante consente di abbattere la complessità

Figura 6.23: SelectionSort: complessità

semplificativa del costo uniforme, può essere interessate discutere il tempo richiesto, rispettivamente, da confronti e scambi. In alcune applicazioni piuttosto settoriali, può capitare di dover ordinare oggetti che non sono conservati nella memoria del processore (né nello *stack* né nello *heap*), ma su dispositivi molto lenti, come la memoria di massa (il disco fisso). In questo caso, è possibile che i confronti avvengano in memoria, e siano veloci, mentre gli scambi avvengono sulla memoria di massa, e siano lenti. L'algoritmo di *SelectionSort*, fra tutti gli algoritmi basati su confronti e scambi, è quello che esegue il minimo numero di scambi: ne esegue uno (ovvero tre assegnamenti) per ogni iterazione del ciclo principale. *InsertionSort*, invece, ad ogni iterazione scala in avanti lungo il vettore tutti gli elementi maggiori di quello corrente, per cui in generale esegue un numero quadratico di assegnamenti.

Quindi, la differenza nel tempo di esecuzione tra tipi diversi di operazioni è forte, un algoritmo che in complesso esegue più operazioni, potrebbe rivelarsi migliore, se esegue meno operazioni lente e più operazioni veloci. Ovviamente, la cosa ha un limite al crescere di n, perché una complessità asintotica di ordine superiore per definizione prima o poi riesce a superare qualsiasi costante moltiplicativa.

Lo scopo fondamentale di questa osservazione è sottolineare il fatto che la complessità asintotica nel caso pessimo è una buona guida, ma ogni informazione aggiuntiva sull'istanza da risolvere (quasi ordinata, piccola, con un insieme base U composto di oggetti veloci da confrontare e lenti da copiare), sulla macchina impiegata (memoria RAM o memoria di massa) e sull'applicazione dell'algoritmo consente di correggere e persino rovesciare le indicazioni che essa fornisce.

6.5 Esercizi

6.5.1 Esercizio

Si implementi la libreria `tab_int` per tabelle ordinate e si osservi come il programma realizzato per esercizio possa usare la nuova libreria senza richiedere alcuna modifica al proprio interno. Si rivalutino le complessità spaziali e temporali.

6.5.2 Esercizio

Si implementi la libreria `tab_int` con l'uso del terminatore e si osservi come il programma realizzato per esercizio possa usare la nuova libreria senza richiedere alcuna modifica al proprio interno. Si rivalutino le complessità spaziali e temporali.

6.5.3 Esercizio

Si trasformi la funzione `InsertionSort` togliendo il livello intermedio di chiamate a procedura, in modo da renderla più efficiente, correggendo il codice nei punti in cui ciò fosse eventualmente necessario.

6.5.4 Esercizio

Si trasformi la funzione `SelectionSort` togliendo il livello intermedio di chiamate a procedura, in modo da renderla più efficiente, correggendo il codice nei punti in cui ciò fosse eventualmente necessario.

Capitolo 7

Liste

Questa lezione e la successiva sono dedicate a un'altra struttura dati astratta, cioè le *liste*, che vengono descritte anche nelle dispense di teoria, ma che riprenderemo rapidamente e di cui discuteremo in grande dettaglio le molte possibili implementazioni, considerando un esercizio piuttosto sofisticato.

Vantaggi e svantaggi delle strutture dati astratte

Prima di dedicarci a questo argomento, volevo aggiungere una chiosa relativa alle tabelle che abbiamo già discusso, ma che merita riprendere in considerazione. Nel risolvere attraverso l'uso di una tabella il problema di caricare un insieme di numeri interi e poi stamparlo, cancellare elementi, aggiungerne, trovare la cardinalità dell'insieme, ecc..., abbiamo introdotto una libreria. Questa è composta da un file intestazione `tabint.h` e un file di definizione `tabint.c`, i quali nel complesso hanno lo scopo di rendere l'implementazione della struttura dati astratta "tabella di interi" invisibile a chi la usa ad alto livello per risolvere questo banale problema di gestire l'insieme di numeri interi.

Nel file `stampa_numero.c` che risolve il problema si vede chiaramente che chi usa questa struttura chiama le funzioni di stampa, di cancellazione, di inserimento e di cardinalità nel `main` o nella funzione di caricamento dei dati (l'altra funzione si limita a interpretare la linea di comando). Chi chiama queste funzioni lo fa senza avere la più pallida idea di che cosa sia una tabella di interi, perché si limita a dichiararla col nome `tabint`. Il compilatore sa che cosa deve fare perché include il file di intestazione che specifica tutto quello che gli serve sapere. Chi scrive il programma, invece, non ha alcun bisogno di sapere in che maniera viene implementata la tabella di interi.

Questo approccio ha degli svantaggi di cui dobbiamo essere consapevoli. In particolare, per un problema così semplice, che si potrebbe risolvere con un vettore statico o dinamico, come abbiamo visto, lo svantaggio fondamentale è che si aggiunge una serie di chiamate di funzione piuttosto macchinose. Come abbiamo visto nel capitolo ??, ogni volta che si chiama una funzione, bisogna valutare i parametri attuali, allocare il record di attivazione sullo *stack*, copiare i parametri attuali nei parametri formali. Quando l'esecuzione termina e si torna indietro, bisogna recuperare il risultato, se c'è un risultato. Tutto questo ha un costo in termini di tempo. Non è un costo enorme: è costante, $O(1)$, perché le strutture dati che stiamo passando sono puntatori, numeri interi, ecc... Però è un costo non trascurabile.

Perché abbiamo scelto di usare questa strada? Per vantaggi che riguardano la facilità d'uso, lo stile, l'eleganza. Il vantaggio fondamentale è che chi risolve il

problema ad alto livello non ha bisogno di sapere in che maniera esso venga risolto a basso livello. Chi scrive `stampa_numero.c` e chi scrive `tabint.h` e `tabint.c` possono essere persone completamente diverse, che non si parlano, tranne per i commenti introdotti nel file di intestazione per indicare dati e risultati di ciascuna funzione. Potreste essere voi che in un progetto d'esame, avendo bisogno di una tabella di interi, non la riscrivete, ma riciclate il codice definito in questa lezione. Oppure può essere la stessa persona a distanza di tempo, che prima costruisce la libreria, e poi la usa nel corso degli anni per progetti sempre nuovi. Il vantaggio evidentemente non è in termini di tempo di calcolo: si paga un pochino per avere questo vantaggio. Vedremo la stessa situazione discutendo le liste.

7.1 La struttura dati astratta "lista"

Quello che adesso faremo è, come per le strutture dati astratte precedenti, darne una definizione ad alto livello e poi proporre una o più implementazioni e mettere le mani nel codice. Qual è lo scopo delle liste intese come strutture dati astratte?

I vettori sono fantastiche strutture dati, molto efficienti in spazio e in tempo: richiedono solo lo spazio strettamente necessario per rappresentare l'informazione associata ai loro elementi e tutte le operazioni su tali elementi richiedono tempo costante. Tuttavia, sono strutture estremamente rigide perché rappresentano n-uple di elementi in cui n è fissato (vedi Figura 7.1). Le tabelle sono un po' più flessibili, perché sono n-uple di elementi con un n che è variabile, purché non superi un valore massimo, e oltre tutto consentono di leggere e scrivere, ma anche di inserire nuovi elementi e di cancellare elementi esistenti. Finché non siamo molto pretenziosi, inserimento e cancellazione sono in tempo costante. Essere non pretenziosi vuol dire aggiungere elementi in fondo alla tabella e cancellare elementi sovrascrivendoli con l'ultimo, dunque cambiandone l'ordine. Se avete bisogno di una tabella ordinata, come abbiamo visto nel caso degli algoritmi *InsertionSort* e *SelectionSort*, allora le cose cominciano a cambiare. In particolare, studiando *InsertionSort* abbiamo visto che per inserire in una tabella ordinata un elemento che non diventi il massimo, bisogna scalare in avanti tutti gli elementi più grandi, creare un buco nella tabella e poi scrivere il nuovo elemento nella tabella in tale buco. Questo comporta un tempo che è $\Theta(n)$, cioè lineare nella dimensione della tabella. Analogamente, una eventuale cancellazione ordinata richiederebbe di scalare all'indietro tutti gli elementi successivi a quello dato, e quindi un tempo $\Theta(n)$. Ci piacerebbe molto una struttura dati che sia anzi tutto completamente dinamica, senza il limite superiore alla dimensione, e poi che sia efficiente nell'inserimento e nella cancellazione, che le esegua in tempo costante, se possibile in qualsiasi posizione, altrimenti almeno in alcune posizioni privilegiate (sarà questo il caso di particolari liste dette *pile* e *code*).

Come si può fare? Intanto diamo la definizione astratta. Questa è piuttosto interessante, perché una lista su un insieme U non ha una definizione in forma chiusa come vettori e tabelle. Ha invece una definizione *ricorsiva*. Questo significa che una lista è una coppia eterogenea di oggetti, il cui primo elemento è un elemento a_1 dell'insieme base U (quindi un oggetto semplice), mentre il secondo elemento è una lista. Questo sembra essere un mangiarsi la coda, perché se una lista è costituita da due elementi di cui uno è una lista, non si vede come andare a termine. Però la definizione non è tautologica, perché ha un caso base, cioè esiste una lista che viene definite in sé e per sé, e non in riferimento ad altre liste. Si tratta della *lista vuota*, che èun semplice insieme vuoto, e come tale è possibile usarla come secondo elemento di una lista, la quale può a sua volta essere inglobata in altre liste. Quindi una lista ha un numero di elementi non definito a priori, illimitato, ma sempre

Un vettore rappresenta una n-upla di elementi con n fissato

Una tabella rappresenta una n-upla di elementi con n variabile ($\leq k$), ma

- l'inserimento di nuovi elementi avviene
  - in posizione terminale in tempo $\Theta(1)$
  - in una posizione desiderata in tempo $\Theta(n)$
    (*si scalano in avanti tutti gli elementi successivi*)
- la cancellazione avviene
  - riordinando gli elementi in tempo $\Theta(1)$
    (*l'ultimo elemento sostituisce quello cancellato*)
  - mantenendo l'ordine originale in tempo $\Theta(n)$
    (*si scalano all'indietro tutti gli elementi successivi*)

Si sente la mancanza di una struttura

- totalmente dinamica
- che consenta inserimenti e cancellazioni efficienti
  mantenendo l'ordine originale

Figura 7.1: Limitazioni di vettori e tabelle

finito, perché prima o poi si arriva ad avere la lista vuota. Nelle dispense di teoria, che si riferiscono alla macchina *RAM*, si dice che gli elementi di una lista hanno indici numerici da 1 a n. Qui useremo una definizione più astratta, dicendo che gli elementi della lista sono indicati da *posizioni*, che costituiscono un insieme P finito, in corrispondenza biunivoca con gli indici numerici da 1 a n, ma non ben definito nella sua natura, cioè totalmente astratto. Queste posizioni compaiono nella definizione della proiezione e della sostituzione, cioè per leggere e scrivere gli elementi. Si può leggere o scrivere un elemento se se ne conosce la posizione. Perché questa astrazione? Perché vedremo che in C, usando quindi un compilatore e un processore con un linguaggio di programmazione ad alto livello, ci sono diversi modi di implementare le liste. Uno dei modi principali fa corrispondere alle posizioni degli indirizzi di memoria, e sappiamo che gli indirizzi di memoria sono numeri interi, ma non sono compresi fra 1 e n e hanno un'aritmetica tutta loro diversa da quella dei numeri interi veri e propri. D'altra parte, un'altra implementazione userà vettori e indici numerici, ancora una volta non necessariamente compresi fra 1 e n. Quindi in sostanza per non legarsi a un'implementazione specifica definiamo le posizioni come oggetti astratti. Indicheremo convenzionalmente una lista come una sequenza $(a_1, a_2, \ldots, a_n)$, ricordando che le posizioni p_1, p_2,..., p_n non sono i valori numerici 1, 2,..., n.

Un primo aspetto fondamentale delle liste è che, conoscendo una lista non si può accedere immediatamente a tutti i suoi elementi, ma solo alla posizione del primo. Data una lista, cioè, si è in grado di conoscere solo p_1. Alcune liste (e qui cominciamo a distinguere casi e sottocasi) consentono di accedere anche all'ultima posizione: data L, si può conoscere p_n. Un secondo aspetto è che, data una lista e una posizione p_i generica, è possibile ricavarne non qualunque posizione, ma solo la successiva p_{i+1}. In un vettore, se conosciamo la posizione 5, possiamo facilmente

Una lista L su un insieme U ha una definizione ricorsiva: può essere

- un insieme vuoto Λ (caso base) oppure
- un elemento $a_1 \in U$ seguito da una lista L_1

La definizione non è tautologica perché il caso base arresta la ricorsione, ma il numero degli elementi non è soggetto ad alcun limite

Per accedere agli elementi della lista, anziché gli indici $\{1, \ldots, n\}$, le operazioni di proiezione e sostituzione usano un insieme di posizioni P

Gli insiemi delle posizioni e degli indici sono in corrispondenza biunivoca

$$\{p_1, \ldots, p_n\} \leftrightarrow \{1, \ldots, n\}$$

Indicheremo per comodità una lista L non vuota come $(a_1, \ldots, a_n)$, ma

- in genere le posizioni non coincidono con gli indici
- data la lista L, solo la posizione p_1 è nota (a volte anche p_n)
- da una posizione p_i si può ricavare solo la posizione seguente p_{i+1} (a volte anche p_{i-1})

Figura 7.2: Liste: struttura dati astratta

conoscere qualunque altra posizione del vettore: la 6, ma anche la 8, perché basta sommare 3 volte la dimensione del singolo elemento per passare da una all'altra posizione. In una lista questo non è vero: in una lista, data la posizione p_5 si può conoscere la posizione p_6, ma non istantaneamente la posizione p_8. Si può conoscerla passando per gradi attraverso p_6, p_7 e finalmente p_8. In alcune liste, e qui torniamo ai casi e sottocasi, è possibile conoscere, data una posizione p_i anche la posizione precedente p_{i-1}.

Quali sono le operazioni astratte che caratterizzano le liste? Si vedano le Figure 7.3 Ovviamente dobbiamo leggere e scrivere: tutte le strutture dati astratte devono consentire lettura e scrittura. Quindi, abbiamo la proiezione e la sostituzione. Che cos'è la proiezione? La *proiezione*, che è rappresentata dalla funzione `legge_lista`, riceve una lista e una posizione e fornisce in uscita un elemento dell'insieme base, cioè banalmente, data una lista e una posizione, indica l'elemento che sta nella lista in quella posizione.

La funzione `scrive_lista` realizza invece l'operazione di *sostituzione* che, data una lista, una posizione e un nuovo elemento dell'insieme base, fornisce in uscita una lista aggiornata, cioè una lista che è identica alla precedente, salvo nella posizione data, nella quale non c'è più l'elemento originale, ma c'è l'elemento che è stato passato alla funzione.

La terza operazione riguarda la distinzione tra le due grandi categorie di liste: quelle vuote e quelle generiche, che sono coppie. Per poter distinguere una lista vuota da una lista non vuota, abbiamo bisogno di una funzione `lista_vuota` che data una lista restituisce un valore booleano, falso o vero, che ci dice se la lista è non vuota o vuota.

Poi cominciamo ad accedere agli elementi, o meglio alle posizioni. Data una lista, è possibile accedere alla posizione del primo elemento attraverso una funzione `primo_lista` alla posizione dell'ultimo attraverso una funzione `ultimo_lista`[1].

[1] Il filmato dice che questo riguarda alcune liste, ma in effetti è possibile su tutte, eventualmente

Qui si apre una piccola questione. Se la lista fosse vuota, che senso avrebbe parlare di prima o di ultima posizione? Non ha effettivamente senso. Quindi, queste funzioni in teoria non dovrebbero essere definite. Teniamo un attimo da parte questa domanda.

Data una posizione in una lista, possiamo accedere alla posizione successiva con la funzione `succlista`, che riceve una lista e una posizione e restituisce una posizione. La funzione `predlista`, invece, data una lista e una posizione, restituisce la posizione precedente. Qui si pone una questione analoga alla precedente: se siamo nell'ultima posizione, e quindi non ne esiste una successiva, oppure siamo nella prima e non ce n'è una precedente, queste funzioni non dovrebbero essere definite. Dovrebbero essere funzioni parziali.

La verifica di appartenenza risolve il seguente problema: data una lista e una posizione, è sicuro che la posizione ha senso per la lista? Data la lista $L = (a_1, a_2, a_3, a_4)$, la posizione p_7 non ha senso per L. In tal caso, si deve poter ottenere una risposta di tipo `boolean`, vero o falso, secondo che la posizione sia sensata o no per la lista. Perché ci serve questo? Perché permetterà di evitare che le funzioni di accesso e scorrimento (`primolista`, `ultimolista`, `succlista` e `predlista`) sfondino i limiti della lista stessa. Questa funzione viene indicata col nome `finelista` per indicare che si sta uscendo dai confini della lista stessa.

Sia $\mathcal{L}$ l'insieme di tutte le possibili liste su U

Le liste ammettono tipicamente le seguenti operazioni

- proiezione: data una lista e una posizione, fornisce l'elemento corrispondente
$$\text{leggelista} : \mathcal{L} \times P \to U$$
- sostituzione: data una lista, una posizione e un elemento inserisce l'elemento nella lista sostituendo quello puntato dalla posizione
$$\text{scrivelista} : \mathcal{L} \times P \times U \to \mathcal{L}$$
- verifica di vuotezza: data una lista, indica se è vuota
$$\text{listavuota} : \mathcal{L} \to \mathbb{B} \qquad (\text{ovvero } \{0, 1\})$$
- accesso alla testa: data una lista, ne fornisce la prima posizione
$$\text{primolista} : \mathcal{L} \to P$$
E se la lista è vuota?
- accesso alla coda: data una lista, ne fornisce l'ultima posizione
$$\text{ultimolista} : \mathcal{L} \to P$$
E se la lista è vuota?

Figura 7.3: Liste: implementazione con puntatori

pagando un alto costo.

Sia $\mathcal{L}$ l'insieme di tutte le possibili liste su U

Le liste ammettono tipicamente le seguenti operazioni

- posizione successiva: data una lista e una posizione, fornisce la posizione successiva nella lista

$$\text{succlista} : \mathcal{L} \times P \to P$$

E se la posizione è l'ultima?

- posizione precedente: data una lista e una posizione, fornisce la posizione precedente nella lista

$$\text{predlista} : \mathcal{L} \times P \to P$$

E se la posizione è la prima?

- verifica di appartenenza: data una lista e una posizione, indica se la posizione appartiene o no alla lista

$$\text{finelista} : \mathcal{L} \times P \to \mathbb{B}$$

Figura 7.4: Liste: implementazione con puntatori

Dobbiamo poi poter inserire elementi e cancellare elementi dalla lista. L'operazione di inserimento riceve la lista, una posizione e un elemento dell'insieme base, e restituisce una lista aggiornata. Dati e risultati sono molto simili a quelli dell'operazione di sostituzione, ma non si tratta di una scrittura, bensì di un'aggiunta: la nuova lista contiene tutti gli elementi della vecchia, nello stesso ordine, ma nella posizione indicata contiene l'elemento che è stato aggiunto. L'elemento che stava inizialmente in quella posizione scala in avanti, e tutti i successivi scalano di un posto. È la stessa situazione che abbiamo visto nel trattare l'algoritmo di *InsertionSort* e che ci siamo posti il problema di eseguire in maniera efficiente, non in tempo lineare, ma costante, se possibile.

Lo stesso problema si pone per la cancellazione. La funzione `canclista` riceve una lista, una posizione e restituisce un'altra lista che è identica alla precendente, ma non ha l'elemento che stava nella posizione indicata. In tale posizione si trova l'elemento successivo, che è scalato all'indietro, richiamando tutti gli elementi successivi un passo indietro. Tutte queste cose sono sofisticate e in diversi punti capita che la funzione restituisca posizioni che non hanno senso (???). Che cosa facciamo in questi casi? Il problema è analogo a quello che abbiamo affrontato per le tabelle considerando inserimenti in tabelle sature o cancellazioni di posizioni che eccedono la cardinalità corrente. La soluzione classica che si adotta nelle liste è di definire una posizione fittizia, che non appartiene ad alcuna lista e che segnala la violazione dei limiti intrinseci della lista considerata (ad esempio, perché la lista è vuota). Questa posizione fittizia viene tipicamente indicata con il simbolo METTERE IL SIMBOLO, una T rovesciata che richiama il simbolo della terra nei diagrammi elettrotecnici. Questa posizione potrebbe essere anche un dato da fornire a funzioni, in particolare `finelista` a indicare posizioni farlocche per le quali la funzione deve restituire il valore `falso`. Questo permette di fare uno scorrimento di una lista, usando ad ogni passo la funzione `succlista` per passare all'elemento successivo e ogni volta interrogarsi sulla posizione che si sta puntando per chiedersi

se sia ancora valida oppure, essendo la posizione che segue l'ultima, è diventata una posizione non valida. Quando `finelista` diventa `vero`, significa che si è usciti dalla lista e quindi si può terminare il ciclo. Insomma, nel caso si eccedano i limiti di una lista sembra più ragionevole procedere in questo modo che interrompere l'intero programma segnalando un errore globale al sistema operativo.

Sia $\mathcal{L}$ l'insieme di tutte le possibili liste su U

Le liste ammettono tipicamente le seguenti operazioni

- inserimento: data una lista, una posizione e un elemento, aggiunge alla lista l'elemento nella posizione data

  $$\text{inslista} : \mathcal{L} \times P \times U \to \mathcal{L}$$

  L'elemento che stava nella posizione data e i seguenti scalano avanti

- cancellazione: data una lista, una posizione e un elemento, cancella dalla lista l'elemento nella posizione data

  $$\text{canclista} : \mathcal{L} \times P \to \mathcal{L}$$

  Gli elementi successivi scalano indietro

Alcune operazioni generano o ricevono posizioni non appartenenti a L

- la testa o la coda di una lista vuota
- la posizione successiva all'ultima o precedente alla prima

Indichiamo le posizioni esterne alla lista col simbolo aggiuntivo $\perp$

Figura 7.5: Liste: implementazione con puntatori

A questo punto, dobbiamo ricordarci del fatto che in C non abbiamo oggetti matematici che possiamo creare con la mente. Probabilmente, avremo a che fare con allocazioni e deallocazioni di memoria dinamica, visto che la struttura è molto dinamica. E allora ci serviranno, come per la tabella, una operazione di *creazione* e una di *distruzione*. La funzione `crealista` ci fornirà una lista, la funzione `distruggelista` la farà sparire nel nulla in qualche modo restituendo tutta la memoria eventualmente occupata al programma.

7.2 Implementazione con puntatori

Vedremo due implementazioni, che nelle dispense di teoria vengono descritte con riferimento alla macchina RAM. La prima è un'implementazione con puntatori, cioè con indirizzi di memoria; la seconda è un'implementazione con vettori e cursori, ovvero indici numerici. L'idea base della prima (vedi Figura 7.7) è che le posizioni non siano oggetti astratti, né valori numerici interi, né indici di registri, ma siano indirizzi di celle di memoria. Sono quindi numeri interi che si comportano in maniera particolare.

In questa implementazione, una lista si rappresenta con la posizione, cioè l'indirizzo, del primo elemento. Se la lista in questione fornisce anche l'accesso all'ultimo

In matematica basta definire un oggetto per crearlo

Nelle implementazioni concrete, però questo non sempre vale:
potrebbe occorrere qualche inizializzazione o allocazione dinamica

Per motivi tecnici, quindi è opportuno definire anche

- creazione: crea una lista vuota

$$\text{crealista} : () \to \mathcal{L}$$

- distruzione: distrugge una lista vuota

$$\text{distruggelista} : \mathcal{L} \to ()$$

Figura 7.6: Liste: implementazione con puntatori

L'idea base è di rappresentare le posizioni con indirizzi di memoria

- l'intera lista corrisponde allora a
  - posizione del primo elemento
  - eventualmente, posizione dell'ultimo elemento
- ogni elemento a_i della lista corrisponde a una struttura con
  - il dato a_i
  - la posizione successiva p_{i+1} ($\perp$ se a_i è in coda)
  - eventualmente, la posizione precedente p_{i-1} ($\perp$ se a_i è in cima)

```
#define NO_ELEMENT NULL                          (posizione esterna alla lista)

typedef elemento *lista;                  (la lista è l'indirizzo della testa)
typedef elemento *posizione;   (la posizione dell'elemento è il suo indirizzo)

typedef struct _elemento elemento;
struct _elemento {
   U a;                                   (U è il tipo dell'elemento generico)
   posizione succ;
   posizione pred;                                 (questo campo può mancare)
};
```

Figura 7.7: Liste: implementazione con puntatori

elemento, sarà rappresentata da una coppia di posizioni: quella del primo e quella dell'ultimo. In questa lezione considereremo solo lezioni con accesso al primo elemento. Nella parte inferiore della Figura 7.7 si definisce una lista con l'istruzione

```
typedef elemento *lista
```

cioè come puntatore a elemento. Anche una posizione è un puntatore a elemento:

```
typedef elemento *posizione
```

In realtà, quindi, lista e posizione sono la stessa, ma l'algoritmista non ha bisogno di saperlo: userà il tipo `lista` per riferirsi a un'intera lista e il tipo `posizione` per riferirsi alla posizione di un determinato elemento im una lista. Il fatto che siano entrambi indirizzi di memoria non dovrebbe riguardarci in alcun modo. Questo è lo scopo della struttura dati astratta.

Un elemento di una lista, invece, è per prima cosa un dato di tipo `U` (per esempio, un intero, come nelle tabelle di interi che abbiamo trattato in precedenza, ma anche una stringa, come nell'esercizio che affronteremo in seguito, o qualsiasi altra cosa, anche molto complicata). Oltre all'informazione vera e propria, questa struttura deve contenere le informazioni ausiliarie che consentono di scorrere la lista: data una lista e una posizione, dobbiamo essere in grado di conoscere la posizione seguente, ed eventualmente anche quella precedente. Questo viene ottenuto introducendo nella struttura una posizione, cioè un puntatore a elemento, `succ` e una posizione `pred`. In questo modo, ogni elemento può accedere al successivo, ed eventualmente al precedente, ma non direttamente ad altri.

Definiremo anche un valore `NO_ELEMENT` per incapsulare il puntatore `NULL`, ovvero nascondere il fatto che la posizione fittizia che non appartiene ad alcuna lista e che viene restituita quando si cerca il primo elemento di una lista vuota o il successivo dell'ultimo elemento, è il puntatore nullo del linguaggio C.

7.3 Tassonomia delle liste

Prima di considerare una per una le operazioni sulle liste, daremo una classificazione a grandi linee dei diversi tipi di lista. Contrariamente alle tabelle, che sostanzialmente hanno due implementazioni principali, per le liste c'è solo l'imbarazzo della scelta. La tassonomia che consideriamo è basata su tre aspetti indipendenti, ciascuno dei quali ammette due possibilità. Di conseguenza, avremo $2^3 = 8$ tipi di lista.

Il primo aspetto, riassunto nella Figura 7.8, è la *direzionalità*:

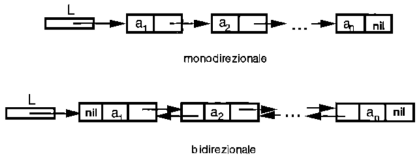
- una lista *monodirezionale* consente di andare da un elemento solo al successivo;
- una lista *bidirezionale* consente di andare da un elemento al successivo o al precedente.

Le liste monodirezionali sono implementate solo con il campo `succ`, oltre al campo informazione, quelle bidirezionali hanno sia il campo `succ` sia il campo `pred`.

Quale è migliore? È chiaro che l'implementazione bidirezionale occupa più spazio in memoria, quindi è meno efficiente rispetto allo spazio. Se però si vuole risalire una lista all' indietro, una lista monodirezionale apparentemente non lo consente. In effetti, si può fare, ripartendo dalla testa ad ogni passo e scorrendola fino a trovare la posizione la cui successiva è la posizione corrente. Quindi, trovare la posizione precedente è possibile, ma costa tempo $\Theta(n)$ nel caso pessimo, anziché $\Theta(1)$. Una

Vi sono tre aspetti implementativi fondamentali nelle liste a puntatori

1. direzionalità, cioè la presenza o assenza della posizione precedente
   - le liste monodirezionali hanno solo la posizione successiva
   - le liste bidirezionali hanno la posizione successiva e la precedente



Si tratta di efficienza spaziale contro efficienza temporale

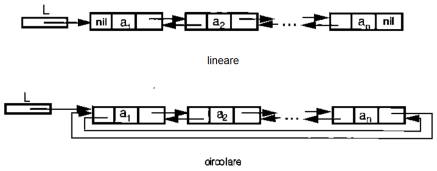
Figura 7.8: Varianti delle liste a puntatori

lista bidirezionale, invece, consente di trovare l'elemento precedente (e quindi di risalire la lista all'indietro) risparmiando tempo. È un classico compromesso fra memoria e tempo.

I vettori sono le strutture più efficienti in assoluto dal punto di vista della memoria, perché contengono solo lo spazio strettamente necessario a rappresentare i dati. Le tabelle possono allocare memoria in eccesso per garantire dinamicità. Le liste monodirezionali contengono gli n dati, n puntatori all'elemento successivo, un puntatore globale all'intera lista, ed eventualmente anche n puntatori all'elemento precedente. C'è un *overhead*, come dicono gli informatici, pari a $n + 1$, oppure a $2n + 1$, indirizzi.

Vi sono tre aspetti implementativi fondamentali nelle liste a puntatori

2. linearità o circolarità
   - le liste lineari terminano nella posizione esterna $\perp$
   - le liste circolari dopo l'ultimo elemento tornano al primo



Si può scorrere ripetutamente la lista, ma come arrestarsi?

Figura 7.9: Varianti delle liste a puntatori

Il secondo aspetto, riassunto nella Figura 7.9, distingue tra:

- liste *lineari*, in cui gli elementi stanno uno dopo l'altro, fino a raggiungere l'ultimo elemento, il cui campo `succ` assume il valore `NO_ELEMENT`;
- liste *circolari*, in cui l'ultimo elemento punta uno dei precedenti (tipicamente il primo, ma non necessariamente: evitiamo di ramificare ulteriormente la tassonomia).

La figura mostra esempi bidirezionali, ma potrebbero essere anche monodirezionali. Nelle liste lineari, è facile sapere se si è arrivati in coda, perché l'elemento successivo è fittizio. Nelle liste circolari, apparentemente si rischia di procedere indefinitamente. In realtà, se la lista si richiude sul primo elemento, basta confrontare l'elemento successivo con il primo della lista per rendersi conto che si è arrivati in coda. Questo determinerà l'implementazione della funzione `fine_lista`, che sarà diversa secondo il tipo di lista considerato.

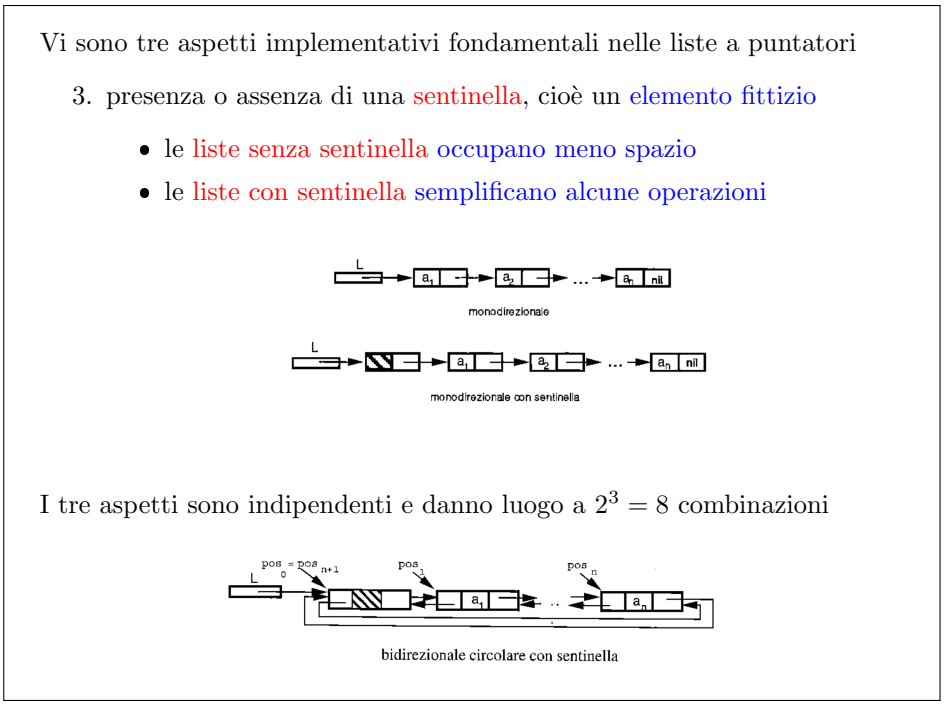


Figura 7.10: Varianti delle liste a puntatori

Il terzo aspetto, descritto nella Figura 7.10, si basa sulla presenza o sull'assenza di un elemento fittizio detto *sentinella*:

- nelle *liste senza sentinella*, il puntatore che rappresenta la lista punta direttamente il primo elemento;
- nelle *liste con sentinella*, il puntatore che rappresenta la lista punta la sentinella.

Nel secondo caso, l'elemento fittizio non ha niente nel campo informazione. La sentinella appare come uno spreco di spazio del tutto inutile. Implementando le funzioni, vedremo che la sentinella semplifica e rende eleganti alcune delle funzioni di base nella gestione delle liste. Ancora una volta, si tratta di capire se questo impiego di spazio aggiuntivo è giustificato dal guadagno in tempo o in semplicità di scrittura che otteniamo.

Abbiamo alluso in precedenza al fatto che si possono implementare liste non solo con puntatori, ma anche con vettori e indici interi. Vedremo questa implementazione nella Sezione **??**. Per ora, osserviamo che anche le liste a vettori e indici ammettono la stessa classificazione in 8 varianti, il che porta il loro numero complessivo a 16. Questo numero potrebbe salire ulteriormente considerando le liste che consentono l'accesso diretto all'ultimo elemento, ed eventuali altre operazioni più sofisticate. Nell'esercizio pratico, tuttavia, ci limiteremo a considerare le liste bidirezionali, circolari con sentinella, concentrandoci sull'implementazione a puntatori, e rimandando a una breve trattazione finale quella a vettori e indici. Di tanto in tanto, commenteremo i punti in cui altri generi di lista richiederebbero modifiche all'implementazione.

7.4 Laboratorio

L'esercizio che affrontiamo è piuttosto lungo[2]. Il suo scopo fondamentale è illustrare in pratica la modellazione di un problema che potrebbe richiedere l'uso di liste, la creazione di una libreria che implementi attraverso puntatori una lista bidirezionale circolare con sentinella, il suo uso per un'applicazione più o meno realistica, e la trasformazione della libreria stessa nel caso si voglia passare a un'implementazione a vettori. Vedremo che questo passaggio può essere compiuto senza fare alcuna modifica al codice che usa la libreria, dato che ci siamo premurati di rendere la struttura dati completamente astratta dall'implementazione.

Come l'esercizio sul gioco dell'Othello, il problema è sufficientemente complesso da assomigliare a un progetto d'esame. Quindi, porteremo avanti in parallelo tre contesti diversi, cercando di chiarire quando si passa da uno all'altro. I contesti sono:

1. il ripasso della programmazione nel linguaggio C e la discussione delle scelte a livello sintattico;
2. l'esercizio in sé, cioè la modellazione del problema, il progetto e la stesura di un algoritmo in modalità *top-down*;
3. l'impostazione di una relazione sul lavoro compiuto.

I tre contesti sono fortemente diversi. Chi ripassa un linguaggio di programmazione è interessato a rivedere la sintassi delle singole istruzioni. Chi segue passo per passo un esercizio è interessato alle singole fasi di realizzazione, fino ai dettagli più elementari. Chi legge una relazione non sa nulla del problema, ma si presume che non sia interessato alla programmazione in C e conosca gli algoritmi fondamentali. Quindi proverà confusione e fastidio a vedersi ricordare le singole istruzioni impiegate e l'implementazione degli algoritmi fondamentali, ma avrà interesse per il modello, per la scelta degli algoritmi e delle strutture dati e per il progetto di algoritmi specifici che non si trovano nel libro di testo. I tre contesti si intrecciano strettamente nella lezione, per cui diventa pesante sottolineare ad ogni passo a quale si sta facendo riferimento e non sempre lo faremo: serve buon senso.

Un concetto fondamentale della lezione è separare nettamente il concetto astratto di lista e la sua implementazione concreta. Questo comporta il dividere fisicamente da un lato la dichiarazione e definizione della struttura dati e delle operazioni che la manipolano (confinate in una libreria di due file) e dall'altro il loro uso (distribuito nel resto del programma). Questo comporta di usare sempre chiamate

[2]Le frequenti ripetizioni nel seguito derivano dall'incompleta fusione della traccia riassuntiva con la sbobinatura della lezione.

a funzioni per accedere ai dati, anche quando un accesso diretto sembrerebbe più semplice e naturale.

Il vantaggio di questo approccio è di poter gestire la struttura dati senza dover conoscere la sua implementazione concreta. È quindi persino possibile cambiare quest'ultima senza modificare il codice del programma, dato che cambia solo quello della libreria, che può essere sostituita in blocco da un'altra.

Questo approccio ha anche qualche svantaggio. Per prima cosa, avvicinarsi il più possibile a una struttura dati astratta con uno strumento concreto richiede un certo sforzo, molti dettagli tecnici di programmazione nel linguaggio C, e non è sempre possibile: a volte ci saranno comunque dissimiglianze tra la definizione teorica e l'implementazione concreta. Anche quando si riesce a mantenere una completa somiglianza, bisognerà introdurre qualche complicazione nella scrittura che a stretto rigore non sarebbe necessaria. In particolare, l'accesso ai dati tramite funzioni introduce per sua natura delle operazioni inutili (tutte quelle relative alla gestione del record di attivazione sullo *stack* di sistema). Alcune di queste inefficienze sono superabili arricchendo la libreria, cioè definendo qualche funzione aggiuntiva rispetto a quelle elencate nella definizione astratta. Altre sono superabili con strumenti tecnologici legati al linguaggio specifico utilizzato (per esempio, le macro in C o le funzioni *inline* in C++); di questi ultimi non parleremo.

7.4.1 La biblioteca

Si vuole scrivere un programma `biblio.c` per la gestione per una biblioteca. Al principio, il programma acquisisce i libri disponibili in ordine alfabetico (prima per autore, poi per titolo). Quindi, il programma riceve un elenco di indicazioni di movimento dei libri, vale a dire dei prestiti e delle restituzioni:

- i libri prestati vanno rimossi subito dall'elenco dei libri disponibili e conservati in un elenco a parte;
- i libri resi non tornano immediatamente disponibili, perché il personale della biblioteca è insufficiente a riordinarli subito; vengono invece conservati al banco del prestito impilati uno sull'altro fino alla sera.

Il programma deve tener traccia dell'ordine delle restituzioni perché alla sera deve fornire al bibliotecario un elenco di istruzioni che lo aiutino a riordinare i libri: in particolare, i libri non hanno una posizione assoluta negli scaffali della biblioteca, ma stanno in un solo scaffale (che si immagina sufficientemente lungo) e vengono conservati in fila, sempre in ordine alfabetico. L'istruzione, quindi, indica per ogni libro della pila (dalla cima al fondo) non una posizione assoluta, ma il libro già presente sullo scaffale prima del quale ciascun libro reso va riposto[3]. Si noti che quando un libro reso è stato riposto diventa immediatamente disponibile, e quindi può fare da punto di riferimento per l'inserimento dei successivi libri resi. Al termine, il programma deve stampare nuovamente la lista dei libri disponibili.

Formato di ingresso e di uscita L'elenco dei libri disponibili segue un formato molto semplice: ogni libro occupa una diversa riga, di lunghezza non superiore a 80 caratteri. Sulla riga vengono stampati l'autore (cognome e iniziale del nome), una virgola e il titolo fra virgolette. Queste informazioni dettagliate non sono significative, perché il problema richiede di gestire l'intera riga in blocco, senza

[3]Ovviamente, è una strana biblioteca, ma questo evita che l'esercizio si riduca ad accedere a un vettore con indici assoluti.

scomporla nelle informazioni dettagliate. I libri dell'elenco sono tutti diversi e sono disposti in ordine alfabetico per autore. Il termine dell'elenco è indicato da una riga contenente solo la parola `FINE`. Il file `input01a.txt` riporta un esempio di elenco dei libri disponibili.

```
Adams S., "Il principio di Dilbert"
Bertossi A., "Algoritmi e Strutture Dati"
Chaucer G., "I Racconti di Canterbury"
Hoeg P., "Il senso di Smilla per la neve"
Hofstadter D., "Godel, Escher, Bach"
Kernighan B. e Ritchie D., "Il linguaggio C"
King S., "L'ombra dello scorpione"
Pennac D., "La prosivendola"
Shakespeare W., "Le allegre comari di Windsor"
Yourcenar M., "L'opera al nero"
FINE
```

L'elenco dei movimenti avvenuti durante il giorno segue un formato simile: ogni movimento occupa una riga, costituita dalla parola chiave `PRESTITO` oppure `RESTITUZIONE`, seguita dal libro (nello stesso formato usato per l'elenco dei libri disponibili). I libri resi possono essere stati presi in prestito il giorno stesso (e quindi figurare nell'elenco iniziale) oppure in giorni precedenti (e quindi essere nuovi). L'elenco dei movimenti termina con una riga che contiene solo la parola chiave `FINE`. Il file `input01b.txt` riporta un esempio di elenco dei movimenti.

```
PRESTITO Bertossi A., "Algoritmi e Strutture Dati"
PRESTITO Kernighan B. e Ritchie D., "Il linguaggio C"
PRESTITO Adams S., "Il principio di Dilbert"
RESTITUZIONE Kernighan B. e Ritchie D., "Il linguaggio C"
PRESTITO Shakespeare W., "Le allegre comari di Windsor"
PRESTITO Pennac D., "La prosivendola"
PRESTITO Yourcenar M., "L'opera al nero"
RESTITUZIONE Bertossi A., "Algoritmi e Strutture Dati"
RESTITUZIONE Shakespeare W., "Le allegre comari di Windsor"
FINE
```

Per quanto riguarda i risultati, il programma deve restituire per prima cosa l'elenco dei libri ancora disponibili al termine della giornata, nello stesso formato usato per acquisirlo al principio. Poi deve fornire una serie di istruzioni. Ogni istruzione consiste in una riga costituita dalla parola chiave `METTERE`, seguita dal libro reso, dalle parole chiave `PRIMA DI` e dal libro disponibile prima del quale il libro reso va riposto. Se il libro va messo per ultimo, si scriverà la parola chiave `METTERE` seguita dal libro e dalle parole chiave `PER ULTIMO`. Al termine delle operazioni, il programma stampa la parola chiave `FINE`. La stampa dell'elenco finale segue lo stesso formato usato per acquisire la lista iniziale.

7.4.2 Modello del problema

Si tratta di una biblioteca molto semplificata, un po' pretestuosa nei suoi meccanismi, costruita in modo da costringerci ad avere come modello un insieme di liste di vario genere.

Per decidere come organizzare i dati del problema (e i risultati intermedi) occorre sapere quali operazioni bisogna compiere su di loro. Le operazioni richieste sono:

- scorrere i libri sullo scaffale per stamparne l'elenco;
- tenere traccia dei libri prestati e dei libri resi;
- determinare la posizione corretta di ciascun libro da riporre in base all'ordine alfabetico e all'elenco dei libri attualmente disponibili (cioè non prestati o già riposti in ordine) e fornire tali informazioni;
- scorrere nuovamente i libri sullo scaffale per stamparne l'elenco.

Il fatto che la biblioteca abbia un unico scaffale più o meno lungo con i libri in ordine alfabetico è abbastanza irrealistico, ma serve a semplificare il problema e a spingere la soluzione verso l'uso di liste. D'altra parte, anche il fatto che sia noto un file con i movimenti dell'intera giornata appare abbastanza irrealistico: bisognerebbe pensarlo più come un flusso di informazioni che arriva dinamicamente nel tempo, per cui ad ogni istante sono noti solo i movimenti precedenti.

Occorre quindi gestire tre insiemi di libri: quelli sullo scaffale, quelli in prestito e quelli restituiti. Sono tre insiemi completamente dinamici. Esiste un sovrainsieme statico che li contiene tutti e di cui costituiscono una partizione: l'insieme di tutti i libri. A stretto rigore, questo insieme può essere ricostruito scorrendo i due file di ingresso e unendo i relativi insiemi, ottenendo i libri che compaiono in almeno uno dei due. Quindi l'esercizio si potrebbe anche risolvere rappresentando i tre insiemi con tabelle, con una dimensione massima stimata a priori (magari per eccesso, sommando i libri inizialmente sullo scaffale e quelli soggetti a movimenti). Però supporremo che il programma debba gestire in tempo reale la situazione, aggiornando via via la situazione, e che quindi durante la sua esecuzione non possa fare questa stima, perché non conosce interamente il file dei movimenti, ma solo la parte già arrivata. Dobbiamo pensare che si scorra il file dei movimenti un passo alla volta durante la giornata, in tempo reale, e non si possa leggerlo una volta per tutte all'inizio. La situazione non è semplicemente dinamica nel senso che durante la scrittura del codice non conosciamo i dati, ma e' totalmente dinamica, nel senso che durante l'esecuzione del codice i dati non sono interamente noti. Nel corso della giornata, oltre a prestare libri noti, possono tornare libri non noti a priori, e il numero di questi ultimi potrebbe essere qualsiasi[4]. Questo rende il problema più realistico, anche se non realizzeremo un programma completamente dinamico, che supera i limiti dell'esercizio.

L'ipotesi di completa dinamicità spinge a scegliere di rappresentare i tre insiemi con liste e a realizzare le operazioni prima elencate come operazioni su liste di libri. Il motivo è proprio questa ipotesi di completa dinamicità senza limiti: potendo usarle, le tabelle sarebbero migliori per l'accesso diretto e il risparmio dei puntatori. Paradossalmente, gli studenti adorano scrivere codice inefficiente in tempo e spazio, che usa le liste per rappresentare insiemi statici. Esistono problemi in cui gli oggetti da manipolare appartengono a un insieme potenziale che si può stimare a occhio, ma è enorme, mentre gli oggetti da manipolare sono molti di meno. In questo caso, l'idea di gestire una tabella che li possa contenere tutti ospitandone in realtà solo una minima parte è assurda, perché richiede un consumo di memoria sproporzionato rispetto all'effettiva necessità. Un esempio di insieme dinamico di questo genere potrebbe essere quello delle posizioni di gioco dell'Othello. Se volessimo enumerarle, è chiaro che sono in numero finito, ma potenzialmente enorme. Inoltre, il loro numero non è facilmente prevedibile a priori, salvo con sovrastime molto cattive, perché alcune posizioni sono effettivamente raggiungibili, mentre altre non lo sono. Generare le posizioni è fattibile: si prende quella di partenza e, valutando tutte le mosse lecite, si generano le posizioni raggiungibili a un passo; poi da queste,

[4]Si potrebbe pensare a donazioni o acquisti da catalogare.

si generano quelle raggiungibili a due passi, e così via. È possibile conservarle in una tabella, ma bisognerebbe allocarla al principio, e quindi avere un'idea di quante siano. La stima sarà probabilmente molto sbagliata per eccesso, e siccome si tratta certamente di un numero grande, l'eccesso sarà sproporzionato. Conviene invece gestire un simile insieme in modo completamente dinamico, per esempio con una lista. Ricapitolando, una lista è una struttura particolarmente adatta per rappresentare un insieme molto dinamico la cui dimensione non è facilmente prevedibile a priori, e per la quale non esistono stime ragionevoli. Fuori da queste situazioni, l'uso di una lista comporta un aggravio nell'occupazione di memoria, a causa del sistema di puntatori, e operazioni di inserimento e cancellazione di elementi che sono efficienti se ne conosciamo la posizione, ma altrimenti non lo sono.

In dettaglio, il primo insieme (libri sullo scaffale) è in ordine alfabetico, prima rispetto all'autore e (nel caso di più libri dello stesso autore) poi rispetto al titolo, e deve rimanerlo. Il secondo insieme (libri in prestito) è in genere privo di un ordine, ma si può anche pensare che sia ordinato in base all'ora del prestito (per esempio, se si volesse aggiungere una funzione di stampa della sequenza dei prestiti ancora aperti). Il terzo insieme (libri resi) è in un ordine determinato dalla sequenza di movimenti di restituzione: i libri sono impilati e i libri resi per primi stanno alla base della pila, quelli resi per ultimi alla cima. Inoltre, il fatto che i libri siano impilati restringe l'accesso all'ultimo libro aggiunto all'insieme, secondo la politica nota come *LIFO* (*Last In First Out*).

L'algoritmo deve tenere traccia di che cosa avviene nel sistema modellato. Se un libro viene preso in prestito, scompare dall'elenco dei libri disponibili e si sposta nell'elenco dei libri che attualmente sono in prestito. Se un libro viene reso, si potrebbe pensare che questi libri vengano immediatamente riposti nell'unico scaffale. In realtà, per complicare un po' le cose, questi libri vengono messi sul bancone della biblioteca, molto rozzamente, impilandoli sopra i precedenti; fino a sera, nessuno li tocca, e quindi non possono nemmeno andare in prestito. Si ipotizza che i movimenti siano tutti sensati, cioè non vengano prestati libri che non sono sullo scaffale e non vengano resi libri che già sono sullo scaffale.

Terminato l'elenco dei movimenti, bisogna fornire ai bibliotecari delle istruzioni per riordinare la biblioteca, prendendo i libri che stanno impilati sul bancone, dall'ultimo che è stato reso al primo, e rimettendoli sullo scaffale. Qui abbiamo un'altra stranezza: sarebbe ragionevole che ogni libro abbia una propria posizione sullo scaffale, ma in tal caso lo scaffale sarebbe modellato correttamente con un vettore, in cui ogni libro avrebbe un indice numerico. Non volendo questo, ipotizziamo che i libri sullo scaffale siano tenuti uno dopo l'altro in ordine alfabetico: via via che i libri vanno in prestito, quelli rimasti si riaggregano verso sinistra, nelle prime posizioni dello scaffale; via via che i libri resi vengono rimessi a posto, vengono infilati nello scaffale spingendo in avanti tutti i successivi. Quindi, le istruzioni che l'algoritmo deve fornire ai bibliotecari consistono nell'indicare quale libro attualmente sullo scaffale deve essere spostato in avanti per fare posto al reso. Fra i libri che sono sullo scaffale, sarà il libro immediatamente successivo a quello da riporre. Quindi, l'istruzione dipende da quali libri sono sullo scaffale, da quali libri sono stati resi e da quali sono in prestito.

Il formato dei libri non ci interessa in modo particolare, perché non sono richieste operazioni che decomponano le diverse informazioni di ciascun libro: il modello per un libro sarà una semplice stringa di caratteri. L'elenco dei movimenti è molto simile: si tratta di operazioni di prestito e di restituzione su libri. Quindi modelleremo un'istruzione con due stringhe di caratteri: una per l'istruzione e una per il libro.

Avremo la parola chiave `PRESTITO` o la parola chiave `RESTITUZIONE`, seguite da uno spazio e dal libro oggetto di questa operazione nel solito formato con autore e titolo. Il file delle istruzioni si conclude con la parola chiave `FINE`, che indica la chiusura della biblioteca, e quindi l'inizio dell'ultima fase dell'algoritmo (il riordino).

Per quanto riguarda la stampa, dobbiamo semplicemente stampare i libri nel solito formato. Per quanto riguarda le istruzioni, dobbiamo stampare riga per riga le operazioni di riordino: la prima parola è la parola chiave `METTERE` a cui segue la stringa che descrive il libro reso, le parole chiave `PRIMA DI` e la stringa che descrive il primo libro da spostare sullo scaffale. È possibile che il nuovo libro vado inserito per ultimo, senza spostarne alcuno; in questo caso, dopo la parola chiave `METTERE` e la stringa che descrive il libro reso scriveremo le parole chiave `PER ULTIMO`. Al termine di tutto questo stamperemo la parola chiave `FINE`.

Nota : Nel contesto di una relazione, che cosa occorre riportare? Senz'altro descrivere il problema, cioè l'organizzazione della biblioteca e il compito dell'algoritmo, cioè i risultati da ottenere. Il formato di ingresso e di uscita è ininfluente, perché puramente convenzionale. Tutt'al più ha un limitato impatto sulla complessità delle operazioni di lettura e scrittura, ma questo in genere non richiede di precisarlo in dettaglio. La modellazione, con la giustificazione della scelta di liste in base al fatto che gli insiemi da rappresentare sono totalmente dinamici e richiedono operazioni di inserimento e cancellazione, è invece parte fondamentale di una relazione.

7.4.3 Decomposizione del problema

Passiamo ora a decomporre il problema nei sottoproblemi, in modo da poter impostare la loro risoluzione con l'approccio *top-down*. Questo ci suggerirà le operazioni da compiere, e quindi potrebbe indurci a tornare sulla modellazione per correggere alcune scelte, fatte prima di aver capito esattamente che cosa occorre fare.

L'algoritmo deve per prima cosa stampare a video l'elenco dei libri sullo scaffale. Al secondo passo, dobbiamo tener traccia dei movimenti in maniera da gestire lo stato della biblioteca, determinando quali libri passano dalla disponibilità sullo scaffale al prestito e quali dal prestito alla resa sul bancone. Si tratta di gestire i tre insiemi dinamici di libri. Quindi, avremo la determinazione delle operazioni di riordino, che richiede di scorrere in modo opportuno l'insieme dei libri resi e capire per ciascuno in quale modo spostarlo nell'insieme dei libri sullo scaffale, in particolare determinando il libro disponibile sullo scaffale prima del quale il libro reso va messo. Questa operazione richiede di gestire da un lato un ordine cronologico inverso (l'ultimo libro reso è il libro da ordinare) e dall'altro un ordine alfabetico (per i libri sullo scaffale) Infine, avremo nuovamente una stampa dei libri sullo scaffale, che è un problema già risolto, e ci suggerisce che la decomposizione probabilmente è corretta e proficua, dato che genera sottoproblemi identici.

7.4.4 Codice iniziale

Il punto di partenza dell'esercizio è il file `biblio0.c`, che già contiene:

1. le inclusioni per gestire ingresso e uscita, memoria, stringhe (dato che abbiamo dati su file, strutture dinamiche e oggetti modellati come stringhe;

2. le direttive per simulare il tipo `boolean`;

3. la costante simbolica `ROW_LENGTH` per gestire le righe in lettura e scrittura[5]
4. le variabili `file_libri` e `file_movimenti` dove conservare il nome dei file dei libri disponibili e dei movimenti effettuati;
5. la funzione `InterpretaLineaComando` che scrive i nomi dei due file nelle corrispondenti variabili recuperandoli dalla linea di comando (`argv[1]` e `argv[2]` rispettivamente).

```
/* Direttive */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define ROW_LENGTH 256


/* Prototipi delle procedure secondarie */

void InterpretaLineaComando (int argc, char *argv[], char
    *file_libri, char *file_prestiti);
```

Il `main` ha la solita struttura. Comprende una parte esecutiva con commenti che indicano il modo in cui abbiamo pensato di decomporre il problema.

```
/* Programma principale */
int main (int argc, char *argv[])
{
  /* Parte dichiarativa */
  char file_libri[ROW_LENGTH];
  char file_prestiti[ROW_LENGTH];

  /* Parte esecutiva */
  InterpretaLineaComando(argc,argv,file_libri,file_prestiti);

  /* Carica l'elenco dei libri disponibili */

  /* Crea le liste (vuote) dei libri prestati e resi */

  /* Esegue i movimenti riportati nel file dei prestiti e dei resi */

  /* Stampa l'elenco dei libri disponibili */

  /* Stampa ed esegue le istruzioni per riporre i libri resi */

  /* Stampa l'elenco dei libri disponibili */

  /* Dealloca le strutture dinamiche */

  return EXIT_SUCCESS;
}
```

La prima fase dell'esercizio richiede di realizzare al livello più astratto le procedure che risolvono i sottoproblemi, tenendole vuote, ma indicandone già i dati e i risultati. Abbiamo decomposto il problema nelle seguenti operazioni:

[5] Il testo indica una dimensione massima di 80 caratteri per ogni libro, a cui ne vanno sommati 12 per le parole chiave `PRESTITO` o `RESTITUZIONE` nel file dei movimenti, uno per lo spazio bianco fra la parola chiave e il libro, uno per l'a capo (se si legge ogni riga con la funzione `fgets`) e uno per il terminatore, per un totale di 94 caratteri. Ho preferito mantenere il valore 256 per andare sul sicuro e non introdurre complicazioni inutili.

1. creare la lista dei libri disponibili, quella dei libri prestati e quella dei libri resi: la prima è caricate da file, le altre due sono vuote;
2. eseguire i movimenti riportati nel file dei prestiti e dei resi;
3. stampare l'elenco dei libri disponibili;
4. stampare ed eseguire le istruzioni per riporre i libri resi;
5. stampare l'elenco dei libri disponibili;
6. distruggere le liste dei libri disponibili, prestati e resi (senza dimenticare gli elementi singoli delle liste stesse).

Siccome il modello assume di rappresentare gli insiemi come liste, e la struttura dati astratta lista è definita attraverso l'elenco di operazioni visto nella Sezione 7.1, possiamo usare queste operazioni, a patto di includere una libreria che le fornisca. Lo scheletro di questa libreria è già disponibile nei file di intestazione `listalibri.h`, che va ovviamente incluso nel nostro codice con un'apposita direttiva. Questa libreria propone un'implementazione a puntatori di una lista bidirezionale, circolare e dotata di sentinella. Questa implementazione consente un costo temporale molto basso per tutte le operazioni, al costo di un'occupazione di memoria maggiore (e un piccolo aumento nel costo temporale per alcune operazioni, dato che ci sono più puntatori da tenere aggiornati).

Il file di intestazione `listalibri.h` dichiara[6]:

- un `libro` come un vettore statico di `ROW_LENGTH` caratteri (come già discusso);
- una `listalibri` come un puntatore a un `elemento` (dato che l'implementazione è a puntatori);
- una `posizione` come un puntatore a un `elemento` (dato che l'implementazione è a puntatori);
- un `elemento` come una struttura contenente un libro (campo `AutoreTitolo`) e due posizioni: quella precedente (campo `pred`) e quella successiva (campo `succ`) (dato che l'implementazione è bidirezionale);
- tutte le funzioni elencate nei lucidi, con piccole differenze che commenteremo.

Il file listato `listalibri0.c` definisce le funzioni, ancora con i corpi vuoti, dato che dovremo riempirle come esercizio sulle implementazioni concrete della struttura dati astratta "lista". Per poter compilare senza messaggi di errore o avvertimento, alcune funzioni restituiscono valori fittizii convenzionali, comunemente detti di *default*: `NO_LIST` per le liste, `NO_ELEMENT` per le posizioni, `TRUE` per i valori logici[7].

È consigliabile, una volta terminato l'esercizio, provare a realizzare alcune delle altre implementazioni a puntatori, se non tutte e sette.

[6] Un aspetto poco gradevole di questa libreria è che i nomi delle funzioni sono totalmente generici, ma le liste che gestiscono sono liste di libri. Questo non crea problemi nell'esercizio, dove si adoperano solo liste di libri, ma potrebbe crearne nel caso in cui si vogliano gestire liste di oggetti di tipo diverso, eventualmente fondendo codici scritti in momenti diversi. Si è preferito conservare i nomi generici per semplificare la scrittura e tenere la coerenza con la parte di teoria.

[7] Questo è un punto sul quale sto riflettendo. Da un lato, consente di compilare il codice senza messaggi, neppure di avvertimento, coerentemente con l'idea che in ogni fase della scrittura dovremmo puntare ad avere codice perfettamente corretto, anche se incompleto. Dall'altro, noto che in pratica è facile dimenticare questi valori fittizii quando si procede a implementare le librerie, e quindi introduce errori spesso insidiosi. Conto di aggiornare dispense e codici dopo aver preso una decisione in proposito.

7.4.5 Prima fase: progetto *top-down*

La prima fase della realizzazione è contenuta nel file `biblio1.c`. Dovremo caricare dal file dei libri l'elenco dei libri disponibili in una lista `Scaffale`. Il modo più semplice di farlo è assegnare alla lista `Scaffale` il risultato della procedura `CaricaLibri` applicata al file `file_libri`.

```
Scaffale = CaricaLibri(file_libri);
```

Per poter scrivere tutto questo, bisogna che i tre simboli siano dichiarati, e quindi dovremo premettere la dichiarazione

```
listalibri Scaffale;
```

che avrà senso solo se includiamo la libreria con la direttiva

```
#include "listalibri.h"
```

in modo che il precompilatore inglobi tutto il contenuto del file di intestazione in cima al listato, e quindi premetta la dichiarazione di una lista di libri. Possiamo anche notare che in realtà il simbolo `ROW_LENGTH` è già definito in `listalibri.h` e quindi è incluso automaticamente e si può cancellare dal file `main.c`[8]

Rimane il simbolo `CaricaLibri`, per il quale bisogna procedere con il solito meccanismo in tre fasi: scritta la chiamata nel codice, si aggiunge la dichiarazione al principio, fra i prototipi, e poi la definizione al termine, fra le procedure secondarie. La funzione restituisce una lista di libri e riceve una stringa di caratteri.

```
/* Carica un elenco di libri dal file file_libri */
listalibri CaricaLibri (char *file_libri);
```

Il corpo vuoto presenta il solito problema che, per definizione, dovrebbe restituire qualcosa. Quindi, compilando con le opzioni rigorose `-Wall -pedantic`, otteniamo due avvertimenti:

```
warning: variable 'Scaffale' set but not used [-Wunused-but-set-variable]}
   listalibri Scaffale;
              \^
biblio1.c: In function 'CaricaLibri':
biblio1.c:89:1: warning: control reaches end of non-void function [-Wreturn-type]
 }
 ^
```

Il primo indica che la variabile `Scaffale` ha ricevuto un valore (il risultato di `CaricaLibri`), ma non è stata usata in seguito per altre operazioni. Il secondo è che il controllo raggiunge la fine di una funzione non-void nella riga 68: effettivamente, alla riga 68 termina la funzione `CaricaLibri`, che dovrebbe restituire una lista di libri, ma non restituisce niente. Se vogliamo evitare il messaggio, possiamo far restituire alla procedura una lista fittizia, `NO_LIST`[9]

Ora dobbiamo creare delle liste vuote per i libri prestati e per i libri presi. Questo è molto semplice, dato che la funzione di libreria `crealista` lo fa. Quindi possiamo

[8] Passaggio piuttosto goffo, ma la definizione serviva a rendere corretto il codice con le definizioni dei nomi dei file. Forse conviene distinguere la lunghezza di un libro da quella del nome di un file.

[9] Come già detto, tendo a pensare che sia un'idea meno buona di quanto pensassi anni fa.

dichiarare le liste di libri `Prestito` e `Resi` e assegnare a ciascuna una lista vuota generata da questa funzione.

```
/* Crea le liste (vuote) dei libri prestati e resi */
Prestiti = crealista();
Resi = crealista();
```

A questo punto, dobbiamo eseguire le operazioni riportate nel file dei movimenti. I dati della procedura sono, oltre al file dei movimenti, le tre liste di libri, dato che bisogna sapere quali libri sono disponibili all'inizio, quali vanno in prestito e quali vengono resi. Si tratta di dati, ma anche di risultati. Quindi, in teoria andrebbero in uscita alla procedura. In C, però, non è possibile restituire più di un risultato semplice. Di conseguenza, passeremo le tre liste (`Scaffale`, `Prestiti` e `Resi`) come argomenti, ma per indirizzo.

Parentesi tecnica Qui i apre una disquisizione tecnica di basso livello, che purtroppo interrompe il flusso dell'esercizio. Usando le strutture dati astratte, si cerca sempre di librarsi negli alti cieli della matematica e si viene continuamente trascinati in basso dal fatto di dover maneggiare oggetti concreti e limitati. Costringere un oggetto fisico a comportarsi in base a regole pensate per oggetti matematici non è banale: si cerca di avvicinarsi il più possibile a questo ideale, non sempre riuscendoci perfettamente. Se si implementa una lista a puntatori con sentinella, il puntatore che rappresenta la lista è l'indirizzo della sentinella (vedi Figura 7.10), e quindi rimane sempre lo stesso durante l'intero uso della lista, qualunque sia il contenuto di questa. Questo significa che le funzioni che modificano le liste potrebbero riceverle passate per valore, anziché per indirizzo, esattamente come si fa quando si passa una stringa a una funzione che ne modifica i caratteri o un vettore a una funzione che ne modifica gli elementi. Solo la deallocazione di una lista con sentinella modifica il puntatore che la rappresenta.

D'altra parte, le liste senza sentinella si comportano diversamente, perché inserendo il primo elemento in una lista vuota o cancellando l'unico elemento di una lista data, il puntatore deve invece cambiare. Quindi, usare il passaggio di parametri per valore significa automaticamente rivelare che la lista è implementata con sentinella e rendere il codice inutilizzabile per un'eventuale implementazione senza sentinella. Siccome stiamo cercando di mantenere il più possibile astratta l'implementazione e indipendente dall'implementazione il codice, ci costringiamo a passare le liste per indirizzo a tutte le funzioni che ne modificano gli elementi, a costo di complicare un po' la scrittura.

Una questione che merita dibattere è se questo comporti dei costi eccessivi. In effetti, il costo è un passaggio aggiuntivo da puntatore a cella puntata, che consideriamo trascurabile, pur non essendo di costo rigorosamente nullo.

A questo punto, dobbiamo stampare l'elenco dei libri. Chiameremo una funzione `StampaLibri`, a cui passeremo `Scaffale` per valore, dato che la procedura non esegue nessuna modifica sulla lista. Siccome nessuna delle 16 implementazioni della lista ha bisogno di cambiare il valore della variabile, sarebbe veramente fuori luogo[10] passare la lista per indirizzo.

```
/* Stampa l'elenco dei libri disponibili */
StampaLibri(Scaffale);
```

[10] E quindi verrebbe penalizzato durante la valutazione di un progetto d'esame, secondo il criterio della struttura del codice.

Dovremo quindi stampare ed eseguire le istruzioni per riordinare i libri resi. La procedura `RiordinaResi` progressivamente svuota la lista `Resi` e riempie la lista `Scaffale`. Siccome le due liste possono cambiare, le passiamo per indirizzo.

```
/* Stampa ed esegue le istruzioni per riporre i libri resi */
RiordinaResi(&Resi,&Scaffale);
```

Per ciascuno di questi tre passaggi, bisogna seguire il movimento in tre fasi (chiamata, dichiarazione, definizione). Fa eccezione la creazione delle liste, dato che la dichiarazione è contenuta nel file di intestazione, incluso a monte, e la definizione è contenuta nel file listato della libreria. Dovremo invece farlo esplicitamente per la stampa della lista, l'esecuzione dei movimenti e il riordino dei libri resi.

```
/* Stampa l'elenco dei libri disponibili */
void StampaLibri (listalibri L);

/* Esegue i movimenti riportati nel file dei prestiti e dei resi
   sulle liste Scaffale, Prestiti e Resi */
void EsegueMovimenti (char *file_prestiti, listalibri *pScaffale,
                      listalibri *pPrestiti, listalibri *pResi);

/* Stampa ed esegue le istruzioni per riporre i libri resi nello
    scaffale */
void RiordinaResi (listalibri *pResi, listalibri *pScaffale);
```

La stampa è una funzione `void` che riceve una lista e la stampa a video. È sensato che il nome della lista sia generico, dato che potremo usare questa funzione su qualsiasi lista di libri, anche se l'esercizio chiede di farlo solo per i libri sullo scaffale. A rigore, una funzione che stampi una lista di libri potrebbe persino essere una buona aggiunta alla libreria, data la sua generalità. Si potrebbe quindi pensare di spostare il prototipo nel file di intestazione e il corpo nel file del listato.

L'esecuzione dei movimenti è una funzione `void` che riceve una stringa con il nome del file dei prestiti e tre puntatori a `listalibri` associati alle tre liste. Come nel resto del corso, adottiamo la convenzione di premettere una `p` ai nomi originali delle variabili quando vogliamo sottolineare che si tratta di puntatori a tali variabili, e non di copie. Nel caso delle liste, sono celle nello *scope* della funzione, che puntano la cella del `main`, che contiene l'indirizzo della sentinella, la quale sta da qualche parte nell'*heap* di sistema. Siccome è facile confondersi, qualunque marchingegno ci aiuti a ricordare la situazione concreta è il benvenuto.

Infine, la procedura di riordino dei resi è una funzione `void` che riceve due liste passate per indirizzo.

Ciò fatto, stamperemo il nuovo elenco dei libri disponibili, e qui per fortuna la chiamata è assolutamente la stessa.

Infine, dovremo deallocare tutte le strutture dinamiche, per liberarne la memoria. La libreria offre una funzione `distruggelista` per distruggere liste. Questa procedura richiede in ingresso una lista di libri passata per indirizzo, dato che la modifica. Questa funzione andrà chiamata su ciascuna delle tre liste che abbiamo definito.

```
/* Dealloca le strutture dinamiche */
distruggelista(&Scaffale);
distruggelista(&Prestiti);
distruggelista(&Resi);
```

Questa funzione è già stata dichiarata e definita nella libreria.

A questo punto, il codice dovrebbe essere corretto dal punto di vista sintattico, e quindi si dovrebbe poter compilare la coppia di file `biblio1.c` e `listalibri0.c` senza errori. Eseguendo il programma, ovviamente, non succede nulla, dato che le funzioni sono ancora tutte vuote. L'unica operazione eseguita è il controllo del formato della linea di comando, cioè che l'utente passi due nomi di file al programma.

7.4.6 Seconda fase: libreria per gestire la lista

La strategia *top-down* comporterebbe ora di scegliere alcune funzioni e svilupparle, risolvendo alcuni sottoproblemi, o eventualmente riducendoli a sottoproblemi di livello ancora inferiore. Invece, sospenderemo per un attimo l'approccio *top-down* e passeremo all'approccio *bottom-up*. Perché?

Premesso che la scelta del modo di sviluppare il proprio progetto è personale e dipende dalla situazione specifica, per cui dare regole generali è difficile, se non impossibile, si possono proporre alcune buone ragioni. Per prima cosa, per poter controllare via via se il codice realizzato è corretto, ha senso che in ogni fase l'algoritmo produca un risultato parziale verificabile. D'altra parte, finché le procedure di gestione delle liste rimangono vuote, questo sarà praticamente impossibile. Rischiamo di scrivere centinaia di righe di codice valutando solo la correttezza sintattica (cioè se il codice è scritto in linguaggio C comprensibile dal compilatore), prima di ottenere qualche risultato che suggerisca se vale anche la correttezza semantica (cioè se il codice esegue le operazioni necessarie a risolvere il problema). A quel punto, potremmo aver generato molti errori, anche insidiosi, che interagiscono fra loro. Non è una buona strategia. Meglio scrivere codice che, oltre ad essere sempre sintatticamente corretto, si avvicini pian piano semanticamente a risolvere il problema complessivo, verificandolo in base a risultati parziali ottenuti via via. Inoltre, in un progetto professionale è probabile che si usino librerie già pronte, per cui completare la libreria ora significa ricondursi al più presto in una situazione che è frequente nelle applicazioni. Infine, avendo appena ripassato la definizione astratta di lista e le caratteristiche fondamentali delle sue implementazioni, passare subito alla realizzazione pratica appare piuttosto naturale dal punto di vista didattico.

Dunque, in questa fase realizzeremo l'intera libreria, completando il file del listato `listalibri.c`, una funzione alla volta. Per capire tutti i passaggi sarà utile basarsi su rappresentazioni grafiche del genere di lista desiderato, che per noi sarà la lista bidirezionale circolare con sentinella (l'ultima nella Figura 7.10). Ovviamente, tutto questo fa parte del contesto "lezione di algoritmi": non va descritto in un'ipotetica relazione, dato che la libreria è standard e al lettore basta sapere che si usa una lista con l'implementazione a puntatori bidirezionale circolare con sentinella, e il motivo della scelta, che abbiamo già discusso.

Nell'ordine, partiremo definendo `crealista()`, che crea una lista vuota come in Figura 7.11, cioè alloca un elemento sentinella, lo fa puntare a sé stesso in entrambe le direzioni (lista circolare) e lo restituisce come valore della lista. I puntatori garantiscono che la lista sia degenere e consista solo della sentinella. L'allocazione può essere effettuata con le istruzioni `malloc` o `calloc`: la seconda produce automaticamente una stringa vuota nel campo `AutoreTitolo`, che e' comodo, anche se basterebbe usare la prima e azzerare il primo carattere della stringa, per farlo coincidere con il carattere terminatore di stringhe. Il puntatore restituito da queste funzioni viene assegnato alla variabile `L` che si restituisce all'esterno e che viene usato come valore dei due puntatori `pred` e `succ`.

```
listalibri crealista ()
{
```

Consideriamo l'implementazione bidirezionale, circolare con sentinella (come nel codice)

`crealista/listavuota`

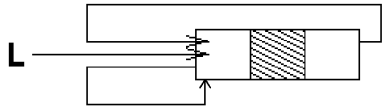


Figura 7.11: Lista vuota

```
  listalibri L;

  L = malloc(sizeof(elemento));
  if (L == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione di una lista!\n");
    exit(EXIT_FAILURE);
  }

  strcpy(L->AutoreTitolo,"");
  L->pred = L;
  L->succ = L;

  return L;
}
```

Si è già accennato al fatto che sono dispute lunghissime su quanto sia giusto o sbagliato convertire il puntatore generico restituito in un puntatore a lista (con un'operazione che non fa effettivamente nulla e che viene eseguita automaticamente comunque dal compilatore). In mancanza di chiare ragioni, adotto questa scrittura per ricordare i tipi del puntatore e dell'oggetto puntato. Completata l'allocazione, verifichiamo che sia andata a buon fine, cioè che il risultato non abbia valore `NULL`, perché nel caso proseguire significa assumere che un'area di memoria sia stata resa disponibile mentre non lo è. Questo genere di errore merita l'abbandono del programma, dopo una stampa sullo standard error per avvertire l'utente. Se invece tutto funziona, una volta riempiti correttamente i campi puntatore e informazione (con una stringa vuota) si può restituire all'esterno come indirizzo della lista quello della sentinella, cioè `L`.

A questo punto, nel `main` le liste `Prestiti` e `Resi` sono già inizializzate correttamente. Non succede lo stesso per `Scaffale`, che è inizializzato in `CaricaLibri`.

Rimandiamo al termine la distruzione della lista, dato che comporta di distruggere anche tutti gli elementi uno per uno, dunque uno scorrimento con cancellazione, per cui aspettiamo di aver prima realizzato scorrimenti e cancellazioni.

Procediamo quindi con *proiezione* e *sostituzione*, cioè lettura e scrittura (vedi Figura 7.3). Qui sorge un problema: la funzione astratta `leggelista` riceve una lista e una posizione e restituisce un elemento dell'insieme base, cioè un libro. Que-

sto corrisponderebbe a una funzione che riceva un oggetto di tipo `listalibri` e un oggetto di tipo `posizione` e restituisca una striga. Ma il linguaggio C non consente a una procedura di restituire un oggetto composto. La libreria adotta invece il classico meccanismo per cui un risultato composto viene trattato come un dato, ma passato per indirizzo. Siccome una stringa è un vettore di caratteri, il passaggio per indirizzo avviene banalmente attraverso il puntatore alla cella di indice 0. L'implementazione non rispecchia sempre perfettamente la struttura dati astratta.

```
void leggelista (listalibri L, posizione p, libro TitoloAutore)
```

Si potrebbe pensare di restituire un vettore dinamico allocato internamente alla funzione. Questo per prima cosa modificherebbe la definizione di `libro`, che è un vettore di caratteri statico, non dinamico. Inoltre, vorrebbe dire allocare nuova memoria ad ogni operazione di lettura, una cosa di un'inefficienza assurda[11]. Un'altra strategia potrebbe essere di restituire una struttura contenente una stringa. Questo richiederebbe di definire un nuovo tipo di dato per la struttura, ma soprattutto sarebbe inefficiente dal punto di vista temporale, dato che comporterebbe una copia al termine della funzione, quando si restituisce il record di attivazione, oltre alla copia che ha luogo internamente alla funzione (dall'elemento della lista alla variabile che contiene il risultato). Infine, si potrebbe congegnare la funzione di lettura in modo che restituisca direttamente l'indirizzo del dato, invece di copiarlo in un vettore passato dall'esterno. Questo sarebbe efficiente, ma allontanerebbe del tutto dalla definizione astratta.

In conclusione, ci limitiamo a passare a `leggelista` la lista `L`, la posizione `p` e l'indirizzo `AutoreTitolo` dove scrivere il risultato e poi copiamo in tale indirizzo il campo `AutoreTitolo` dell'elemento in posizione `p` della lista `L`, con la consueta funzione `strcpy` per la copia di una stringa in un'altra. Come si trova il campo dell'elemento desiderato? Dipende dall'implementazione scelta. Nell'implementazione a puntatori, un elemento è semplicemente l'oggetto puntato da una posizione, cioè **p*. Ai suoi campi si accede con l'operatore `->`, da cui:

```
void leggelista (listalibri L, posizione p, libro AutoreTitolo)
{
  strcpy(AutoreTitolo,p->AutoreTitolo);
}
```

La funzione di sostituzione `scrivelista` è assolutamente complementare: riceve una lista, una posizione e un libro e lo copia nel campo informazione dell'elemento in posizione data nella lista data. Il risultato è la lista modificata, in modo da rispecchiare esattamente la definizione astratta. Per ottenere questo risultato basta copiare la stringa data nel campo `AutoreTitolo` dell'elemento in posizione `p` della lista `L`, dunque:

```
listalibri scrivelista (listalibri L, posizione p, libro AutoreTitolo)
{
  strcpy(p->AutoreTitolo,AutoreTitolo);
  return L;
}
```

Possiamo notare che di per sé sarebbe già sufficiente modificare il campo senza restituire la lista, dato che l'operazione modifica la lista iniziale, senza crearne

[11] Ma non rarissima nei progetti consegnati dagli studenti.

una nuova. In effetti, attraverso il puntatore `p` abbiamo passato per indirizzo il campo da modificare. Ancora una volta, però, stiamo cercando di avvicinarci il più possibile alla definizione astratta di lista, che riceve una lista, una posizione e un oggetto dell'insieme base e restituisce una lista. Siccome in questo caso simulare la struttura astratta costa poco, lo facciamo senz'altro.

Proseguiamo con l'identificazione di una lista vuota. In base alla Figura 7.11, una lista bidirezionale circolare con sentinella è vuota quando la sentinella punta sé stessa con entrambi i puntatori, dunque `L->pred` e `L->succ` coincidono con `L`. D'altra parte, possiamo limitarci a testare una delle due condizioni, dato che se una vale e l'altra no, la lista è corrotta, cosa che dovrebbe risultare impossibile quando si inizializza e si modifica la lista attraverso procedure corrette. Se ricordiamo che in C si possono direttamente scrivere espressioni logiche e ricavarne il valore, possiamo direttamente restituire il valore del confronto fra `L->succ` e `L`.

```
boolean listavuota (listalibri L)
{
  return (L->succ == L);
}
```

È una buona idea interrogarsi su come sarebbe possibile realizzare questa funzione in implementazioni diverse (in particolare, prive di sentinella).

Possiamo quindi realizzare le funzioni che determinano la prima e l'ultima posizione in una lista data, cioè `primolista` e `ultimolista`, che ricevono una lista e restituiscono la posizione desiderata in essa. Sempre riferendosi alla Figura 7.10, siccome la sentinella precede il primo elemento e segue l'ultimo, l'indirizzo del primo elemento è contenuto nel campo `succ` della sentinella e quello dell'ultimo nel campo `pred` della sentinella.

```
posizione primolista (listalibri L)
{
  return L->succ;
}

posizione ultimolista (listalibri L)
{
  return L->pred;
}
```

L'accesso quasi diretto all'ultimo elemento è uno dei vantaggi delle liste circolari. Se la lista fosse lineare, per raggiungere l'ultimo elemento, bisognerebbe partire dal primo e scorrerli tutti, con un'operazione di complessità $\Theta(n)$, se n è il numero di elementi della lista. Ancora una volta, conviene meditare su come sarebbero implementate queste funzioni nel caso di liste senza sentinella (circolari o lineari, monodirezionali o bidirezionali). Discuteremo poco più avanti che cosa accade quando si cerca il primo o l'ultimo elemento di una lista vuota.

Le altre due funzioni che caratterizzano la struttura dati astratta lista erano l'accesso alla posizione seguente e precedente una posizione data, cioè le funzioni `succlista` e `predlista`. Facendo come sempre riferimento alla Figura 7.10[12], quando si conosce la posizione `p` di un elemento, per conoscere quelle del precedente e del successivo basta leggere e restituire il contenuto dei campi `pred` e `succ` di tale elemento:

[12] Che in realtà è tutt'altro che chiara per questo scopo, e andrebbe riadattata.

```
posizione succlista (listalibri L, posizione p)
{
  return p->succ;
}

posizione predlista (listalibri L, posizione p)
{
  return p->pred;
}
```

Qui viene spontanea un'osservazione: perché passare alle funzioni `succlista` e `predlista` sia la lista `L` sia la posizione `p`, quando basta quest'ultima per trovare il risultato? Il motivo è, ancora una volta, l'aderenza alla definizione astratta, che usa entrambi gli argomenti (il concetto di una posizione non legata ad una lista è privo di senso), e l'indipendenza dall'implementazione. Se, infatti, tutte le implementazioni a puntatori potrebbero evitare l'uso della lista, vedremo al termine del capitolo che le implementazioni a vettori e indici non possono prescindere dall'indicazione di entrambi, perché la posizione rappresentata come indice intero non è sufficiente a ricavare il risultato cercato. D'altra parte, è vero che passare due parametri anziché uno aumenta il consumo di memoria (il record di attivazione è più grande) e il tempo richiesto per la chiamata della procedura (vengono copiati due dati, di cui uno inutile, anziché uno solo). L'uso delle strutture dati astratte ha un costo, che va confrontato con il vantaggio in termini di chiarezza di scrittura (i puntatori scompaiono completamente) e flessibilità nella gestione del codice (vedremo al termine del capitolo che sarà possibile cambiare implementazione senza modificare il codice, sostituendo semplicemente una libreria con un'altra).

Anche in questo caso, rimandiamo per un istante la questione di che cosa accada quando si cerca l'elemento successivo all'ultimo o precedente il primo. E anche in questo caso, conviene meditare su come potrebbe essere implementata la funzione `predlista` qualora non vi fosse un campo puntatore `pred`: bisognerebbe scorrere la lista dalla testa per poter determinare l'elemento che ne precede uno dato.

Si noti che in tutti questi casi la posizione e la lista devono essere coerenti fra loro: se il puntatore `p` non indica un elemento della lista `L` queste funzioni sono del tutto scorrette, ma l'idea è che, usando solo le funzioni della libreria e usando posizioni e liste fra loro associate, non cadremo mai in casi del genere.

L'ultima procedura semplice è `finelista`, che determina se una data posizione descrive situazioni di errore per una lista data, cioè le situazioni in cui si cerca il primo o ultimo elemento di una lista vuota, l'elemento successivo all'ultimo e l'elemento precedente il primo. In questi casi, le funzioni dovrebbero fallire, ma è più conveniente imporre loro di restituire un valore convenzionale, che corrisponda a tale fallimento, in modo da poterlo immediatamente registrare, reagendo correttamente. La procedura `finelista` dovrebbe valutare una posizione `p` per una lista `L` e concludere che rappresenta una di queste situazioni di errore. Ora, in base al modo in cui abbiamo definito le quattro funzioni precedenti, si vede che la posizione `p` che restituiscono è sempre quella della sentinella. Ma allora il test risulta molto semplice.

```
boolean finelista (listalibri L, posizione p)
{
  return (p == L);
}
```

È chiaro che, se `p` fosse un indirizzo del tutto scorretto, esterno al "mondo" circoscritto della lista in questione, questa funzione non lo rileverebbe, ma se la lista viene gestita in maniera accorta e facendo uso solo delle funzioni di libreria, questo non dovrebbe mai verificarsi. È ovviamente utile chiedersi come realizzare questa funzione (e le quattro precedenti) nel caso in cui non ci fosse una sentinella: il candidato naturale è l'indirizzo degenere *NULL*, che si potrebbe ribattezzare `NO_ELEMENT`[13]

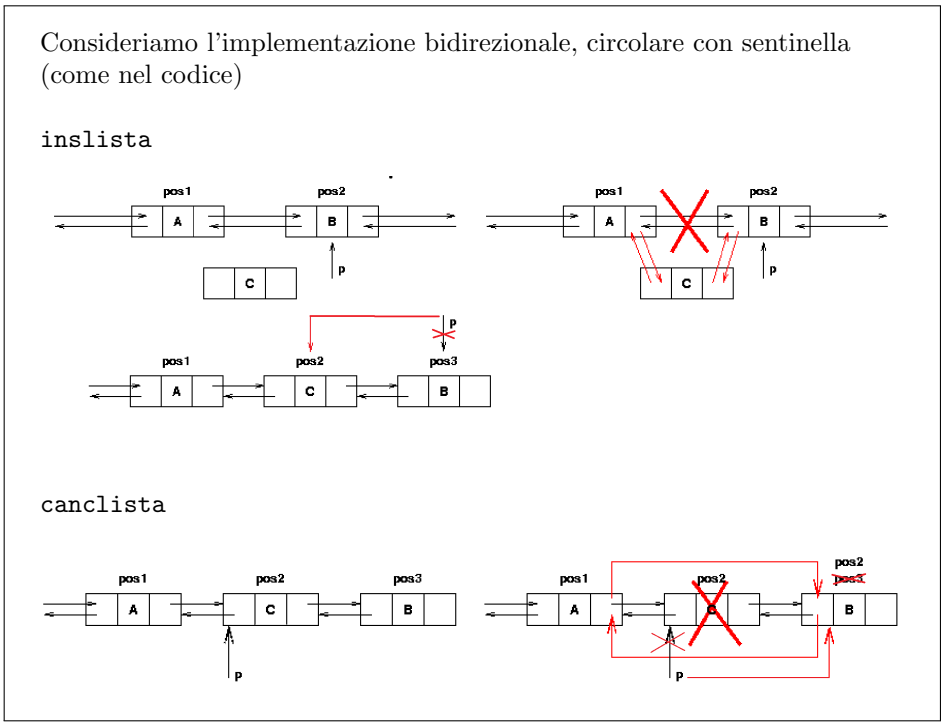


Figura 7.12: Inserimento e cancellazione

Consideriamo ora l'inserimento di un nuovo elemento in una lista. La parte superiore della Figura 7.12 descrive il meccanismo. In base alla definizione astratta, inserire un elemento in una posizione significa creare un nuovo elemento che va a occupare quella posizione e fa scalare di un passo in avanti l'elemento che occupava inizialmente la posizione stessa e tutti gli elementi successivi. La rappresentazione grafica è particolarmente utile per ordinare i passaggi dell'operazione evitando di perdere informazioni necessarie. Il primo passo consiste nell'osservare che la funzione `inslista` deve ricevere una lista, una posizione e un oggetto dell'insieme base, cioè un libro. Nella rappresentazione grafica, invece, i dati sono una lista, una posizione e un elemento, cioè una struttura contenente un libro e due campi puntatore. Questo significa che prima di inserire l'elemento bisogna allocarlo, esattamente come si è allocata la sentinella nel creare la lista. Nel far ciò, automaticamente si riceve un indirizzo di memoria `q`, che possiamo considerare come la "posizione" del nuovo elemento C. Il campo informazione viene riempito copiando l'argomento corrispondente.

Poi bisogna aggiornare i puntatori del nuovo elemento e di quelli fra i quali esso andrà inserito. Nel farlo, bisogna evitare di perdere l'accesso alla parte finale della

[13] Controllare: mi viene il dubbio che con l'implementazione data la costante simbolica `NO_ELEMENT` sia in effetti del tutto inutilizzata.

lista. Fortunatamente, l'indirizzo `p` è sufficiente a evitare questo rischio, perché punta la testa della parte finale della lista. Possiamo agganciare il nuovo elemento ai vecchi assegnando opportunamente i suoi campi `pred` e `succ`: il secondo coincide banalmente con `p`, mentre il primo riceve il valore `p->pred` (e qui viene buona la bidirezionalità della lista). Si possono poi rompere i vecchi collegamenti per far sì che il nuovo elemento sia effettivamente parte della catena, e non semplicemente appeso ad essa. L'elemento A (che è `p->pred`) deve avere come successivo il nuovo elemento C, anziché B (dunque `p->pred->succ = q`), e l'elemento B (che è `p`) deve avere come precedente il nuovo elemento C, anziché A (dunque `p->pred = q`). In tutto questo marchingegno, l'unico aspetto critico è che l'operazione che modifica `p->pred` sia l'ultima, perché altrimenti le due operazioni precedenti che fanno uso del valore `p->pred` sarebbero scorrette. La freccia che da B risale verso A deve spostarsi verso C solo all'ultimo momento. Da questo punto di vista, la rappresentazione grafica è estremamente utile.

```
listalibri inslista (listalibri L, posizione p, libro AutoreTitolo)
{
  posizione q;

  q = (posizione) malloc(sizeof(elemento));
  if (q == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione di un elemento!\n");
    exit(EXIT_FAILURE);
  }
  strcpy(q->AutoreTitolo, AutoreTitolo);
  q->pred = p->pred;
  q->succ = p;

  p->pred->succ = q;
  p->pred = q;  /* NB: p->pred va aggiornata per ultima, o le
      istruzioni precedenti sono scorrette! */

  return L;
}
```

Si noti che la funzione restituisce la lista stessa come risultato finale. Ancora una volta, questo non è rigorosamente necessario, perché la lista viene comunque modificata, ma ci mantiene vicini alla definizione astratta. Inoltre, se la lista non avesse una sentinella e si volesse inserire un elemento in testa, la funzione `inslista` dovrebbe modificare la lista stessa, e questo non sarebbe possibile nel modo in cui l'abbiamo dichiarata (con un parametro passato per valore). Quindi, è piuttosto importante, se vogliamo mantenere la struttura dati astratta, che il risultato di `inslista` venga assegnato alla variabile che descrive la lista stessa: per le liste con sentinella, non cambia nulla, ma per quelle senza sentinella il codice risulterebbe scorretto.

La rappresentazione grafica nasconde però un aspetto sottile della definizione astratta. Questo aspetto è il fatto che un puntatore, ovvero indirizzo, in realtà non è esattamente la stessa cosa di una posizione. Se consieriamo l'inserimento appena eseguito, infatti, il puntatore `p` che indicava la posizione iniziale dell'elemento C, al termine dell'operazione indica la posizione finale dello stesso elemento. Questo non è ciò che ci si aspetterebbe da una "posizione" astratta in una lista: nell'esempio, la posizione inizialmente puntata da `p` è la seconda della lista, ma dopo che l'elemento C viene inserito in tale posizione, al termine della procedura, la posizione `p` è diventata la terza della lista. Volendo essere precisi, la funzione `inslista` dovrebbe anche aggiornare il puntatore `p` assegnandogli il valore di `q`. Quello che

è automatico per un oggetto matematico, non lo è però per un indirizzo fisico, il contenuto di una cella, che non cambia se il processore non lo modifica. D'altra parte, modificare il valore di `p`, come indicato in figura, richiederebbe di passarlo per indirizzo. Una simile scelta avrebbe perfettamente senso, ma da un punto di vista pratico il vantaggio di aggiornare la posizione sarebbe piuttosto ridotto. D'altra parte, ci sarebbe un netto svantaggio pratico. Molto spesso l'inserimento in lista non viene fatto in una posizione indicata esplicitamente da una variabile, ma in una posizione calcolata come risultato di una funzione. In altre parole, nella chiamata a `inslista` si usa come parametro una chiamata a funzione (per esempio, `primolista`, per porre l'elemento in testa). Tuttavia, il risultato di una funzione non si può passare per indirizzo, perch'e non ha un indirizzo stabile, dato che risiede in una cella del record di attivazione, che viene deallocato al termine della chiamata. Per amor di semplicità, quindi, non adottiamo la scelta più precisa. Purtroppo, nel caso della cancellazione ci sono ottimi motivi pratici, che discuteremo in seguito, per adottare la strategia opposta, cioè per aggiornare il puntatore. Insomma, la questione è sottile, ogni scelta questionabile, e non riesco ad adottare una posizione del tutto soddisfacente.

Prima di passare alla cancellazione, si valuti che cosa cambierebbe se la lista non fosse bidirezionale: per conoscere `p->pred` occorrerebbe ripartire dalla testa della lista per ricostruire l'indirizzo corretto. È vero che non sarebbe necessario assegnare il suo valore a `q->pred`, ma sarebbe necessario conoscerlo per aggiornare `p->pred->succ`. Se invece non ci fosse una sentinella o se la lista non fosse circolare, le operazioni che abbiamo semplicemente descritto presenterebbero molti casi particolari, dato che abbiamo considerato scontata l'esistenza di un elemento precedente e un elemento successivo, ciascuno dei quali potrebbe venire a mancare. L'uso della lista circolare con sentinella serve proprio a evitare di dover elencare uno per uno tutti questi casi particolari e di dover identificare in quale caso ci si trovi. C'è quindi un vantaggio in termini di tempo.

Il meccanismo della cancellazione è descritto nella parte inferiore della Figura 7.12. Per cancellare l'elemento in posizione `p`, bisogna fare in modo che il precedente punti il successivo e il successivo punti il precedente. Fatto ciò, si può deallocare l'elemento C (non occorre neppure rompere i suoi puntatori), senza correre il rischio di perdere spezzoni della lista. Torniamo al problema dell'aggiornamento di `p`: dobbiamo decidere se aggiornare tale indirizzo o non preoccuparcene. Da un punto di vista matematico, la posizione andrebbe aggiornata, assegnando a `p` l'indirizzo dell'elemento B, che è risalito a occupare la posizione cancellata. Questo, però, comporta di passarla per indirizzo e di eseguire operazioni apposite. Nel caso dell'inserimento, abbiamo optato per non farlo. In questo caso, faremo il contrario, per due motivi. Anzi tutto, avere un puntatore a un indirizzo deallocato è sempre pericoloso, perché ci si può dimenticare che lo sia e cercare di usarlo, producendo comportamenti impredicibili (di solito, l'interruzione del programma). Inoltre, la cancellazione viene spesso utilizzata all'interno di cicli (per esempio, come vedremo, per distruggere una lista), usando una posizione come indice del ciclo. In questo caso, avere una posizione che punta celle di memoria deallocate porta sicuramente a problemi al termine dell'iterazione, quando si aggiorna la posizione o se ne testa il valore.

La ricomposizione dei puntatori da elemento precedente a successivo e da successivo a precedente richiede di essere precisi nell'ordine delle operazioni, per evitare di cancellare informazioni necessarie.

Inoltre, è opportuno ricordarse che non stiamo lavorando con la posizione `p`, ma con un puntatore ad essa, indicato con `pp`. Per semplificare un pochino questa complicazione, possiamo ricavare la posizione `*pp`, dereferenziando il puntatore, e attribuirla a una variabile locale `p`. Questo serve a evitare di usare costrutti

complicati. Una volta che abbiamo la copia della posizione desiderata, possiamo anche aggiornare l'originale, spostando la posizione sull'elemento successivo, che ne prende il posto. Ciò fatto, basta deallocare l'elemento puntato da `p`.

```
listalibri canclista (listalibri L, posizione *pp)
{
  posizione p;

  p = *pp;
  *pp = p->succ;

  p->pred->succ = p->succ;
  p->succ->pred = p->pred;

  free(p);

  return L;
}
```

Come sopra, la funzione `canclista` restituisce al termine la lista, ed è buona norma assegnarla alla variabile che descrive la lista nella funzione chiamante perché altrimenti in alcune implementazioni la funzione risulterebbe scorretta (precisamente, nelle implementazioni senza sentinella, cancellando l'elemento in cima alla lista, la lista stessa non risulterebbe aggiornata correttamente).

Anche in questo caso, è utile meditare sull'implementazione di questa funzione con liste monodirezionali (che richiederebbero la determinazione dell'elemento precedente, in tempo lineare), lineari o senza sentinella (che comporterebbero una serie di casi particolari, legati all'esistenza o meno di un elemento precedente e di uno successivo).

L'ultima operazione da realizzare è la distruzione di una lista. Per eseguirla, bisogna anzi tutto cancellare uno per uno tutti i suoi elementi, in un ordine qualsiasi. Il modo più comodo è cancellare ripetutamente il primo elemento finché la lista non è vuota. Questo si può fare sfruttando le operazioni già realizzate. Quando la lista è vuota, rimane ancora una sentinella, che va anch'essa deallocata. Infine, il puntatore `L`, che ancora punta l'area precedentemente occupata dalla sentinella, deve diventare pari a `NO_LIST`, ovvero `NULL`. Siccome la lista viene passata per indirizzo alla funzione, l'argomento `pL` è un puntatore alla lista, e quindi la lista stessa è `*pL` e la sua prima posizione si recupera con la funzione `primolista` applicata a `*pL`. Per cancellarla, non si può passare direttamente alla funzione `canclista` la chiamata a `primolista`, perché è richiesto un puntatore a una posizione. Quindi bisogna raccogliere questa posizione in una variabile locale `p`, spezzando la determinazione del primo elemento della lista dalla chiamata di `canclista` sull'indirizzo di questo elemento. Uno dei motivi per cui si è lasciato il passaggio per valore della posizione a `inslista`, perdendone l'aggiornamento, è di poter scrivere chiamate a funzione direttamente nella lista degli argomenti. Solo ora possiamo procedere a cancellare la posizione `p`, passata per indirizzo, recuperando la lista stessa e assegnandola alla variabile `*pL`. Può sembrare barocco passare la lista l come dato e recuperarla come risultato anche se la lista dato è stata modificata, ma se vogliamo una totale indipendenza dall'implementazione dobbiamo procedere in questo modo. L'alternativa sarebbe passare anche la lista per indirizzo. Deallocare la sentinella è molto semplice, perché la sentinella è l'oggetto puntato da `L`, cioè `*pL`. Al termine, `*pL` può essere assegnato a `NULL`.

```
void distruggelista (listalibri *pL)
{
```

```
  posizione p;

  while (!listavuota(*pL))
  {
    p = primolista(*pL);
    canclista(*pL,&p);
  }

  free(*pL);
  *pL = NULL;
}
```

Tutto questo *tour de force* è ancora privo di risultati visibili, perché non abbiamo ancora caricato libri nelle liste e non siamo in grado di stamparle. Però abbiamo una libreria funzionante per gestire le liste e siamo a un passo dal poter ottenere risultati, e trovare almeno alcuni degli errori eventualmente commessi nel frattempo. A questo dedicheremo le prossime sezioni.

Ricordiamo ancora una volta che tutta questa sezione, con i suoi problemi tecnici di dettaglio non ha alcun legame con la relazione del progetto, dato che in quella sede si tratta solo di giustificare la scelta della struttura dati lista e dell'implementazione a puntatori, bidirezionale, circolare e con sentinella in base alle operazioni richieste e alla memoria disponibile e presumibilmente richiesta dai dati. I dettagli tecnici si discutono in un testo di programmazione o di algoritmi, o in una lezione di ripasso di programmazione e di studio di algoritmi.

7.4.7 Terza fase: caricamento e stampa dei libri

Torniamo ora all'approccio *top-down*, sviluppando alcune delle funzioni che risolvono i sottoproblemi ad alto livello individuati nella Sezione **??** Il risultato sarà il codice riportato nel file listato `biblio2.c`.

In particolare, realizzeremo la procedura `CaricaLibri` e la procedura `StampaLibri`, in modo da poter cominciare ad avere risultati verificabili, cioè una relazione fra dati e risultati che non è ancora quella desiderata, ma comincia a consentire di valutare e risolvere eventuali errori, prima che si accumulino e comincino ad interagire fra loro in modi complessi. Questo è un buon metodo di lavoro.

La procedura `CaricaLibri` apre il file dei libri e ne inserisce il contenuto nella lista `Scaffale` in ordine alfabetico. Il file (si veda l'esempio `input01a.txt`) contiene riga per riga autore e titolo di un libro, che formano un'informazione unica. La procedura deve leggere una riga alla volta finché trova una riga costituita solo della parola chiave `FINE`, la quale indica che il file termina. Diamo per scontato che il formato sia rispettato, per evitare complicazioni che potrebbero facilmente diventare sofisticatissime, dato che non ci sono limiti al modo in cui un file di dati può essere sbagliato. Al limite, ammetteremo che un file sia privo di libri e contenga solo la parola chiave `FINE`. Questo è un principio che seguiremo durante l'intero corso: assumere che il formato dei dati corrisponda a quanto indicato nel testo. D'altra parte, non assumeremo che il formato dei dati coincida con quello dell'esempio oltre ciò che è riportato nel testo. Per esempio, non faremo mai assunzioni dettagliate sul numero di caratteri che separano le informazioni, sulla presenza o assenza di righe vuote alla fine del file e cose del genere[14] Quindi la funzione legge una riga alla volta e verifica se la sua prima parola (o l'intera riga) è la parola chiave di termine.

[14]Per capirsi, siccome il testo del progetto non specificherà mai i dati con assoluta precisione, qualsiasi procedura di caricamento dei dati che proceda un carattere alla volta è votata a fallire: bisogna procedere un'unità informativa alla volta. Nel caso di questo esercizio, una riga per volta.

In quel caso, interrompe il ciclo di lettura. In caso contrario, tratta la riga come l'informazione associata a un libro nuovo (autore e titolo) e inserisce questo libro nella lista `Scaffale`. Avendo aperto il file, al termine dovremo chiuderlo.

La procedura `StampaLibri` riceve una lista di libri, la scorre elemento per elemento, legge il libro nella posizione considerata e lo stampa a video, fermandosi quando la lista è terminata. La libreria `listalibri.h` fornisce tutto quel che serve alla bisogna.

Passiamo alla realizzazione commentata di `CaricaLibri`[15].

```
/* Carica un elenco di libri dal file file_libri */
listalibri CaricaLibri (char *file_libri)
{
  FILE *fp;
  boolean fine;
  listalibri Scaffale;
  char Riga[ROW_LENGTH];
  posizione p;


  /* Apre il file file_libri */
  fp = fopen(file_libri,"r");
  if (fp == NULL)
  {
    fprintf(stderr,"Errore nell'apertura del file %s!\n",file_libri);
    exit(EXIT_FAILURE);
  }

  /* Crea una lista Scaffale vuota */
  Scaffale = crealista();

  do {
    /* Legge una riga saltando l'a capo finale */
    fscanf(fp,"%[^\n]\n",Riga);

    /* Se la riga e' il comando "FINE", esce dal ciclo */
    fine = (strcmp(Riga,"FINE") == 0);
    /* altrimenti, inserisce il libro nella lista Scaffale in coda,
       dopo l'ultimo (cioe' prima del successore dell'ultimo) */
    if (fine == FALSE)
    {
      p = succlista(Scaffale,ultimolista(Scaffale));
      inslista(Scaffale,p,Riga);
    }

  } while (fine == FALSE);

  fclose(fp);

  return Scaffale;
}
```

Per aprire il file dei libri, occorre una variabile puntatore a file (`FILE *`), che per semplicità chiameremo `fp`, anche se potremmo distinguere i due file coinvolti nell'esercizio (però è ovvio quale sia il file usato in questa funzione). La variabile riceve il risultato della funzione `fopen`, alla quale si fornisce il nome del file dei libri e l'indicazione che il file andrà aperto in lettura (`"r"`).

[15] Quanto segue è una descrizione commentata del codice, e ha come obiettivo di descrivere passo per passo l'implementazione e di ripassare la sintassi del linguaggio C. Nello stendere una relazione diretta a un lettore che sa programmare e vuol conoscere solo il risultato finale del progetto, ci si guardi bene dal procedere in questo modo pedante e dettagliato: che i file vadano aperti e chiusi è ovvio; così pure quali siano i nomi delle istruzioni da usare, e così via.

Dopodiché, è buona norma andare a controllare che i file sia effettivamente stato aperto, per evitare che le operazioni seguenti diano risultati irragionevoli. In caso di errore, dovremmo segnalarlo all'esterno, magari specificando il nome del file (potrebbe essere sbagliato, e in tal modo chiarire all'utente dove stia l'errore). Infine, potremmo abbandonare il programma, segnalando al sistema operativo che qualcosa è andato male. In caso contrario, possiamo proseguire. Conviene subito passare al termine della funzione e chiudere il file (`fclose`), per non dimenticarlo. Quando ci sono coppie di operazioni complementari, che si svolgono al principio e al termine di un programma, o di una procedura, scriverle immediatamente entrambe riduce il numero di cose da tenere a mente e consente di concentrarsi sugli aspetti più "interni" e importanti. Per lo stesso motivo, ci occupiamo subito di dichiarare, inizializzare e restituire il risultato della funzione, cioè di aggiungere le istruzioni `listalibri Scaffale`, `Scaffale = crealista()` e `return Scaffale`. In questo modo eliminiamo il messaggio di avvertimento di cui avevamo parlato in precedenza. La procedura non è ancora completa, ma manipola già oggetti del tipo richiesto.

La lettura richiederà certamente un ciclo. I tre tipi di ciclo (conteggio, condizione iniziale e condizione finale) sono in realtà trasformabili l'uno nell'altro, ma uno sembra più naturale. Non sarà un ciclo a conteggio, perché non sappiamo quanti libri contiene il file. Siccome il file contiene certamente qualcosa (al limite, la parola chiave `FINE`, nel caso di biblioteca vuota), il ciclo sarà a condizione finale, perché almeno una riga verrà letta. La condizione di permanenza non è ovvia a priori, per cui cominciamo a definire una variabile logica `fine` (in gergo *flag*), per cui il ciclo proseguirà fin tanto che `fine == FALSE`[16]. La condizione può essere inizializzata, ponendo `fine = FALSE` prima del ciclo, oppure riassegnata ad ogni iterazione (come faremo in questo caso, dato che è semplice farlo).

Il corpo del ciclo consiste nella lettura di una riga e nella sua interpretazione. Procediamo scrivendo commenti, che chiariscano a noi stessi che cosa vogliamo fare, e poi trasformandoli in codice. Per ogni riga del file, bisogna caricarla in una stringa, per conservare la quale ovviamente occorre dichiarare una variabile. Il testo riporta una lunghezza massima per i libri, che potremmo usare, aggiungendo ovviamente il terminatore. Considerando che poi lavoreremo anche sul file dei movimenti, le cui righe sono più lunghe, perché contengono libri e istruzioni varie, potremmo definire due costanti *ad hoc*. Qui per semplicità abbiamo usato una sola lunghezza (la stessa `ROW_LENGTH` già usata in precedenza), che le eccede entrambe. È un piccolo spreco di memoria, decisamente trascurabile.

Per leggere una riga esiste la funzione `fgets`, che già conosciamo. Per cambiare, usiamo un altro costrutto quasi equivalente, che consiste nel riconoscere, attraverso la funzione `fscanf` una riga, definendola come sequenza di opportuni caratteri (con la specifica `%` seguita dalle parentesi quadre), specificando non direttamente i caratteri accettabili, bensì quelli non accettabili (l'a capo, `\n`) preceduti da `^`. Questo costrutto ha due differenze rispetto a `fgets`: come tutte le specifiche di formato, salta i caratteri separatori iniziali[17] (non quelli eventualmente presenti all'interno dopo i primi caratteri riconosciuti) e non include nella stringa acquisita l'a capo terminale. Entrambe le differenze possono essere desiderabili, dato che i separatori iniziali e l'a capo finale non fanno parte della stringa che descrive un libro. Ci si potrebbe chiedere che cosa succeda all'a capo che non viene letto. Esso rimane accodato nello *stream* d'ingresso, ma non creerà problemi alla prossima lettura perché verrà saltato insieme agli altri caratteri separatori iniziali.

Una volta letta la riga, bisogna determinare se essa si riduca alla parola chiave

[16] In C la condizione finale dei cicli è di permanenza, cioè quando la condizione è soddisfatta si rimane nel ciclo; in altri linguaggi esistono costrutti `repeat ... until` in cui la condizione invece è di terminazione, cioè quando la condizione è soddisfatta si esce dal ciclo.

[17] Questo va verificato: mi è venuto il dubbio che non sia così, anche se la logica lo suggerisce.

`FINE` o sia l'identificatore di un libro. Siccome escludiamo che possa esserci un terzo caso, per farlo, basterà confrontare la riga con la parola chiave (`strcmp`) e, nel caso coincidano, modificare il *flag* per terminare il ciclo. Nell'esempio, si è voluto farlo assegnando direttamente una condizione logica alla variabile. È questo che rende semplice rivalutare la condizione di terminazione ad ogni iterazione, anziché (come spesso si fa) inizializzarla prima del ciclo e modificarla solo quando il ciclo va terminato.

Se la riga letta è un'indicazione di terminare, usciamo dal ciclo. Se invece la riga corrisponde a un libro, bisogna definire in quale posizione inserirlo nella lista `Scaffale`. Il testo indica di accodare via via i libri, dato che il file li riporta nel loro ordine sullo scaffale. Quindi la posizione corretta è in fondo alla lista. Con questo non si intende l'ultima posizione occupata, dato che questo spingerebbe in avanti l'attuale ultimo elemento, inserendo il nuovo al penultimo posto. Si intende invece la posizione dopo l'ultima, che non è un concetto del tutto ovvio. Separiamo i vari termini della questione per maggiore chiarezza. Dichiariamo una variabile `posizione p` per conservare questa posizione e separiamo la ricerca della posizione corretta dall'inserimento. La seconda operazione è semplice: la procedura `inslista` riceve una lista di libri, una posizione e un libro e restituisce la lista aggiornata, che assegniamo alla variabile `Scaffale`. Ennesima tecnicalità: la lista viene aggiornata comunque ("distruggendo" la precedente, da un punto di vista logico, non di allocazione di memoria), per cui non è necessario assegnare la nuova lista alla vecchia variabile. Però, volendo adeguarsi al concetto di struttura dati astratta, questa è la scrittura più aderente. Tornando alla determinazione della posizione, la posizione che segue l'ultima non è direttamente accessibile con le funzioni della libreria, ma per ottenerla basta applicare `succlista` al risultato di `ultimolista`. Fisicamente, questo significa puntare la sentinella, il che rende il compito della funzione `inslista` molto semplice. Ne risulta un'espressione abbastanza barocca da suggerire che possa essere utile aggiungere alla libreria una funzione `accodalista` per inserire un nuovo elemento in coda a una lista chiamando una sola funzione anziché tre. Ciò fatto si possono anche combinare le due istruzioni in una sola, spostando le chiamate a `succlista` e `ultimolista` come argomenti di `inslista`. Come già discusso, questo sarebbe impossibile se avessimo adottato il passaggio per indirizzo della posizione, come nel caso di `canclista`, perché un programma C non è in grado di gestire l'indirizzo del risultato di una funzione in modo che sopravviva alla terminazione della funzione stessa. Per lo stesso motivo, dopo l'inserimento la variabile `p` continua a puntare la sentinella, mentre in teoria dovrebbe puntare il nuovo elemento aggiunto. Siccome ad ogni iterazione `p` viene ricalcolata, questo non ci tocca.

Passiamo ora alla procedura `StampaLibri`, che si riduce a un classico ciclo di scorrimento.

```
/* Stampa l'elenco dei libri disponibili */
void StampaLibri (listalibri L)
{
  posizione p;
  libro AutoreTitolo;

  for (p = primolista(L); !finelista(L,p); p = succlista(L,p))
  {
    leggelista(L,p,AutoreTitolo);
    printf("%s\n",AutoreTitolo);
  }
  printf("FINE\n");
}
```

Occorre dichiarare una posizione generica `p`, che parte dalla prima posizione della lista (`primolista(Scaffale)` e prosegue di passo in passo fino a sforare il termine della lista, cioè finché il risultato di `finelista(Scaffale,p)` diventa vero. Ad ogni iterazione, si aggiorna `p` sostituendole la posizione successiva `succlista(Scaffale,p)`. Nello scrivere queste chiamate, conviene sempre tenere a fianco le loro dichiarazioni nel file `listalibri.h` per controllare che l'elenco dei parametri sia corretto[18] Nel corpo del ciclo, la funzione `leggelista` ci fornisce, data la lista e la posizione, il libro corrente, da caricare in una opportuna stringa di caratteri `AutoreTitolo`, dichiarata al principio della funzione. Quindi, possiamo stamparla a video, aggiungendo l'a capo che non abbiamo caricato nella lettura.

Si può osservare che questa procedura di stampa non è molto efficiente, dato che la lettura copia la stringa in un'altra, che viene poi stampata. Per renderla efficiente si potrebbe aggiungere alla libreria una funzione `stampalista` che essendo interna alla libreria stessa, avrebbe il permesso di accedere direttamente al dato.

A questo punto, compilare ed eseguire il codice aiuta a verificare se tutto è stato fatto correttamente. Nella situazione attuale, il codice si limita a stampare due volte la lista dei libri presenti nel file dei libri. Questo perché viene stampata la lista iniziale, l'esecuzione dei movimenti è ancora vuota e viene stampata la lista finale, che coincide con quella iniziale.

7.4.8 Quarta fase: esecuzione dei movimenti

Nella fase successiva realizziamo la funzione `EsegueMovimenti`, che riceve il file dei movimenti e modifica corripondentemente le tre liste (`Scaffale`, `Prestiti` e `Resi`). Ricordiamo che si immagina che l'esecuzione avvenga in tempo reale durante la giornata, via via che nuovi movimenti vengono comunicati e nuove righe si aggiungono in fondo al relativo file. Il risultato di questa fase sarà il file `biblio3.c`.

Le tre liste sono sia dati sia risultati della procedura, dato che vengono modificate via via. Per tale motivo, le liste vengono passate per indirizzo. Per questo motivo, gli argomenti non sono liste, ma puntatori a lista; per ricordarlo meglio, abbiamo anche cambiato i nomi delle tre variabili, premettendo a ciascuno una `p`, per sottolineare che sono puntatori alle vere variabili, le quali risiedono nello *scope* del `main`.

```
/* Esegue i movimenti riportati nel file dei prestiti e dei resi
   sulle liste Scaffale, Prestiti e Resi */
void EsegueMovimenti (char *file_prestiti, listalibri *pScaffale,
                      listalibri *pPrestiti, listalibri *pResi);
```

Anche qui, procediamo descrivendo per punti la procedura sotto forma di commenti. Per prima cosa, apriremo il file dei movimenti (con il solito controllo di che l'operazione abbia avuto successo), e ovviamente lo chiuderemo al termine della procedura. Fra le due cose, avremo un altro ciclo che lavora riga per riga finché non trova la parola chiave `FINE`. Se non lo è, interpreteremo la riga come un comando, che potrà essere una restituzione o un prestito. La procedura è molto simile alla precedente, salvo che i comandi possono essere due e vanno distinti. Potremmo procedere come nel caricamento, ma il fatto che ogni riga comincia con una parola chiave suggerisce di leggere separatamente tale parola e l'eventuale resto della riga, anziché la riga intera. Questo consente di usare un ciclo a condizione iniziale che confronta direttamente il comando con `FINE` senza usare *flag* e nel corpo del ciclo lo

[18] Uno dei problemi più grossi, e una delle fonti di errore più insidiose, sta nel fatto che liste e posizioni sono entrambi puntatori, e quindi invertire l'ordine degli argomenti non viene riconosciuto come errore dal compilatore, e neppure genera avvertimenti.

confronta con `PRESTITO` e `RESTITUZIONE` per decidere che cosa fare. A questo punto, il prestito richiederà di cercare un determinato libro (che è la parte residua della riga) nella lista `Scaffale`, toglierlo da lì e metterlo nella lista `Prestiti`. Nel caso in cui il libro non sia nella lista, segnaleremo l'errore e usciremo[19] Dove inseriamo il nuovo libro nella lista dei prestiti? Il testo non lo specifica: mentre la lista dello scaffare è ordinata alfabeticamente e quella dei resi è ordinata cronologicamente in senso inverso (dal reso più recente al più vecchio), la lista dei prestiti non è ordinata. Quindi possiamo mettere il libro dove vogliamo. Se invece il comando è una restituzione, dobbiamo cercare il libro nella lista dei prestiti. Che fare se non lo troviamo? Potremmo segnalare un errore e terminare, ma il fatto è che i libri resi saranno in generale stati prestati in giorni precedenti, e quindi non compariranno nella lista dei prestiti. Quindi qui il trattamento del caso in cui la ricerca fallisce è diverso: accetteremo tutti i resi, senza protestare, ma cercheremo il libro fra i prestiti per cancellarlo, se lo troviamo. In tutti i casi, lo inseriremo nella lista `Resi` in cima, come indicato nel testo. Fatto tutto ciò, possiamo leggere il comando successivo. Sulla base di questa traccia, realizziamo il seguente codice.

```
/* Esegue i movimenti riportati nel file dei prestiti e dei resi
   sulle liste Scaffale, Prestiti e Resi */
void EsegueMovimenti (char *file_prestiti, listalibri *pScaffale,
                      listalibri *pPrestiti, listalibri *pResi)
{
  FILE *fp;
  char Riga[ROW_LENGTH];
  char Comando[ROW_LENGTH];
  char AutoreTitolo[ROW_LENGTH];
  posizione p;


  /* Apre il file dei movimenti */
  fp = fopen(file_prestiti,"r");
  if (fp == NULL)
  {
    fprintf(stderr,"Errore nell'apertura del file
        %s!\n",file_prestiti);
    exit(EXIT_FAILURE);
  }

  /* Legge la prima riga saltando l'a capo finale */
  fscanf(fp,"%[^\n]\n",Riga);

  /* Per ogni riga fino al comando FINE */
  while (strcmp(Riga,"FINE") != 0)
  {
    /* Separa il comando e il libro */
    sscanf(Riga,"%s %[^\n]",Comando,AutoreTitolo);

    /* Se il comando e' PRESTITO */
    if (strcmp(Comando,"PRESTITO") == 0)
    {
      /* Cerca il libro sullo Scaffale */
      p = TrovaLibro(AutoreTitolo,*pScaffale);

      /* Se non c'e', termina il programma */
      if (p == NO_ELEMENT)
      {
        fprintf(stderr,"Il libro %s non appartiene alla lista
            Scaffale!\n",AutoreTitolo);
        exit(EXIT_FAILURE);
      }
```

[19] Può succedere? Il testo non lo specifica, ma sembra pericoloso dare per scontato che il libro ci sia sempre e lasciare che il programma prosegua per conto suo, e non costa molto aggiungere il test e le istruzioni per terminare.

```
      /* Altrimenti sposta il libro da Scaffale a Prestiti */
      *pScaffale = canclista(*pScaffale,&p);
      p = succlista(*pPrestiti, ultimolista(*pPrestiti));
      *pPrestiti = inslista(*pPrestiti, p, AutoreTitolo);
    }
    /* Altrimenti, e' RESTITUZIONE */
    else /* if (strcmp(Comando,"RESTITUZIONE") == 0) */
    {
      /* Cerca il libro nei Prestiti */
      p = TrovaLibro(AutoreTitolo, *pPrestiti);

      /* Se lo trova lo cancella, altrimenti non fa nulla */
      if (p != NO_ELEMENT) canclista(*pPrestiti,&p);

      /* Inserisce il nuovo libro in cima ai Resi */
      p = primolista(*pResi);
      *pResi = inslista(*pResi, p, AutoreTitolo);
    }

    /* Legge la riga seguente */
    fscanf(fp,"%[^\n]\n",Riga);
  }

  /* Chiude il file dei movimenti */
  fclose(fp);
}
```

Come anticipato, leggeremo il comando con la funzione `fscanf` e la metteremo in una stringa `Comando` di lunghezza `ROW_LENGTH` (solo per comodità: sarà molto sovradimensionata). La condizione di permanenza del ciclo verifica che `Comando` differisca dalla parola chiave `FINE`, cioè che il risultato di `strcmp` non sia nullo. Nel ciclo, confrontiamo `Comando` con le due parole chiave che corrispondono alle operazioni. Siccome in entrambi i casi si opera sul libro contenuto nella parte rimanente della riga[20], si recupera anche questo in una stringa `AutoreTitolo`, ancora con la funzione `fscanf` e la specifica `%[^n]`. Questa ha il vantaggio di saltare i separatori compresi fra il comando e i veri e propri dati del libro e di non includere l'a capo finale.

Proseguiamo distinguendo tra prestito e restituzione con il confronto fra `Comando` e le due stringhe possibili. Qualora il risultato di un confronto sia nullo, si eseguono le relative operazioni. Se entrambi falliscono, segnaliamo l'errore e terminiamo[21]. I due casi richiedono operazioni diverse, ma simili. In entrambi, si cerca il libro in una lista. Questo non è un compito elementare e non ci sono funzioni pronte per farlo. Adottiamo la strategia *top-down*, supponendo di avere una funzione `TrovaLibro`, che, dato un libro e una lista, trova la posizione della lista in cui si trova il libro, cioè riceve il libro `Riga` e la lista `Scaffale` (scritta come `*pScaffale`, dato che `Scaffale` è passata per indirizzo, e in questa funzione non è accessibile direttamente), e restituisce una variabile `p` di tipo `posizione`. Se il libro non fa parte della lista, abbiamo bisogno di definire un valore restituito dalla funzione quando fallisce. La libreria fornisce una costante simbolica `NO_ELEMENT` che rappresenta proprio una posizione insensata, che adottiamo come valore restituito in quel caso. Quando `p` ha quel valore, l'algoritmo termina con un messaggio di errore. Altrimenti (ma non occorre il costrutto `else`, dato che nel caso precedente si abbandona del tutto il programma), spostiamo il libro da una lista all'altra.

[20] Qui lo scopo è solo di non duplicare codice per eleganza: non è una questione di efficienza.

[21] Nell'ultima versione, assumiamo che, tolto il prestito, si tratti sempre di restituzione: questo espone a errori, ma semplifica il codice.

Come si sposta un elemento da una lista a un'altra? La libreria non offre procedure per farlo. O ne costruiamo una noi o sfruttiamo le funzioni esistenti (che sono state concepite per essere sufficienti a una trattazione base delle liste). Per spostare un elemento da una lista a un'altra si può leggerlo, cancellarlo dalla posizione corrente nella lista corrente e inserirlo nella nuova posizione nella nuova lista. Quindi, nel caso specifico, applichiamo `canclista` alla lista `*pScaffale` e alla posizione `p` (passata per indirizzo) e assegniamo il risultato alla lista `*pScaffale` (come già detto, questo non è necessario: serve solo a restare aderenti alla definizione astratta). Poi applichiamo `inslista` al nuovo libro `AutoreTitolo`, alla lista `*pPrestiti` e alla combinazione delle operazioni `succlista` e `ultimolista`. Ancora una volta, assegniamo il risultato a `*pPrestiti`. Il codice riutilizza la variabile `p` per conservare questa posizione: si può considerare un uso un po' sporco, ma non crea realmente problemi, dato che viene usata ogni volta subito dopo essere stata reinizializzata.

Ovviamente, la funzione `TrovaLibro` va anche dichiarata e definita, per cominciare ancora con un corpo vuoto (e non essendovi un valore di ritorno otterremo comunque un messaggio di avvertimento in compilazione).[22] Questa funzione riceve un libro e una lista e restituisce la posizione del libro nella lista (eventualmente `NO_ELEMENT`, se il libro non appartiene alla lista).

```
/* Cerca il libro AutoreTitolo nella lista L e ne restituisce la
     posizione
   (NO_ELEMENT se il libro non e' nella lista) */
posizione TrovaLibro (libro AutoreTitolo, listalibri L);
```

Il secondo caso è molto simile: bisogna trovare il libro nella lista `*pPrestiti`. Se lo troviamo, lo cancelliamo. In ogni caso, lo inseriamo in cima alla lista `*pResi`. Questo è semplice, dato che la posizione iniziale è semplicemente il risultato di `primolista`, conservata magari nella solita variabile `p`.

Per concludere, leggeremo la nuova riga.[23]

Notiamo che spostare un libro da una lista a un'altra applicando in sequenza una cancellazione e un inserimento è abbastanza inefficiente, dato che la deallocazione e l'allocazione di memoria dinamica potrebbero facilmente essere risparmiate se ci limitassimo ad aggiornare i puntatori dell'elemento che contiene il libro in modo da sganciarlo dalla prima lista e agganciarlo alla seconda. Questo è impossibile con la libreria nella sua forma base, ma potremmo aggiungere una funzione `sposta_lista` che esegua l'operazione nel modo più efficiente (si veda l'Esercizio X).

Eseguire il codice a questo punto è problematico, perché `TrovaLibro` non restituisce valori sensati, ma il codice reagisce in base al valore restituito (che è l'incontrollabile contenuto delle celle corrispondenti al risultato all'interno del record di attivazione). D'altra parte, restituire temporaneamente un valore convenzionale (per esempio, `NO_ELEMENT`) non risolve il problema, perché abbiamo stabilito

[22] Se si dimentica di definirla, il compilatore assume che sia una funzione che restituisce un intero, e quindi fornisce un avvertimento piuttosto criptico, secondo il quale si sta cercando di convertire un intero in un puntatore senza un'esplicita operazione di *cast*. È un avvertimento assurdo ma che il compilatore cerca di mettere in piedi sulla base di quello che gli abbiamo dato è di qualche ipotesi di fondo sbagliata.

[23] COMMENTO INTERESSANTE, CHE NON SO DOVE METTERE, RIGUARDO I BLOCCHI ALL'INTERNO DELLE FUNZIONI: Si noti che le parentesi graffe dei due casi del costrutto selettivo suggeriscono l'esistenza di livelli anche all'interno di una singola procedura, senza bisogno di chiamare sottoprocedure: l'intera struttura del codice è gerarchica. E quando un blocco compreso fra parentesi graffe diventa abbastanza lungo e importante, magari con variabili che vengono usate solo al suo interno (e che il linguaggio C consentirebbe di dichiarare all'inizio del blocco, come se si trattasse di una parte dichiarativa), viene naturale sostituirlo con una procedura, usando il blocco stesso come corpo.

che il codice termini immediatamente quando `TrovaLibro` restituisce questo valore. Quindi, prima di provare a eseguire il codice, realizziamo anche `TrovaLibro`.

La funzione è molto semplice e somiglia a `StampaLibri`: scorre la lista, ad ogni passo legge l'elemento corrente in una variabile `AutoreTitolo2` e, invece di stamparla, la confronta con il libro `AutoreTitolo` che stiamo cercando. Se coincidono, abbiamo la posizione corretta e semplicemente la restituiamo all'esterno. Se invece esauriamo la lista e arriviamo in fondo, restituiamo `NO_ELEMENT` a indicare che l'elemento cercato non appartiene alla lista. Anche questa funzione si candida naturalmente ad essere inclusa nella libreria di gestione della lista.

```
/* Cerca il libro AutoreTitolo nella lista L e ne restituisce la
    posizione
   (NO_ELEMENT se il libro non e' nella lista) */
posizione TrovaLibro (libro AutoreTitolo, listalibri L)
{
  posizione p;
  libro AutoreTitolo2;

  for (p = primolista(L); !finelista(L,p); p = succlista(L,p))
  {
    leggelista(L,p,AutoreTitolo2);
    if (strcmp(AutoreTitolo,AutoreTitolo2) == 0) return p;
  }

  return NO_ELEMENT;
}
```

Eseguendo il codice sull'esempio, otteniamo ancora una stampa di due liste, ma questa volta le liste sono composte da 4 libri, anziché da 10, precisamente dai 4 libri, dei 10 iniziali, che non sono andati in prestito. L'algoritmo, infatti, stampa l'elenco finale, dopo l'esecuzione dei movimenti, prima e dopo il riordino dei resi. Siccome però non abbiamo ancora eseguito il riordino dei resi, i due elenchi sono identici e considerano solo i prestiti e il fatto di impilare i resi sul bancone.

7.4.9 Quinta fase: riordino dei resi

Questa fase dell'esercizio realizza la procedura `RiordinoResi` che stampa ed esegue le istruzioni di riordino, cioè scorre la pila dei resi dalla testa e sposta via via i suoi elementi nel posto giusto dello scaffale, stampando in che posizione li mette, relativamente ai libri correnti. Il risultato è il file `biblio4.c`.

La procedura riceve la lista dei libri sullo scaffale e quella dei libri resi e restituisce le due liste aggiornate. Quindi, entrambe vengono passate per indirizzo.

```
/* Stampa ed esegue le istruzioni per riporre i libri resi nello
    scaffale */
void RiordinaResi (listalibri *pResi, listalibri *pScaffale);
```

Come al solito, procediamo a dividere il problema in sottoproblemi. In questo caso, non ci sono file da gestire. Avremo un altro ciclo, finché la lista dei resi non è vuota. Leggeremo il primo elemento e lo cancelleremo dalla lista. Poi determineremo quale sia la sua posizione corretta nella lista `Scaffale`, cioè quella del primo libro che lo segue in ordine alfabetico. Se non esiste un tal libro, cioè che si va fino in fondo alla lista, il libro va messo in posizione finale. Questo problema è molto simile alla ricerca di un libro in una lista. Quindi bisogna inserire il libro nella posizione

trovata e stampare il messaggio corretto. Al termine, stampa la solita parola chiave `FINE`.

```
/* Stampa ed esegue le istruzioni per riporre i libri resi nello
    scaffale */
void RiordinaResi (listalibri *pResi, listalibri *pScaffale)
{
  libro AutoreTitolo, AutoreTitolo2;
  posizione p;


  /* Finche' Resi non e' vuota */
  while (!listavuota(*pResi))
  {
    /* Legge il libro in cima alla lista Resi */
    p = primolista(*pResi);
    leggelista(*pResi,p,AutoreTitolo);

    /* Cancella il libro dai Resi */
    *pResi = canclista(*pResi,&p);

    /* Cerca la posizione corretta (primo libro successivo) per i
        libro AutoreTitolo nello Scaffale */
    for (p = primolista(*pScaffale); !finelista(*pScaffale,p); p =
        succlista(*pScaffale,p))
    {
      leggelista(*pScaffale,p,AutoreTitolo2);
      if (strcmp(AutoreTitolo,AutoreTitolo2) < 0) break;
    }

    /* Inserisce il libro AutoreTitolo nello Scaffale nella posizione
        corretta */
    inslista(*pScaffale,p,AutoreTitolo);

    /* Stampa l'istruzione corretta */
    if (finelista(*pScaffale,p))
      printf("METTERE %s PER ULTIMO\n",AutoreTitolo);
    else
      printf("METTERE %s PRIMA DI %s\n",AutoreTitolo,AutoreTitolo2);
  }
  printf("FINE\n");
}
```

Vediamo una possibile soluzione commentata in dettaglio. Per sapere se la lista `*pResi` è vuota o no, basta chiamare la funzione `listavuota`. Finché il risultato è falso, restiamo nel ciclo. Il corpo legge il primo libro, determinandone la posizione con `primolista` e assegnando il libro alla variabile `AutoreTitolo` con la funzione `leggelista`. Si potrebbero combinare le due chiamate in un'istruzione sola o dividerle usando una variabile ausiliaria `posizione p`. La seconda opzione è adatta al nostro caso perché vogliamo anche cancellare l'elemento dalla lista, e la funzione `canclista` richiede una posizione modificabile, dato che viene passata per indirizzo. Come sempre, eseguita una modifica di una lista, assegniamo il risultato alla lista stessa.

Ora cerchiamo la posizione corretta della stringa `AutoreTitolo` nella lista `*pScaffale`. Il problema è simile alla ricerca di un libro, ma ha soluzione diversa. Possiamo fare una funzione ad hoc oppure scrivere direttamente le istruzioni, che non sono molte. Avremo un ciclo di scorrimento, interrotto non quando il libro corrente coincide con quello cercato, ma quando lo segue in ordine alfabetico. Quindi dovremo leggere il libro corrente con `leggelibro` in una stringa ausiliaria `AutoreTitolo2` e terminare il ciclo quando `strcmp` applicata alle due stringhe restituisce un valore

negativo, anziché nullo[24]. Possiamo interrompere il ciclo usando un *flag*, ma un modo più semplice è adoperare l'istruzione `break`. Questa istruzione è considerata un po' anomala perché distrugge il principio di fondo che un blocco di istruzioni (qui rappresentato dalle parentesi graffe che racchiudono il corpo del ciclo) dovrebbe avere un solo punto di ingresso e un solo punto di uscita, rispettivamente alla prima e all'ultima riga del blocco. Il motivo è che è facile perdere il controllo su che cosa succede in un modulo se ingressi e uscite avvengono ovunque (è la famigerata questione dell'istruzione `goto`). D'altra parte, per un ciclo corto come questo, in cui `break` compare all'ultima riga, il problema è decisamente meno grave. Trovata la posizione `p`, possiamo applicare la solita funzione `inslista` alla funzione `*pScaffale` per aggiornarla. Anche qui, lo spostamento di un libro da una lista a un altro 'e stato eseguito cancellandolo dalla prima e inserendolo nella seconda, ma si sarebbe potuto realizzare una funzione per spostare l'elemento, risparmiando allocazioni, deallocazioni e diverse copie del campo che contiene le informazioni.

Terminato il ciclo, stampiamo il messaggio corretto. Questo dipende dalla posizione in cui abbiamo inserito il libro. Per distinguere se la posizione era fittizia o corrispondeva a un libro effettivo, basta verificare se `finelista` in posizione `p` è vera o falsa[25]. Nel primo caso, scriviamo `METTERE` prima del libro reso (`AutoreTitolo`) e poi `PER ULTIMO`. Nel secondo, scriviamo `METTERE` prima del libro reso (`AutoreTitolo`) e poi `PRIMA DI` e il titolo del libro in posizione `p`, che è `AutoreTitolo2`[26].

Compilando ed eseguendo il codice, dovremmo ottenere il risultato corretto dell'intero esercizio: prima la stampa dei libri rimasti sullo scaffale alla fine della giornata (che sono 4, come visto sopra); poi le istruzioni di riordino (un libro in fondo, perché i libri rimasti erano successivi, poi due libri prima di altri due libri inizialmente presenti, infine un libro prima di uno che era in prestito, ma è stato reso dopo, e quindi reinserito prima sullo scaffale); infine, la stampa dei libri presenti sullo scaffale al termine delle operazioni di riordino (che sono 9: i 4 iniziali e i 5 resi).

7.5 Implementazione con vettori e cursori

Come anticipato nella Sezione 7.2, esiste un altro importante gruppo di implementazioni delle liste, che consente di ottenere gli stessi 8 tipi di lista discussi in precedenza (monodirezionali o bidirezionali, lineari o circolari, con o senza sentinella). Nel seguito discutiamo in dettaglio queste implementazioni e consideriamo un esercizio di implementazione. Lo scopo sarà verificare che, adottando il punto di vista delle strutture dati astratte, l'algoritmo realizzato nel file `biblio4.c` può rimanere intatto a fronte di una modifica completa dell'implementazione delle liste, purché si usi una nuova libreria, cioè si includa un nuovo file di intestazione `listalibri-cursori.h` e si compili il tutto con un nuovo file listato `listalibri-cursori.c` (ovviamente, se la nuova libreria ha nomi identici alla vecchia per i due file, cosa che qui non facciamo per chiarezza, non cambia assolutamente nulla: basta ricompilare il tutto).

Nell'implementazione a vettori e cursori, una lista è conservata in un vettore (vedi Figura 7.13). Ogni elemento del vettore è una struttura con due o tre campi:

[24] Potrebbe succedere che le due stringhe siano identiche? Se esistono più copie dello stesso libro, sì. In tal caso, l'ordine è ininfluente (secondo il testo), per cui è indifferente considerare una condizione di valore negativo o non positivo.

[25] Si noti che confrontare `p` con `NO_ELEMENT` sarebbe sbagliato, dato che, nel caso di sforamento della lista, il ciclo assegna a `p` l'indirizzo della sentinella.

[26] Si noti che in questo caso non aver aggiornato `p` durante la funzione `inslista` ci torna utile, perché vogliamo conoscere il libro che occupava quella posizione prima dell'inserimento.

L'idea base è di rappresentare le posizioni con indici in un vettore

- l'intera lista corrisponde allora a
  - un vettore (cioè un puntatore alla prima cella)
- ogni elemento a_i della lista corrisponde a una struttura con
  - il dato a_i
  - l'indice successivo p_{i+1} ($\perp$ se a_i è in coda)
  - eventualmente, l'indice precedente p_{i-1} ($\perp$ se a_i è in cima)
- gli elementi inutilizzati del vettore stanno in una "lista libera"

Non occorrono allocazioni e deallocazioni

Ovviamente, si perde la totale dinamicità

Figura 7.13: Liste: implementazione con vettori e cursori

un campo informativo (nella Figura 7.14), numeri interi), un campo `succ`, ed eventualmente un campo `pred`, che consentono di muoversi lungo la lista. Questi campi, però, non contengono indirizzi, ma gli indici numerici degli elementi corrispondenti nel vettore. Nell'esempio, l'elemento di indice 4 ha valore 5; il suo precedessore è l'elemento di indice 7 e il successore quello di indice 1 Ovviamente, occorre un indice speciale per accedere al primo elemento della lista, oppure (come nel caso della figura) alla sentinella, che non ha un campo informativo dotato di significato. È facile rendersi conto, scorrendola, che la lista dell'esempio è circolare.

La caratteristica fondamentale di questa implementazione è che gli elementi sono tutti allocati fin dal principio. Questo è un vantaggio, perché non occorre allocarli e deallocarli durante l'uso, cosa che costa tempo e può generare problemi nella gestione in tempo reale della memoria (le aree allocate possono frammentarsi, creando zone disponibili, ma troppo piccole per poter servire davvero a qualcosa; inoltre, accedere consecutivamente ad aree di memoria vicine è spesso più veloce per motivi tecnologici). È anche uno svantaggio, perché il vettore ha una dimensione fissata al principio e quindi la lista non può crescere oltre una determinata dimensione: si perde la totale dinamicità (a meno di riallocare il vettore, il che comporta una ricostruzione non sempre banale del suo contenuto, e a volte la sua intera copia in altre aree di memoria).

Inoltre, anche se gli elementi del vettore sono già allocati, non è chiaro come fare a crearne di nuovi per inserirli nella lista corrente. Bisogna sapere quali elementi del vettore sono attualmente parte della lista e quali invece sono liberi, dunque utilizzabili per essere inseriti. Questo richiede di definire sullo stesso vettore una seconda lista, nota in gergo come *lista libera*, complementare alla prima. La lista libera, cioè, contiene tutti gli elementi del vettore che non fanno parte della lista primaria. Nella Figura 7.14 vediamo che la lista contiene solo gli elementi di indice 1, 7 e 4 (nell'ordine). Gli altri elementi fanno invece parte della lista libera e i loro campi `pred` e `succ` sono predisposti a concatenarli. Inoltre, esiste un indice ausiliario per la testa (la sentinella, in effetti) della lista libera, che nell'esempio è l'elemento di indice 8. Quando si inserisce un nuovo elemento nella lista, questo viene preso dalla lista libera e spostato. Viceversa, cancellare un elemento significa spostarlo nella lista libera. Questa è l'idea di fondo dell'implementazione.

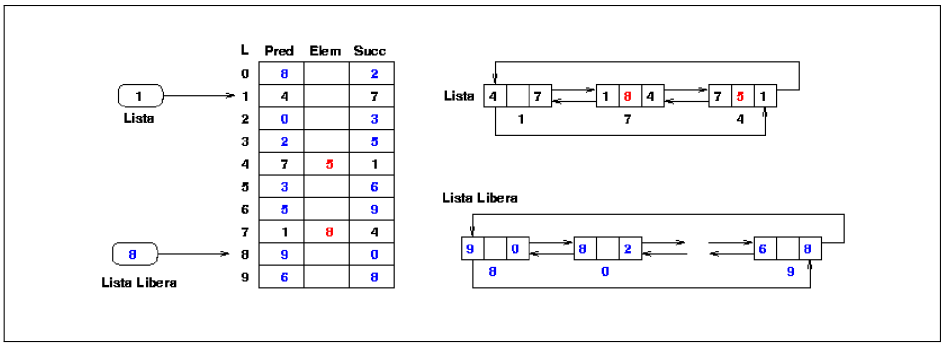


Figura 7.14: Liste: implementazione con vettori e cursori

La Figura 7.15 riporta la dichiarazione in C delle strutture dati necessarie. È utile confrontarla elemento per elemento con la dichiarazione a puntatori della Figura 7.7.

Una lista è ancora una volta un puntatore a elemento, ma si tratta in effetti di un vettore dinamico, anziché dell'indirizzo di un solo elemento, cioè automaticamente sappiamo dove stanno gli altri elementi della lista (anche se non abbiamo informazioni sulla loro sequenza effettiva).

Perché non un vettore statico? Potrebbe esserlo, ma cè un problema puramente tecnico: vogliamo conservare esattamente le dichiarazioni delle funzioni, e in particolare di `crealista`. Ma questa funzione non riceve dati e restituisce una lista, per cui:

- per allocare tutti gli elementi, deve conoscerne per altra via il numero: questo si ottiene attraverso una costante simbolica `LIST_SIZE`, che specifica una volta per tutte un numero massimo possibile di elementi;
- in quanto funzione C, deve restituire un oggetto semplice, e quindi non può generare un vettore statico, ma solo dinamico.

Per questo motivo abbiamo la bizzarra situazione di un vettore di dimensione costante allocato dinamicamente[27]

Gli elementi del vettore sono strutture contenenti un libro (cioè una stringa) e due posizioni, esattamente come nell'altra implementazione. Tuttavia, le posizione non sono puntatori ad elemento, ma numeri interi. Questa è la grossa differenza.

Per accedere alla lista e alla lista libera, occorrono indicazioni sulla posizione delle due sentinelle[28]. Queste indicazioni sono fornite da altrettante costanti simboliche (sempre perché le dichiarazioni delle funzioni non offrono altra via per ottenerle): la sentinella è l'elemento di indice 0 e la sentinella libera nell'elemento finale `LIST_SIZE+1` (gli elementi significativi vanno quindi da 1 a `LIST_SIZE` e `LIST_SIZE` è esattamente il loro numero massimo). Va detto che le sentinelle potrebbero andare ovunque, e forse l'idea di mettere quella libera in ultima posizione non è la più brillante, se si volessero rompere i limiti di questa struttura e ammettere la possibilità di aumentare la dimensione del vettore riallocandolo e conservando in qualche modo la dimensione corrente.

[27] In generale, cosa da evitare per chiarezza, e penalizzata nella valutazione dei progetti.

[28] A rigore, la lista libera potrebbe anche essere monodirezionale e senza sentinella, dato che inserimenti e cancellazioni avverranno sempre dalla testa, ma evitiamo complicazioni di dettaglio.

```
#define NO_ELEMENT -1                          (posizione esterna alla lista)
#define NO_LIST NULL

#define LIST_SIZE 100                          (lunghezza del vettore)
#define SENTINELLA 0                           (indice della sentinella)
#define SENTINELLA_LIBERA LIST_SIZE+1          (indice della sentinella
per la lista libera)

typedef elemento *lista;                (la lista è l'indirizzo del vettore)
typedef int posizione;           (la posizione dell'elemento è il suo indice)

typedef struct _elemento elemento;
struct _elemento {
   U a;                                 (U è il tipo dell'elemento generico)
   posizione succ;
   posizione pred;                          (questo campo può mancare)
};
```

Figura 7.15: Liste: implementazione con vettori e cursori

7.5.1 Implementazione della libreria

La fase finale dell'esercizio non riguarda l'algoritmo in sé, ma un'implementazione alternativa della libreria per gestire le liste di libri. È già dato il file di intestazione `listalibri-cursori.h` e un file listato iniziale `listalibri-cursori0.c`, in cui tutte le definizioni hanno il corpo vuoto, mentre il risultato finale sarà il file `listalibri-cursori.c`.

Si sono già discussi gli svantaggi in termini di efficienza delle strutture dati astratte. È quindi utile vedere concretamente uno dei vantaggi non ovvii.

Ripartiamo dalla soluzione `biblio4.c` dell'esercizio già svolto. Sostituiamo l'inclusione di `listalibri.h` con `listalibri-cursori.h` (cosa non necessaria se semplicemente ci limitiamo a dare lo stesso nome alle due due librerie). A questo punto, dobbiamo ripetere l'esercizio della Sezione 7.4.6, cioè riempire i corpi vuoti di tutte le funzioni di gestione della lista. È utile procedere confrontando i singoli passaggi con le corrispondenti implementazioni per la libreria a puntatori, per valutare differenze e somiglianze.

Cominciamo con la creazione di una lista vuota.

```
listalibri crealista ()
{
  listalibri L;
  posizione p;

  L = (listalibri) calloc(LIST_SIZE+2,sizeof(elemento));
  if (L == NULL)
  {
    fprintf(stderr,"Memoria insufficiente per allocare una lista di
        libri!\n");
    exit(EXIT_FAILURE);
  }

  /* La lista contiene solo la sentinella */
  L[SENTINELLA].succ = SENTINELLA;
```

```
  L[SENTINELLA].pred = SENTINELLA;

  /* Gli altri elementi allocati stanno nella "lista libera" */
  for (p = 1; p <= LIST_SIZE; p++)
  {
    L[p+1].pred = p;
    L[p].succ = p+1;
  }
  L[1].pred = SENTINELLA_LIBERA;
  L[SENTINELLA_LIBERA].succ = 1;

  return L;
}
```

Anziché allocare un singolo elemento che facesse da sentinella e assegnare ai suoi puntatori l'indirizzo dell'elemento stesso, dobbiamo per prima cosa allocare un intero vettore di `LIST_SIZE+2` elementi `L`. Questa bizzarra operazione di allocazione dinamica con una dimensione costante in linea di principio andrebbe evitata perché è molto più comodo e chiaro definire un vettore statico, ma qui siamo costretti a farlo dalla necessità di restituire questo vettore alla procedura chiamante, imposta dal formato della dichiarazione di `crealista`, che non è modificabile, se vogliamo mantenere la struttura dati astratta.

Controllato che l'allocazione abbia avuto successo, inizializziamo gli indici del predecessore e successore della sentinella, che è l'elemento di indice `SENTINELLA` nel vettore: saranno entrambi uguali allo stesso indice `SENTINELLA`. Il campo informazione della sentinella si può ignorare perché non verrà mai usato. Se l'intero vettore viene allocato con `calloc`, i suoi elementi sono tutti inizializzati a 0, e questo significa che i libri sono tutti inizializzati a stringhe vuote (dato che il carattere terminatore coincide con lo zero). Dal punto di vista pratico, non è un vantaggio, e ha un costo temporale (lineare nella dimensione della lista, anziché costante). Può essere gradevole se si sta usando un ambiente di compilazione con un *debugger* che consente di vedere il contenuto delle variabili, perché allora le stringhe appaiono tutte vuote.

Mentre l'implementazione a puntatori a questo punto è terminata, quella a vettori richiede di strutturare la lista libera, in modo che contenga tutti gli elementi. Per farlo, conviene che ogni elemento punti il successivo e il precedente (se esistono), la sentinella libera abbia come successore il primo elemento e come predecessore l'ultimo, il primo elemento abbia come predecessore la sentinella libera e l'ultimo elemento abbia come successore la sentinella libera. Questo produce due liste complementari all'interno del vettore: una lista vuota e lista libera che contiene tutti gli elementi esistenti, nell'ordine (cosa non richiesta, ma fatta per semplicità). A questo punto, la lista è creata e si può restituire il vettore all'esterno per chi voglia usarlo.

Consideriamo ora la procedura `distruggelista`, che è invece molto più semplice della corrispondente. Infatti, anziché distruggere un elemento alla volta, si può deallocarli tutti in blocco con la funzione `free`.

```
void distruggelista (listalibri *pL)
{
  free(*pL);
  *pL = NO_LIST;
}
```

Se vogliamo essere molto chiari, possiamo rendere visibile a tutti che l'indirizzo del vettore non corrisponde più a un'area di memoria allocata assegnandogli il valore `NULL`. Questo permette di controllarlo ed evitare che si cerchi di accedere ancora alla stessa area di memoria attraverso quell'indirizzo.

La lettura di un elemento è molto semplice, dato che la posizione `p` è l'indice dell'elemento cercato nel vettore `L` e quindi basta copiare il campo `AutoreTitolo` nel risultato attraverso la funzione `strcpy`.

```
void leggelista (listalibri L, posizione p, libro AutoreTitolo)
{
  strcpy(AutoreTitolo,L[p].AutoreTitolo);
}
```

L'istruzione è praticamente identica, salvo che diventa finalmente chiaro perché la procedura `leggelista` riceva anche la lista, oltre alla posizione e al libro da leggere: contrariamente all'implementazione a puntatori, in quella a vettori conoscere la posizione `p` di un elemento non è sufficiente a determinarlo.

La scrittura di un elemento della lista funziona alla stessa maniera.

```
listalibri scrivelista (listalibri L, posizione p, libro AutoreTitolo)
{
  strcpy(L[p].AutoreTitolo,AutoreTitolo);
  return L;
}
```

Si noti che `scrivelista` restituisce la lista modificata (cosa non strettamente necessaria, ma aderente alla definizione astratta), e questo rende necessario che il vettore `L` sia dinamico, altrimenti non potremmo restituirlo con una funzione C.

Il test che determina se una lista è vuota consiste nel valutare gli indici del predecessore e successore della sentinella, per vedere se puntano la sentinella stessa. Anche qui non testiamo entrambe le condizioni, assumendo che la lista sia gestita solo con le funzioni di libreria, e quindi le due condizioni valgano o non valgano simultaneamente.

```
boolean listavuota (listalibri L)
{
  return (L[SENTINELLA].succ == SENTINELLA);
}
```

Il primo elemento della lista è sempre il successore della sentinella (originariamente, `L->succ`).

```
posizione primolista (listalibri L)
{
  return L[SENTINELLA].succ;
}
```

mentre l'ultimo è il predecessore (originariamente, `L->pred`).

```
posizione ultimolista (listalibri L)
{
  return L[SENTINELLA].pred;
}
```

Le posizioni degli elementi successivo e precedente si trovano prendendo i campi `succ` e `pred` dell'elemento di indice `p` nel vettore `L`.

```
posizione succlista (listalibri L, posizione p)
{
  return L[p].succ;
}

posizione predlista (listalibri L, posizione p)
{
  return L[p].pred;
}
```

Ancora una volta, si constata la necessità di conoscere non solo la posizione, ma anche la lista.

Una lista è finita quando si arriva alla sentinella, cioè l'indice numerico `p` coincide con `SENTINELLA`.

```
boolean finelista (listalibri L, posizione p)
{
  return (p == SENTINELLA);
}
```

Veniamo alle operazioni più sofisticate. L'inserimento comportava una nuova allocazione. In questa implementazione, invece, tutto è già allocato, per cui bisogna sganciare dalla lista libera un elemento, inserirvi l'informazione nuova e agganciarlo alla lista nella posizione indicata.

```
listalibri inslista (listalibri L, posizione p, libro AutoreTitolo)
{
  posizione q;

  q = L[SENTINELLA_LIBERA].succ;
  /* La lista libera e' vuota: non si possono inserire libri
     NOTA: si potrebbe allargare la memoria allocata per il vettore */
  if (q == SENTINELLA_LIBERA)
  {
    fprintf(stderr,"Memoria insufficiente per allocare il libro
        %s!\n",AutoreTitolo);
    exit(EXIT_FAILURE);
  }
  strcpy(L[q].AutoreTitolo,AutoreTitolo);

  /* Toglie l'elemento in posizione q dalla lista libera */
  L[L[q].pred].succ = L[q].succ;
  L[L[q].succ].pred = L[q].pred;

  /* Mette l'elemento nella lista vera */
  L[q].pred = L[p].pred;
  L[q].succ = p;
  L[L[p].pred].succ = q;
```

```
  L[p].pred = q; /* NB: L[p].pred va aggiornata per ultima, o le
      istruzioni precedenti sono scorrette! */

  return L;
}
```

L'elemento da sganciare è uno qualsiasi, dato che fa solo da contenitore per l'informazione che è fornita in ingresso alla funzione. Per semplicità, prendiamo il primo elemento della lista libera. Come nell'implementazione a puntatori dobbiamo verificare che l'allocazione non sia fallita, così qui dobbiamo verificare che la lista libera non sia vuota. Le due situazioni sono logicamente equivalenti, anche se in questo caso il fallimento sopravviene prima, dato che si è esaurito solo il vettore, e non l'intera memoria. Termineremo quindi l'intero programma con un messaggio di errore[29]

Se la lista libera non è vuota, procediamo a sganciare il suo primo elemento, in maniera analoga a quanto si fa per cancellare un elemento da una lista, dato che l'elemento viene sganciato dalla lista prima di deallocarlo. Ovviamente, il procedimento va tradotto nelle nuove strutture dati: anziché l'elemento corrente, il successore deve avere come predecessore il predecessore e il successore deve avere come predecessore il predecessore.

Il nuovo elemento va modificato inserendogli l'informazione associata al nuovo libro.

Sganciato l'elemento, possiamo inserirlo nella lista nella posizione data, usando le quattro operazioni già note, ma tradotte nelle nuove strutture dati: l'elemento da inserire deve avere come successore l'elemento nella posizione data e come predecessore il predecessore di questo; l'elemento predecessore deve avere come successore quello nuovo, e l'elemento nella posizione data deve avere come predecessore il nuovo elemento (questa operazione va fatta dopo la precedente, per non perdere il predecessore)[30]. Come di consueto, restituiamo la lista `L`.

Inserimento e cancellazione hanno lo stesso problema già discusso sul modificare o no il valore della posizione a valle della funzione. Dato che la dichiarazione delle funzioni deve rimanere la stessa, adottiamo la stessa soluzione incoerente: conservare il vecchio valore di `p` dopo l'inserimento e aggiornarlo dopo la cancellazione.

```
listalibri canclista (listalibri L, posizione *pp)
{
  posizione p;

  p = *pp;
  *pp = L[p].succ;

  /* Sgancia l'elemento dalla lista */
  L[L[p].pred].succ = L[p].succ;
  L[L[p].succ].pred = L[p].pred;

  /* Inserisce l'elemento in cima alla lista libera */
  L[p].pred = SENTINELLA_LIBERA;
  L[p].succ = L[SENTINELLA_LIBERA].succ;
  L[L[SENTINELLA_LIBERA].succ].pred = p;
```

[29] Si potrebbe pensare di riallocare il vettore con dimensioni superiori, raffinamento che qui non discutiamo, dato che comporta diverse questioni tecniche.

[30] Ma forse in effetti potremmo usare le informazioni che sono appena state scritte nel nuovo elemento.

```
  L[SENTINELLA_LIBERA].succ = p; /* NB: L[SENTINELLA_LIBERA].succ va
      aggiornata per ultima, o le istruzioni precedenti sono
      scorrette! */

  return L;
}
```

Occorre conservare temporaneamente il valore della posizione `p`, cioè l'indice intero `*pp`, in modo da poter spostare `pp` sull'elemento successivo senza perdere l'informazione. Fatto ciò, si tratta di sganciare l'elemento dalla lista (con le due operazioni già viste sopra. Infine, non deallocheremo l'elemento cancellato, ma lo inseriremo nella lista libera. Siccome la posizione di inserimento è arbitraria, per semplicità lo inseriremo in cima, fra la sentinella e il primo elemento, con le consuete quattro operazioni: il nuovo elemento ha come predecessore la sentinella e come successore il primo elemento della lista libera; il primo elemento ha come predecessore quello nuovo e la sentinella ha come successore l'elemento nuovo. Al termine, restituiamo la lista.

Ora, se compiliamo `biblio4.c` con `listalibri-cursori.c` anziché `listalibri.c` (o se usiamo lo stesso nome per le due librerie), il risultato è perfettamente corretto. Questo è il concetto fondamentale: adottando strutture dati astratte, è possibile sostituire in blocco i moduli contenenti l'implementazione di strutture dati a basso livello senza dover modificare i moduli contenenti gli algoritmi ad alto livello. Questo consente ad alcuni di pensare algoritmi senza sporcarsi le mani con l'implementazione delle strutture dai, e ad altri di lavorare all'ottimizzazione delle strutture dati stesse senza distruggere l'opera di chi ha scritto algoritmi usando le precedenti versioni.

7.6 Leggere una riga di testo (un sordido problema informatico)

Nella terza fase dell'esercizio, si pone il problema di leggere una riga del file di ingresso, decidendo se:

1. includere l'a capo finale,
2. includerlo e subito cancellarlo,
3. non includerlo.

La prima soluzione è la più semplice: basta applicare la funzione `fgets`. Però bisogna ricordare nel seguito che i libri terminano con un a capo. Questo influisce sulle successive operazioni di stampa (ogni libro andrà automaticamente su una riga diversa, senza indicarlo esplicitamente) e di confronto (l'a capo compare in tutti i libri letti in questo modo, per cui non dovrebbero esserci problemi). Soprattutto, crea problemi alla stampa delle istruzioni di riordino, nelle quali si stampa il libro reso seguito dalla posizione in cui inserirlo e tutto questo andrebbe fatto su una sola riga. Questo tende a escludere tale soluzione.

Nelle prime edizioni del corso ho applicato la seconda soluzione, applicando la funzione `fgets` per leggere la riga compreso l'a capo e poi cancellando quest'ultimo. Per troncare una stringa in qualsiasi posizione basta sovrascrivere il primo carattere che va escluso con il terminatore `'\0'`. In questo caso, il carattere è `'\n'`) e sta in ultima posizione, per cui si può applicare l'istruzione `Riga[strlen(Riga-1)]`

`= '\0';` e quasi sempre non si avranno problemi[31]. Purtroppo, sistemi operativi diversi gestiscono in modo diverso l'andare a capo: alcuni lo rappresentano con un singolo carattere, altri con due. Bisogna ricordare le macchine da scrivere meccaniche, che andavano a capo eseguendo due movimenti: tirare il carrello verso sinistra e poi ruotare il tamburo per spostare il foglio di carta. I compilatori C dovrebbero adeguarsi tutti in modo che il programmatore possa ragionare sempre in termini di un singolo carattere.

Purtroppo, se si apre con un sistema operativo un file di testo creato con un altro e non riadattato durante il trasferimento, sono possibili problemi. In particolare, gli esempi `input01a.txt` e `input01b.txt` sono stati creati sotto Windows, e ogni riga termina con i caratteri `'\r'` (rientro del carrello) e `'\n'` (passaggio alla riga seguente). Usandoli sotto altri sistemi operativi senza una traduzione, possono ingannare il compilatore C, che cancellerà solo `'\n'`, lasciando `'\r'` che crea nella stampa un effetto bizzarro di rientro al principio della riga e conseguente sovrascrittura. Il tutto è ben spiegabile solo alla luce del meccanismo. Una soluzione pratica è adattare i file al nuovo sistema con i seguenti comandi da terminale:

```
sed -i 's/\r//g' input01a.txt
sed -i 's/\r//g' input01b.txt
```

Probabilmente, funziona anche aprire i file, e cancellare e ricreare gli a capi a mano uno per uno.

La terza soluzione è quella adottata correntemente: consiste nel leggere la riga con l'istruzione `fscanf(fp,"%[^\n]\n",Riga);` dove la specifica `"%[^\n]"` riconosce qualsiasi sequenza di caratteri che non (`^`) includa l'a capo, mentre il termine `\n` finale legge l'a capo (in effetti, legge qualsiasi sequenza di separatori: a capi, spazi bianchi e tabulazioni) senza scriverlo da nessuna parte. Non ho testato questa soluzione su altri sistemi operativi, quindi invito a farmi sapere se dà luogo a problemi.

[31] Un problema di questa istruzione è che, applicata su una riga vuota, cioè di lunghezza nulla, va a scrivere nella posizione `Riga[-1];` provocando certamente problemi, ma le righe lette da `fgets` dovrebbero essere tutte non vuote.

7.7 Esercizi

7.7.1 Implementazione senza sentinella di listavuota

È una buona idea interrogarsi su come sarebbe possibile realizzare questa funzione in implementazioni diverse (in particolare, prive di sentinella).

7.7.2 Inserimento di nuovi elementi in coda a una lista

`succlista` al risultato di `ultimolista`. Ne risulta un'espressione abbastanza barocca da suggerire che possa essere utile aggiungere alla libreria una funzione `accodalista` per inserire un nuovo elemento in coda a una lista chiamando una sola funzione anziché tre.

7.7.3 Stampa di una lista

Si può osservare che questa procedura di stampa non è molto efficiente, dato che la lettura copia la stringa in un'altra, che viene poi stampata (per non parlare delle chiamate a funzione per scorrere la lista, che richiedono tutto il macchinario di allocazione e deallocazione dei record di attivazione sullo *stack* e che potrebbero essere sostituite da semplici accessi a puntatori). Per renderla efficiente si potrebbe aggiungere alla libreria una funzione `stampalista` che essendo interna alla libreria stessa, avrebbe il permesso di accedere direttamente al dato.

7.7.4 Spostamento di elementi da una lista a un'altra

Costruiamo una nuova funzione per spostare un elemento da una lista a un'altra. Va notato che la funzione è piuttosto semplice se sposta elementi da una lista a un'altra, ma potrebbe venire la tentazione di usare la stessa funzione per spostare un elemento da una posizione a un'altra nella stessa lista. Vi sono casi in cui questo potrebbe portare a errori nella gestione dei puntatori. Le soluzioni sono due: vietare lo spostamento interno a una lista (con un controllo esplicito) oppure implementare la funzione con molta attenzione, in modo che sia corretta anche nei casi più insidiosi.

7.7.5 Ricerca di un elemento in una lista

Aggiungere la funzione `TrovaLibro` alla libreria e usarla.

7.7.6 Altre implementazioni

Ovviamente, ciascuna delle altre 14 implementazioni presentate in precedenza (7 a puntatori e 7 a cursori) costituisce un utile esercizio.

7.7.7 Implementazione con indirizzi separati del primo e ultimo elemento

7.7.8 Impaccamento di liste

Realizzare una libreria "impaccamento di liste" che gestisca un numero (dato o qualsiasi) di liste multiple definite sullo stesso universo (a puntatori o a vettore e

cursori), con la limitazione che un elemento possa appartenere al massimo a una delle liste. La libreria deve consentire spostamenti fra liste o fra liste vere e lista libera (nel caso dei vettori).

Rigorosamente parlando, è anche possibile gestire più liste su un solo vettore, finché sono mutuamente esclusive. Tecnicamente, questo richiede $l+1$ posizioni aggiuntive per le sentinelle delle l liste e la sentinella della lista libera (non necessaria se l'impaccamento è in effetti una partizione). All'indubbio vantaggio in termini di spazio fa da contrasto lo svantaggio che le dichiarazioni e le definizioni riportate nella libreria diventerebbero specifiche per il caso particolare della partizione di un insieme dato in $l+1$ liste. Si tratta valutare se la maggior specificità è compensata dalla maggiore efficienza. Va anche detto che ospitare più liste su un solo vettore confligge con la struttura dati astratta che abbiamo definito e usato finora. Occorrerebbe in effetti definire una struttura dati astratta per la gestione di una pluralità di liste disgiunte.

Esercizi sulle liste

Esercizio 1 Si scriva un codice `progress.c`, che legge da tastiera un numero `n`, inserisce in cima a una lista `L` i primi `n` numeri positivi e li stampa nell'ordine che seguono lungo la lista.

Esercizio 2 Si aggiunga al codice precedente una funzione che scorre la lista `L` per cercare la posizione occupata dal numero `n/2` e poi stampa il contenuto della lista da quella posizione alla fine.

Esercizio 3 Si implementi una libreria per gestire liste di numeri interi, costituita dai file `listaint.c` e `listaint.h`. Si usi tale libreria per leggere da file un insieme di interi aggiungendo i valori letti alla lista in cima, oppure in fondo, oppure in posizione crescente o decrescente.

Si aggiungano quindi alla libreria:

- una funzione di inserimento ordinato (nel caso la lista sia bidirezionale, si può anche scegliere se aggiungere l'elemento dalla cima o dal fondo per confronto col primo e con l'ultimo elemento della lista);
- una funzione di concatenamento di liste;
- una funzione di conteggio del numero di elementi della lista;
- una funzione di ricerca che restituisce la posizione di un elemento in una lista.

Esercizio 4 Si scriva un programma `listamult.c` che gestisce su una tabella (quindi con un'implementazione a indici) un numero variabile di liste (al massimo 10) e obbedisca a comandi del tipo[32]:

- aggiungere un elemento a una lista di indice dato (da 1 a 10);
- stampare una lista di indice dato;
- contare gli elementi di una lista di indice dato;
- concatenare due liste di indice dato;
- cercare elementi in una lista;
- cancellare una lista;
- ordinare le liste per cardinalità decrescente;
- ecc...

[32] Questo esercizio è un'idea peregrina buttata lì per stimolare la fantasia.

Esercizio 5 Dato il vettore `int V[8]` di valore `[ 5 15 34 54 14 2 52 72 ]` e i due puntatori `int *p, *q`, inizializzati rispettivamente con
```
p = &V[1];
q = &V[5];
```

- quanto vale `*(p+3)`?
- quanto vale `*(q-3)`?
- quanto vale `q-p`?
- è vero o falso che `p < q`?
- è vero o falso che `*p < *q`?

Esercizio 6 Supponendo che `s`, `d` e `m` siano puntatori a elementi di un vettore, si vuole scrivere un'istruzione che faccia puntare `m` all'elemento intermedio fra `s` e `d` (nel caso vi siano due elementi intermedi, si consideri quello di sinistra; ad esempio, per `s = &V[3]` e `d = &V[6]` sia `m = &V[4]`). L'istruzione `m = (s + d)/2;` è scorretta. Perché? Come ottenere un'istruzione corretta, usando l'aritmetica dei puntatori?

Esercizio 7 Sia `int V[10]; int *p;` e `p = V;`
Indicare se le seguenti espressioni sono lecite o no, e per quelle lecite se sono vere o false:

- `p == V[0]`
- `p == &V[0]`
- `*p == V[0]`
- `p[0] == V[0]`

Capitolo 8

Grafi

Questo capitolo è dedicato a una delle strutture dati più comuni e più importanti della matematica discreta: i *grafi*. Ci sono decine di migliaia di problemi applicativi o teorici che sono legati ai grafi e che li rendono un oggetto di particolare interesse. La prima sezione è teorico-descrittiva, definisce i grafi e descrive le loro proprietà, e si sovrappone a un'analoga sezione delle dispense di teoria. Il suo scopo è di sottolineare alcuni aspetti che ci interessano più particolarmente in questa sede.

8.1 Definizioni per i grafi non orientati

Ogni relazione binaria su un insieme base finito $V = \{v_1, \ldots, v_n\}$
si può descrivere elencando le coppie di elementi di V in relazione

$$E = \{\{i, j\} : i \in V, j \in V, i \text{ e } j \text{ are related}\} \Rightarrow E \subseteq V \times V$$

Un modo standard di rappresentare una relazione binaria è il grafo $G = (V, E)$, cioè una coppia di insiemi:

- un insieme V di oggetti elementari detti vertici
- un insieme E di coppie non ordinate di oggetti di V detti lati

Un grafo si rappresenta disegnando i vertici come punti (o cerchi)
e i lati come linee

$V = \{v_1, v_2, v_3, v_4, v_5\}$

$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_1, v_5\}, \{v_2, v_3\},$
$\{v_2, v_4\}, \{v_3, v_4\}, \{v_3, v_5\}, \{v_4, v_5\}\}$

Si notino le parentesi grafe:
la coppia non è ordinata

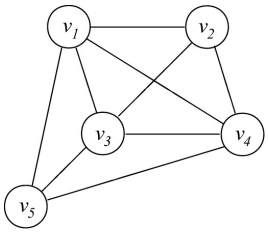


Figura 8.1: Grafi

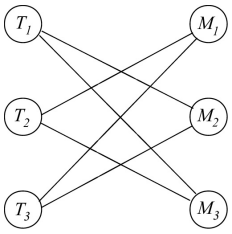
Un *grafo non orientato* descrive una relazione binaria simmetrica tra elementi di un insieme base finito $V = \{v_1, v_2, \ldots, v_n\}$, i cui elementi vengono convenzionalmente chiamati *vertici*.

Si tratta di oggetti elementari, cioè quello che ci interessa di questi oggetti, a livello di struttura dati astratta grafo, è che ci sia una relazione binaria su di loro, cioè che alcune coppie di questi oggetti siano in relazione. Queste coppie (non ordinate) vengono dette *lati*.

La classica rappresentazione grafica di un grafo è costituita da puntini o cerchi, che rappresentano i vertici, e da linee che li collegano, che rappresentano i lati. Le linee non sono necessariamente rette perché di questi lati interessa solo la topologia, cioè i due estremi che essi collegano. Nella Figura 8.1 c'è un esempio con 5 vertici e 9 lati. Si sottolinea in particolare il fatto che il lato (v_1, v_2) è il lato (v_2, v_1): non c'è una ordine preciso.

Tipicamente il grafo viene indicato quindi attraverso una coppia di oggetti (V, E), dove V è l'insieme base (e sta per "vertices", in inglese), mentre E è un insieme di coppie degli oggetti dell'insieme base (e sta per "edges").

- reti stradali: i vertici sono città, i lati strade
- reti elettriche: i vertici sono impianti, stazioni o utenti, i lati linee elettriche
- reti di telecomunicazione: i vertici sono trasmettitori, ripetitori e ricevitori, i lati collegamenti
- reti sociali: i vertici sono utenti, i lati relazioni umane
- giochi: i vertici sono posizioni, i lati mosse
- relazioni di (in)compatibilità; i vertici sono oggetti/persone, i lati coppie di oggetti/persone (in)compatibili



V include operazioni (*task*) e macchine
$V = \{T_1, T_2, T_3, M_1, M_2, M_3\}$

Il lato $\{i, j\}$ indica che il task i può essere eseguito dalla macchina j

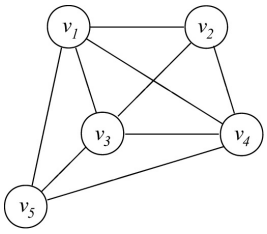
Non si può eseguire un task su un task, né una macchina su una macchina

Figura 8.2: Esempi

Vediamo un po' di esempi tipici di applicazioni. I grafi servono moltissimo a rappresentare reti stradali, perché i vertici si prestano a rappresentare città e i lati strade, oppure reti elettriche (i vertici sono i punti importanti della rete elettrica, come impianti, stazioni, utenti, ecc...) e i lati sono le linee elettriche. Nelle reti sociali, i vertici sono utenti, i lati sono relazioni fra gli utenti (per esempio, amicizie), purché simmetriche. Nei giochi, i vertici rappresentano posizioni del gioco, i lati delle mosse. Bisogna osservare che un gioco si presta ad essere modellato con un grafo non orientato solo quando le mosse sono perfettamente reversibili, mentre in molti giochi le mosse non lo sono. Negli scacchi, i pedoni possono andare solo in avanti, per esempio. Per tali giochi ci vorranno i grafi orientati. Una classica

applicazione sono poi le relazioni di compatibilità o di incompatibilità. Il disegno in basso a sinistra nella Figura 8.1 si può interpretare dicendo che ci sono lavorazioni ("task"), T_1, T_2 e T_3, che possono essere eseguite su tre macchine, M_1, M_2 e M_3, ma non in maniera assolutamente libera. La lavorazione T_1 può essere eseguita solo sulla seconda o sulla terza macchina, T_2 solo sulla prima e sulla terza e T_3 sulla prima e la seconda. Ci si potrebbe chiedere se sia possibile assegnare conteporaneamente ogni lavorazione a una macchina rispettando queste compatibilità. Questo è un classico problema su grafo. Non è di quelli che affronteremo nel corso, ma è un esempio molto semplice e molto chiaro.

- i e j sono i vertici estremi del lato $\{i, j\}$
- due vertici i e j sono adiacenti se il lato $\{i, j\}$ esiste
- il lato $\{i, j\}$ è incidente nei vertici i e j
- il grado δ_v di un vertice v è il numero di lati incidenti



- v_1 e v_2 sono i vertici estremi di $\{v_1, v_2\}$
- v_3 e v_4 sono adiacenti (il lato $\{v_3, v_4\}$ esiste)
- il lato $\{v_3, v_5\}$ è incidente ai vertici v_3 e v_5
- il grado del vertice v_3 è $\delta_{v_3} = 4$

Figura 8.3: Topologia di un grafo

La Figura 8.1 illustra un po' di nomenclatura. I due vertici che compongono un lato si dicono *vertici estremi* del lato. Si dice che sono fra loro *adiacenti* e che il lato è *incidente* in uno e nell' altro. Il numero di lati che incide in un vertice è detto *grado* di quel vertice. Per esempio, nel disegno il centrale v_3 ha 4 lati che incidono in esso, cioè (v_1, v_3), (v_2, v_3), (v_3, v_4) e (v_3, v_5) e il suo grado è 4.

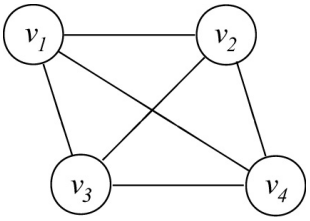
Un grafo che ha tutti i possibili lati si dice *grafo completo*, e siccome i lati sono tutte le coppie possibili, il numero dei lati di un grafo completo coincide con il numero delle combinazioni di n oggetti a due a due, $C_{n,2}$. Se non ammettiamo le ripetizioni, questo numero è espresso del coefficiente binomiale $\binom{n}{2}$, cioè è $n(n-1)/2$. Se invece ammettiamo che le coppie di vertici identici $((v_1, v_1)$, ecc. . .), questo numero cresce di n, e quindi diventa $n(n+1)/2$. I lati che hanno due estremi coincidenti tecnicamente si chiamano *autoanelli*.

Il concetto che ci interessa particolarmente per l'esercitazione pratica è quello di *sottografo*. Un sottografo è in un certo senso un sottoinsieme di un grafo. Siccome, però un grafo è una coppia di oggetti, la definizione va precisata. Un sottografo di un grafo $G = (V, E)$ è un grafo a sua volta, $H = (U, X)$, il cui insieme di vertici U è un sottoinsieme dell'insieme V dei vertici del grafo di partenza, mentre l'insieme dei lati X è un sottoinsieme dell'insieme E dei lati del grafo di partenza. Occorrono quindi tre proprietà:

1. che il primo sottoinsieme sia sottoinsieme del primo sottoinsieme di partenza;

Un grafo è completo quando ogni coppia di vertici corresponde a un lato

$$E = \{\{v_i, v_j\} : v_i \in V, v_j \in V, i < j\}$$



Tutti i grafi con n vertici hanno

$$m \leq \binom{n}{2} = \frac{n\,(n-1)}{2} \text{ lati}$$

(con l'uguaglianza per i grafi completi)

Se sono ammessi gli autoanelli, un grafo completo ha

$$E = \{\{v_i, v_j\} : v_i \in V, v_j \in V, i \leq j\} \qquad m = \frac{n\,(n+1)}{2}$$

Figura 8.4: Grafi completi

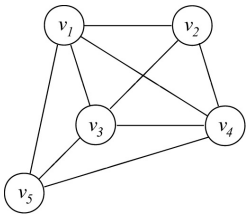
$H = (U, X)$ è un sottografo di $G = (V, E)$ se

- è un grafo
- $U \subseteq V$ e $X \subseteq E$

È un sottografo ricoprente quando $U = V$

È un sottografo indotto quando $X = E_U = \{\{u, v\} \in E : u, v \in U\}$

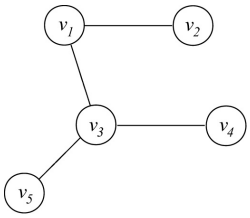
$G = (V, E)$



$V = \{v_1, v_2, v_3, v_4, v_5\}$

$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_1, v_5\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}, \{v_3, v_5\}, \{v_4, v_5\}\}$

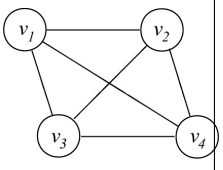
$H_1 = (U_1, X_1)$



$U_1 = \{v_1, v_2, v_3, v_4, v_5\} = V$

$X_1 = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_3, v_4\}, \{v_3, v_5\}\}$

$H_2 = (U_2, X_2)$



$U_2 = \{v_1, v_2, v_3, v_4\}$

$X_2 = \{\{v_1, v_2], \{v_1, v_3], \{v_1, v_4], \{v_2, v_3], \{v_2, v_4], \{v_3, v_4]\} = E_{U_2}$

Figura 8.5: Sottografi

2. che il secondo sia sottoinsieme del secondo sottoinsieme del grafo di partenza;

3. che il primo e il secondo sottoinsieme siano fra loro legati dalla relazione di costituire un grafo, cioè che X non contenga coppie qualsiasi, ma coppie di elementi che appartengono ad U.

Vediamo un esempio nella Figura 8.1. Il grafo a sinistra è il solito con 5 vertici e 9 lati. Esso ammette come sottografo in mezzo, che ha 5 vertici e soltanto 4 lati. Quei quattro lati sono un sottoinsieme dei 9 lati di partenza, come i 5 vertici sono un sottoinsieme dei 5 vertici di partenza. Si tratta di un sottografo. *En passant*, si tratta di un caso particolare di sottografo, che contiene tutti i vertici originali. Si parla allora di *sottografo ricoprente*, e torneremo a palarne in uno degli ultime capitoli, quando andremo a cercare sottografi ricoprenti con proprietà opportune. Il grafo a destra è un sottografo anch'esso, perché è fatto di 4 vertici, che sono un sottoinsieme dei 5 iniziali, e di 6 lati, che sono un sottoinsieme dei 9 lati di partenza. Questo sottografo ha un'altra proprietà interessante: i suoi 6 lati sono tutti e soli i lati che collegano i 4 vertici anche nel grafo di partenza; non ne abbiamo perso nessuno. Questo significa che, una volta indicati i 4 vertici, non occorre indicare quali lati sono interessanti: si prendono tutti. Un simile sottografo si chiama *sottografo indotto* da un sottoinsieme di vertici. Questo è il tema al quale dedicheremo l'esercitazione pratica. Un ultimo esempio: se prendessimo ancora i vertici v_1, v_2, v_3 e v_4, ma aggiungessimo al sottografo il lato (v_1, v_5), che pure appartiene al grafo di partenza, non otterremmo un sottografo, perché i suoi due estremi non sono entrambi parte del grafo di arrivo. Verrebbe a mancare la proprietà che U e X siano fra loro coerenti, che costituiscano un grafo.

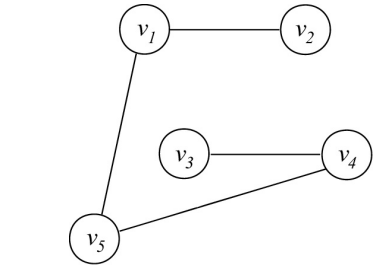
- Un cammino è una sequenza di lati, in cui ognuno condivide un estremo col lato precedente e l'altro col lato successivo (*se esistono*)

$$P = \left(\{v_{\pi_0}, v_{\pi_1}\}, \{v_{\pi_1}, v_{\pi_2}\}, \ldots, \{v_{\pi_{k-1}}, v_{\pi_k}\}\right)$$

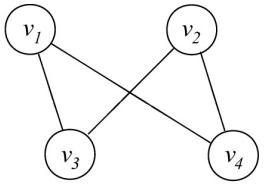
I vertici estremi v_{π_0} e v_{π_k} sono connessi

- Un ciclo è un cammino il cui primo e ultimo vertice coincidono

$$v_{\pi_k} = v_{\pi_0}$$



$P = (\{v_1, v_2\}, \{v_1, v_5\}, \{v_4, v_5\}, \{v_3, v_4\})$



$P = (\{v_1, v_3\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_1, v_4\})$

In un grafo connesso ogni coppia di vertici è connessa da un cammino

Figura 8.6: Connessione

Ora consideriamo altre proprietà che serviranno in capitoli seguenti, anche se non in questo. Definiremo *cammino* un insieme di lati che ha la proprietà di essere

una sequenza ordinata tale che in ogni lato un estremo coincide con un estremo del lato successivo e l'altro estremo coincide con un estremo del lato precedente (si veda la Figura 8.6). Questi lati sono in un certo senso concatenati: formano anche una sequenza di vertici. Si dice che tutti i vertici di un cammino, in particolare i vertici estremi, sono fra loro *connessi*.

La connessione è una proprietà importante per i grafi. Se un vertice a è connesso a un vertice b, per l'assenza di orientamento dei lati, anche il vertice b è connesso al vertice a. Quindi, la connessione è una relazione simmetrica. D'altra parte, se a è connesso a b e b è connesso a c, allora i due cammini concatenati ne formano uno da a a c (non è un problema qui che il cammino possa tornare su sé stesso), e quindi a è connesso a c e la connessione è una relazione transitiva. Di conseguenza, la connessione è una relazione di equivalenza e i sottoinsiemi di vertici fra loro connessi formano classi di equivalenza.

C'è un caso particolare di cammino in cui il primo e l'ultimo vertice della sequenza, gli unici non dotati (rispettivamente) di predecessore e di successore, coincidono. In tal caso, il cammino si chiama *ciclo*. I cicli possono essere utili, ad esempio perché rappresentano percorsi chiusi in una rete stradale, lungo i quali un veicolo deve servire dei clienti, oppure maglie in reti elettriche. In molte applicazioni si cercano esplicitamente cicli con proprietà desiderabili.

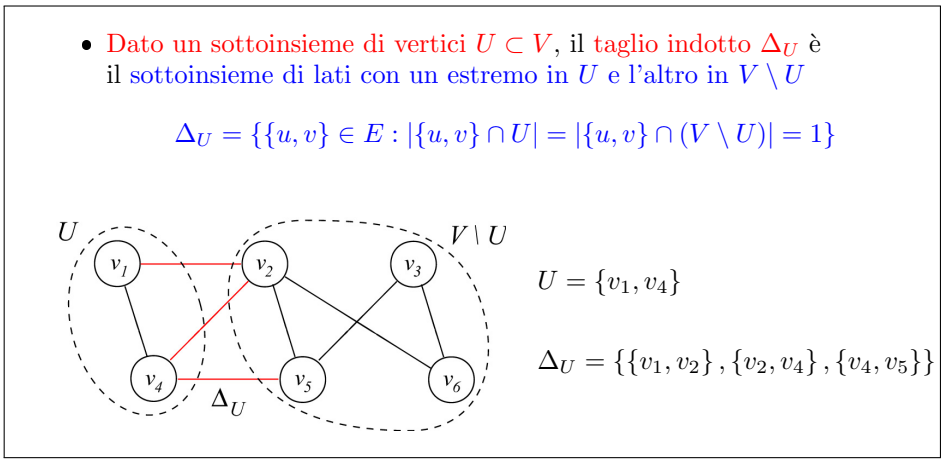


Figura 8.7: Tagli

L'ultima definizione, in realtà, non verrà usata nel corso, ma è comune in molte applicazioni. Dato un sottoinsieme di vertici in un grafo (per esempio, $U = \{v_1, v_4\}$ nella Figura 8.7), abbiamo definito il sottografo indotto. Il sottoinsieme complementare di vertici, $V \setminus U = \{v_2, v_3, v_5, v_6\}$ induce anch'esso un sottografo. Mentre alcuni lati appartengono a uno e all'altro sottografo indotto, altri (indicati in rosso nella figura) non appartengono a nessuno dei due.

stanno né nel primo o nel secondo perché perché hanno un estremo nel sottoinsieme indicato è un estremo nel sottoinsieme complementare questi lati è una caratteristica che stanno a cavallo e costituiscono il cosiddetto taglio taglio indotto dal primo sottoinsieme in realtà indotto anche dal secondo A cosa servono i tagli sono interessanti perché determinano Quali sono le vie dalle quali bisogna necessariamente passare se si vuole andare da uno dei due insiemi l'insieme complementare quindi per esempio vie di passaggio in caso di guerra in caso di commercio in caso di trasmissioni idrauliche o elettriche o qualsiasi altra cosa potete facilmente inventare molte applicazioni Ecco nelle applicazioni è tipico il fatto che ai vertici o ai lati si

associano dei informazioni per esempio valori numerici e allora si può dire che il grafo è pesato può essere pesato sui vertici può essere pesato sui lati può essere pesato su entrambi per esempio se il grafico rappresenta una rete stradale è tipico che sui lati ci siano delle lunghezze Quanto è lunga la strada o dei tempi di percorrenza o potrebbero esserci dei flussi Quante automobili percorrono quella strada in un'unità di te e cose analoghe si possono pensare per tutte le applicazioni che abbiamo visto per esempio nel caso dei giochi si può pensare che una posizione sia vincente per uno dei due giocatori o per l'altro che sia più forte per uno per l'altro dei due giocatori quindi si possono pensare a molte di queste applicazioni Il che vuol dire che dovremo inventarci un modo di rappresentare anche queste informazioni eventualmente Allora qui c'è una carrellata di modelli di problemi basati su grafi che ho pensato di saltare perché in effetti non hanno nessun legame con lo pochi legami con la lezione nessuno con la lezione di oggi pochi con le lezioni successive e richiedono un discreto impegno però può valere la pena di correre queste applicazioni perché quantomeno convincono della importanza della assoluta ogni presenza del concetto di grafo in nella matematica applicata detto ciò passo velocemente al concetto di grafo orientato che è molto simile a quello di grafo non orientato l'unica differenza sta nel fatto che descrive le azioni binarie asimmetriche Quindi quelle in cui non è detto che sia in relazione con BBC e relazione con a Qui c'è una nomenclatura che alcuni differenziano nettamente è molto in modo molto pignolo da quella dei grafi non orientati mentre altre assolutamente confondono io tendo a stare un po' nel mezzo nella direzione del essere un po' più pignolo e 15

8.2 Definizioni per i grafi orientati

Da quella dei grafi non orientati mente altri assolutamente confondono io tendo a stare un po' nel mezzo nella direzione del essere un po' più pignolo e quindi cercherò di distinguere con l'uso della delle parole corrette se il grafico di cui parliamo sia orientato non orientato Però ogni tanto mi sbaglierò e quindi poi mi corregge rossum si usa dire che un grafo non orientato è una coppia di insiemi come quello non orientato Dov'è il primo insieme n o insieme dei nodi è costituito da Oggetti elementari e quindi in effetti non ha nulla che lo distingue da l'insieme dei vertici mentre il secondo insieme quello degli archi a è costituito da coppie di oggetti DN ma queste coppie sono coppie ordinate quindi conta il fatto che una stia al primo posto è uno Steward secondo nel disegnino sotto vedete la rappresentazione grafica nella quale si vede che i nodi sono come sempre cerchi mentre gli archi sono si delle linee che collegano coppie di nodi Ma come applicazione Cerco una freccia non solo si vede abbastanza chiaramente che mentre n 2 in relazione con n 1 n 1 non è in relazione con n 2 perché c'è una freccia solo in una direzione mentre In altri casi Noi abbiamo per esempio farne una delle cinque la relazione In entrambe le direzioni e quindi Però ci vogliono due archi che sono diversi Infatti il grafo è molto simile a quello non ho nient'altro da cui siamo partiti a 5 nodi anziché 5 vertici ma invece di avere nove lati come avevano in precedenza di Latina 13 che non sono 9 e non sono neanche 18 cioè il grafo non è perfettamente simmetrico ma non è neanche ridotto semplicemente alle relazioni che avevamo inizialmente alcune sono doppie detto ciò proseguendo con la nomenclatura e quando abbiamo i due nodi di un latte un arco questi nodi sono sempre gli estremi dell'Arco ma in particolare il primo che nel tuo code il secondo testa come se si trattasse di un animale che va dal primo al secondo nodo Dopodiché Largo esce dal primo nodo ed entra nel secondo e quindi il grado cioè il numero dei nodi degli archi incidenti in un nodo va diviso in un grado entrante e un grado uscente la cui somma è che vado totale avremo ancora dei cammini che però tecnicamente dovrebbe chiamarsi cammini orientati E pretendono che la sequenza di Archi rispetti anche la direzione Cioè non è più sufficiente che ogni altro

abbia in comune un estremo con l'acqua successivo e l'altro estremo col precedente ma bisogna che la coda di una così ha la testa del precedente e che la testa dell'Arco sia la coda del solo a questo punto si può dire che il cammino è diventato Altrimenti se abbiamo degli archi posizionati orientati in direzione opposta Non si può seguire il cammino e Come seguire una strada in cui a un certo punto ci sia un senso unico sbagliato e quindi non si può entrare a questo punto si dice che i due estremi o in realtà qualunque coppia di nodi lungo il cammino gli entrato in cui il primo nodo sia precedente al secondo è fortemente connessa e la connessione forte come avete visto nella lezione di teoria non è una relazione di equivalenza perché non è simmetrica detto ciò in particolare può capitare Anche qui che l'ultimo nodo di un cammino coincida col primo e allora si parla di ciclo o più precisamente di circuito e si può anche parlare del corrispondente del taglio cioè si può andare a prendere un sottoinsieme di no di individuare il sottografo indotto che è formato da tutti gli altri che hanno estremi in quei nodi prende il sottoinsieme complementare individuale sotto mica ha condotto e cercare solo gli archi che sono a cavallo che hanno un estremo nel primo è un estremo nel secondo tutti insieme e questi Archi formano la cosiddetta sezione che per distinguerla dal taglio la quale sezione a sua volta Poi è fatta di archi che vanno da u ha il complemento di oche sono la sezione uscente e Archi che tornano indietro dal complemento di uva su che sono la sezione entrante e questo è particolarmente importante per problemi di flusso per esempio di cui non ci occuperemo in questo corso qui scivolo avanti ricordo che ci sono la quantità di modelli sui grafi orientati per esempio nelle reti sociali non ci sono amici come su Facebook ma ci sono li deve follower follower Lidl come su Twitter e nel molti giochi le posizioni hanno un orientamento c'hai le mosse sono irreversibili i pedoni vanno in avanti Quindi una mossa evidentemente ne deve precedere un'altra nelle strade

Chiedere un'altra nelle strade ci sono le reti stradali Ci sono strade a senso unico e in nella descrizione dei progetti può capitare di rappresentare certi eventi importanti che segnano la fine di un'attività è l'inizio di un'altra con dei Nodi e le attività visto che collegano due di questi eventi la l'inizio dell'attività con la fine si possono naturalmente spontaneamente rappresentare con degli Archi ma ci sono altri modelli di rappresentazione di progetti e quali invece si privilegia la relazione di precedenza fra attività e quindi le attività sono nodi e la precedenza fra attività è descritta da un arco quindi in basso a sinistra noi vediamo un modello Activity on Ark con una Milestone un evento che raccoglie il termine dell'attività i è il termine dell'attività iota e solo allora quando entrambe sono terminate fa partire l'attività K abbiamo lo stessa situazione descritta con un modello Activity on Node in cui le attività sono dei nodi ai jhk e il fatto che hai preceda AK è un arco che collega il primo secondo e fatto che hai J preceda debba precedere AK precede la k cioè che debba terminare prima che AK cominci è rappresentato da un arco che collega i due nodi a questo punto tutta la parte modellistica Eh direi completata tutto quello che dobbiamo sapere e anche di più lo sappiamo possiamo concentrarci sulla tutto dati astratta e sulle implementazioni Dunque struttura dati astratta prima considerazione vi parlerò solo di arti grafiche orientati perché visto che ci sono anche quelli non orientati per un motivo molto semplice e il motivo l'abbiamo accennata in precedenza abbiamo già visto che quando un 2 nodi sono in relazione metrica è un po' come se fossero collegati da una relazione binaria non ordinata Cioè In effetti fra n 1 e 2 n 5 Sì ci sono due archi Ma si si potrebbe anche mettere un lato e sarebbe la stessa cosa Questo non è vero in tutti i casi perché questa relazione è descritta del grafo rappresentato non è simmetrica Ma se avessimo un grafo simmetrico Ecco che allora la scelta tra avere due alti uguali opposti o un lato sarebbe puramente una scelta di gusto viceversa se avete un grafo non orientato come avevamo All'inizio la scelta fra avevano descritto come una linea

ovvero descritto con due archi uno che vale uno che torna sarebbe perfettamente di gusto equivalente quello che sto dicendo è che c'è una relazione biunivoca diciamolo in termini matematici c'è una relazione biunivoca 11 fra i grafi orientati simmetrici e grafi non orientati Questo vuol dire che se ho un modo di rappresentare entrati simmetrici posso usare quel modo per rappresentare i grafi non orientati ma in realtà io andrò a presentarvi una rappresentazione per i grafi orientati generici non solo quelli simmetrici e quindi risolvo un problema più ampio e in questo modo non devo preoccuparmi del problema più semplice Questo è il motivo per cui non parlerò più da qui in poi ti grafi non orientati tuttalpiù andrò a raccontarvi grafico orientati simmetrici grafi non orientati descritti con grafi orientati simmetrici Allora tutto e la tipa stratta noi dobbiamo fornire tutto l'insieme totale poi le operazioni che si possono compiere sugli elementi di questo insieme farò una semplificazione centro dovrò l'insieme dei grafi non di tutti i grafi orientati ma dei grafi orientati che hanno un dato insieme di nodi n perché fisso l'insieme dei nodi per sempre la vita e non avere operazioni del tipo Aggiungi un nodo togli un nodo che sono perfettamente fattibile non aggiungono moltissimo alla comprensione logica dell'argomento ma complicano abbastanza l'implementazione è molto spesso non sono poi necessari cioè di solito uno sa quali sono gli elementi su cui lavora la sua relazione non sempre Tenete presente che comunque l'estensione è banale come siamo passati dei vettori che avevano la dimensione fissata le tabelle che avevano dimensione libera fino a un certo punto così possiamo passare dai grafi su un dato insieme dei nodi ai grafi su insieme di nodi qualunque basta consentire l'aggiunta e la cancellazione di droni mini torniamo a noi abbiamo un grafo con un dato insieme di no di quello che non è Dato l'insieme degli archi le operazioni che io ipotizzò di introdurre sono l'aggiunta dell'Arco a un grafo dato la cancellazione di un

Io ipotizzò di introdurre sono l'aggiunta dell'Arco a un grafo dato la cancellazione di un arco da un grafo dato è una test di esistenza di un arco vediamole una alla volta la giunta dell'Arco Che cosa è rappresentata dalla funzione in Sarco riceve in ingresso un grafo su un dato insieme di nodi riceve in ingresso un nodo di quelli insieme e un altro modo se una coppia ordinata perché il prodotto cartesiano e restituisce in uscita un altro grafo che altro grafo semplice il grafo di partenza a cui è stato aggiunto un arco compreso fra quei due nodi che va dal primo al secondo la domanda che nasce spontanea è Ma se questo arco già c'era Cosa si fa E qui la risposta è a gusto si può reagire dicendo che si è commesso un errore inaccettabile quindi abbandonare tutto oppure invece ragionare in termini di Unione di insiemi va bene Ho aggiunto un elemento un insieme che ce l'aveva già non è cambiato niente Questa è un'altra possibilità una terza possibilità è quella di dire No ma io ipotizzo che in realtà la algoritmista programmatore che chiama questa funzione si è già sincerato che l'alcol non esiste già e ci pensa lui sono tre possibilità e discuteremo poi in seguito mettendo le mani nel codice del minazione dell'Arco funziona in maniera analoga si prende un grafo si prende una coppia di no di ordinata e si produce un altro grafo che non è altro che il primo qui è stato tolto largo che corrisponde alla coppia di nodi data e anche quel problema se per caso l'acqua che devo togliere non c'è cosa faccio potrei mandare un messaggio di errore potrei non fare nulla perché tanto ho tolto qualcosa che non esisteva Oppure potrei richiedere all'utente che questa verifica la faccia lui prima infine proprio da queste due domande che ci siamo posti Prima evidente il fatto che l'utente deve in qualche maniera sapere poter sapere se una data coppia di nodi ordinata corrisponde un corrisponde un arco del grafo e quindi una funzione esiste Arco che prendo un grafo prendo una coppia ordinata di nodi e restituisca in uscita uno se la coppia corrisponde un arco e zero Se non corrisponde è una funzione che augurabile che esista Se no te le funzioni banalissime ce ne saranno molte altre interessanti e in effetti nel corso dell'esercizio aggiungeremo delle funzioni che ci renderemo conto

essere abbastanza interessanti un po' come nel caso delle liste abbiamo visto che poteva valere la pena di avere una funzione di stampo una funzione di ricerca una funzione di spostamento da una lista all'altra lì non l'abbiamo fatto perché l'esercizio era molto sofisticato qui lo faremo perché partiamo da una base veramente Povera un altro esempio che vedremo invece nell'esercizio se il grafo fosse pesato Beh viene spontaneo avere delle funzioni di accesso al che so al peso del vertice 3 o al costo dell'Arco che va dal nodo i dal nodo uno al mondo Quattro ho detto vertice invece di nodulo avevo preannunciato va bene quindi l'idea comunque che se la struttura sintattica tratta è implementata con l'approccio delle strutture dati astratte tutti gli accessi avvengono attraverso delle funzioni può essere vantaggioso No Ne abbiamo già parlato Vediamo le implementazioni no Dimenticavo siccome sono comunque oggetti concreti non oggetti astratti sarà necessario esattamente come per le liste sarà necessario gestire la memoria e quindi sarà necessario avere delle funzioni per creare e per distruggere grafi la funzione che crea un grafo creerà allo crea tutta la memoria dinamica necessarie l'unica informazioni di cui ha bisogno è qual è l'insieme dei nodi perché così crea un grafo che a quelli insieme di nodi di partenza proprio perché io ho voluto bloccarmi su un dato insieme di nodi in generale invece se genera siamo grafi di qualunque dimensione non avremmo neanche bisogno di sapere quelle informazioni che avremo un grafo generico cioè un grafo completamente vuoto e privo non solo di Archi ma anche di nodi invece la distruzione distruggerà il grafo riducendo lo appunto un grafo nuovo giù è vuoto senza Senza niente Senza contenuto Ok Adesso possiamo passare a Elencare le implementazioni nella lezione di teoria voi avete visto in realtà alcune implementazioni e le implementazioni che mi presento non sono perfettamente identiche perché non lo sono perché in realtà ho preferito seguire le implementazioni che useremo nelle lezioni di laboratorio e che più comunemente mi capita di usare nell'attività mia di ricerca che poi è dedicata

Che useremo nelle lezioni di laboratorio e che più comunemente mi capita di usare nell'attività mia di ricerca che poi è dedicata effettivamente a realizzare algoritmi di ottimizzazione su grafi Quindi questo è il settore del quale mi dedico effettivamente al quale mi dedico effettivamente la differenza con le implementazioni che avete visto nella lezione teorica e minima cosa succede succede che la mia prima implementazione quella che vedete indicata come lista degli archi è banalmente la definizione del grafo grafo è un insieme di nodi è un insieme di coppie di nodi che sono alti perfetto mettiamo la presentiamo queste coppie come una lista di coppie seconda la matrice di adiacenza è una delle implementazioni che avete visto in teoria non cambia nulla solo la terza è leggermente diversa ma con leggermente intendo che è equivalente a una delle applicazioni descritte nella lezione di teoria non è esattamente identica delle differenze sono complessivamente minime entriamo un pochino più nel dettaglio come si fa a rappresentare su processori in linguaggio C l'insieme dei nodi ora se il grafo è pesato la cosa può avere delle complicazioni Ma io inizialmente inizialmente ipotizzato che il grafo non si ha pesato cioè che ci interessi solo la topologia i nodi e gli altri però se i nodi non sono strutturati sono solo oggetti elementari L'unica cosa che ci interessa sapere che sono un insieme finito quindi sono in corrispondenza biunivoca coi numeri interi positivi da 1 a n Ma i numeri interi positivi da 1 N si rappresentano nel modo più efficiente possibile con il numero intero n quando io so che i nodi sono 7 non ho bisogno di sapere che chiamano 1 2 3 4 5 6 7 e questo è quello che in pratica Volevo dire cioè la rappresentazione dell'insieme dei nodi è un nome per quanto riguarda gli archi invece abbiamo tre possibili rappresentazioni La prima è la cosiddetta lista degli archi vi dicevo è la definizione gli archi sono coppie di nodi io la presenterò coppie di numeri interi andiamo a vedere un esempio questo grafo qua con 6 nodi e nove Archi Esatto È rappresentabile con il numero 6 che mi dice Quanti sono i nodi e 9 coppie le coppie 12 15 2335 eccetera eccetera fino alla coppia 56 Questa è la mia

prenotazione del grafo comunista di Archi vedete subito un vantaggio enorme la compattezza occupazione spaziale è minima non si può togliere niente da questa rappresentazione senza perdere parte del grafo quindi l'occupazione spaziale eta di a e lineare nel numero degli archi annuncio subito che è una convenzione non universale ma molto diffusa parlando di archi di chiamare n il numero n piccolo il numero dei Nodi e m piccolo il numero degli archi e quindi tendenzialmente Io userò sempre tetta DN e tetta di m per dire lineare nel numero dei nodi lineare del numero degli archi a volte mi ricorderò di spiegarlo altre volte no qual è lo svantaggio fondamentale di questa operazione Finché si tratta di aggiungere togliere Archi si può sostanzialmente ridursi alla Giunta eliminazione di elementi da una lista o da una tabella io ho detto lista ma potrebbe tranquillamente anche essere una tabella eventualmente dipende un po' dalle operazioni che poi bisogna fare su questi Archi bene quindi tempo costante presumibilmente A meno che non vadano tenuti ordinati Insomma ne abbiamo già parlato ma per quanto riguarda la operazioni di verifica di esistenza data una coppia di nodi Esiste un esiste quest'arco Come si può fare questa operazione e abbiamo già fatto qualcosa del genere nella ricerca dei libri nella lista dei prestiti nella lista dei resi e mi ha visto che la soluzione va lineare in a correre esiste l'arco 41 vediamo 12 no 15 no 2 3 No 35136 no 41 l'ho trovato nel caso pessimo bisogna guardarli tutti e quindi la complessità di questa operazione è detta dm30 della cardinalità di Ah quindi è piuttosto inefficiente e questo è il grosso svantaggio della implementazione di un grafo come lista di Archi non passano qui ho anche un grafo pesato sui vertici e sugli Archi Come si può rappresentare una funzione di peso sui vertici essendo i vertici un numero noto il modo più banale è efficiente di rappresentare una con

Peso sui vertici e scendi vertici un numero noto il modo più banale è efficiente di rappresentare una funzione peso sui vertici di avere un vettore lungo come i vertici in maniera che istantaneamente io posso sapere che il peso del vertice due è il secondo elemento l'elemento di indice due del vertice W quindi è pari a 1 Analogamente il costo degli archi e se uno sa qual è l'indice dell'Arco va direttamente sulla elemento del vettore dei corti e siamo a posto un'altra possibilità potrebbe essere quella di avere non delle Mura lista di coppie 12 15 23 ma di Verona lista di terne cioè potremmo mettere le informazioni del costo insieme alle due informazioni topologiche dei nodi estremi può essere più vantaggioso Se per caso non abbiamo gli indici degli archi però per un po' noi non useremo non useremo grafi Pesa quindi questo discorso l'ho fatto ma l'ho accanto nero Qual è la seconda implementazione la seconda implementazione quella matrice di adiacenza ed è una risposta alla inefficienza nella ricerca di un arco dati due estremi Qual è il modo più semplice di determinare un valore dati due da Tuning dato due indici è quello di avere una vettore un vettore con due ingressi c'è una matrice è quello di avere una matrice come questa qua nel quale righe corrispondono ai nodi le colonne corrispondono ai nodi è l'elemento che sta nella riga quarto e Nella colonna 1 vedete che è un numero 19 corrisponde esattamente al costo dell'acqua che va da 4 a 1 questo è pesato sigla fo non è pesato li deve tua basterebbe metterci dei valori booleani degli Uni e degli zeri e quindi la rappresentazione matrice di adiacenza consiste in una matrice con tante righe tante colonne Quanti sono i nodi quindi quadrata e che contiene al suo interno valori booleani se ci interessa solo la topologia quindi uno in corrispondenza le coppie che corrispondono dal 30 a quelle che non corrispondono oppure se abbiamo un grafo pesato ci mettiamo i valori del peso tenendo conto del fatto che però allora in corrispondenza alle coppie che non corrispondono ad Archi bisogna inventarsi un valore fittizio che ci fa capire che l'arco lì non c'è quel trattino deve corrispondere a un qualche numero Qual è il vantaggio montaggio è che se io voglio sapere se esiste come abbiamo fatto prima l'arco 56 prendo la Liga 5 la colonna 6 e istantaneamente so che esiste questo arco in tempo costante lo svantaggio grosso è

che l'occupazione di memoria è enorme perché è tardi n quadro e n quadro in questo caso è 9 sono 9 elementi mentre invece il teta DM che avevamo prima Scusate n 436 perché sei per sei mentre lemmi che avevamo prima il tetto di a Milano prima era soltanto un nuovo elementi finché è piccolo la differenza non è molta c'è già qua riesci ad essere quattro volte tanto ma se pensate anche la po' di 1000 nodi e non è tanto Voi avrete una matrice 1000 x 1000 cioè di un milione di elementi che magari di Alchimia a 2000 3000 4000 quindi siamo fattore che c'è la terza di salto fra L'occupazione di memoria Dai quella che sarebbe strettamente necessario che altro si può dire niente che c'è una terza possibile rappresentazione con la quale chiudiamo la carrellata la terza rappresentazione è quella che io definisco vettore delle fourstar o eventualmente vettore delle pecore sta viene anche chiamata Vittoria delle liste di incidenza o di adiacenza seconda dei testi e degli autori e questa idea è intermedia cioè il DEA Sostanzialmente è di non avere una cella per ogni potenziale nodo di non avere un'unica lista per tutti gli archi ma di avere tante liste precisamente una lista per ogni nodo e la lista e la lista degli arti uscenti Per questo si parla di follower sta perché stare e la stella Cioè è l'insieme degli archi che escono che hanno tutti la stessa origine come i raggi di una stella mentre fuoco due dire che va in avanti Allora qui Vedete un vettore anche se è largo dovete pensarlo come un vettore è il cui primo elemento è una lista 2 12-5-87 Il secondo è una lista Un solo elemento 311 e via dicendo l'ultimo elemento è una lista vuota col classico con la classica rappresentazione a t rovesciata a cosa rappresenta che cosa corrisponde questo al fatto che il nodo sei non sa a chi uscenti e quindi la lista degli altri uscenti è vuota mentre il nodo uno anche scienze

Mentre una lista vuota col classico con la classica rappresentazione a t rovesciata a cosa rappresenta che cosa corrisponde questo al fatto che il nodo sei non sa a chi uscenti e quindi la lista degli altri uscenti è vuota mentre il nodo uno ha anche uscenti 12 15 Allora dentro quella lista ci sta l'acr 12 che a questo 12 elac 15 che accosto 87 perché sentito due 12:05 87 perché se uno volesse risparmiare al massimo memoria riportare l'uno è inutile visto che tutti gli altri uscenti dalla cella numero 1 di quel vettore contengono come nodo di partenza 1 quindi non è strettamente necessario riportarli e gli sono stati tolti ancora una volta il 12 87 eccetera sono costi Perché il gruppo è pesato Ma se il grafo non fosse pesato quella sarebbe banalmente una lista di numeri interi cioè la lista dei nodi adiacenti Qual è lo svantaggio questa rappresentazione che se noi vogliamo andare a cercare il famoso Ako 56 quello che dobbiamo fare entrare nella cella 5 esco averla tutta e quindi alla terzo passo abbiamo un 223 un 4 10 e finalmente 16 17 e ti fa capire che l'acqua 56 esiste e che costa 17 allora qual è la complessità di questa ricerca è tempo costante per arrivare al elemento giusto del vettore cioè la lista giusta e poi tempo proporzionale al numero di elementi di quella lista che è il grado uscente del nodo e quindi nel caso Pessimo è è nemmeno uno o n se ci sono gli auto anelli e teta dnt solito è minore di 30 dm e ricordate potrebbe essere che in un grafo da 1000 nodi gli altri potrebbero anche essere un milione Quindi si può risparmiare anche molto Certo non è tardi uno non è com'è l'occupazione in Spazio l'occupazione spazio se vedete la lista ha esattamente le liste hanno esattamente nove elementi quindi sembrerebbe identica all'occupazione della lista unica Ma non è identica perché questo vettore a una costola sapete che è un vettore di liste a un vettore di puntatori ed eventualmente di sentinelle e poi ci sono tutti gli altri elementi Quindi c'è un overhead Quindi questa espressione qui a questa struttura una occupazione Cat ADM per le varie liste più tetta di n per il vettore delle teste di quelle liste di conseguenza è intermedia fra le altre due che altro svantaggio c'è lo svantaggio e che se io volessi sapere quali sono gli archi che escono dal nodo due lo vedo istantaneamente ma se voglio sapere gli altri che entrano nel mondo 2 Devo guardarmi tutte le liste esattamente come se avessi la struttura a lista di archi e di conseguenza non ho una fantastica efficienza

nel momento in cui mi interessa sapere quali sono gli archi entranti in un nodo per avere la potrei raddoppiare il vettore delle formule sta raggiungendo un vettore delle pecore stare il che mi porta una occupazione memoria doppia perché ogni ora compare in uno e nell'altro esattamente doppia occupazione Ma avrei questa informazione aggiuntiva con la stessa complessità tutto sommato abbastanza ridotta ora a questo punto quello che sostanzialmente faremo è riprendere il problema di individuare il sottografo indotto da un certo sottoinsieme di nodi e cercheremo di impostarlo come problema da risolvere con un programma in C scegliendo una e poi l'altra e poi l'altra delle tre librerie Quindi quello che faremo in pratica sarà risolvere un problema algoritmico è risolverlo con tre strutture dati diverse tra implementazioni diverse della stessa struttura dati astratta non solo quello che faremo è risolvere questo problema con tre algoritmi leggermente diversi tra num problema così facile che consente di usare tanti algoritmi sono molto simili però sono diversi e quindi avrei mai maltrattare algoritmi ad alto livello intersecati combinati con tre implementazioni a basso livello è che ci porterà il 9 possibili soluzioni algoritmi che ognuna con la sua analisi di complessità per cui oltre fare un ripasso dici È un esercizio di programmazione top down realizzazione di librerie eccetera eccetera faremo anche un bell'esercizio di analisi della complessità temporale è spaziale che fino adesso abbiamo abbastanza lasciato un pochino nel dimenticatoio dopo averla introdotta nella sezione apposita

Si è un esercizio di programmazione top down realizzazione di libreria eccetera eccetera faremo anche un bel esercizio di analisi della complessità temporale è spaziale che fino adesso abbiamo abbastanza lasciato un pochino nel dimenticatoio dopo averla introdotta nella lezione apposita Ben ritrovati in questa seconda parte della nona lezione è in tutta la decima lezione che seguirà ci dedicheremo a un esercizio abbastanza lungo e intricato che serve a lavorare con la struttura dati astratta grafo e con le sue implementazioni in particolare considereremo tutte e tre le implementazioni che ho introdotto nella prima parte cioè l'implementazione a lista degli archi quella matrice di adiacenza è quella vettore telefono astaro vettore delle liste di incidenza affronteremo un problema molto semplice addirittura banale apparentemente che è il problema di andare a determinare dato un grafo è dato un sottoinsieme di nodi di questo grafo il sottografo indotto Ti ricordo cosa hai sottografo indotto si tratta pratica di andare a prendere una parte dei nodi del grafo di partenza per esempio qui avevamo i 4 nodi V1 V2 V3 V4 in questo caso l'esempio era su un grafo non orientato e noi dovremmo su un grafo orientato ma non cambierà molto e di questi prenderemo questi nodi E prenderemo tutti gli archi che hanno entrambi gli estremi in questi nodi non un estremo dentro uno fuori o tutti e due fuori devono entrambi gli estremi essere nella sottoinsieme dato questo Individua univocamente sottografo indotto una volta fissato sottoinsieme di di nodi e di conseguenza diciamo il problema di per sé abbastanza semplice da enunciare e da descrivere vedremo che nonostante questo ci sono diversi semplici algoritmi per risolvere questo problema è il contenuto fondamentale della lezione uno dei contenuti fondamentali Sara Osservare in che maniera diversi algoritmi abbiano ciascuno Una complessità che dipende dalla implementazione delle strutture dati implementazione concreta cioè seconda che il grafo venga implementato come lista di Archi come matrice di adiacenza o come vettore delle star e ciascuno degli algoritmi che considereremo assumevano una complessità diversa anche se ovviamente strettamente legata alle operazioni che l'algoritmo richiede vedremo che in realtà il concetto di struttura dati astratta ci in generale ci permette di dire che qualunque algoritmo potrà essere eseguito su qualunque struttura dati su quello che implementazione della struttura dati a patto che ci siano le operazioni ha fatto di consente alle operazioni che l'algoritmo di chiede che però d'altra parte alcune implementazioni si prestano particolarmente bene la verità ve alcuni algoritmi Perché Perché in S implementazioni certe operazioni sono

particolarmente scorrevoli particolarmente efficienti dal punto di vista del tempo ma anche particolarmente facili da realizzare scrivendo codice al punto che in realtà nel momento in cui si concepisce un algoritmo in parte si sta già pensando alla implementazione più opportuna anche se le due cose sarebbero da un punto di vista retratto indipendenti e separabili fra loro Quindi sotto certi aspetti Stiamo dicendo due cose opposte ma che in realtà sono semplicemente due lati della stessa medaglia Allora cattiva e che parte di questa lezione è dedicata al concetto di complessità computazionale che abbiamo abbastanza trascurato negli esercizi precedenti tanto per fare un salto indietro quando abbiamo parlato del esercizio sul gioco dell' Otello nel non solo nel descrivere abbiamo descritto Il problema Abbiamo descritto alla modellazione del problema cioè la scelta di strutture dati con cui rappresentarlo e della algoritmi con cui risolverlo dopodiché abbiamo descritto gli algoritmi modalità top-down e magari ci siamo spinti addirittura a commentare il codice in realtà quello che mancava che mancherebbe in una relazione è l'analisi della complessità di questo algoritmo non l'ho fatta per evitare di sovraccaricare delle lezioni già pesanti che cosa si potrebbe dire della complessità di una parte della esercizio sull'hotel per esempio

Evitare di sovraccaricare delle lezioni già a pesanti che cosa si potrebbe dire della complessità di una parte della esercizio sul Otello per esempio prendiamo all'esecuzione delle mosse e si potrebbe dire che noi abbiamo una sequenza di mosse abbiamo un ciclo che scorre le mosse per ogni mossa dobbiamo andare a valutare se quella mossa È ammissibile o no E che effetto ha abbiamo visto che questa cosa comporterebbe per ogni mossa di andare a vedere in ciascuna delle 8 direzioni verticale orizzontale e diagonale se in quella direzione c'è una sequenza io lo chiamavo filotto di Celle avversarie che godono di certe proprietà punto di essere avversarie contigue di terminare in una cella dello stesso colore del giocatore e non invece di andare a finire sulla sull'esterno della scacchiera e dopodiché se tutto questo è valido Allora vuol dire che la mossa illecita che si può eseguire questo post certo modificazione della scacchiera molto rapidamente tutto questo che complessità potrebbe avere per ogni mossa quindi ci sarà un certo numero di mosse abbiamo 8 direzioni da esplorare e per ognuno di questi otto direzioni dobbiamo scorrere un Cirotto di Celle quanto sarà lungo questo filotto e potrebbe anche essere lungo Zero cioè potrebbe interrompersi immediatamente ma nel caso pessimo andrà da una parte all'altra della scacchiera cioè come partiamo da una cella ben precisa si estenderà Al massimo per altri 7 Celle e dopodiché Dopodiché per ognuna delle Celle così attraversate dovevo fare un certo numero di testa Comunque il numero costante riassumendo abbiamo le mosse per otto direzioni per 7 Celle al massimo in ogni filotto per un numero costante di test sembrerebbe che tutto l'algoritmo sia lineare nel numero delle mosse qualcuno Però potrebbe osservare che siccome ad ogni passo si aggiunge una pedina sulla scacchiera Siccome la scacchiera nel 64 e all'inizio ce ne sono quattro in realtà le mosse non possono essere più di 60 e quindi in effetti l'intero algoritmo è di complessità costante e che è giustificato dal fatto che stiamo parlando di un gioco Comunque sono a scacchiera finita e con una continua aggiunta di mo c'ho il procedimento se vogliamo irreversibile potremmo generalizzare o dicendo Ma allora a scacchiera potrebbe essere nel perenne Beh allora il numero di mosse a diventerebbe eventualmente proporzionale nei quadro allora ogni filotto le direzioni sarebbero sempre otto in filotti potrebbe anche essere lunghi ennemenouno i test sarebbero sempre in tempo costante Insomma se arriverebbe una conclusione approssimativamente n cubo di complessità possiamo fare un'analisi analoga per la gestione della libreria per esempio potremmo fare d'esecuzione delle mosse dire che anche lì abbiamo una sequenza di questa qui non si sa bene di che lunghezza sia sono m sono prestiti di libri che sono inizialmente disponibili ma anche Resi di libri che sono esterni e noi non sappiamo Quanti sono i libri e l'esterno quindi dobbiamo dare

un numero queste mosse m per esempio Dopodiché ogni mossa potrebbe essere una mossa terminale E allora questa è in tempo costante si esegue potrebbe essere una mossa di prestito che va a cercare un libro nello scaffale Quanti libri ci sono sullo scaffale Quanto dura la ricerca eliminare il numero dei libri sullo scaffale quindi Qui abbiamo un termine n dopodiché se il libro viene trovato si va a metterlo nella lista dei reni e questo è un tempo costante Se invece però la mossa e divertiti tuzione e allora se vi ricordate quello che succedeva e che ho il libro andava cercato all'interno della lista dei prestiti perché poteva essere un libro reso che era stato preso in prestito il giorno stesso Quanti sono i libri in prestito e non si sa esattamente ma sono al massimo tanti Quanti erano i libri inizialmente sul sullo scaffale Ti amo un nome un numero a questo dato e potremmo chiamarlo n Però potrebbe anche essere che il libro fosse fuori porta Tanto noi lo cerchiamo Comunque nella lista dei libri in prestito e se non lo troviamo ci fermiamo per cui questa operazione è proporzionale al numero dei Libri in prestito che al massimo n e poi cosa c'è da fare niente prendere il libro peso e metterlo sul bancone anche questa in tempo costante sto andando molto velocemente un po' grezza mente ma capite che la valutazione della complessità di questa esecuzione delle mosse è verosimilmente teta dmn dove m sono le mie

Della complessità di questa esecuzione delle mosse è verosimilmente teta dmtn dove m sono le mosse elencate nel secondo file dei del dei dati mentre n in uno dei libri inizialmente disponibili che sono elencati nel primo file dei dati ok Questo per darvi un'idea in questo problema noi Affronteremo una situazione un po' diversa dalle precedenti non è un problema pratico quindi non c'è una fase di modellazione in cui decidiamo che sto tu dov'è che oggetti matematici usare ci viene detto di usare un grafo il grafo è dato Quindi questa fase si salta e possiamo passare invece direttamente alla fase algoritmica eccetera eccetera Allora cominciamo a ragionare un attimino il file ti dicevo è dato che nella di file già disponibili e dato un esempio che questo fa il grafo 1 questo file descrive un grafo in un formato esternamente Direi autoesplicativo per inciso è lo stesso grafo che compare nei lucidi nelle proiezioni è un piccolo grafo di 6 nodi e di 3 e di 9 archi in un grafo orientato ti ho detto perché i grafi non orientati sono in corrispondenza biunivoca con i grafi orientati simmetrici e quindi qualunque rappresentazione valida per i grafi orientati vale anche per quelli non ho niente Tati e non c'è molto di più da dire qualcosina ma non molto si può risparmiare qualcosa per esempio evitando di raddoppiarli anche se si dovrebbe usare un grafo simmetrico comunque torniamo al nostro discorso il formato dicevo è autoesplicativo nel senso che semplicemente c'è l'elenco degli archi ogni Arco è una coppia di indici sì che rappresentano i nodi sono separati da virgole per chiudere fra parentesi tonde non c'è scritto quanti sono i nodi questo in parte un errore nel senso che evidentemente i nodi toccati D'Archi sono sei in questo caso ma potrebbero tranquillamente essere 7-8-10 anche un numero molto alto poiché esistono nodi non toccati da Archi cioè no di isolati io per semplicità sto ipotizzando che questo programma lavori solo su Craft privi di no di isolati è una semplificazione di conseguenza possiamo usare questo formato molto semplice per descrivere i grafi che e dovremmo caricare e andare a inserire in un'opportuna struttura in memoria Cosa dobbiamo poi fare dovremmo andare a recuperare un sottoinsieme di no di questo tutto insieme di nodi potrebbe stare in un altro file di testo ma ho pensato di fare una piccola variazione chiedendo che il l'elenco sottografo venisse passato da linea di comando noi Scriveremo il nostro programma lanceremo il nostro programma fornendo di come primo argomento il nome del file di testo che contiene il grafo e come successivi argomenti gli indici numerici dei nodi del sottografo maniera da poterlo lanciare molte volte velocemente sullo stesso grafo dando diversi sottografi diversi possibili sottografi questo ci richiederà un pochino di operazioni per recuperare da linea di comando l'insieme dei nodi del sottografo abbiamo già un punto di

partenza che è il file sottografo 0.5 che ha le solite direttive a una solita definizione per la lunghezza delle stringhe a una definizione di tipo ci serviranno vettore di interi dinamici e allora ho definito il tipo vintage che è un puntatore intero usato per rappresentare un vettore poi c'è il programma principale nel quale già predisposto una scaletta che decompone in modalità top Town il nostro problema generale in sotto problemi e poi sotto andremo a scrivere le procedure secondarie quindi la prima fase che ripropongo di fare eventualmente arrestando il procedendo È quella di andare a riempire questo main in Conde le opportune funzioni cosa si tratta di fare si tratta di leggere da linea di comando il nome del file che contiene il grappo e anche l'elenco degli indici dei nodi del sottografo Dopodiché dobbiamo caricare il telefono dal file che abbiamo detto prima e poi dobbiamo stamparlo a video e stampare a video il fotografo che è indotto dagli indici dei nodi sul grafo che Abbiamo caricato dal file Tutto abbastanza semplice ovviamente richiede che si si decide in qualche maniera Come rappresentare gli indici dei nodi del sottografo e Come rappresentare il grafo Allora prima di Gli indici dei nodi del sottografo e Come rappresentare il grafo Allora prima di procedere per i fatti vostri vi indicherò che noi useremo per il grafo la implementazione a Matrice di adiacenza Infatti ha già incluso il wpt una libreria che adesso andiamo Qui adesso andiamo a dare un'occhiata mentre per l'indice dei nodi del sottografo la più semplice struttura che si possa pensare a quella di un vettore che contenga un insieme di numeri interi che sono questi indici e quindi bisogna sapere quanto bisogna locale questo vettore anche sapere quant'è lungo potremmo definirlo vettore potremmo definire la tabella in realtà una tabella la cui dimensioni non cambia Durante l'esecuzione dell'algoritmo per cui non è rigorosamente una tabella ogni tanto la chiamerò tabella perché si potrebbe anche pensare che questo sotto insieme di nodi possa eventualmente allargarsi e stringersi non succederà nel nostro algoritmo per cambi molto C'è un vettore una tabella dimensione fissa sono la stessa cosa allora la libreria grafo emea Cosa contiene contiene la definizione di grafo l'implementazione con matrice di adiacenza del grafo che abbiamo già visto cioè numero dei nodi n numero degli archi m e matrice booleana di adiacenza che per ogni riga e per ogni Core che ha le righe corrispondenti ai nodi le colonne corrispondenti ai nodi destinazione e in ogni cella contiene uno se c'è l'arco dalla riga alla colonna e zero invece ce l'ha ancora non c'è avremo delle procedure per creare e distruggere il grafo e avremo tre procedure fondamentali che devono inserire nuovi al chi è questa la useremo pesantemente nel caricamento da file no preciso per cancellare Artic in realtà non ci servirà è una procedura per valutare se una data coppia origine-destinazione effettivamente corrisponde a un arco oppure no all'interno del grappo dato e questa funzione ci servirà abbastanza pesantemente per risolvere il nostro problema Quindi direi che a questo punto se volete potete provare a restare e a riempire Con eventualmente corpi vuoti queste funzioni io lo farò Dopodiché cominceremo a riempire invece le funzioni stesse poi nella parte finale di questa lezione che si interrompe Rai il resto verrà svolto nella prossima andremo a riempire i corpi di queste cinque funzioni della libreria grafo che sono attualmente vuoti il che ci permetterà di andare a quantomeno a caricare il grafo che abbiamo visto e di andare a stamparle maniera da verificare che il caricamento dei dati sia andato a buon fine Allora la prima cosa che dobbiamo fare e interpretare la linea di comando solita funzione che si ritrova praticamente sempre e che consiste nell'andare a decifrare a partire dalla zia dalla Grecia dal numero di argomenti sulla linea comando e dagli effettivi argomenti ciò che dobbiamo fare che cosa ci troveremo su questi argomenti abbiamo detto che la linea di comando deve fornire il nome del file nel quale sta il grafo quindi il nome del file dei dati questo sarà una stringa di caratteri vettore costante di caratteri che andremo a definire con la solita dimensione Rolex per non andare a immaginare chissà che cosa dopodiché però dovremmo andare a leggere questo vettore dinamico di indici dei nodi Caprioli non sappiamo quanto è lungo e quindi andrà allocato qui dentro come abbiamo

già visto in una esercizio precedente questa funzione delle restituisce un vettore che viene allocato dentro Comunque ho detto che glielo dobbiamo passare per indirizzo Diamogli il nome lo chiamerò è sempre dire sottoinsieme e dobbiamo anche sapere quanto è lungo e la lunghezza la chiamerò NS numero dei nodi nel sottoinsieme S ovviamente ci servono le relative variabili S che cosa esce un vettore dinamico Quindi è un contatore intero e lo chiamo vinta per sottolineare che è un vettore e non è un puntatore di per sé Dopodiché RnS a questo punto solito meccanismo a tre fasi abbiamo realizzato la chiamata Possiamo aggiungere in alto la dichiarazione e poi andremo a mettere sotto la definizione questa sera una funzione void riceve il numero intero di argomenti

Che riceve il numero intero di argomenti da linea di comando e il vettore delle stringhe poi avremo la stringa in uscita la stringa con i falli i dati sul nome del file dei dati avremmo questo Vinz puntato PS avremo infine il numero di elementi N S puntato perché sono i risultati sono dei finti dati che sono passati per l'indirizzo terza fase andiamo a realizzare la definizione momentaneamente col corpo vuoto cosa facciamo dopo che richiama il grafo da file si arriva una bella funzione cavità grafo la quale riceve dalle file del dati il grafo stesso e come si fa a mettere qualcosa in un grafo nella struttura della Foce l'abbiamo e qua nell'altro file che abbiamo già incluso quindi molto semplicemente possiamo definirci dichiararci un grafo g e possiamo decidere che questo sarà il risultato della funzione carica grafo quindi glielo passiamo per indirizzo Questa è la chiamata Possiamo salire e scrivere la dichiarazione nella dichiarazione forse possiamo essere un po' più specifici possiamo dire che carichiamo il grafo G dal file fai dati e dobbiamo specificare il tipo per cui fa i dati e un puntatore a carattere EG in realtà è un puntatore a grafo Pigi visto che viene passato per indirizzo stessa passaggio qua sotto dove andiamo a scrivere il corpo vuoto della funzione Poi stampa video Allora qui dovrebbe la spontanea tentazione di scrivere sta un po' eccetera eccetera farò una piccola modifica tanto per fare le cose un po' diversamente dalle altre volte Cioè mi prova un problema che ci siamo già posti sul libreria per le liste non è che per caso una funzione che stampi un grafo sia una buona aggiunta alla libreria dei grafi Qui non c'è ma possiamo anche toccarla questa libreria tutto sommato e potrebbe essere utile avere una funzione che stampa i grafi perché potrebbe essere utile va bene tutto perché il formato che abbiamo che ci è stato imposto informato molto standard nella descrizione di grafi E poi perché in realtà nel momento in cui andremo a scrivere iscrivere cosa vedrete che quando cambiamo l'implementazione verrà abbastanza spontaneo cambiare anche l'implementazione della stampa e allora cambiare per cambiare visto che sostituiremo una libreria con un'altra Tanto vale a riscrivere completamente anche la stampa dentro la libreria invece che cambiare la libreria e poi andavi scriverla nel file di partenza Cerchiamo di tenere il file di partenza più fisso possibile vedremo che in realtà questa volta non ce la faremo a tenerlo completamente fisso perché in realtà scopriremo che davvero gli algoritmi sono si realizzabili ciascuna algoritmo e realizzabile in tutte le implementazioni però ci sono dei forti legami per cui effettivamente certi algoritmi si prestano particolarmente essere realizzati in certe concerti implement e quindi diciamo è un po' più difficile tenere i file principale del tutto indipendente dalle dalle librerie che gli si sostituisce diventa uno sforzo che non vale la candela vedremo più avanti Non adesso credo che non sia particolarmente chiaro quello che voglio fare però adesso E va bene Supponiamo che stampa grafo non sia una funzione che scrivo io ma funzioni di libreria Per questo la chiudo tutta minuscola come sono minuscole le funzioni cui una mia convenzione tanto per te cos'è che vogliamo stampare il grafo G attenzione grafo potrebbe essere passato per copia però ultraform oggetto grosso e copiare un grafo mentre si passa a un record di attivazione è fastidioso allora in questi casi si usa passare comunque l'oggetto per indirizzo perché così passiamo una cella lettina che è molto nella è molto efficiente

Quindi in effetti alla funzione Passeremo un puntato adesso dobbiamo scriverlo qua sopra ma abbiamo detto che lo mettiamo nella libreria Quindi in realtà lo mettiamo in che l'acqua è mia H per cui è comunque sopra Cioè dobbiamo andare in grafene h e aggiungere qui sotto la funzione che stampa video Il grafo G questa funzione è una funzione Void e riceve un puntato 1:07:11 Qui sotto la funzione che stampa video Il grafo G questa funzione è una funzione Void e riceve un puntatore a grafo PC non cambia assolutamente nulla Quello che però poi bisogna farli andarlo a mettere da qualche parte dove non qua sotto ma dentro nella libreria grafo emmeazero nella quale andiamo ad aggiungere la nostra funzione con il corpo vuoto ok Questo verrà poi sistemato dopo torniamo a noi ci manca l'ultima funzione è quella che stampa in realtà costruisce e poi stampa video e sottografo indotto e ne abbiamo una stampa sottografo indotto la quale riceve il grafo che dobbiamo da cui dobbiamo partire anche qui e lo passiamo per l'indirizzo non perché debba essere modificato ma per efficienza Dopodiché dobbiamo passare i dati cioè il vettore S questo non va Passato per indirizzo perché è già un indirizzo Noi stiamo passando il contenuto di quell' vettore non andiamo a modificare il vettore e non è sufficiente passare l'indirizzo di un vettore perché il vettore già un indirizzo quindi è già molto compatto e poi ci serve che è il numero dei nodi basta perché non abbiamo risultati è una stampa quindi non ha risultati quindi andiamo in alto a dichiarare il tutto andiamo a stampare a video il fotografo è il sottografo Scusate il fotografo indotto su G o sul grafo G dal sottoinsieme s e d NS Lodi così si capisce un po' di più cosa stiamo facendo è una funzione che restituisce Void che ha come dati un puntatore a grafo un vettore di interi è un intero NS e queste cose andiamo a copiarle qua sotto lasciando per il momento un corpo del tutto vuoto A questo punto quello che possiamo fare andare a riempire le singole funzioni neanche una volta se volete interrompere procedere al riempimento quello che faremo sarà riempire interpretare il comando e carica grafo ci lasciamo stampa sottografo indotto per la lezione prossima facciamo questa prima fase nella seconda fase finale della lezione andremo a riempire le 5 e ormai diventate sei Funziona cine della libreria ok Come si interpreta la linea di comando procediamo anche qui drown visto che deve fare un po' di cose per quello che dobbiamo fare Sarà necessariamente andare a capire un attimino che numero di argomenti sia quello giusto quindi andremo a controllare il numero degli argomenti sulla linea di comando poi andremo a prendere il primo di questi argomenti cioè andremo a determinare il nome del file dei dati e non dovremmo scrivere nella variabile nella struttura opportuna cosa ho fatto Mi sono perso Ok quindi avendo determinato il nome del file dei dati andremo finalmente a prendere gli argomenti successivi e aumenti successivi che cosa faranno prima capire intanto Quanti sono questi nodi sottografo indotto il numero di nodi del sottografo indotto perché ci serve sapere il numero di nodi perché sapendo il numero dei nodi potremmo finalmente al locale il vettore dei nodi del sottografo indotto una volta che lo avremmo allocato possiamo andare a riempire questo Vettore E questo dovrebbe sistemarci in realtà tutta la procedura di interpretazione della linea di comando procediamo Come si fa a sapere qual è il numero di argomenti argomenti non è fisso a priori Però sicuramente di argomenti cioè devono essere ci deve essere il nome del programma che è l'argomento zero e ci dev'essere il nome del file del capo possono non esserci no di sì perché è un caso del genere ma qual è il sottografo indotto da un insieme vuoto di no di un fotografo vuoto Non è assunto Quindi in effetti ci rimettiamo a chiedere che ci siano almeno 2-3 argomenti se di argomenti ce ne sono meno di 2 ma invece qui 1:12:00 Argomenti se di argomenti ce ne sono meno di 2 da Invece qui la cosa non funziona e quindi andremo a protestare per segnalare che c'è un errore nella linea di comando e così potrebbe anche spiegare come dovrebbe essere informato ma ci siamo per adesso questa volta non lo facciamo dicendo il formato corretto è nome del programma nome del nome del file i dati elenco della lista dei nodi Una volta fatto questo il nome del file dei dati c'è sta in Albi uno si tratta banalmente

di andare a copiare questo nome rendendola tardi uno e andando a copiare in file dati con la solita funzione sto incopy Ok adesso quanti sono i nodi del grafo indotto e se abbiamo detto che i parametri In tutto sono laici e i primi due sono il nome del programma che Stiamo lanciando è il nome del file dei dati è chiaro che tutti gli altri in teoria dovrebbero essere invece dovrebbero essere i nomi dei nodi del fotografo Quindi qual è il numero di questi nodi E N S puntato da pns e questo non è altro che ac-12 adesso possiamo al Locarno il vettore hai detto che si chiama TCS dobbiamo assegnare a un vettore di interi il risultato della chiamata di culloch assegnato pns più uno perché fa un vettore che parte da uno Eva numero totale questa per convenzione che segue in tutto il corso è la dimensione di ciascun elemento è la dimensione di un intero conviene sempre andare a verificare che la locazione sia andata a buon termine e quindi se per caso la locazione è fallita cioè il vettore Nullo possiamo andare a stampare un qualche messaggio di errore e a uscire messaggio di avere questa volta sarà un errore nella allocazione del vettore dei nodi del sottografo indotto altro questa volta dobbiamo riempirlo come lo Riempiamo devi andare a prendere due av3v 4 eccetera e metterli in S1 S2 S3 quindi gli indici sono diminuiti di 1 Come si fa il ciclo for che parte da uno Evra fino a * pns ho chiamato P l'indice tanto per riferirmi a parametri Ma possiamo prendere un nome qualsiasi e si tratta di andare a prendere Questi elementi andarli a scrivere Dove andavi a scrivere in psdp Dov'è * PSV ha messo fra qua fra parentesi tonde perché l'operatore quadre ha una priorità che supera quella dell'asterisco e quindi creerebbe dei pasticci 15 aprile eseguire l'asterisco su PS per ottenere as e poi seguire le quadre quel problema è che contiene delle stringhe mentre noi dobbiamo mettere queste cose in un numero intero qui esiste una funzione a tui che fa la conversione da stringa a intero io tendenzialmente preferisco usare una funzione forse più pesante ma che fa un po' più di analisi e la funzione s-scan F che somiglia la scala f ma in sostanza lavora su una stringa Qual è la stringa la stringa è hard di dov'è tipi più uno perché mi han detto che si parte dal 22 23 24 in corrispondenza più alla $1 + 2^2 + 2^3$ e in questa stringa io vado a cercare che cosa vado a cercare un numero intero Se lo trovo bene se lo trovo lo vado a scrivere e commerciale perché passo per indirizzo questa cella in cui devo scrivere e bene e il la funzione Scan F mi dice che ha trovato un oggetto corretto con il risultato è 16 risultato uno Siamo contenti se vedi il Tato non è uno non siamo contenti vuol dire che da linea di comando non abbiamo passato dei numeri interi Ma magari le passato nelle parole delle altre cose e questa volta c'è un errore nel formato della linea di comando potremmo anche spiegare un po' meglio 1:16:57 Potremmo anche spiegare un po' meglio che genere di E vabbè ma direi che possiamo tranquillamente fermarci qua Ok questo punto mi ha sistemato l'interpretazione della linea di comando Come facciamo a caricare grafo può valere la pena di fermarsi un attimo fare una ri compilata e compilata di che cosa hai compilata del file sottografo 0 il quale fare il da solo non basta se proviamo semplicemente a compilarlo avremo degli errori perché c'è la funzione stampa grafo qui dentro che non è dichiarata quindi dobbiamo compilare contemporaneamente anche la nostra libreria la quale libreria si chiama grafo e mia zero.com adesso effettivamente il tutto con Fila ovviamente non succede nulla nel senso che se scriviamo semplicemente grafo Dovrebbe esserci un errore non mi spieghi esattamente perché non protesti Se invece andiamo a passare quantomeno il nome del file questa volta c'è un altro perfetto L'errore che in realtà Arci deve essere maggiore o uguale a 2 e 4 è minore di 2 che sei un pasticcio Adesso possiamo ricompilare mi aspetto che la cosa funzioni se non scrivo nulla c'è un errore se scrivo soltanto il nome del file in teoria la cosa potrebbe anche andar bene se scrivo dei numeri 123 124 va bene se invece di scrivere dei numeri invece di scrivere i nomi andiamo a scrivere delle parole Ecco se una volta informato quindi siamo abbastanza contenti di come funziona questa cosa anche se non fa ancora nulla come facciamo a caricare il grafo Allora per scaricare il grafo dobbiamo andare ad aprire questo

file che è un formato molto semplice Ora il problema è che si noi dobbiamo andare a caricare il grafo ma il grafo andrà in qualche maniera ha creato e la libreria crea grafo ha una funzione che richiede il numero dei nodi perché abbiamo deciso che dobbiamo sapere quanti sono i nodi per semplicità Noi non sappiamo Quanti sono i nodi e qui non c'è scritto quanti sono i nodi però sappiamo che non ci sono No di isolati quindi i nodi ammesso che non ci siano dei buchi anche questo Diciamo che ve lo garantisco il i nodi sono tanti Qual è il massimo numero che si legge qua dentro per cui leggeremo tutto questo grafo andando la prima volta a cercare Qual è il massimo indice questo ci dirà Quanti sono i nodi potremmo creare il grafo creato il gruppo potremmo tornare a leggere gli altri aggiungendogli uno per uno stesso chiediamo per fargli quello che ci proponiamo di fare in sostanza quello che si può paghiamo di fare è di ovviamente aprire il file e alla fine dovremmo ovviamente chiuderlo poi dovremmo andare a correre il file e lo scopriremo una prima volta per contare i nodi del grafo e salva il massimo indice a questo punto vi torna all'inizio del grappo con all'inizio del file e Coreglia inserendoli nel grafo questo è tutto quello che dobbiamo fare tutto veramente semplice in buona parte l'abbiamo anche già fatto per aprire il file abbiamo bisogno di una variabile contatore per esempio questa variabile non fa altro che prendere il risultato di F Open applicato a file dati e il file lo apriamo in lettura Dopodiché Dobbiamo controllare che la lettura sia andata a buon fine e se è fallita diremo che c'è stato un errore nel apertura 1:21:43 E se fallita diremo che c'è stato un errore nel apertura del file e magari scriviamo il nome del file Ok Qua in fondo andremo a chiudere i file così non ce lo dimentichiamo Dopodiché per contare i nodi del grafo dobbiamo leggere gli altri come si fa a leggere gli archi dunque ci sono tanti modi di farlo però qui la struttura è piuttosto semplice Ognuno di questi anni che si apre con una (in teoria potrebbe esserci degli spazi Prima non dobbiamo dare per scontato che si parte subito con una (quanti spazi magari zero come qui magari un numero qualsiasi non è un problema perché sapete che le stringhe di formato consentono di rappresentare Zero Più spazi con un semplice spazio Comunque poi ci vuole una tonda poi ci vuole un numero intero magari preceduto da spazio che facevo la, magari preceduta da spazi un numero intero e via e viene via Come si fa a scrivere tutte queste belle cose è abbastanza semplice in realtà quello che dobbiamo fare e dire non sappiamo Qual è il numero dei nodi Quindi per adesso diciamo che è zero che ce l'ho lo chiamo Dopodiché finché riusciamo a leggere roba dal grafo cioè finché riusciamo a leggere dal file AFP che cosa una qualche opportuna stringa di formato andremo avanti Che cos'è che dobbiamo leggere da stringa di formato Allora dobbiamo leggere eventualmente spazi seguiti da una (seguiti eventualmente da spazi da un numero intero Qui non metto gli spazi perché per cento di voi già dire che si possono essere degli spazi all'inizio invece li metto spazio, perché se mettessi, subito vorrebbe dire che poi immediatamente la virgola attaccata al numero intero poi faccio un altro percento.di o non fa male Se mettete lo spazio anche qua semplicemente non è necessario spazio parentesi chiusa e dovrebbe essere a posto cosa stiamo leggendo due numeri interi ti interessano i numeri interi Sì perché Perché l'indice Massimo fra quelli che leggiamo è il numero dei nodi Allora come si chiama questi due numeri di solito no deve non chiamarti io iota ma io in questa esercizio userò le lettere o è di origine destinazione quindi qui andiamo a leggere l'intero eleggiamo l'intero di se tutto questo avuto successo abbiamo letto due numeri interi Quindi fintanto che ne leggiamo due andiamo avanti avanti avanti a ogni passo cosa facciamo ogni passo abbiamo letto o ID è il massimo fra questi è il massimo fra tutti vario e di diventa il numero dei nodi n finisce per caso o è maggiore di n Allora n diventa ho E se invece di è maggiore di n Allora m diventa di e questo è tutto quello che dobbiamo fare per determinare il numero dei nodi del grafo se valgono le ipotesi che abbiamo fatto a questo punto possiamo tornare indietro nel file stesso cioè chiamare Rewind sul file e possiamo andare qui sto dicendo già inserire gli archi nel grafo ma non abbiamo creato il gruppo quindi ho

dimenticato un piccolo commento Dobbiamo creare un grafo vuoto.in nodi visto che adesso sappiamo quante n Come si fa a creare un grafo vuoto di n nodi abbiamo la nostra funzione Cina momentaneamente vuota ma poi la riempiremo e ci basta copiarla creiamo il drappo di n nodi e questo gruppo si chiama Pigi è la variabile PG che andiamo a esportare Adesso possiamo scorrere gli archi nuovamente come facciamo scorrere gli archi come abbiamo fatto qui con lo stesso pezzo di codice soltanto che questa volta Stiamo leggendo o è di non per calcolare Quanti sono i Quanti sono i modi ma lo stiamo leggendo per andare a inserire gli archi Come si fa a inserire un arco c'è la funzione Insalaco che prende i due estremi e va a inserirli nel grafo dato quindi abbastanza semplicemente andremo a inserire l'arco o di nel grafo PG e questo in effetti è tutto fate non servono neanche le parentesi grafe per cui a questo punto noi dove 1:26:42 Di nel grafo PG e questo in effetti è tutto fatti non servono neanche le parentesi grafe per cui a questo punto noi dovremmo aver caricato tranquillamente inserito e grafo Proviamo a vedere se sintatticamente il tutto funziona non farà assolutamente nulla di interessante Perché Perché non abbiamo le procedure di creazione di inserimento e via dicendo però quantomeno possiamo vedere che la sintassi corretta ci sarebbe stampa sottografo indotto ma abbiamo detto che la facciamo dopo procuriamoci Adesso una controprova del fatto che questo pezzo di codice funzioni cioè Riempiamo la libreria Quindi a questo punto Io suggerisco di fermare provare voi a implementare questa libreria che è molto semplice tenendo conto del fatto che noi vogliamo creare distruggere un grafo inserire un nuovo Arco cancellare un nuovo Arco valutare se esiste un arco e stampare un grafo ipotizzando che il grafo sia in sostanza matrice di adiacenza Quindi è una implementazione di una banalità assoluta molto molto semplice Allora cerchiamo di creare questo famoso grafo Che cosa vuol dire creare un grafo dato il numero dei Nodi e dobbiamo andare Prendiamoci la lista della spesa dobbiamo andare a valorizzare e dare un valore ANM nella matrice di adiacenza queste sono le tre campi che potrebbero aver bisogno di un valore Quant'è il numero dei nodi n viene da fuori e il valore n Quant'è il numero degli archi me0 perché abbiamo detto che questo telefono che creiamo n nodi nessuna Arco perfetto e per quanto riguarda la matrice Amatrice deve essere una matrice n x n booleana Anzi una matrice indicizzata da1n da1n quindi l'altra una matrice dnp1 righe dnp1 colonne Come si fa ad allocare una matrice dinamica dovresti ricordarmelo da una delle vecchie Des delle vecchie elezioni Comunque in sostanza si tratta di prendere questo puntatore matrice adiacenza e assegnargli un doppio Inter realizzato dalla funzione culloch a Quale funzione culloch che cosa va a prendere n + 1 abbiamo detto perché andiamo da zio n n + 1 g n + 1 o se volete potete anche scrivermi più 1 visto che abbiamo già la variabile qui potrebbe essere più semplice che cosa deve essere questo questi devono essere la costola della matrice che deve essere il vettore dei puntatori al detto che corrispondono alle righe in cui dentone andiamo a riempire dei puntatori a intero Dobbiamo verificare che questa locazione sia andata a buon fine e qui da qualche parte avevamo una allocazione che fosse si può sfruttare per non dover riscrivere Il tutto sempre pericoloso ma otteniamo un po' sotto controllo il copia-incolla quindi andiamo a vedere Se pigi matrice 200 è nullo in questo caso c'è stato nel lavoro e nella locazione della matrice di adiacenza del grafo c'ho fatto nel possiamo abbiamo solo la costolina non abbiamo le righe quindi dobbiamo elemento per elemento andare a riempire le righe Come si chiamano le righe detto che le chiamo o poi li andiamo a fare però che va da 1 fino a n il solito pgn volendo andiamo a darlo calci una per una le righe Quindi abbiamo qualche cosa di molto simile a questo blocco di istruzioni e metto dentro le parentesi grafe e dobbiamo andare ad allocare la riga o esima dpg matrice adiacenza che restituisce un puntatore intero la lunghezza è sempre n + 1 e gli elementi sono degli interi qui è una causa di enormi errori se andate a sbagliare la dimensione dell'intero o a sbagliare la definizione dei tipi quindi bisogna Prestare particolare attenzione e ovviamente bisogna controllare che

la matrice di adiacenza o non si sa nulla perché se è nulla sostanzialmente avete fatto un errore nella locazione della riga di un'opportuna riga della matrice di adiacenza se vogliamo possiamo anche stampare qual è questa riga e la riga o a questo punto abbiamo una Fanta 1:31:25

Della matrice di adiacenza se vogliamo possiamo anche stampare qual è questa riga e la riga o a questo punto abbiamo una fantastica matrice di adiacenza locata della Dimensione giusta e siccome abbiamo usato culloch è tutta piena di Zeri Cosa vuol dire questo vuol dire che non ci sono alti e quindi siamo contenti perché effettivamente non volevamo nessuna Arco e quindi in effetti cui possiamo anche tutto sommato fermarci Questa è l'implementazione della costruzione della creazione di un grafo Come si fa a distruggere un grafo in maniera opposta noi dobbiamo andare a cancellare una per una due righe e poi a cancellare la costola a quel punto abbiamo dei tre campi che potrebbero avere dentro dei numeri qualsiasi sarà il momento opportuno andare a segnarci che non ci sono nodi che non ci sono Archi che non c'è una matrice è importante procedere in ordine inverso Cioè se io seguo via la costola poi non sono più in grado di usarla per andare a via locare le righe quindi devo farlo in ordine assolutamente inverso fine avrò bisogno per prima cosa di scorrere tutte le della famosa matrice e queste righe una per una andranno deallocate 1 Free dpg matrice adiacenza li ho quando abbiamo già locato tutte le righe possiamo andare a del locale la matrice nel suo complesso c'è la famosa costola a cui sono appese una per una tutte le righe e a questo punto già che ci siamo per chiarire ulteriormente il concetto possiamo andare dire che la variabile il campo matrice adiacenza è un puntatore non deve continuare a puntare a quella area di memoria che è stata liberata lo mettiamo a nulla ma lo mettiamo nulla Adesso non prima di averlo bloccato se no perdiamo della memoria che è ancora lì impegnata che non è più utilizzabile Analogamente possiamo andare a prendere il numero di Arch m e settarlo a 0 è il numero di nodi n 70 A questo punto la No il nostro grafo è uno scatolotto con dentro tre variabili tutte nulle e si capisce abbastanza bene che quella un grafo completamente distrutto Andiamo sulle cose un pochino più corposo interessanti come si fa a inserire un arco Allora in realtà è molto semplice perché inserire un arco origine-destinazione vuol solo dire che la Celletta dell'amico vigile e Nella colonna destinazione deve diventare Pavia uno quindi Uno potrebbe dire io prendo PG matrice adiacenza di videochiamate origine test anziché ho i bimbi e li metto uguali a 1 se vogliamo Guardare TV più corretto però questo crea un pasticcio perché noi abbiamo un conteggio del numero degli archi e quindi in effetti il numero degli archi D1 A questo punto si pone un problemino problemino Eh ma non è che per caso quella lì c'era già è qui l'ho scritto perché il chiusa questa libreria si è avvertito non c'è nessuna verifica che la con un esista qui si può fare in vari modi una cosa che si può fare alzare la verifica fai la verifica poi se si scopre che la coda esiste Si può reagire dicendo No non va bene usciamo abbandoniamo il programma che è un po' drastico un'altra cosa che invece si potrebbe fare e forse sarebbe più note si potrebbe fare invece è quella di dire Ma se noi Fissiamo una quel a corteggiare tutto sommato non è un problema lo lasciamo già fissato Quindi potremmo dire L'unica cosa che non dobbiamo incrementare il numero degli archi sennò questo diventa incoerente quello che potremmo dire che se per caso la matrice 1:36:00 Non esista selaco per caso esiste Io semplicemente non faccio nulla che può essere una idea saggia però può anche essere un problema cosa facciamo nella cancellazione dell'Arco il criterio assolutamente lo stesso Cosa facciamo per cancellarlo dobbiamo diminuire di 1 il numero degli archi dobbiamo andare a settaria falso la Celletta e però tutta questa cosa la facciamo soltanto se la Celletta è vero Perché se per caso invece era già falsa non cambia assolutamente nulla quindi piuttosto banale l'implementazione di aggiunte cancellazione di Archi altrettanto banale è decidere se un arco Esiste un esiste Come si fa a capire se un arco esistono esiste Si va a leggere la

matrice di adiacenza e quindi si restituisce l'elemento test la matrice di adiacenza questa è talmente banale che c'è da chiedersi se davvero valga la pena di usare una funzione Anzi è chiaramente molto inefficiente al locale un record di attivazione copiando tre campi semplicemente per accedere a un campo di una matrice è un elemento di una matrice Diciamo che questa cosa ha senso semplicemente perché vogliamo una struttura dati astratta e noi vogliamo del codice che non dipenda da come effettivamente implementiamo questa funzione esiste Arco Infatti poi vedremo che implementeremo la il gelato come lista di Archi ma la funzione esiste funzionerà esattamente nella stessa maniera di prima ci vediamo un attimo con la stampa del grafo Come si fa a stampare un grafo lo vogliamo stampare in questo formato quindi dobbiamo scordarci tu chi è uno per uno andare a scrivere origine-destinazione si tratta abbastanza banalmente visto che abbiamo una matrice Discover e tutte le righe tutte le colonne cioè tutte le celle E a seconda dei casi andare a scrivere o non scrivere la coppia tra parentesi tonde e quali sono i due casi è il caso in cui ci sia largo il caso in cui non c'è l'arco in sostanzialmente si tratta di fare un ciclo Ford su ho chiamata 1 fino al numero di nodi n incrementando ogni volta o poi ci sarà dentro un ciclo questa volta fatto sui nodi di destinazione e ad ognuna di queste cose andremo a verificare se esiste l'arco o di nel grappo ci qui possiamo dire esiste anche = Oppure possiamo anche semplicemente scrivere il distacco perché tanto è un operatore booleano quindi viene il suo risultato dire interpretato come avevo falso e se per caso la coesiste bello e andremo a stamparlo quindi andremo a stampare a video Nel formato (numero intero, numero intero) spazio andremo a scrivere i valori o è di questo In effetti dovrebbe semplicemente chiudere la nostra implementazione della funzione stampa Arco se tanto ci dà tanto a questo punto possiamo ricompilare tutto probabilmente senza errori possiamo andare a prendere il gelato facciamo questa volta Un sottografo indotto totale quindi dovremmo stampare in teoria tutti quanti i lati e vediamo che gli effetti ci sono tutti quanti i 9 i lati sono in realtà nello stesso ordine perché erano già in ordine lessicografico cioè ordinati per il primo estremo e poi per il secondo Ma in effetti possiamo anche tranquillamente cominciare a togliere dei nodi possiamo mettere per esempio soltanto quelli pari 246 e ovviamente non cambia nulla perché la verità è che non abbiamo implementato la procedura di stampa del sottografo indotto Quindi quello che effettivamente stiamo facendo è eseguire all'interno del main questa funzione stampa grafo che stampa le interrogavo questo direi conclude per ora l'esercizio quello che faremo la prossima volta sarà andare a riempire stampa sottografo indotto e Vi propongo tre diversi algoritmi per farlo tre algoritmi ad alto livello algoritmi quindi diciamo su oggetti matematici li implementeremo usando le librerie le funzioni di questa libreria e andremo a discutere la complessità computazionale spaziale e temporale di questi tre algoritmi Dopodiché Ci proveremo il problema di dire ma è il meglio del meglio e se per caso non cambiassimo la nostra struttura dati Now Think azione la struttura di non si potrebbe migliorare lì si aprirà una serie di altri discorsi però per ora concludiamo qui

0:00:00 Buongiorno Benvenuti alla decima lezione del modulo di laboratorio del corso di algoritmi la decima lezione del seguito della nonna è come penso ricorderete abbiamo avviato la realizzazione di un lungo esercizio implementativo dedicato a realizzare una serie di possibili algoritmi e di possibili implementazioni per il problema di determinare il sottografo indotto su un dato grafo orientato qui esempio su un grafo non orientato una volta dato un sottoinsieme di nodi di questo grafo abbiamo realizzato nella prima parte un po' di procedure ad alto livello decisamente siamo andati a costruirci una procedura per interpretare la linea di comando nella quale Abbiamo ipotizzato che si passasse come primo argomento il nome del file che contiene il grappolo questo semplice è abbastanza autoevidente formato di lista diarchia chiusi Fra che sono coppie di indici di nodi separati da virgole racchiusi

fra parentesi tonde e poi invece Daniela di comando passiamo un elenco di indici di nodi che va andiamo a mettere all'interno di un vettore dinamico essere che viene allocato dinamicamente dentro la funzione interpreta linea comando è che a una certa lunghezza NS prima di cominciare proseguire e prima di darvi un quadro generale della lezione devo fare una piccola correzione per fortuna una correzione che non avrebbe avuto nessun effetto Però è una correzione comunque pericolosa che riguarda la funzione crea grafo nella quale la matrice di adiacenza che è dichiarata come doppio puntatore a vuole Anna cioè matrice di valori logici in realtà era stata allocata come doppio puntatore a interi e qui avevamo Dopodiché anche qua sotto c'erano gli Inter in realtà bisogna scrivere boolean perché dico che il problema il lavoro e non aveva nessun effetto perché in realtà abbiamo dichiarato il tipo boolean come sostituto come alias del tipo Inter ma in realtà questo è uno dei classici e tipici errori che si fanno nella locazione di direttorio di matrice sbagliare il tipo Te l'ho già detto a Lucato Che significa sbagliare la dimensione allocata e e allo Chiama dimensione più grande ancora ancora ce la caviamo Ma se lo chiamo per me sono più piccolo e classico caso in cui poi usando la matrice Speriamo nessuno ci avverte e quando alla fine del dell'algoritmo dell'esecuzione dello chiamo il tutto il programma salta per aria senza spiegarsi il perché ecco questa è una delle classiche situazioni se vediamo che il programma salta per aria specialmente dialogando ma a volte anche usando delle matrici dobbiamo andare a controllare se la diamo a Locate è giusta Quindi se abbiamo scelto il tipo giusto di dato e se abbiamo scelto la dimensione giusta corretto questo errore Io passo a darvi un pochino di struttura generale della lezione perché è parecchio lunga e perché ci sono un po' di cose di impostazione da dire però è quello che in effetti faremo è presentare una serie di algoritmi per la determinazione del sottografo indotto dato un grafo è Dato l'insieme A è un sottoinsieme di nodi precisamente vedremo 3 algoritmi e poi un quarto che non sono pochi e li realizzeremo potenzialmente su tre implementazioni diverse del grafo che sono le 3 descritte nella lezione precedente cioè la matrice di adiacenza alla lista di archi e il vettore delle liste di incidenza ovvero vettore del Fuoco Star D'altra parte anche il sottoinsieme dei nodi può essere rappresentato in vari modi anche questo in tre modi non vado a entrare nel dettaglio adesso e potete ben capire che siccome ogni algoritmo in linea di principio può essere implementato su qualsiasi implementazione delle sue strutture Dati qui rischiamo di avere 4 algoritmi per tremendamente azioni di grafo per tre impianti azioni del tetto insieme dei nodi per un totale di 36 coppie algoritmi implementazione che sarebbe folle che cosa succede che cosa si fa in teoria Cosa si dovrebbe fare in teoria e si dovrebbe fare le analisi di tutte queste coppie che andare a concludere che una di queste detto sì Coppi algoritmo implementazione si presenta come la più promettente o vedremo alla fine della lezione non è detto che ci sia una vincitrice assoluta poiché anche molto interessante Dopodiché dovremmo implementarla in pratica quello che spesso succede che si segue una idea che sembra ragionevole Haven

0:04:52 In pratica quello che spesso succede che si segue una idea che sembra ragionevole avendo fatto una pre-analisi e poi magari dopo aver fatto un implementazione ci si rende conto che si poteva fare meglio allora si torna un attimino indietro si corregge l'implementazione si sostituisce magari venuta mentazione con un'altra se abbiamo fatto una libreria Da cosa è particolarmente indolore ma non completamente Insomma c'è una specie di movimento di retrazione di ciclo di convezione quello che noi faremo sarà ancora diverso perché questo più che essere un esercizio di implementazione è anche un esercizio che da una panoramica Quindi in effetti il nostro obiettivo sarà di far vedere diverse implementazioni anche se avremmo potuto a priori sceglierne una questo vorrei fosse abbastanza chiaro non è quello che facciamo oggi non è il modo giusto di affrontare il problema è un modo che serve a vedere tanti modi diversi non faremo Però le 36 azioni Perché sarebbe

troppo e quindi vedrete che ogni tanto in Todo ho dei nuovi elementi farò delle scelte eccetera eccetera però che cos'è che effettivamente abbiamo fatto quello che abbiamo fatto è partire subito scegliendo un'implementazione per il grafo quello alla matrice di adiacenza e abbiamo anche scelto implementazione per il sottoinsieme dei nodi abbiamo scelto di metterli in un vettore come elenco di indici Perché Perché sembra una rappresentazione è particolarmente compatta mentre quella matrice di adiacenza non è compatta però sembrano implementazione particolarmente semplice e abbiamo scelto con criterio molto euristico e che mostrerà i suoi limiti nel seguito però questa è stata la strategia che abbiamo adottato Allora a questo punto cominciamo a presentare un po' di possibili algoritmi per il problema di trovare sottografo indotto dato un grafo e dato un sottoinsieme di no di un possibile algoritmo lo vediamo su questo documento che è fornito io vedete è sostanzialmente consiste in questo consiste nel dire avere tutte le coppie di nodi o di dell'insieme dei nodi n e per ogni coppia di nodi Quindi abbiamo un ciclo suino di origine è un ciclo fino a di destinazione e per ogni coppia andiamo a verificare tre cose La prima cosa che devi ti chiamo è che il nodo origine stiano e sottoinsieme perché se non sta né sottoinsiemi non si interessa perché guardiamo le coppie perché gli altri sono coppie ne stiamo esplorando un sole insieme per vedere quali sono gli elementi che effettivamente ci interessano e che dovremmo stampare poi Quindi abbiamo tre test dicevo il primo è che l'origine stia nel sottoinsieme dei nodi dati Il secondo è che la destinazione Thiene sottoinsieme dei nodi dati e il terzo è che la coppia sia nel sottoinsieme degli archi a questo punto se tutte e tre le condizioni valgono stamperemo l'arco domanda Quanto ci costa tutto algoritmo possiamo fare una stima astratta a priori senza prendere in considerazione le implementazioni per farla dovremo dare dei nomi a che cosa alle complessità delle operazioni che abbiamo assunto come operazioni di base Però intanto il fatto di scorrere i nodi lo considereremo come un qualcosa che si può fare in tempo costante per ogni nodo Non lo dico neanche ma è abbastanza ragionevole che si possa fare visto che i nodi sono numeri interi che vanno da 1 N D'altra parte ci interessano i tre test appartenenza a s dell'origine appartenenza S della destinazione e infine appartenenza da della coppia ok Come si può ottenere tutto questo ci sono due cicli annidati sapete che i cicli corrispondono delle sommatorie avremo due sommatoria annidate somme Remo però che va da 1 ad n il che corrisponde a questo ciclo esterno una somma per di che va da 1 a n che corrisponde a quest'altro ciclo interno 3 operazioni in realtà quattro che sono il test di appartenenza di Hope as quello di di as queste due operazioni sono concettualmente identici i denti che sono due operazioni di appartenenza due operazioni di valutazione di appartenenza di un elemento ha un sottoinsieme e ho chiamate p.s. per brevità Dopodiché avremo la valutazione di appartenenza di una coppia di nodi a l'insieme degli archi e poi la stampa quanto ci costeranno queste operazioni dipende dalla implementazione Però io direi che immediatamente possiamo liberarci di Tpr perché la stampa di un arco non dipende Tanto dal implementazione forniremo i due stamperemo è il tempo sarà verosimilmente costante 0:09:45 Uniremo i due indici di stamperemo e il tempo sarà verosimilmente costante quindi la parte interessante Sara TS Sara Tia e come potete facilmente immaginare questi due termini che sono delle funzioni cui non l'ho scritto ma qua sotto evidenziato sono funzioni di che cosa è di qualche parametro adimensionale queste due funzioni dipenderanno da Come è stato implementato il sottoinsieme Dei Nodi e da Come è stato implementato il sottoinsieme degli archi avevi direi proviamo a implementare a scrivere il codice relativo a questa questo possibile algoritmo e vediamo poi di fare una valutazione adattata A quale implementazione a quelle che abbiamo appena usato cioè S è un vettore di indici di Lunghezza n s e a insieme il grafo è un rappresentato con una matrice di adiacenza quindi possiamo ripartire da dove eravamo arrivati andare nella nostra funzione stampa sottografo indotto che al momento è del tutto vuota è sostanzialmente realizzare la implementazione che abbiamo definito come algoritmo

Uno cioè la lo scorrimento doppio dei nodi di n e valutazione di appartenenza di po' e d f e d o d Alba seguita dalla stampa questo è più o meno l'algoritmo che dobbiamo implementare cosa dobbiamo fare Dobbiamo scorrere dobbiamo scrivere tutti i nodi che abbiamo detto solo numeri da 1 N sostanzialmente in questo mondo qui in effetti mi ha fatto qualcosa di leggermente più astuto cioè di definire un tipo nodo come se fosse l'intero di conseguenza avremo delle variabili o è di tipo nodo che però in realtà sono sostanzialmente degli interi è semplicemente per far capire meglio che cosa stiamo facendo e faremo scorrere il nodo da 1 fino al valore del numero dei nodi del grafo Quant'è il valore del numero dei nodi del grafo il grafo è* PG. quindi PG freccia renne e poi Passeremo al nodo successivo troviamo esattamente la stessa cosa per le destinazioni e a questo punto ci rimarrà da fare i tre test in cosa consistono i tre test e trovate il consistono nel valutare se un certo modo per questo punto numero intero appartiene a un vettore Questo è lo stesso e questa qui è una lavatrice lavatrice Scusate è il test di appartenenza di un arco a un grafo rappresentato come matrice di adiacenza questo terzo test in realtà è già risolto ce l'abbiamo nella libreria perché abbiamo la funzione esiste Arco il primo No il primo e secondo che cosa facciamo o ci ragioniamo su Oppure diciamo no procedura top Town Supponiamo di avere una funzione che ci chiede che ci dice se un gatto ha un nodo appartiene a un vettore di no di una tabella di no di qui Vi ho già detto io uscirò fra Vettore rigore un vettore perché è costante ma non cambierebbe molto se fosse una tabella servirebbe soltanto il fatto che avessimo allocato un vettore un po' più lungo e avessimo una variabile che ci dice quanti dei nodi quanti degli elementi del vettore sono usati quindi fatemi usare il termine più generico Allora mi sto chiedendo se l'elemento o appartiene alla tabella S che attualmente a Lunghezza NS se questa cosa è vera e se è vera analoga condizione Imposta sul nodo di e infine se esiste Arco quale arco abbiamo detto l'arco origine-destinazione grafo Quindi o dpg Se queste te le condizioni sono tutte e tre verificate Ecco che noi possiamo andare a stampare l'arco Ci serve una funzione diciamo di no diciamo che ci limitiamo a stampare a video che cosa Largo Nel formato richiesto (numero intero, al tuo numero intero chiuso la tonda spazio così non sono appiccicati l'uno all'altro ma posto Quasi Nel senso che Abbiamo ipotizzato l'esistenza di una funzione che in questo momento non esiste e al solito avendo la chiamata possiamo tornare a Monte aggiungere questa funzione 0:14:30 Ci serve una funzione diciamo di no diciamo che ci limitiamo a stampare a video che cosa l'arco Nel formato richiesto (numero intero, altro numero intero chiuso la tonda spazio così non sono appiccicati l'uno all'altro ma posto Quasi Nel senso che Abbiamo ipotizzato l'esistenza di una funzione che in questo momento non esiste e al solito avendo la chiamata possiamo tornare a Monte aggiungere questa funzione a fra le dichiarazioni fai prototipi in cima e poi metteremo la definizione che cosa fa questa funzione questa funzione determina se il nodo o appartiene alla tabella S di Lunghezza n s Questo è quello che fa restituisce un valore booleano usa un nodo o intero adesso qui dipende potremmo anche stabilire che in realtà questa Sì come no da intero sono la stessa cosa dobbiamo anche stabilire se di vendere questa funzione un pochino più generica adattarla funzionari sunter generici quindi Definiamo la presente in questo modo visto che il vettore In fondo è un vettore di interi e che la dimensione è un intero perché l'ho messa con valori minuscoli semplicemente perché somiglia molto a una funzione di libreria Allora potremmo dirci Ma perché non facciamo anche una libreria tabella. H con queste funzioni perché non voglio farla molto lunga e perché per adesso non so se mi servirà veramente un'intera libreria Quindi come abbiamo fatto nel gioco dell'hotel quando abbiamo fatto tutto il problema e poi alla fine abbiamo detto Peppe sono un po' di funzioni per gestire scacchiere facciamo una libreria scacchiera Ecco anche qui potremmo fare lo stesso È sempre una questione di vedere anche in che modalità si intende gestire il tutto c'è una certa libertà quindi andiamo a copiarci di sotto la l'intestazione in maniera da poter aggiungere un corpo vuoto allora Al

momento questo non è un codice corretto perché la funzione dovrebbe restituire un boolean tanto per renderlo temporaneamente corretto dal punto di vista sintattico Diciamo che restituisce per esempio falso Giusto per non dire troppo a questo punto può valere la pena di compilare tanto per vedere se quello che stiamo facendo ha un senso dal punto di vista sintattico e dobbiamo compilare il solito fai sottografo zero con la libreria che stiamo usando che è a questo punto si chiama grafo e mi ha zero e il tutto lo andremo a scrivere in sottografo EX c'è una dichiarazione implicita della funzione appartiene tabella Perché ho prima messo e poi tolto questo diciamo che sto cercando di tenere una certa coerenza nelle convenzioni ma non sempre ci riesco Dopodiché Qui abbiamo un problema secondo il quale stiamo cercando di stampare un intero ma in realtà non li stiamo passando niente Infatti avevano gli stiamo passando ne ho né di Vediamo se questo è sufficiente dovrebbe essere sufficiente Infatti perché bevo che ho i di sono nodi Ma è anche vero che i nodi sono interi e quindi a meno di essere molto pignoli sei molto pignoli potremmo fare una conversione forzata ed esplicita da tutte e due le parti Però tutto sommato Direi che per questa volta non lo faccio Giusto per essere un po' incoerente in maniera esplicita Allora a questo punto dovremmo avere un codice che è sintatticamente corretto potremmo anche lanciarlo ma siccome la funzione appartiene tabella restituisce sempre falso non mi aspetto che si stampi nulla assolutamente c'è mi aspetto che come prima ci sia soltanto la stampa del intero grafo a video Ecco Vi faccio notare che stiamo stampando il grafo e poi stampando il fotografo e in effetti non abbiamo niente che distingue queste due stampe Quindi è un buon momento forse per fare un pochino di cornice Cioè per iniziare la stampa del grafo in qualche modo per esempio potremmo dire che G è uguale mentre invece a posteriori può valere la pena di andare semplicemente perché semplicemente perché credo che nella stampa del grafo non abbiamo messo questa cosa Ovviamente nulla vieta di andare a mettere la capa invece in fondo a questa funzione Se si ritiene che sia meglio dettagli poi Analogamente qui faremo una 0:19:05 Ovviamente qui faremo una stampa del sottografo indotto Fatemelo chiamare GS e anche qui andremo a capo e a questo punto ci aspettiamo qualche cosa di un pochino più gradevole alla vista possiamo anche provare a lanciare il codice per avere una qualche controprova dove vediamo che lodigraf oggi viene stampato e il sottografo GS al momento non viene stampato per ovvi motivi cosa ci manca da fare Ci mancano implementare appartiene tabella avendo scelto di descrivere la tabella dei nodi come un semplice elenco è abbastanza di abbastanza semplice questo detto per inciso è un'operazione di quelle standard su cui si scrivono libri di algoritmica E l'operazione di soia al ching di ricerca all'interno di un insieme ne avete sentito parlare in una lezione di te avete sentito parlare del alternativa possibile che la ricerca binaria che è più efficiente a patto che l'insieme sia ordinato qui insieme non è ordinato perché l'abbiamo semplicemente caricato da linea di comando però Mettiamo da parte questa osservazione per un momento Cos'è che dobbiamo effettivamente fare realizzare la partenza la tabella quindi diciamo che abbiamo un intero i che scorre gli indici della tabella da 1 fino a NS ad ogni passo ci chiediamo se per caso l'elemento che c'è stato passato è uguale a SDI perché in quel caso l'abbiamo trovato non è nel caso non è opportuno proseguire restituiamo tu D'altra parte siamo arrivati in fondo vuol dire che non l'abbiamo trovato e quindi è corretto restituire forse questo dovrebbe salvarci la situazione in teoria in realtà non c'è nulla da stupirsi Sì perché in effetti non stiamo chiedendo di stampare il sottografo indotto sul grafo uno da un insieme vuoto e chiaramente insieme vuoto dobbiamo completare la nostra linea di comando per cominciare potremmo mettere tutti quanti i nodi non importa che siano in ordine e avremo esattamente lo stesso font Dopodiché potremo decidere di togliere il nodo sei e spariranno i due archi 3656 che incidono nel modo sei e poi possiamo anche decidere di fare qualche cosa di più estremo quindi per esempio tenere solo 4 e 5 e ci rimane solo l'acqua 54 Quindi in effetti Diciamo che al netto che abbiamo fatto

solo tre o quattro prove questo algoritmo sembra essere corretto A questo punto possiamo cominciare In effetti a ragionare sulla sua complessità in dettaglio avendo scelto la implementazione delle insieme s e l'implementazione del grafo e possiamo dire quanto ci costa andare determinare l'appartenenza di un elemento al sottoinsieme SD un nodo al sottoinsieme S benissimo È chiaro è uno scorrimento è un condimento di tutto insieme Ness ce l'abbiamo qua sotto questo scorrimento qui la complessità età DNS quindi ti dsz dnf da altra parte Qual è la complessità di esiste Arco questa Come state da DNS viene ripetuta due volte poi c'è esiste ancora sistemato è una semplice consultazione della matrice di adiacenza quindi attenta di uno ne deriva che stiamo sommando però è di che vanno da 1 ad n zd1 per l'età di uno per la appartenenza a l'insieme degli archi tenda DNS per l'appartenenza ad essere moltiplicata per due età di uno per la stampa come sono tutte indipendenti da ho i dadi non si fa altro che sommarle per n volte e poi ancora che vende volte il risultato quindi otteniamo n quadro vo questa complessità dove NS prevale tu le altre due quindi abbiamo un test adn quadro nvs è una buona complessità è cattiva ci possiamo aggiornare su osserviamo che dipende da due parametri Perché Perché i dati dipendono da due parametri è vero che nel sembrata è più piccolo di n per cui Se volessimo dare una complessità nel caso pessimo si potrebbe anche più grezza mente dire che NS è più grande di N O se volete tetta DN nel caso pessimo e quindi il tutto viene 30 DM cubo però È un'analisi un pochino più grossolana è corretta a livello progetto d'esame accettabile è più brutta di quell'altra Perché vedremo che poi ci servirà distinguere un po' meglio Qual è la complessità spaziale di questo algoritmo è la complessità spaziale non è molto complicata Qui c'è soltanto l'uso di due variabili nel complesso Che cos'è che abbiamo usato è in sostanza nel complesso torniamo al 0:23:50 Pochino più grossolana è corretta a livello progetto d'esame accettabile più brutta di quell'altra Perché vedremo che poi ci servirà distinguere un po' meglio Qual è la complessità spaziale di questo algoritmo Beh la complessità spaziale non è molto complicata Qui c'è soltanto l'uso di due variabili nel complesso Che cos'è che abbiamo usato è in sostanza nel complesso torniamo al main abbiamo usato un grafo e abbiamo usato un vettore di interi il vettore a complessità spaziale intenda DNS il grappo a complessità tetta DN quadro Perché è una matrice doppia che è stata locata l'abbiamo visto prima è stata locata n x n e quindi abbiamo ennequadro pns Siccome ne sei più piccolo di n qui non ha senso scrivere n 4 più NSN se sparisce all'interno della valutazione Ok Quindi la nostra il nostro algoritmo uno implementato in questo modo a questa complessità si può fare di meglio allora abbiamo già accennato al fatto che potremmo fare una ricerca più astuta che si rifletterebbe direttamente su questo NS vediamo anche quanto se la ricerca fosse binaria noi il tetto DNS lo ri dovremmo contenta di logaritmo DNS che è sicuramente meglio D'altra parte questa cosa richiederebbe di aver precedentemente ordinato il insieme in s e quanto costa ordinarlo insieme ne abbiamo visto un paio di algoritmi di ordinamento in una corsa a lezione erano entrambi quadratici Ma dovresti anche aver già visto nelle lezioni di teoria e noi li prenderemmo in una successiva lezione di laboratorio degli algoritmi più efficienti che hanno complessità e nei log n in questo caso nsns quindi in effetti mi potremmo ridurre questa complessità Patto di sommarsi un termine teta DNS log n f lo facciamo lo facciamo ancora perché abbiamo ancora due algoritmi e c'era speranza che questi due algoritmi in realtà migliori no Senza dover fare più sforzi in questa direzione ma lo vediamo in seguito quindi temporaneamente accantoniamo l'idea e passiamo un secondo algoritmo cioè una ovvia modalità di miglioramento di questo semplicissimo codice e la modalità ovvia è qui stiamo eseguendo un'operazione appartenenza della tabella alla tabella S dell'insieme dell'elemento che avremmo potuto tranquillamente fare fuori dal ciclo for su di sé l'avremmo fatta una volta invece di farla n volte e quindi è un discreto spreco questa operazione l'algoritmo due che vi presento è semplicemente una piccola variante dell'algoritmo uno con-

siste nello scorrere tutte le origini nello valutare se l'origine sta nel tutto insieme poi scorrere eventualmente tutte le destinazioni e valutare le altre due gli altri due testi abbiamo già introdotto in precedenza quindi definirlo un nuovo algoritmo è quasi eccessivo Però effettivamente fa le operazioni in un ordine diverso e ne fa di meno perché ne fa di meno perché se per caso non appartiene ad esse il ciclo interno non viene eseguito possiamo dare una valutazione è un pochino più complicato ma abbiamo Certamente una via suo Dopodiché all'interno ci sarebbe tutto questo è tutto questo vuol dire fare una singola operazione di test di appartenenza ed è questo TS a questo TS dovrebbe seguire in teoria all'interno della sommatoria suo che va da 1 ad n che cosa ho niente nel caso in cui o non appartenga ad esse oppure nel caso in cui ho appartenga desse questa operazione Qua noi per fortuna sappiamo quante volte o appartiene ad esse ti appartiene NS volte quindi invece di fare un TS più o 0 o qualcosa è semplicemente contiamo il caso in cui c'è 0 da una parte e dall'altra Diciamo che in NS casi Quindi tutti questi anni casi soltanto NS casi che sono questi Dov'è l'indice t o vada 1nf Che cos'è io io e l'indice con il quale Noi stiamo scorrendo la tabella s e questo indice va invece da 1 a n f in complesso quindi per ognuno dei casi in cui entriamo all'interno del del IV sul sull'appartenenza di ho adesso noi eseguiamo quanto hai scritto qua dentro cioè la sommatoria per di che va da 1 DL n di questo test.ts di questo test ti ha e di questa stampa con le stesse complessità che abbiamo visto in precedenza quindi 0:28:20 Vediamo quanto è scritto qua dentro cioè la sommatoria per di che va da 1 ad n di questo test.ts di questo test Tia e di questa stampa con le stesse complessità che abbiamo visto in precedenza Quindi cosa significa significa che qui abbiamo n volte TS ETS ETA DNS non dipende da ot mentre 15 abbiamo un tetto di ns11z di uno vince il teta DNS che va X N X questo ciclo interno Eva x n s per il ciclo esterno perché Perché teta DNS non dipende né da di ne dai o conclusione abbiamo un test adn x n s e qui abbiamo un test adn per NS quadro che vince come rispetto alla precedente Prima avevamo Eni quadrupede NS adesso abbiamo n per n f quadro sembra lo stesso ma è meglio perché ne esce più piccolo di n per caso Pessimo è identico non è cambiato nulla nel caso pessimo aver portato fuori il test non ci ha portato nessun vantaggio ma se per caso abbiamo l'istanza in cui vogliamo calcolare Un sottografo indotto da un insieme molto piccolo su un grafo molto grande questa piccolissima modifica ci ha fortemente migliorato la nostra complessità implementiamo Quindi direi che mettiamo tra parentesi quanto già fatto che ci può anche servire per ispirarci e cerchiamo di implementare questa volta l'algoritmo 2012 Che cos'è che fa in effetti algoritmo due fa uno scorrimento dei nodi origine in N e valutazione di appartenenza poi c'è uno scorrimento dei nodi di destinazione n e valutazione di appartenenza di dhsv potremmo dire appartenenza di o as quindi di as e di o di ad a e questo sistema direi la descrizione In che cosa consiste tutto questo il codice assomiglia molto provo a fare un adattamento diretto che cosa dobbiamo effettivamente fare uno scorrimento esterno su o all'interno del quale Per prima cosa Per prima cosa andiamo a testare la partenza di o a s che in realtà l'unica cosa che bisogna fare è soltanto nel caso in cui ho appartenga Adesso possiamo proseguire con un ciclo for tutto punto direi che poi mentre servono un po' meno parentesi grafe di quello che sembrava più una lista un bel ciclo for all'interno del quale ne possiamo andare a valutare la partenza di di as e l'esistenza dell'Arco e direi che questo è quanto anche qui sopra avremmo potuto risparmiare qualche parentesi Allora c'ho fatto possiamo provare a valutare la correttezza simpatica e provare a fare un po' di esempi esempio complessivo quello con il 6 in meno quello solo con quattro e cinque è corretto probabilmente si è è più veloce per saperlo dovremmo cronometrare con Esistono delle procedure per farlo li useremo in una delle ultime elezioni però ci serve sicuramente un grafo molto più grande per valutare la differenza qui è talmente piccolo che si va sui millesimi probabilmente sui milionesimi di secondo e la differenza non si percepisce Allora abbiamo finito ci stiamo ancora il fatto che avremmo potuto valutare

Questa TS questa TS in tempo tenta di log nsns E quindi cosa sarebbe successo che questa parte qui Sarebbe migliorata ma tanto è trascurabile questa sarebbe migliorata vuol dire che sarebbe rimasto Leonessa esterno rimasto Lenne 0:33:10 Ma tanto è trascurabile questa sarebbe migliorata vuol dire che sarebbe rimasto l'ennesima esterno rimasto Lenne è un altro degli annessi sarebbe diventato un logo ens con cui abbiamo avuto nsns come qua sopra avremmo avuto ennequadro Logan S lo facciamo prima di decidersi a farlo mi presento il terzo dei possibili algoritmi per risolvere questo problema l'idea Qual è l'idea che fino adesso abbiamo scorso nodi valutando se appartengono o no alla tabella ma noi abbiamo la tabella potremo scorrere direttamente la tabella senza andare a correre tutti i nodi e poi chiedersi se stanno nella tabella sarebbe meglio Forse sì perché in fondo la tabella È una sequenza esco averla non costa nulla costa tempo costante si tratta solo di andare a leggere S quindi quello che vi propongo di fare l'algoritmo 3 alla seguente struttura andiamo a correre la tabella per chiarire il fatto che sto scrivendo una tabella dirò che ho un indice numerico io ho che vada uno ha la cardinalità di esse quella che nel codice Abbiamo indicato con NS Dopodiché però mi serve il nodo Allora il nodo lo vado a recuperare all'interno del vettore io ho all'interno del vettore che rappresenta S nell'indice io ho questo vettore Io l'ho chiamato Qui nello pseudocodice S piccolo noi nel codice abbiamo chiamato S grande io ho usato S piccolo qui per distinguerlo dal insieme S che ovviamente nel codice non c'è Dopodiché fatto questo possiamo scorrere la tabella per cercare le destinazioni e andare dalla indice di destinazione a ricavare la destinazione il nodo infine Possiamo valutare la partenza della coppia o di a l'insieme degli archi ed eventualmente stamparlo Qual è la complessità di tutto questo abbiamo un ciclo esterno per io che vada 1ns internamente abbiamo questa operazione qui di ricavare dato un indice di un nodo in una tabella il l'indice del nodo operazione che è una proiezione quindi è in tempo costante però ho voluto sottolineare la dandole un nome per essere estremamente Pignolo dopodiché abbiamo una sommatoria interna sulle destinazioni da 1 NS la stessa ti Find il test di appartenenza Tia e la Stampa Cosa costa tutto questo non è difficile Nel senso che dobbiamo solo capire quanto ci costa ti Finder ma l'ho già detto ti fa in the costante anche questa è costante conclusione abbiamo una bella sommatoria da 1 NS dei tenta di uno di una sommatoria di un tetto di 1 più tetta di uno che è il tempo di valutare l'appartenenza di un arco ad una coppia di nodi a l'insieme degli archi più in fretta di uno che ha il tempo di stampa conclusione questi tre Fanno un unico zd1 X NS questo zd1 compare è dominato il teta DNS ho tenuto internamente va moltiplicato ancora per NS otteniamo età DNS quadro ora attenta DNS quadro è certamente meglio di questo NS quadro perenne Perché chiaramente n maggiore di 1 e A maggior ragione è migliore di quest'altro perché abbiamo detto che questo era peggiore di questo perché n quadro NSP n x n S4 per il semplice motivo che NS più piccolo di n Quindi il terzo algoritmo è certamente meglio data l'implementazione che abbiamo costruito Ma se cambiassimo implementazione se finalmente ci decidessimo a fare quella famosa ricerca binaria qui non migliorerebbe di niente qui non c'è perché non c'è nessuna ricerca non esiste un TS quindi qui non abbiamo miglioramente Potremmo avere un miglioramento nei primi due allora provo a affiancare le scritte e proviamo a fare un piccolo ragionamento in particolare vediamo questo qua sotto se li avessimo NS quadro e qui avessimo n per NS log NS sarebbe meglio o peggio e lasciate perdere il logo NSN per NS sicuramente più dnf quadro per il semplice motivo che onesta in comune e che n più grosso di annesse quindi non servirebbe a niente Ma se io mi inventassi un'implementazione ancora più furba allora si può ragionare addirittura lasciando perdere l'implementazione e guardando le espressioni astratte Perché l'espressione astratta allora perché questa espressione astratta qui 0:37:55 Perché l'espressione astratta allora perché questa espressione astratta qui diamine vai tra il primo e il secondo algoritmo È abbastanza evidente che il secondo è migliore perché esegue esattamente le stesse

operazioni Salvo che ogni tanto non esegue quelle del primo cioè abbastanza evidente che quando non appartiene ad esse non si fa nulla quando appartiene adesso e fa le stesse operazioni Quindi direi che nessuno può essere dubbioso sul fatto che l'algoritmo due sia meglio del 1 su 3 si potrebbe ragionare Perché sono delle espressioni abbastanza diverse ma vi faccio notare che abbiamo Intanto un tira e un Tpr che sono contenuti attraverso una sommatoria da 1 a innesto e un'altra sommatoria di 1 NS Qui abbiamo tia itpr che sono contenuti in una sommatoria tra 1 NS ma in una da1n quindi più lunga conclusione per questa parte l'algoritmo due peggio dell'algoritmo tre e poi cosa succede succede che il tel era un T find racchiusi in una doppia sommatoria su NS e unti Finder ha chiuso una semplice sommatoria su NS PS racchiuso una sommatoria su nsns è un TS racchiuso in una sommatoria su N cioè questo TS e questo ti Find sono racchiusi in sommatorie simili Salvo che qui è un doppio NS e qui c'è un ms un n conclusione questa sommatoria è più lunga di quest'altra non solo tds e certamente peggio di T Find Perché ti fa è una singola operazione di lettura per cui per quanto TS voi possiate migliorarla non avete mai a vendere questo pezzo migliore di questo è la stessa cosa vale per questa banale sommatoria da 1 a n su TS confrontata con la sommatoria da 1 a n s su t Find quindi non c'è nessun modo in cui l'algoritmo due possa essere meglio dell'algoritmo 3 conclusione al limite potrebbero essere identici nel caso in cui nn fosse uguale n e lì si andrebbe a giocarsela sulle costanti di moltiplicazione di proporzionalità qui probabilmente non ha senso non avendo senso Noi terremo da qui in poi questo terzo algoritmo ignoreremo i primi due e non ci potremmo Almeno per il momento il dubbio di andare a implementare la ricerca in nfs con un ricerca binaria preceduta da un ordinamento questo per il momento quello che invece ovviamente dobbiamo fare è andare a implementare questo algoritmo 3 Dunque l'algoritmo 3 è un pochino diverso dai precedenti andiamo a comunque commentare via il 2 e andiamo ad aggiungere un algoritmo 3 nel quale si fa uno scorrimento degli doppio degli indici in S traduzione in nodi e valutazione di appartenenza Dio di Ada molto stringato ma più o meno c'è tutto Cos'è che dobbiamo fare abbiamo bisogno di scorrere gli indici quindi ci servono io e ID ci serve un bel doppio ciclo con ovviamente indici e estremi diversi io deve andare da uno fino a n s e questo è io ho e ID deve andare da uno a NS questo è lì prima di lanciare tutto questo Giusto per andare a risparmiare al massimo andiamo a ricavare Oh dai oh E come facciamo semplicemente andiamo a leggere l'elemento io esimo del vettore S A questo punto facciamo la stessa cosa di sotto e quindi andiamo a recuperare di come elemento i di esimo del dottor S E tutto quello che ci resta da fare e valutare l'esistenza dell'Arco che deve attualmente stamparlo se esiste l'arco Print F si può fare leggermente di meglio in realtà si si può fare leggermente di meglio È abbastanza evidente che qui le variabili Heidi 0:42:48 In realtà si si può fare leggermente di meglio È abbastanza evidente che qui le variabili o i di non servono a niente perché Potremmo direttamente andare a scrivere SDI può essere Dio di dentro nella chiamata della funzione di stampa e questo ci porterebbe a eliminare un paio di copie con un micragnoso guadagno ma anche con una forma tutto sommato più compatta E forse più elegante voluto scrivere quella estesa per sottolineare tutti i singoli passaggi della derivazione L'unica cosa è che questa lettura dsd io viene fatta dentro il ciclo fuori interno antiche con l'esterno ma sono assolutamente quisquiglie Allora ha detto ciò abbiamo riassumiamo abbiamo valutato 3 algoritmi diversi abbiamo valutato la loro complessità tratta quindi in termini di indipendenti dall'implementazione e poi l'abbiamo Calata in una specifica implementazione per il grafo è una specifica implementazione per il sottoinsieme e abbiamo anche vagamente discusso una implementazione alternativa che eri sotto insieme che abbiamo per lo stampato che abbiamo però scartata perché abbiamo visto che migliora due algoritmi comunque dominati e quindi in effetti ci teniamo il terzo è per il momento lasciando a perdere lasciamo da parte l'implementazione a vettore ordinato te sottoinsieme S Eventualmente potremmo riprenderla in seguito

cosa facciamo Adesso proviamo a cambiare l'implementazione del grappo Cosa vuol dire cambiare l'implementazione del grafo vuol dire che invece di utilizzare l'implementazione a Matrice di adiacenza potremmo usare quella lista di Archi Come si fa Beh allora vi ho costruito è predisposto una libreria lista Archi h e una libreria lista Archi cx30c quest'ultima è corpi di funzioni vuote la prima invece vi Fornisce la struttura struttura è un grafo che contiene numero di nodi numero di Archi esattamente come prima invece della matrice di adiacenza alla lista degli archi perché può essere interessante intanto per valutare giustamente la come dire per valutare direttamente un'altra implementazione E poi perché in effetti i nostri algoritmi erano tutti basati su scorrimenti di o del sottoinsieme o scorrimento di nodi e poi testi appartenenza sottoinsieme e la mossa Vincente è stata scorrere il sottoinsieme Ma allora questa operazione di scorrimento non si potrebbe fare anche sugli Archi cioè tutto sommato lo il test di appartenenza che Scusate non si potrebbe fare anche sugli Archi risposta Sì effettivamente si può fare a patto che gli archi siano rappresentati in modo che lo consenta è uno dei possibili modi è la lista di Archi Allora ipotizziamo di rappresentare la lista di Archi abbiamo le solite 5 funzioni che abbiamo visto prima a cui abbiamo aggiunto la sesta funzione stampa grafo e qui sotto Ci sono tre funzioni 4 funzioni aggiuntive la cui utilità valuteremo nel seguito Quindi per adesso limitiamoci a considerare le 6 funzioni che già conosciamo la creazione la distruzione inserimento e cancellazione di Archi il test di appartenenza e la stampa a video che cosa dobbiamo fare Dobbiamo sostanzialmente implementare questa libreria è chiaro che in una applicazione concreta noi avremmo già probabilmente la libreria e quindi questa fase la salteremo non è necessaria Quindi adesso stiamo passando a un esercizio di alimentazione Però possiamo fare un'analisi a priori l'analisi è avendo scattati i primi due algoritmi avendo tenuto il terzo questa nuova struttura dati questa nuova implementazione ci viene buona Oppure Mica tanto probabilmente è la risposta è che è difficile che ci venga buona per il semplice motivo che se qui in questa espressione 5 andiamo a sostituire all'implementazione matrice di adiacenza quella con gli stadi Archi L'unica cosa che cambia Non cambia la stampa non cambia la ricerca del nodo corrispondente a un elemento del vettore S Cambiala i testi appartenenza e come si fa a sapere se una coppia o di appartiene non appartiene all'insieme degli archi avendo una lista di Archi bisogna scorrere la lista è una ricerca all'interno di una lista quindi la sua complessità Sara testa di m 6 m e il numero degli archi e qui avevamo zd1 per 0:47:30 La cosa che cambia Non cambia la stampa non cambia la ricerca del nodo corrispondente a un elemento del vettore S Cambiala i testi appartenenza e come si fa a sapere se una coppia o di appartiene non appartiene all'insieme degli archi avendo una lista di archi per una cover salita una ricerca all'interno di una lista quindi la sua complessità Sara tetta di m 6 m e il numero degli archi e qui avevamo un tetta di uno per cui quello che ci aspettiamo e che compaiono zdm che complessivamente la complessità temporale di 20 m x n f quadro che potrebbe essere molto peggiore Se abbiamo un grafo di 1000 nodi denso Quindi con un milione di Archi questo algoritmo diventa un milione di volte più lento che non è molto bello evidentemente allora che senso Potrebbe avere un senso potrebbe essere quello di migliorare la complessità spaziale perché è spaziale perché in sostanza rappresentiamo non con un enorme matrice n 1000 x 1000 tutte le potenziali coppie di nodi ma rappresentiamo soltanto gli altri che effettivamente esistono e questo cosa significa significa che la complessità spaziale zdm anziché tetta DN quadro andiamo un attimino a disquisire Eccoci qua la complessità dello spazio diventa 30 dm e questo m più piccolo in generale di n quadro quanto più piccolo dipende dal tipo di grafo sei un grafo sparso nel noi avevamo un grafo di con 9 Archi su 6 x 6 6 No dice sei per sei trentasei potenziali Archi quindi era relativamente spasso non spessissimo ci possono essere grafiche sono decisamente spazi e in tal caso occupare uno spazio proporzionale denne può essere un grosso vantaggio rispetto ad avere uno spazio di complessità asintotica te3n

quadro se per oggi Raffo in realtà è denso o addirittura completo 6 completo non cambia nulla Abbiamo m oggetti che sono in Ecuador anziché ennequadro oggetti in più questi oggetti non sono delle Celle di una matrice booleana ma sono degli oggetti allocati con un puntatore trecce magari un punto magari buttato Restuccia un contatore prezzo magari delle sentinelle Insomma qualche cosa di molto più sofisticato questo cosa vuol dire che per quanto riguarda la complessità temporale scelta è decisamente sbagliata sempre comunque se usiamo l'algoritmo tre mentre per quanto riguarda la complessità spaziale può essere una buona scelta sembra più sparso e una cattiva scelta se arriva poi è molto denso ci sono dobbiamo valutare la situazione Qui cominciamo a vedere il fatto che un algoritmo può essere meglio di un altro per alcuni stanze peggio per altre meglio per alcuni aspetti spaziali peggio per altri temporali completiamo comunque con continuiamo Comunque la nostra analisi Allora vale la pena di cercare di implementare questa libreria Come la si implementa Questa famosa libreria e bisogna andare a riempire crea distrugge eccetera domanda Che cos'è una lista di Archi una lista di anche un qualcosa che va a sua volta implementato e per fortuna per non rendere l'esercizio decisamente troppo lungo io vi ho fornito completamente implementata una libreria per gestire liste di Archi dove un arco è individuato una lista di alchil rappresentata come un implementazione bidirezionale circolare con sentinella Esattamente come la lista dei libri e di conseguenza Questa è una tempesta brutto allora di conseguenza una lista è un puntatore a un arco che sarà la sentinella anche una posizione è un contatore un arco e l'arco che cosa sarà la corsa da una struttura che contiene al suo interno due nodi origine-destinazione e i nodi sono degli interi e conterrà due posizioni successive precedente ci serve tutta questa roba Non necessariamente ho deciso di usare questa realizzazione solo perché mi era più familiare però si poteva usare neanche un'altra questa libreria contiene le solite funzioni che abbiamo già visto quindi per creare e distruggere leggere scrivere eccetera eccetera eccetera e quindi questa è già predisposta è utilizzabile quella che invece va realizzata è la libreria relativa al crafond a questo punto chi se la sente fa un bel esercizio a mettere in pausa e INPS Che cosa le prime 6 funzioni quelle che abbiamo già visto in precedenza magari ispirandosi anche se la struttura è completamente diversa da quelle della matrice di adiacenza E invece io lo farò fra un attimo Allora come si fa 0:52:10 A questo punto chi se la sente fa un bel esercizio a mettere in pausa e implementare Che cosa le prime 6 funzioni quelle che abbiamo già visto in precedenza magari ispirandosi anche se la struttura è completamente diversa da quelle della matrice di adiacenza E invece io lo farò fra un attimo Allora come si fa a creare un grafo lista della spesa dobbiamo assegnargli NM è il puntatore alla lista degli archi me la porto di fianco in maniera da non dimenticare nulla e osservo che il campo n del grafo G Vale esattamente n perché stiamo ipotizzando di creare un grafo dato il numero dei nodi che rimane invariato Dopodiché dobbiamo andare a decidere Quanti sono gli archi ma il grafo è vuoto e quindi gli altri saranno 0 molto semplicemente infine dobbiamo andare a stabilire la lista Come si fa a costruire la lista di Archi vuota per saperlo andiamo a ispirarci alla libreria lista Archi libri alistar chi ha una funzione crea distrarci il cui scopo è quello di costruire una lista di Archi vuota Ed è esattamente quello che ci serve Quindi a questo punto con tre mosse abbiamo creato il nostro della po' vuoto Come si fa a distruggere il Granfo potete immaginare che non sarà una cosa molto diversa dobbiamo settare a valori opportuni e dialogare tutte le strutture dati dinamiche quindi dovremmo andare a scrivere che il dei nodi di questo che non è più un grafo stava a zero che il numero degli archi di questo che non è più un grafo diventerà 0 e dobbiamo andare a cancellare a distruggere la lista degli archi Come si fa a distruggere una lista di Archi c'è una funzione distrugge di talchi che prende un contatore alistar Chi è Di conseguenza qui andremo ad applicare distrugge lista Archi all'indirizzo dpg freccia l Tutto questo ci restituirà andrà a distruggere qualunque elemento ci sia in questa lista distruggere la sentinella e metteva a zero

annullo anche il puntatore l maniera che chiunque possa guardare questo gran fai capire che non è un grafo è solo una struttura scatola vuota proseguiamo Come si fa ad aggiungere un arco a un grafico esistente non passano nella scorsa lezione ho deciso un po' così al volo di cambiare la definizione di questa funzione rispetto a quella che ho usato fino all'anno scorso stabilendo che la funzione testa se l'arco esiste o non esiste è quello là l'arco esista non fa nulla mentre Qualora non esista invece lo aggiunge Seguirò questa strada anche qui in maniera che tutte le 3 librerie siano coerenti Ma come si fa ad aggiungere sostanzialmente un arco è quello che dovevamo fare in sostanza e andare a scorci la lista degli archi cercando ad ogni passo se lo troviamo o no Come si fa scorrere una lista di Archi abbiamo il nostro concetto di posizione quindi un tipo Rosa Arco e abbiamo le nostre funzioni che partono dal primo Lee starkey arrivano lui procedono finché non sia alla fine della lista archi e ad ogni passo si spostano successivo nella lista degli archi Proviamo a scrivere ce lo abbiamo bisogno di un po' da arco a cui potremmo dare il nome né Monica mente significativo come ti ha potremmo imporre che pia parta dal primo Lee starkey del grafo PG E questo dovrebbe essere corretto prima lista Archi prendo una lista di archi che è PG 3 Celle Ecco e a questo punto noi potremmo andare a testare finché non siamo in fondo alla lista degli Archi del della lista pgl Finché la posizione Pia non è l'ultima della lista PDL chiamo che effettivamente finalista archivi richiede la lista degli archi che la posizione quindi la lista degli archi epgl la posizione Pia e a questo punto sostituiamo ogni ph con il successivo nella lista PG freccia l della posizione Pia Questo è il nostro ciclo di scorrimento identico allo scorrimento delle liste di libri ad ognuno di questi passi noi dobbiamo andare a leggere Largo corrente per 0:56:45 Edilibri ad ognuno di questi passi noi dobbiamo andare a leggere la corrente per andare a capire chi sono i suoi estremi Come si fa a leggere un arco ci sarà una funzione legge Arco che data una lista una posizione recupera i due nodi estremi per indirizzo quindi abbiamo bisogno di andare a leggerci i 2 nodi nodo o in modo di e quindi possiamo andare a dire legge Arco la legge arkowcy Prendiamoci la sequenza degli argomenti in maniera da nonna perderli la lista degli archi si chiama PG freccia l la posizione corrente l'abbiamo chiamata Pia il modo in cui andiamo a scrivere eh & O e l'altro è commerciale di adesso abbiamo letto i due nodi possiamo confrontarli possiamo dire che possiamo dire che se l'abbiamo trovato il nodo Cioè se ho è uguale al nodo Ah scusa ti hanno fatto solo ora che questi nodi si chiamano alla stessa maniera e questo Evidentemente è impossibile è inaccettabile quindi andiamo a leggere nella lista le leggeremo origine è destinazione quindi se per caso o è l'origine e di è uguale al destinazione vuol dire che abbiamo effettivamente trovato l'arco Che significa che l'acqua esiste già ma se l'acqua esiste già non ha senso aggiungerlo Quindi saltiamo banalmente fuori mentre se invece l'arco non lo troviamo vuol dire In sostanza che quando usciamo siamo arrivati in fondo alla lista degli archi quindi possiamo usciti col break e quindi siamo in una posizione Pia che è ancora valida oppure non siamo usciti col break siamo usciti regolarmente e quindi la nostra posizione attuale Pia è alla fine della lista degli archi in questo caso l'arco va effettivamente aggiunto Domanda Dov'è che l'arco ma effettivamente aggiunto Questa è una buona domanda non c'è una risposta indicata dal Testo Possiamo metterlo dove vogliamo e io per analogia con le liste dei libri farò quella cosa un po' barocca che abbiamo visto in precedenza di andare a metterlo dopo l'ultimo della della posizione stessa quindi andremo a prendere l'operazione di inserimento che è questa qua in lista Archi 0:59:35 Andremo inserire questo famoso Arco nella lista PG freccia l in una posizione che è questa che adesso dobbiamo andare a stabilire e daremo un nome e l'arco che dobbiamo inserire è largo o di Allora come andiamo a determinare la posizione ci servirà una seconda posizione per esempio cu questa posizione Q abbiamo detto che sarà la successiva nella lista degli archi e gl e che cosa dell'ultima credo di aver scritto correttamente ultimo lista dovrebbe ricevere una lista e restituire la posizione dell'ultimo elemento e noi

vogliamo andare nel successivo elemento e a quel punto inserire li correttamente questo nuovo al con che stiamo aggiungendo direi che sento il bisogno di valutare se il codice asintoticamente corretto quindi andiamo a compilare questa volta la0 e vediamo che cosa ci dice una quantità di cose le cose principali li avevi principali sono questi di collegamento di Linking perché noi abbiamo anche utilizzato la libreria li stacchi e di conseguenza va inclusa nella compilazione Però questo non risolve tutto per risolvere tutto andiamo a leggere un po' meglio gli errori e scopriamo che c'è un avvertimento secondo cui nella riga 164 di sottografo Quindi qua è proprio sbagliato in precedenza devono errore commesso in precedenza cioè il fatto che valutava mooa.it ma abbiamo tolto o e t e gli abbiamo sostituiti con SDI o SDI di Quindi questo è un lavoro che abbiamo commesso in precedenza notate che è segnalato come Warning come avvertimento Perché in teoria o e d sono stati dichiarati semplicemente però non si sa che valore hanno Quindi questo è un errore abbastanza insidioso non segnalato come errore compiliamo nuovamente e scopriamo che Dio non sono utilizzate Il che è corretto teoricamente potremmo anche toglierli di mezzo e poi vediamo se ci serviranno in un qualche futuro a questo punto arriviamo alla funzione Insalaco nella quale c'è una dichiarazione implicita della funzione ultimo lista perché lui sta dicendo che l'ultimo lista è dichiarata implicitamente perché in realtà si chiama l'ultimo li stacchi e quindi ultimo lista non è una funzione che esista è di fronte a una funzione di cui non sa nulla Il compilatore dice ma magari me lo dice vai da qualche altra parte quindi proseguiamo E questa volta il tutto funziona Ho però un dubbio di fondo il dubbio di fondo è che questa funzione sta ancora utilizzando la libreria grafo Enea mentre dovrebbe usare la libreria g l a sintatticamente tutto funziona proseguiamo proseguiamo ne andiamo a implementare la procedura di cancellazione Quindi ricapitolando qui dentro Se l'arco già esiste saltiamo fuori dal break Siccome non è arrivato in fondo la lista Non si fa nient'altro Se invece trova l'arco entriamo qui dentro lo aggiungiamo e siamo dimenticati di andare a incrementare il numero di Archi stesso a questo punto dovremmo essere a posto la cancellazione svolge una funzione abbastanza simile Nel senso che anche qui abbiamo deciso che se l'arco non esiste non si fa nulla Mentre se esiste lo si cancella per cui la struttura sarà piuttosto simile vado a ispirarmi per fare prima sto svegliando quello che combino andiamo a correre tutti gli altri e questo perfetto Li leggiamo valutiamo e se la troviamo usciamo se no andiamo in fondo Chiaramente la cosa è esattamente complementare a prima Cioè se siamo arrivati in fondo vuol dire che la Marco non c'è e quindi è sbagliato fare qualunque cosa cioè non bisogna fare niente quindi questo le operazioni Vengono eseguite solo se non si è alla 1:04:20 Cosa è esattamente il complementare a prima Cioè se siamo arrivati in fondo vuol dire che l'arco non c'è e quindi è sbagliato fare qualunque cosa cioè non bisogna fare niente quindi questo le operazioni Vengono eseguite solo se non si è alla fine della lista degli archi Dopodiché dobbiamo cancellare Come si fa a cancellare l'arco che abbiamo trovato e che è puntato dalla posizione Pia molto semplice Qui c'è una operazione di cancellazione di una lista di archi di un elemento da una lista di Archi la lista si chiama PG freccia l e la posizione è Pia qui va Passata per indirizzo per quella complicata questione che abbiamo discusso all'epoca è che non riesci Però riguardo al fatto che cancelliamo e Pia continua a puntare una cella che sballata qui in realtà non ci interessa che piove ingaggio Ornato tanto quello che voleva messo lo cancellare l'arco in questo caso Inoltre andiamo a diminuire il numero degli archi stessi Direi che possiamo proseguire fare una piccola compilata per vedere che la variabile In effetti non è stata usata e quindi la possiamo cancellare e andare a cercare l'esistenza dell'Arco Resistenza è particolarmente importante perché era il test che avevo preannunciato probabilmente allungherà la il tempo peggiorerà la complessità del nostro algoritmo Come funziona una determinazione se una coesistono è molto simile a quello che abbiamo già visto la rigore avremo delle tue ha potuto in queste funzioni chiederci se la coesisteva non esisteva però abbiamo voluto ho preferito da

implementare il tutto per evitare troppe chiamate di funzione che comunque non sono il massimo dell'efficienza Allora a questo punto noi andiamo a scorrere il la nostra lista di Archi leggere elemento per elemento l'arco stesso e se per caso lo troviamo non c'è neanche bisogno che usciamo possiamo usciamo dal sito possiamo restituire direttamente tu al mondo esterno altrimenti andremo a restituire foals E questo dovrebbe essere quanto fine della implementazione di questa fase stampa del grafo come la funzionava la stampa del grafo in precedenza avevamo tutti gli indici dei nodi da 1 N O vigine da1n destinazione interrogava mo l'esistenza dell'Arco in realtà questa cosa fatta qui funziona ma è particolarmente inefficiente Perché Perché avremo un ciclo for su o con un ciclo post di interno con un vip esiste Arco e Però questo qua sopra è una complessità che è tetta chi può per l'eredità di n Anche questo è un ciclo o gn e questo qua sotto Abbiamo appena visto che è uno scorrimento di tutti gli altri quindi io ho DM conclusione n x n x 1 m che può anche essere né quadro si può far di meglio Si si possono direttamente scorrere gli archi che sono questi Quindi questa prima implementazione che è quella che deriverebbe dalla stampa grafo contenuta in un grafo m-up apro vado a cercarla per mostrarla e vi faccio vedere che stampa grafo a un ciclo for su A1 sudi è un IV ed è se vogliamo il modo più pulito di farlo con una matrice di adiacenza Qui invece è un modo assurdo perché è nettamente più pesante è il modo più semplice e naturale è quello di andare a correrси con un classico ciclo tutti gli archi e a questo punto Siccome gli archi sono Archi Non abbiamo da fare nient'altro che stamparli posso andare a rubare la operazione di stampa operazioni di stampa richiede o cady e quindi l'unica cosa che dobbiamo fare è dato l'arco andare a recuperare i due estremi Eccoli qua quindi cancelliamo queste questa prima 1:09:10 Eccoli qua quindi cancelliamo queste questa prima possibile implementazione andiamo a leggerci ho ID e cambiamo il video e questo è quanto andiamo a compilare e a questo punto dovremmo avere il nostro algoritmo 3 perfettamente funzionante perché Perché l'algoritmo III Usa la funzione esiste Arco che abbiamo implementato Vediamo se funziona Funziona nel caso del intero cancellando sei e prendendo soltanto una coppia Qui c'è un errore perché qui si stampa 55 e il motivo sostanzialmente che sto usando una funzione legge Arco che in effetti non è stata non è stata dove leggere il colore gioco è stata incrementata Comunque cerchiamo di ragionare nell'alta la stampa del grafo è corretta per cui qui gli errori non ce ne sono eventualmente gli errori sono sulla stampa del fotografo in effetti stampa dei autoanelli fasulli Allora In che in che consiste la stampa del sottografo scopriremo abbastanza banalmente che abbiamo scritto due volte SD io e SD o di Perfetto allora adesso mi aspetto che tutto vada bene con l'intero grafo con una sua parte E con addirittura un singolo Arco Ok allora dove siamo arrivati Siamo arrivati a implementare l'algoritmo tre Vi faccio notare senza nessuna modifica al programma stesso tranne la inclusione della libreria corretta quindi la modifica sostituzione di una libreria con un'altra abbiamo implementato questo algoritmo per giovannone la complessità siamo passati a Tetta DNS quadro per via di questo doppio ciclo attenta DNS quadro per m a causa della esiste Arco che scorre tutti gli archi perché l'abbiamo fatto perché in realtà volevo portare alla luce un quarto potenziale algoritmo e quanto potenziale algoritmi basato sull'idea anziché scorrere due volte tutti i nodi del sottoinsieme che poi andare alla ricerca di un arco di scorletti archi e poi andare alla ricerca degli no dire sottoinsieme Cioè questo qua il tuo algoritmo che Vi propongo consiste nel dire che corriamo agli archi per ognuno di questi Archi andiamo a leggere gli estremi dell'Arco e poi andiamo a vedere testare la partenza dei due estremi a sottoinsieme S in quel caso stampiamo la complessità astratta di tutto questo si può descrivere con questa espressione in cui abbiamo una sommatoria da 1 m che corrisponde a questo famoso ciclo però a appartenente alla grande Dopodiché dentro ci sono l'operazione di lettura degli estremi le due operazioni di test e l'operazione di stampa e quindi avremo un ti extremi più due volte TS più Tpr Quinto dando per scontato esattamente come ho fatto negli

altri cicli e il passaggio da una col successivo sia fattibile in tempo costante e quindi ci siano dei zd1 che si possono complessivamente trascurare Qual è la complessità di determinare gli estremi di un arco dato un arco stesso probabilmente è costante è la funzione legge Arco che abbiamo visto in precedenza ma funziona Scusate la funzione già con non l'abbiamo L'abbiamo l'abbiamo vista implementata a livello di a livello di lista di Archi Eccolo qua Mentre questa operazione già la conosciamo e questa anche per cui inverosimilmente che complessità Ci aspettiamo da tutto questo ci aspettiamo una complessità data da una costante tetta di uno seguita da due volte questo tds che abbiamo finora considerato un tetta di Eccoci qua tetta DNS e poi da una stampa complessivamente m volte te da DNS società dmns cometa dmns rispetto al tetto DNS quadro è meglio o è peggio sms in comune e poi c'è mxns Yaki anziché i nodi del sottoinsieme sono più bianchi o sono fino a Lidl tutti insieme nel caso pessimo 1:13:54 DNS quadro è meglio o è peggio se un annesso in comune e poi c'è m anziché NS gli archi anziché i nodi del sottoinsieme sono più via che sono fino a Lidl sottoinsieme nel caso pessimo grafo denso piatti sono n quadro quasi dell'ordine viene quadro mentre in un grafo molto spasso ma molto molto spasso gli altri potrebbero essere proporzionale al numero dei nodi potrebbe persino essere meno dei nodi per essere meno dei nodi del sottoinsieme bisognerebbe andare a cercare col lanternino ogni stanza con grappo bassissimo che abbia addirittura meno Archi che i nodi del sottoinsieme di cui cerchiamo il sottografo indotto oltretutto Abbiamo ipotizzato all'inizio che i nostri grafi non avessero nodi isolati e non avevano ad isolati significa che necessariamente gli archi sono almeno n meno 1 perché in quel caso vai in un albero il grafo sarebbe connesso eccetera eccetera eccetera solo questi dettagli anche se in teoria dovresti avere già sentito parlare conclusione questo non è un gran algoritmo però implementiamo lo stesso perché a questo punto Vi amo Buona l'idea di migliorare la complicità attenta DNS è già sappiamo che possiamo vedere al logaritmo DNS e quindi la cosa comincia a diventare interessante ma vediamo che vedremo che si può ridurla ulteriormente con una terza possibile implementazione del sottoinsieme dei nodi ma prima di fare questo passiamo invece alla implementazione che però rimanderei a dopo la pausa Salvo dire che in cosa consiste consiste nel consentire uno scorrimento degli archi e una lettura dell'Arco loro condimento degli artici e chiederà come minimo di implementare funzioni per trovare il primo Arco la cui successivo e determinare siamo arrivati in fondo alla lista e la lettura richiederà una funzione che legge gli estremi dell'Arco questo si legge estremi Arco probabilmente o Infatti ho dimenticato di adeguare dirgli di aggiornare il WiFi domanda ma perché ho bisogno di prima Real Consult Sharko eccetera Non posso direttamente usare la lista non posso direttamente usare queste funzioni primo li stacchi fine degli Stark eccetera no perché perché voglio chiamare queste funzioni da un algoritmo realizzato qua dentro e quindi ho bisogno Avrei bisogno di conoscere queste funzioni ma queste funzioni non sono descritte in gara vla o includere la libreria lista Archi qui ma è un andare ad aprire la scatola grafo e guardarci dentro oppure creo un filtro un livello di funzioni di passaggio dentro grafo l a queste funzioni qui che mi consentono di accedere alle funzioni che stanno di là e chiave tutto questo è inefficiente Ma va nella direzione delle strutture dati astratte e questo è quello di cui ci nella seconda parte della lezione quindi concludere questa implementazione introdurre il l'ultimo algoritmo quello sempre algoritmo quato Ma come una diversa implementazione e Poi cercheremo di passare il tutto a una terza implementazione su grafo cioè quella con il vettore delle follower Star Devi trovarti allora abbiamo sia procedere poi concludere questa lunga esercizio Eravamo rimasti a metà della implementazione della libreria grafo lista archi a metà perché perché abbiamo implementato completamente le funzioni di base quelle di creazione e distruzione inserimento cancellazione testi esistenza e stampa del grafo senonché nel algoritmo 4 che adesso ci proponiamo di realizzare che cosa succede succede che ci proponiamo di scorrere gli archi e per scorrere gli archi non abbiamo delle funzioni

adatte nella libreria di partenza la libreria di partenza aveva. inserimento cancellazione ma niente di più interessante e specifico di conseguenza questo suggerisce che sia il caso di arricchire la libreria ora Cosa vuol dire a richiedere libreria Sì come intrinsecamente questa implementazione include al suo interno una lista di Archi avere quantomeno le funzioni di base per poter la cover e quindi quantomeno avere come sono indicate qui in questa in questa 1:18:50 Questa implementazione include al suo interno una lista di Archi avere quantomeno le funzioni di base per poterla correre quindi quantomeno avere come sono indicate qui in questa in questa file di intestazione in questo thread and file la funzione per ricavare la posizione del primo Arco quella per ricavare la posizione dell'Arco successivo è quella per capire se si è arrivati in fondo è chiaro che siccome noi sappiamo che in realtà l'implementazione è bidirezionale e automaticamente sarebbe il caso di implementare ultimo Arco preda Arco e sarebbe anche poi il caso di andare a vedere tutte le altre possibili di venire qui abbiamo una funzione per leggere gli estremi degli Arco perché non andava inserire invece anche una libreria per scriverli semplicemente perché al momento non ci servono Qui si tratterebbe di capire se abbiamo sovradimensionato la libreria oppure no per fare in fretta implementiamo solo unicamente le librerie che ci le funzioni che ci servono strettamente che sono primo Arco successo e fine Archi al solito l'indicazione che vi do è provare direttamente perché sono veramente molto banali queste tre e la libreria di lettura Dopodiché trovare vedere la soluzione che vi che vi presento Questa soluzione è molto semplice e consiste visto che la funzione la libreria grafo include al suo interno una libreria di gestione lista Archi che abbiamo già visto e questa libreria presenta delle funzioni per determinare il primo elemento della lista degli archi il successivo e per capire se siamo arrivati in fondo si tratta di sfruttare pesantemente queste funzioni quindi per esempio per andare a determinare il primo Arco invece di restituire noark non faremo altro che calcolare il primo elemento della lista quale lista degli architetti la lista si chiama Pigi freccia l e questo è tutto quello che in effetti ci serve per ottenere il primo Arco talmente banale che Uno potrebbe pensare ma cosa serve avere una locazione di questo record di attivazione che poi immediatamente ci chiama un'altra locazione di un altro record di attivazione Eccolo qua che poi è tutto ci passa un elemento DL freccia Sud non potremmo direttamente usare il PG friselle freccia successo a quel punto potremo direttamente qua dentro nello scorrere la lista degli Archi del grafo usare il PC frecce freccia su ci stiamo annegando in questo modo il principio della dell'approccio delle strutture dati astratte con un vantaggio inefficienza ma con un enorme svantaggio in chiarezza di scrittura stiamo costringendo chi usa chi implementa questo algoritmo a sapere come si gestiscono le liste puntatori stiamo inchiodando cia una lista gestita puntatori e B deve suonare con sentinelle e via dicendo invece che magari un'altra quindi preferiamo usare quest'altro meccanismo al solito vi dico ci sono anche delle strumenti tecnologici per far sì che queste non siano delle vere chiamate di funzione con un po' di tempo si risparmi però un conflitto c'è l'abbiamo già detto Lo devo ancora Maria via sempre meno spesso che cosa dobbiamo fare per l'anno successivo decisamente Ovvio è esattamente la stessa cosa andiamo a cercare la funzione per determinare il successivo in una lista di archi e adattiamo gli argomenti di questa chiamata quindi la lista degli altri non sarà la generica l Ma sarà PG freccia l e d'altra parte la posizione dell'Arco e invece è semplicemente questa posizione Pia non cambia nient'altro Per sapere se siamo in fondo alla lista degli archi chiameremo la funzione fine lista Archi vedremo con gli opportuni parametri c'è ancora una volta la chiameremo con posizione Pia e con lista degli archi PG freccia l e questo è quanto rimane la lettura degli estremi dell'Arco e abbiamo una grafo al suo interno possiamo calcolare la lista abbiamo la posizione nella lista semplicemente si tratta di andare a recuperare la funzione che legge un arco è questa funzione c'è la riportiamo qua dentro la funzione Sara Play gearco dovremmo applicarla alla lista PG freccia l alla posizione Pia e i due nodi

saranno nient'altro che p.o.e. PD e sono due puntatori che possiamo direttamente passare all'interno senza nessuna modifica allora che sto. siamo effettivamente in grado di tornare nel nostro programma e di implementare 1:23:30 Ovviamente in grado di tornare nel nostro programma e di implementare questo algoritmo 4 per cui procedo a commentare l'algoritmo tre Anche se non sembra poi così male quindi Eventualmente poi lo apriremo abbiamo visto che comunque funziona e potremmo andare a definirci un algoritmo 4 presentandolo in breve in breve che cosa fa questo algoritmo che tu algoritmo fa uno scorrimento degli archi in a poi ricavo valutazione deter- minazione degli estremi e valutazione di appartenenza di o e ds e questo è quello che ci accingiamo a fare come si fa a scorrere la lista degli altri gli altri del grafo chiamando le relative funzioni di questa libreria quindi le funzioni che abbiamo appena implementato primo Arco su Ciacco fine Archi ci servirà una corso re che sarà un po' d'Arco posato Qui non è definito indefinito in lista Alchimia di star Chi è inclusa all'interno di grafo l AEG Raffaele a è Ecco qui è incluso dentro il nostro fare Quindi siamo in grado di vedere che cos'è un po' d'Arco quindi siamo anche in grado di dichiarare una variabile posar co che io continuo a chiamare Pia questa variabile la uso come cursor in un ciclo che parte da Marco del nostro grafo PG Dopodiché andremo a controllare se siamo arrivati alla Come si chiama fine della lista degli articoli fine Archi sempre dpg nella posizione Pia e infine sostituiremo a ciascun ti ha il lago successivo nel grafo PG data la posizione corrente piacuto il ciclo di scorrimento degli archi sotto ciclo di scorrimento degli archi corrisponde a questo ciclo qui c'ho fatto dovremmo andare a recuperare gli estremi dell'Arco stesso e gli estremi si recuperano con la funzione che legge gli estremi dell'Arco quindi la applicheremo al grafo Pigi alla posizione Pia e andremo a recuperare i due valori che a questo punto abbiamo già le variabili no riattiviamo le variabili o e di che Abbiamo commentato per evitare di avere dei messaggi di avvertimento andiamo quindi a recuperare in uscita il nodo o è il nodo di una volta che abbiamo determinato questi due nodi si tratta abbastanza semplicemente torniamo qui di testarne l'appartenenza alla famosa tabella quindi possiamo chiederci se appartiene tabella o sns e Analogamente appartiene tabella di sms e qualora entrambi effettivamente appartengano noi possiamo tranquillamente spingerci a fare il nostro Arco il quale arco si chiama odi quindi lo copio da qua a questo punto dovremmo essere grado di aver sistemato tutto quindi dovremmo poter compilare il nostro codice che consiste della fai principale sottografo Feo della libreria grafo ns0 e della libreria lista Archi tutto questo Dovrebbe darci noto fai sottografo abbiamo due variabili non usate che sono io ho i di se proprio vogliamo garantire di non aver nessun messaggio Sì basta toglierle e dovremmo poter tranquillamente lanciare il nostro 1:28:10 E dovremmo poter tranquillamente lanciare il nostro programma sull'Inter insieme dei nodi cambio l'ordine giusto per far vedere che non me dipende tolgo il 6 mancano un paio di archi e mi riduco come prima ai nodi 4 e 5 e vediamo che c'è soltanto un arco che li collega quindi dovremmo avere l'implementazione anche di questo algoritmo però abbiamo già studiato a Monte questa implementazione ci siamo resi conto che non è molto soddisfacente Perché a causa delle operazioni di ricerca di appartenenza di un no da un sottoinsieme il costo ci viene zdm per NS Eccolo qua a questo punto ci chiediamo ma non si potrebbe migliorare la operazione di testi appartenenza abbiamo già visto che se facessimo un ordinamento nel nostro insieme s e facessimo poi la ricerca binaria riusciremo a ridurre questo tetta DNS un tetto di Logan NS complessivamente otterremo zdm per la e qui si potrebbe aprire una discussione se sia peggio o meglio DNS quadro m rischia sempre di essere nel quadro e quindi in un grafo denso e sicuramente peggio in un grafo sparso m tenderà ad essere NS e quindi avremo un sms con trasporto nel quadro conclusione Potremmo effettivamente migliorare dobbiamo tener conto del fatto che però bisogna ordinare questo bettor e questo cinquantenne sms Non è non è grave non peggiora il risultato complessivo Ma non si può fare di meglio e la risposta è che si in realtà c'è un'altra

implementazione io non vi faccio quindi quella ricerca binaria anche se potrebbe essere ragionevole vi faccio quella a vettore di incidenza Cosa vuol dire vettore di incidenza molto semplice un vettore di incidenza non è altro che un vettore di n valori logici che dunque un vettore che ho usato per descrivere sottoinsieme di un insieme viene allocato lungo come l'insieme complessivo e sono valori logici quindi per ogni elemento dell'insieme complessivo c'è un valore che può essere vero o falso 10 e in base a che cosa si decide che sia 10 in base al fatto che l'elemento appartengono al sottoinsieme quindi rappresentiamo il sottoinsieme attraverso gli uni che appaiono all'interno del vettore stesso Cosa vuol dire questo che ha rispetto la tabella precedente non abbiamo NS numeri interi ma abbiamo n valori binari n più grosso DNS In teoria il binario sarebbe più piccola dell'intero anche se però noi abbiamo definito con un test il booleani come numeri interi Quindi in effetti stiamo sprecando più spazio però magari risparmiamo tempo anzi il concetto che nel momento in cui hai un vettore di incidenza il tuo test tds è bella banalissimo perché si tratta mente un attimo che mi sono perso si tratta semplicemente di andare a testare se questo ho detto Vieni incidenza avevo falso in un'opportuna posizione quindi rivedendo il nostro codice Eccolo qua anche se starebbe meglio su una pagina sola Allora in pratica si tratta di costruire il nostro vettore di incidenza Come si fa a costruire un vettore di incidenza prima non si mette tutto falso e poi si decide che per gli elementi che appartengono al sottoinsieme si mettono i corrispondenti elementi del vettore avevo A questo punto possiamo partire con l'algoritmo 4D Se volete cioè scorrere tutti gli altri determinare gli estremi dell'Arco e per questi estremi andare con un test istantaneo con una lettura del vettore di incidenza a vedere se l'elemento o il nodo fa parte del tutto insieme e se il nodo ti fa parte del sotto insieme e in tal caso stampare da Arco Qual è la complessità di tutto questo è la seconda frase somiglia molto alla prima con la differenza che invece di avere due volte tetta dns80 di Loch Ness e ha due volte tetta di uno quindi è molto meglio andiamo giù a vedere la quotazione in sostanza stiamo correndo al solito gli archi e invece di avere due volte tetta DNS abbiamo tratta di un'etichetta di uno però abbiamo una parte per eliminare l'inizializzazione di tutti gli elementi a falso e degli elementi da sottoinsieme avevo.in punta DNS Ovviamente questo è dominato da questo il teta DNS d'altezza DN è questa somma qui è dominata dal è tutta costante quindi si moltiplica per m e il risultato finale età di npm poi 1:33:06 È tutta costante quindi si moltiplica km è il risultato finale età di npm poi in generale è più piccolo di M ma qui Voglio essere assolutamente generico a mettere anche il caso di grafi bassissimi con m più piccolo di n e quindi scriverò teta di npm A questo punto ci può avremo il problema di confrontare questa implementazione con la precedente ma prima di forse lo e direi che vale la pena di andare effettivamente a implementare questo algoritmo per vedere che cosa comporta Allora che cosa comporta comporta intanto di commentare l'algoritmo 4 nella sua implementazione semplice Dopodiché andiamo a vogliamo da aggiungere algoritmo 4 con vettore di incidenza siamo più precisi con S implementato come le prove di incidenza l'algoritmo è lo stesso di prima quindi possiamo anche copiarsi la descrizione in effetti e a questo punto è come l'algoritmo è uguale Abbiamo una forte somiglianza con la sostituzione di queste due funzioni di queste due chiamate appartiene tabella Se volessimo potremmo conservare le chiamate andare a cambiare il contenuto di appartiene tabella Però cerchiamo di essere un pochino più grezzi più semplici per arrivare più rapidamente alla lo scioglimento finale si serve un vettore di incidenza e questo ho detto Vincenzo è un vettore di booleani sarebbe un puntatore a Buriano che va allo eccetera Facciamo una cosa Visto che abbiamo definito un tipo wint è ragionevole A questo punto per coerenza con noi stessi definire un tipo di boolean che è costituito da puntatori a boolean questa è il modo in cui si legge questo è un puntatore a boolean ed è v boolean e lo dichiariamo questo vettore potremmo chiamarlo per esempio vs non qua ho sbagliato dobbiamo dichiararlo qui esatto nella funzione stampa sottografo indotto

fino adesso è un Puro e Semplice puntatore andrà al locato andrà locato con ids sarà un vi boolean ottenuto con una funzione callo la Quale funzione callo che vi chiede Quanti elementi ha questo lettore di incidenza e quanti sono l'insieme complessivo quindi i nodi per cui PG freccia n + 1 al solito perché andiamo da 1 a n e la dimensione di ciascuno è quella di un boolean a tutto questo possiamo far seguire il solito controllo che la locazione sia andata a buon fine potremmo anche provare a cercare qualche cosa di utile Qui c'è per esempio l'apertura di un file ma facciamo errore nel allo azione del vettore di incidenza DS E ovviamente non ci sono parametri Quindi a questo punto usciamo ci ricordiamo di idealo Carlo in uscita perché altrimenti questa area di memoria va nello heap poi si esce si perde il puntatore vs quell'aria di memoria ma l'area di memoria rimane occupata e non è più né utilizzabile negli allocabile Finché non esce dal programma quindi idea lo chiamo la prima di uscire e a questo punto dobbiamo costruire la Come si fa a costruire bisogna inizializzare tutto il vettore a 0 Ma la Carlo che lo fa già di per sé e dobbiamo mettere a true gli elementi che stanno in esse è sostanzialmente è una cosa che abbiamo già visto corriamo gli elementi del vettore S di partenza Quindi non sparisce in realtà è una rappresentazione doppia del nostro sottoinsieme sia come tabella sia come vettore di incidenza Altrimenti avremmo dovuto nella funzione avremmo anche potuto in questa funzione interpreta Nina di comando allocare riempire il nostro vettore dei incidenza ma avremmo dovuto sapere quanti sono i nodi quindi avremmo dovuto aprire il file di testo capite Quante quante cose che si intersecano nel che vanno tenute in considerazione quindi diciamo che non facciamo niente di tutto questo e di conseguenza andiamo a semplicemente a riempire vsd 1:37:50 Siderazione quindi diciamo che non facciamo niente di tutto questo e di conseguenza andiamo a semplicemente a riempire vsd dsd Chi è a metterlo a true quindi siamo andati a prendere l'elemento io esimo della tabella S che è il valore o in altri punti abbiamo scritto o uguale SDI o e poi invii ASD questo nodo abbiamo messo il valore True questo dovrebbe risolvere l'intera questione di conseguenza possiamo passare effettivamente a realizzare questa operazione di appartenenza sottoinsieme Com'è che organizziamo questa operazione banalmente con un accesso al vettore vs In quale posizione nella posizione hot ci chiediamo se riesce e tu possiamo tranquillamente lascialo così tanto è un valore booleano e dsdd e questo diventa il nostro doppio test Evidentemente in tempo costante tutto è andato correttamente dovremmo poter compilare trovando che ci manca la variabile io occhiali semplice motivo che l'avevamo commentata temporaneamente ID continua a non servirci e non sbaglio e a questo punto dovremmo poter fare le nostre solite prove su tutto il grafo su una parte è semplicemente su una coppia e sembra che tutto sia a posto allora la valutazione di questo algoritmo come vi ho detto porta a concludere un tetta dnpm cometa di npm rispetto a te t'ha DNS quadro e sono già abbastanza scorrelati Nel senso che NS più piccolo di n e probabilmente anche di M ma questi sono lineari e questo qui invece quadratico Allora se il grafo denso MSN quadro dunque più grosso DNS quadro Ma se la puoi invece mrn tendono ad essere la stessa cosa e dipende da quante NS da quanto è grosso il sottoinsieme che ci interessa Quindi se abbiamo tutti insieme molto piccolo e un grafo denso ci converrà l'algoritmo tre mentre se abbiamo un sottoinsieme grosso e un grafo sparso si converrà usare l'algoritmo quattro pur di implementare il grafo come lista di archi e di implementare il sottoinsieme S come vettore di incidenza prima di passare all'ultima implementazione che ci dirà qualcos'altro di nuovo vi faccio salvare che abbiamo cambiato la libreria la nostra vecchia libreria Aveva soltanto le prime 6 funzioni ne abbiamo aggiunte altre quattro vuol dire che questa è una struttura dati astratta diversa da quell'altra ni nel senso In realtà da un punto di vista teorico si può tranquillamente andare a riprendere la nostra libreria grafo emmea. h che aveva meno funzioni e si può andare ad aggiungere 1 2 3 e 4 funzioni qua dentro a patto che si dia un significato al concetto di posar co è l'unica cosa che qui manca del tutto vieni qua in qualche

modo Bisogna definire Che cos'è la posizione di un arco vi do un'idea così in realtà è un'idea l'autista pratico non ha nessun senso è inefficiente Però Uno potrebbe dire che la posizione di un arco ad essere un puntatore a una cella di questa matrice booleana chiamiamo puntatore a boolean è un po' D'Arco a questo punto primo Arco dovrebbe andare nella matrice e scorrere Tutti gli Zeri finché becca il primo 1 primo in un qualche ordine bisogna dare un ordine agli archi del grafo descritto con matrice di adiacenza che di per sé non hanno un water però se uno va a Riga per riga e poi colonna per Colonna può dire che il primo Arco che trovano in ordine lessicografico è il primo Arco del grafo e quindi questo potrebbe restituire un puntatore a quella posizione Dopodiché su charko partendo da questo puntatore potrebbe tornare alla Celletta a recuperare indirettamente gli indici di riga e di colonna con un po' di aritmetica dei puntatori continuare a correre la matrice fino al successivo e poi di nuovo restituire l'indirizzo di questa posizione successiva e via dicendo è possibile sia cervellotico assolutamente barocco a inefficiente Sì perché per trovare il primo Arco invece di fare la banale operazione che abbiamo implementato di chiamare primo Lee starkey cioè di chiamare alle sue Fare la banale operazione che abbiamo teste implementato di chiamare Primoli starkey cioè di chiamare alle successe tempo costante e lì invece si tratterebbe di andare nella matrice scorri finché non si becca un 1 quindi sostanzialmente sarebbe un incrementazione piuttosto inefficiente ed è per questo che neanche Vi propongo di farla ma potrebbe essere un esercizio carino Giusto per capire invece come funziona uno il linguaggio di programmazione e due il concetto di struttura dati astratta anche in situazioni in cui non ha veramente una giustificazione da un punto di vista di uso pratico però questo è possibile non lo facciamo di conseguenza possiamo provare invece a passare alla terza implementazione a te ha sempre mentazione di ricordo consisteva nel gestire gli archi non in un'unica lista ma in tante liste una per ogni nodo di origine quindi di avere un vettore di liste di Archi un vettore di liste di incidenza ho detto anche detto vedi può stare perché perché questi articoli ha la caratteristica di avere tutti allo stesso modo di partenza lago 12 LARC 15 qui quest'ultima sarà l'arco 52 54 e 56 quindi Tre Archi che hanno che Pokémon una stella ed ecco perché è una star e fuoco e Perché escono tutti dallo stesso modo Va bene allora ha detto ciò che cosa si tratta di fare ancora una volta si tratta di abbandonare la libreria che stavamo utilizzando e passare questa volta da una libreria grafo Ele a passare una libreria grafo FS che anche sta già fornita possiamo un po' di pulizia andiamo a chiudere quelle che non ci servono più e andiamo ad aprire la libreria grafo FSH in questa libreria abbiamo una nuova definizione di grafo che è costituita da numero dei nodi dal numero degli archi n ed m e da un puntatore agli starchicken a te un vettore questo motore dinamico di liste di archi che ho chiamato FF Cosa vuol dire tutto questo vuol dire che dovremmo trattare le solite funzioni creazione di istruzioni inserimento cancellazione e via dicendo alla nuova situazione perché altrimenti questa semplice inclusione non ci permette di far funzionare tutti gli algoritmi che abbiamo già visto io passo anche temporaneamente Poi vedremo perché a commentare l'algoritmo 4 e riattivare invece da lui 4-bis e a riattivare l'algoritmo tre che comunque non abbiamo scattato del tutto solo perché è più semplice e quindi sarà il primo che faremo funzionare allora in che cosa dobbiamo adesso cimentarci sostanzialmente dobbiamo cimentarci nella costruzione di questa libreria grafo ff e poi nella valutazione di tutte le varie operazioni per i due algoritmi che ci sono rimasti in ballo in Lizza Allora cosa dobbiamo fare Abbiamo una libreria di partenza grafo fs0 che è completamente piena di Cat costituita da funzioni completamente vuote e al solito andiamo a riempire le singole funzioni fatelo voi Dopodiché fate ripartire la registrazione Allora la prima cosa da fare per creare il grafo è al solito creare le sue strutture e sono NM e questo vettore dinamico Quanto vale il campo n campo n vale il valore di n fornito dall'esterno Quanto vale il numero degli archi in questo caso come sempre sarà dicevo perché abbiamo Graffiti vuoti Dopodiché dovremmo

andare a costruire questo vettore PG FS freccia FS Che cos'è questo vettore questo vettore è un vettore dinamico di tipo detto negli Star Chi è lo allo chiamo con la solita culloch Quanto lungo Beh vogliamo da 1 a n ognuno dei nodi da 1 m deve avere la sua forma alta Quindi solito pgnt uno è i singoli Senti che cosa sono i singoli elementi sono delle lista anche che mi ricordo sono dei Semplici puntatori alle sentinelle poi avremo la solita valutazione di 1:47:20 Bene vogliamo da un'ora n ognuno dei nodi da 1 m deve avere la sua formula Star quindi solito PG n + 1 e i singoli elementi che cosa sono i singoli elementi sono delle lista anche che mi ricordo sono dei Semplici puntatori alle sentinelle poi avremo la solita valutazione di correttezza della locazione vado a procurarmene una e mi assicuro di sostituire bene le cose e avevo viene la locazione del vettore delle Forward e a questo punto non è che basti perché questo vettore adesso contiene dei puntatori che sono tutti Nulli grazie al fatto che la culloch riempie di Zeri l'area di memoria che all'Oca quindi in ognuno di questi elementi per o che va da 1 fino a Gigi freccia n 3 Ognuno di questi elementi noi dobbiamo creare una lista vuota Ma la Dobbiamo creare intanto ci serve un po' che è un nodo ovvero un intero Dopodiché non meno di questi casi noi dobbiamo andare ad assegnare a vgf SD o che cosa una lista di archi e questo si fa con la solita funzione crea lista Archi che crea una lista vuota cioè all'Oca il nostro la nostra sentinella Una volta fatto questo è relativamente semplice esattamente il contrario andare a distruggere e grafo Come si distrugge il grafo andiamo a mettere a zero i nostri due valori in realtà ci conviene queste due cose farle dopo per il semplice motivo che dobbiamo elemento per elemento distruggere le singole liste e vogliamo sapere quanti sono gli elementi con il pgn Non può essere reso Zero immediatamente Quindi queste due operazioni le spostiamo in basso per ultime e per ognuna di queste liste noi dobbiamo andare semplicemente a distruggere la lista di distruggere lista Archi operazione che al solito usiamo quello che ci dice lei del file in maniera da non sbagliare andremo a distruggere un punto la lista la lista si chiama tgfs che è un vettore o questa è la lista e noi vogliamo andare a prendere l'indirizzo di questa lista abbiamo finito No perché c'è un vettore che è stato allocato quindi fatta la distruzione delle singole liste andiamo a deallocare l'intero vettore Ovviamente questo va fatto dopo la distruzione delle singole liste perché sennò non abbiamo modo di accedere di accedere a elemento del pgf possiamo procedere Come si fa a inserire un arco è molto semplice io mi iscriverei a quello che abbiamo fatto andiamo da pelo di qua per la lista degli archi Qui abbiamo inserito un arco facendo un po' di un po' di ricerche Qui abbiamo ricercato e eventualmente abbiamo effettivamente compiuto l'inserimento fidata di fare la stessa cosa con una differenza la differenza è che non c'è un'unica lista di Archi ma c'è una ben precisa lista di archi che è quella fuoco star del nodo o quindi prendo un po' di cose poi vediamo che cosa si tratta di cambiare nel dettaglio Sempre con molta attenzione corriamo loro la lista generica di Archi Ma la Focus Star del nodo o che è un dato e questo lo ripetiamo nel fine e nel sud per il resto Stiamo scoprendo una ben precisa lista di Archi leggiamola così puntato Ricky beviamo origine-destinazione Le confrontiamo e se l'abbiamo trovato perfetto vuol dire che usciamo usciamo e l'acqua esiste già non facciamo nulla Mentre se la con esiste ci abbiamo fatto tutto questo ciclo andiamo a Cioè in quest'altro caso andiamo a calcolare la posizione che sta dopo l'ultima posizione sempre di quella lista lì e andiamo inserire in quella lista lì in quella posizione la odi e adesso possiamo incrementare PGM per inciso avendo un vettore di fuoco Star potrebbe valere la pena di avere anche un lettore di gradi che vadi uscenti perché no per avere così l'informazione se ci serve non lo sto facendo 1:52:00 Perché no per avere così l'informazione se ci serve non lo sto facendo Non ci serve per l'algoritmo che stiamo realizzando però per altri algoritmi potrebbe servire a questo punto la cosa analoga la facciamo per la cancellazione quindi ancora una volta mi iscriverò a quanto abbiamo fatto per l'unica lista Eccoci qua e adattiamo la Nel senso che correremo al solito non tutta la lista l che non esiste Ma la lista fuoco a star del nodo

ho questo va sostituito. per punto metodicamente per fortuna se ci dimentichiamo il compilatore dirà che non conosce la lista Pdl e quindi non sarà probabilmente un grosso errori ma per il resto dovremmo essere a posto così proviamo a compilare visto che mi ha fatto un certo numero di indicazioni qui scopriamo che c'è un ID non dichiarato e delle variabili non utilizzate questo fa riferimento alla funzione qua sotto dove In effetti no alla funzione stampa sottografo indotto dovrei ho tolto l'algoritmo quattro e quindi tutte le sue variabili sono non più usate e ho riacceso con quindi due variabili che invece vengono usate per cui devo andare anche a riattivare alcune variabili e presumibilmente di attivarne altre questo probabilmente sistema il tutto Ci sono anche Pia o i d che giustamente non vengono più usate Non è necessario farlo in fondo questo è l'unico avvertimento che quasi sempre è innocuo ignorare però per pignoleria tengo conto anche di questo vediamo l'esistenza dell'Arco ci Speriamo pesantemente all'altro codice perché abbiamo un simpatico scorrimento dovremmo andare a scorrere non tutta la lista degli Arch ma soltanto la lista degli altri che ci interessa quindi andremo a scorrere la lista di Jeep FS del nodo origine cambiando ogni singola occorrenza e andremo a leggere l'arco ed eventualmente comportarci Di conseguenza anche qui nessun particolare problema poi abbiamo la stampa del grafo Ecco la stampa del grafo è leggermente diversa perché perché c'è un punto fondamentale ed è che non esiste come precedentemente avevamo un'unica lista e quindi non è possibile copiare direttamente questo codice però si può fare una cosa molto simile cioè quello che possiamo fare stampando il grafo è stampare nodo per nodo le liste una in fila all'altra cioè quello che possiamo fare in sostanza è stampare un ciclo qui poi vi devo dire una cosa ma la dico dopo per non aggravare il discorso guardo tutti i nodi uno per uno per ogni nodo scorro la specifica formula star del nodo stesso cambiando queste tre occorrenze Dopodiché è così qua compare un piccolo problema Il piccolo problema è che io sto leggendo al nodo e lo sto anche sovrascrivendo con questa operazione è vero che se tutto va bene nella foresta del nodo o i nodi hanno come prime tre mo Quindi io posso anche sovrascrivere questa variabile perché tanto gliela sovracchiaro col suo stesso valore se questa cosa mi da fastidio Allora userò come cursore non la stessa Variabile o che sto usando per leggere qua dentro ma userò un'altra variabile e diciamo che comunque mi metto al riparo da un errore ma evidentemente ne sto commettendo un altro dettaglio programmazione allora Cosa vuol dire tutto questo che in teoria visto che il nostro programma con l'algoritmo numero 3 Chiama poi solo la funzione di 1:56:50 Numero 3 Chiama poi solo la funzione esiste Arco e Abbiamo caricato il grafo con la funzione Insalaco siamo a posto così dovremmo essere in grado di lanciare la nostra funzione di ottenere i risultati corretti sui vari casi che abbiamo visto in precedenza ovviamente in linea di principio vale la pena di avere Inoltre casi lo stesso grafico ma un bel numero di grafi un bel numero di casi e se velocemente sistemiamo visualizziamo 30 40 100 un certo numero diciamo di dicasi Allora abbiamo una buon conforto del fatto che non sto algoritmo sia corretto domanda Quanto ci costa questa implementazione dell'algoritmo 3 e che cos'è che è cambiato in sostanza torniamo alla espressione generale dell'algoritmo tre quella dovrebbe essere questa espressione generale generale Eccola qua sopra scorre tutti i nodi del sottoinsieme per ognuno trova il nodo nel grafo di partenza va di nuovo a correre in ordine un sottoinsieme trova il nodo di destinazione in grado di partenza e per la coppia va a cercare l'arco E infine lo stampa e abbiamo detto che questo con la lista degli altri non è una buona idea perché la NS quadro dovuto al doppio ciclo annidato si moltiplicava poi non per 11 come nel caso della matrice di adiacenza ma per un tetto di m che è parecchio lungo qui cosa succede e qui succede che la valutazione di esistenza di un arco ci costa andiamo a cercarlo Quanto ci costa ci costa uno scorrimento di una foto e Star Quanto è lunga una foto alta definizione lunga Delta più di ho cioè numero degli archi uscenti da o che nel caso pessimo n In altri casi potrebbe essere più piccola Diciamo che c'è un Delta massimo se

diciamo che c'è un Delta massimo in sostanza stiamo dicendo che ti dia al posto di questo tetra di uno a teta di questo Delta Massimo e quindi ho risultato finale è un tetta grande di n4s per il Delta massimo dei nodi che stiamo considerando il che non è bello perché è sicuramente molto peggio sicuramente probabilmente Può darsi che tutti i nodi in realtà abbiano una cruscente quindi è lo stesso ma è molto difficile probabilmente è peggio per Comunque meglio delle nel quadro S perenne che avevamo tenuto con implementazione precedente non solo dal punto di vista dell'occupazione Spaziale la rappresentazione con fuoco Star che occupazione a questo l'abbiamo visto già nella lezione di teoria Ma possiamo vederlo guardando la implementazione Ecco che nella creazione Noi abbiamo creato un vettore lungo n n + 1 e poi abbiamo creato delle liste di Archi ogni Arco compare una volta in una di queste liste Quindi abbiamo una occupazione che proporzionale a mnp mjn più a me è peggio della tetta di m che si aveva con la sola lista di Archi ma è in generale meglio del test adn quadro che si aveva con la matrice di adiacenza dove In generale significa non sempre sempre perché ci sono graphic come quelli completi dove In realtà avere mrn quadro booleani è meglio che avere m che poi ancora quadro strutture contenenti no d'origine nodo destinazione il predecessore successore più tutte le varie sentinelle più la costoletta il vettore delle follower sta Quindi c'è un rischio che però riguarda solo i grafi molto densi per i gatti sparsi tutto sommato conviene quello che stiamo facendo adesso allora ha detto ciò uno dice perfetto Passiamo al largo di to4 algoritmo 4 o 4 bis deve fare il 4 il 4 bis Direi che non c'è storia perché il 4 bis è nato per migliorare la complessità dell'algoritmo 4 me li faccio di nuovo vedere sostanzialmente l'algoritmo quattro aveva questa complessità m per NS dovuta in buona parte a questo teta DNS mentre il 4 bis riusciva ridurre questa complessità ha un semplice mpn perché perché il teta DNS sostituito da un tenta di uno quindi rimane solo zdm ma tutto questo richiede una delle operazioni per il 2:02:00 Inizializzazione di un vettore di incidenza che costano 30 DN Quindi direi che non ci piove sul fatto che questo mpn è meglio della mprms per cui quattro semplice lo ignoriamo da qui in poi direi che non è una questione di lo chiamo mentre invece Continuiamo con questo 4 bis allora qual è il problema che mi vuole così del 4 bis che richiede di scorrere gli archi per scorrere gli archi noi avevamo introdotto nella libreria tra quelle a delle funzioni la funzione primo Arco sul palco e fine Arco qui dobbiamo fare lo stesso però domanda è Che senso ha parlare di primo arco in una implementazione a fuoco a star dove ci sono tante liste in una cosa vuol dire primo Arco era risposta potrebbe essere potrebbe essere il primo Arco del primo nodo Eh ma se il primo no donarti uscenti allora secondo e anche lui prima eco del terzo Il primo è il primo modo che altri utenti capite che è un pochino più complicata questa implementazione di quella che abbiamo fatto noi dobbiamo fatto come implementazione Eccola qua Siamo semplicemente andati a prendere il primo lista chiede la lista Qui invece dovremmo cominciare a dire finché troviamo nodi che hanno liste vuote andiamo avanti quando troviamo il primo nodo che ha una lista non vuota prendiamo il primo arco della sua lista che complessità questo non è proprio il massimo ma ci serve veramente il concetto di primo Arco e la risposta è che se torniamo a vedere l'algoritmo in questione scopriamo che in realtà quello che noi vogliamo non è trovare il primo Arco è scorrere gli archi e per scrivere gli archi abbiamo già ho trovato una soluzione applicabile lo scorrimento degli archi l'abbiamo fatto nella stampa lo scorrimento degli archi lo abbiamo fatto nella stampa del grafo eccolo qua Però questo l'abbiamo fatto con la lista dunque lo scorrimento degli archi lo abbiamo già fatto Non mi ricordo più dove Ecco adesso mi sono ricordato lo abbiamo già fatto nella famosa ricerca esiste Arco No nella famosa ricerca Eccola qua finalmente l'abbiamo appena fatto adesso nella stampa del grafo certo anticipato la stampa del grafo rispetto a quello che volevo fare prima aveva senso l'abbiamo fatto nella stampa del grafo è che abbiamo detto scriviamo tutti i nodi e poi per ogni nodo scopriamo la lista degli archi uscenti Allora questa cosa qui

si può tranquillamente applicare pari pari anche alla implementazione dell'algoritmo quato per cui spegniamo commentiamo via l'algoritmo 3 riaccendiamo l'algoritmo 4 e andiamo a vedere che cos'è che bisogna cambiare se proviamo a lasciarlo così troveremo un errore dovuto al fatto che non esistono le funzioni primo Arco fine Archi su cerco e neanche la funzione legge estremi arco in realtà è quello che dobbiamo fare poi anche ovviamente attivare un po' di variabili per non sentirselo dire seguito ma dovevo sostituire questo ciclo questo ciclo sostituiamo con lo stesso ciclo che abbiamo usato nella stampa del grafo Perché in fondo funziona e 2:05:40 Ciclo sostituiamo con lo stesso ciclo che abbiamo usato nella stampa del grappo perché in fondo funziona è questo che cosa vuol dire vuol dire che è così qua Andremo per ciascuno dei nodi qui bisogna capire se ci è lecito usare o come variabile comincio a metterla dentro Poi vediamo se riusciamo a evitare il bisticcio nella lettura degli estremi mi per ognuno dei nodi o per ognuno degli archi uscenti dal nodo o quello che bisogna fare torniamo allo pseudocodice che ci può effettivamente aiutare quello che dobbiamo fare Andare a determinare gli estremi della Ma noi nell'altra uno dei due estremi lo sappiamo noi basta alla destinazione ed era questo a cui mi riferivo dicendo Vediamo se riusciamo a evitare il bisticcio sulla dopodiché abbiamo un doppio test Un test sull' appartenenza del primo estremo in III l'appartenenza del secondo di questo dovremmo parlare ancora allora cominciamo a sistemare la questione della lettura Ci serve veramente nel momento in cui la lì il traffico è implementato a fuoco a star e quindi per accedere un arco Io so già l'origine ti serve veramente leggere l'intero Arco origine-destinazione forse no tutto sommato E infatti qui sotto Io ho stabilito di leggere non tutto un arco Ma la destinazione di un arco che esce dal nodo o è che a una certa posizione nella lista degli archi uscenti dal nodo o Dopodiché ho sostituito primo Arco successo e fine Archi con un primo Arco FS un successo e feste un fine al tfs che svolgono la stessa funzione ma non su la lista complessiva che non esiste ma sulla lista fuoco star del nodo o infatti c'è un parametro o aggiuntivo per cui quello che dobbiamo fare queste funzioni qui usarle qua dentro quindi andare a cercare non il primo a tutta la lista ma semplicemente il primo Arco nella follower star di che cosa del grafo G uscente dal nodo o andare in fondo al lista degli archi Vorwerk star del grafo G del nodo nella posizione piace dire se la posizione Pia è la fine della lista degli anchi fuoco star del nodo del grafo g e infine aggiornare una posizione con la successiva nella foga Star sempre del grafo G del nodo o data la posizione Pia corrente se facciamo questo andremo a recuperare la destinazione dell'Arco PG nodo o determinato dalla posizione Pia il risultato non ne ho perché già lo conosciamo ma è di e siccome è un risultato semplice non abbiamo bisogno di passarlo per indirizzo Ma possiamo passarlo come risultato della nostra funzione Infatti legge del talco restituisce un nodo fatto questo potremmo andare a testare se ho i d appartengono al nostro grafo e a questo punto saremo a posto Cosa ci manca da fare 1 2 3 e 4 funzioni che dobbiamo andare implementare qua dentro dopodiché faremo l'analisi di complessità e scopriremo un altro piccolo simpatico dettaglio che ci permetterà di migliorare ulteriormente questo algoritmo quindi andiamo a implementare primo Arco su Cerco fine Archi relativi a una ben determinata follower Star sono sostanzialmente le funzioni qua sotto al momento vuote queste quattro funzioni qua l'invito al solito provateci voi e poi vedete la soluzione che sarà oltremodo banale come potete facilmente immaginare Allora la soluzione consiste nell'andare a prendere nella lista opportuna il primo Arco quindi andremo ad accedere alla funzione primo lista Archi alla quale dovremmo passare la lista degli archi stessa Qui abbiamo la nostra funzione possiamo vedere che a questo parametro e restituisce un po' d'Arco quindi la lista Come si chiama Si chiama cugi follower star del nodo o è questo è quanto è assolutamente analoga la ricerca dell'Arco successivo al Cerca dell'Arco successivo consiste nel prendere il successivo Arco nella lista degli archi della fuoco star del nodo data 2:10:30 Analoga la ricerca dell'Arco successivo ricerca dell'Arco successivo consiste nel prendere il successivo

Arco nella lista degli archi della forma star del nodo data la posizione corrente Pia e Determinare se siamo alla fine della lista degli archi consiste banalmente nel vedere se siamo alla fine della lista degli archi PG e feste o data la posizione corrente pa3 semplici banalità per quanto riguarda la destinazione la cosa poco più complicata Siccome dobbiamo al solito usare le procedure della libreria di lista Archi useremo questa procedura di lettura dell'Arco e anche se in realtà noi conosciamo il nodo di partenza Quindi dovremmo andare a cercare quale arco l'arco che sta nella follower star del nodo o in posizione di a e recupereremo in uscita due numeri dei quali il primo in realtà lo conosciamo già possiamo andare a metterlo in una variabile tanto per E questo sarà fuorilegge se uno volesse potrebbe controllare che sia identico è uno spreco di tempo se tutto è andato bene lo spreco di tempo Dopodiché la cosa interessante invece sarà recuperare la variabile di perché Perché è la variabile che dobbiamo restituire all'esterno questo ci permette di determinare la destinazione dell'Arco stesso a questo punto siamo perfettamente in grado di compilare ed eseguire il nostro algoritmo con la variante 4B quella più efficiente controlliamo questo non è corretto perché dobbiamo sostituire la libreria lista articona libreria for all star lista a chi va benissimo la libreria grafo Elea composta vediamo se ci sono errori ce ne sono abbiamo un assegnamento che fa un pointer da un intero senza un cast vuol dire che c'è probabilmente una funzione con il nome sbagliato siamo nella Liga 191 che è questa e vuol dire presumibilmente che ho sbagliato il nome di una di queste di queste funzioni vediamo quale potrebbe essere sarà stato sul palco il primo Arco e sul palco e questo punto pulizia compilazione stiamo avendo una variabile il nome utilizzata I Die Ich commenteremo via e abbiamo una o che invece non è dichiarata quindi dobbiamo risalire la o in effetti in grafo fs0 alla riga 27 in grafo fs0 alla riga 27 Questa è una delle prime cose che abbiamo fatto nella distruzione ci siamo dimenticati di dichiarare la variabile cursore no dopo Dopodiché invece avevamo qui ho una variabile ID che in effetti non è necessaria perché serve all'algoritmo tre e non serve all'algoritmo 4 perfetto Vediamo se il tutto funziona quantomeno sugli esempi non funziona Allora quello che succede è che in effetti mi ha fatto un po' di pasticci con gli indici nella stampa del grafo e qui l'errore È già nella stampa del grafo stiamo facendo un ciclo su oyku e poi in realtà stiamo andando a correre questa lista del nodo o che non è ancora stato usato Quindi bisogna essere un po' co eventi ovunque e riportare Origins anche qua a questo punto vediamo se se questo è già sufficiente compilando eri eseguendo non è sufficiente non lo è perché anche nella lettura dobbiamo usare origia o nasce in questo. di conseguenza forse adesso ci siamo ci siamo Possiamo valutare l'altra versione e infine anche la 45 tanto per continuare a fare sempre gli stessi test e a questo punto possiamo procedere ha cercato di valutare la complessità di questo algoritmo è alta la conquista l'abbiamo valutata ed è già visto che siamo mpn 2:15:04 La complessità di questo algoritmo è alta la conquista l'abbiamo valutata abbiamo già visto che siamo mpn che potrebbe non essere male ma l'avevamo valutata sulla lista degli archi cambia qualcosa se la valutiamo qua dentro Allora valutando la qui dentro abbiamo sempre questo scorrimento teta di n per allocare inizializzare a 0 il vettore di incidenza un teta DNS per inizializzare a uno gli elementi del vettore che degli elementi del sottoinsieme poi abbiamo questo scorrimento che scorre su tutti gli altri quindi è sempre una ciclo che va da 1 a da 1 m però qua dentro cosa succede che abbiamo una destinazione Letta in tempo costante senza particolari difficoltà due test in tempo costante e quindi sembra che siamo assolutamente nella stessa situazione di prima questo è Ma attenzione attenzione Ecco che si può effettivamente migliorare la situazione e ulteriormente con un trucco che abbiamo già applicato all'inizio di questa lezione cioè andando osservare che questo test di appartenenza Non è necessario farlo a ogni Arco del nodo o si può fare una volta una volta per tutte per ogni singolo no do cioè possiamo spostare il test vs di ho qua in alto solo se questo test ha esito positivo andare a correre gli archi uscenti e di conseguenza non farlo più qua dentro Sembra

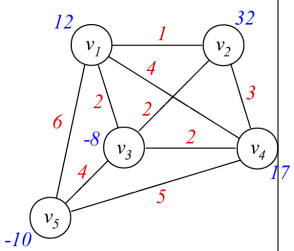
una stupidaggine ma il vantaggio è questo ciclo esterno che estratto eseguito n volte in realtà non richiede poi tutte le n volte di eseguire il ciclo interno e di conseguenza il ciclo intero che quello che costa di più impatta di meno sul risultato finale però ridevo di più se vi ricordate nell'algoritmo Trek questa era la modifica della lo dico un algoritmo due nell'angolino a te poi noi passeremo direttamente a scorrere gli elementi DS e noi abbiamo ancora il dottor S quindi non c'è bisogno di scorrere tutto il vettore di incidenza per poi andare a testare elemento per elemento se è 120 non possiamo direttamente scorrere il sottoinsieme S come si fa andando a correre io o che va da 1 fino a NS questo test non serve più perché gli elementi di Io sono tutti in necessariamente parte del sottoinsieme stesso però abbiamo bisogno questo punto di conoscere o è chi è oh oh è semplicemente o è semplicemente SDI quella ti Find che avevamo definito parecchio tempo fa e che a costo di uno costante unitario in realtà c'è altro da fare qua dentro no come cambia la complessità è molto più semplice abbiamo una somma su NS di questo ciclo uscente Quindi abbiamo una somma da uno abbiamo la inizializzazione esterna del vettore di incidenza a cui segue una somma da 1 NS di che cosa della somma sugli Archi uscenti dal nodo e quindi da 1 a delta più di o di una costante la costante è questa lettura più questo test di appartenenza più stampa e questo cosa vuol dire qui viene Delta più di o dopodiché se ho considero tutti gli altri uscenti dai nodi del sottoinsieme Sto contando tutti gli altri che escono dai nodi del sottoinsieme e questa cosa è sicuramente più piccola di m perché è più piccola di tutti gli altri tutti gli altri che escono da un sottoinsieme di nodi e meno che tutti gli archi conclusione la vera stima di complessità di questo algoritmo non è neanche npm ma n più Potremmo chiamarli MS o m degli archi che escono dai nodi del sottoinsieme s e questo è verosimile che sia ancora più efficiente ci sarà comunque il caso in cui il l'insieme S è molto grande e il grafo è molto denso e quindi questo msd ventenne quadro e quindi è peggio DNS quadro Ma ci saranno probabilmente dei casi in cui invece è meglio tirando un po' di somme direi che mi evita di fare una prova per vedere se il tutto torna in maniera da avere il codice corretto e a occhio e croce potrebbe esserlo Dopodiché mi evita di fare un po' un riassunto cosa abbiamo visto 2:20:00 Dopodiché mi evita di fare un po' un riassunto cosa abbiamo visto una galoppata di algoritmi su un problema che sembrava banalissimo alcuni di questi algoritmi erano dominati da un punto di vista teorico non c'era bisogno neanche di decidere come implementarli si poteva banalmente capire che l'algoritmo due era meglio dell'algoritmo uno perché faceva delle operazioni in meno per l'algoritmo tre era un pochino più complicato ma anche lì se arrivava capire che era meglio degli algoritmi 1 e 2 dopodiché abbiamo cambiato l'implementazione siamo andati a fondo nell'implementazione a Matrice di adiacenza e poi abbiamo testato le altre abbiamo visto che a volte queste implementazioni in peggio avano sostanzialmente migliorano lo spazio nei grafi sparsi non si usa mai una matrice di adiacenza per un Granfo spasso ma peggiorava nell'implementazione dal punto di vista temporale però aprivano la strada ad algoritmi che in teoria sono realizzabili anche con la prima libreria però introducendo delle funzioni definite in modo un po' buffo e queste nuove questi nuovi algoritmi sulla nuova struttura dati avevano un aspetto perlomeno promettente subito subito non tanto nel senso che l'algoritmo 4 implementato adesso mi sto perdendo ma lo dico quattro nell'implementazione tempi c'era questo MNS non era particolarmente promettente Ma facendo una buona implementazione delle insieme eh se poteva diventare più promettente o addirittura particolarmente adatto per certe istanze e abbiamo visto che poi l'ultima implementazione del grasso ci ha permesso di ulteriormente migliorare questa complessità sostituendo m con un sottoinsieme di m che potrebbe potenzialmente essere anche molto piccolo conseguenza siamo arrivati a concludere che cosa che ci sono quantomeno due algoritmi ragionevoli cioè l'algoritmo 3 e l'algoritmo 4 con implementazione del grafo guastare è con implementazione del sottoinsieme S sia come vettore di incidenza sia come lettore come lista Petrella perché

qui abbiamo usato la implementazione a tabella abbiamo scorso il vettore S questa implementazione cui si presenta la migliore per lo meno su una certa categoria di problemi mentre l'altra probabilmente la migliore su un'altra categoria di problemi e questa situazione molto variegata in realtà è tipica di problemi che non siano assolutamente semplici e banali e direi che questo dovrebbe permettersi di chiudere la lezione avendo visto un sacco di elementi interessanti vi faccio solo notare Questa non è un esempio di progetto d'esame ma nei progetti di esame richiesta un'analisi di complessità che si può condurre con queste tecniche non necessariamente con la completa pignoleria che ho adottato Ma insomma in questo modo cioè descrivendo l'algoritmo Ecco lo facciamo un esempio semplice descrivendo l'algoritmo non dico con un codice ma quantomeno come descrizione sintetica parole ricavandone una stima di complessità che non è detto debba passare attraverso due fasi Se mi dite direttamente che avete applicato una certa implementazione qui dentro al posto di tst a etpr potreste direttamente scrivere le espressioni finali ma capite che scorrevole avere la descrizione dell'algoritmo seguita dalla valutazione di complessità perché stanno bene insieme e mi dicono Quanto mi costa lagoritmo che mi hai appena raccontato e la valutazione di complessità dipende dal fatto che hai fatto operazioni e poi che hai scelto una certa implementazione che era particolarmente adatta a quelle operazioni questo direi è un modo molto scorrevole è logico di raccontare le cose quindi molto efficace nel comunicazione A questo punto chiudiamo effettivamente la lezione

Figura 8.8: Grafi pesati

Si possono definire uno o più pesi sui vertici/lati

- Un grafo pesato sui vertici (V, E, w) è un grafo $G = (V, E)$ i cui vertici sono associati a informazioni quantitative $w : V \to \mathbb{R}$
- Un grafo pesato sui lati (V, E, c) è un grafo $G = (V, E)$ i cui lati sono associati a informazioni quantitative $c : E \to \mathbb{R}$



Applicazione	vertici	lati
reti stradali	viaggi generati o attratti	lunghezze, tempi o costi di viaggio
reti elettriche	energia prodotta o consumata	costo di costruzione delle linee
reti di telecomunicazione	domanda di traffico	capacità o costo dei collegamenti
reti sociali	valore individuale	forza della relazione
giochi	valore della posizione	probabilità o costo della mossa
relazione di (in)compatibilità	utilità dell'elemento	forza della (in)compatibilità

8.3 Lezione 9-10

La lezione è centrata su un esercizio per impratichirsi sui seguenti argomenti:

- grafi rappresentati con la matrice di adiacenza

Figura 8.9: Modelli basati su grafi (1)

Qual è l'insieme massimo di persone che posso raggiungere per conoscenza?

L'insieme dei vertici V include tutti gli individui (io sono il vertice $i \in V$); l'insieme dei lati E tutte le conoscenze (coppie di individui che si conoscono)

Si trovi il sottoinsieme di massima cardinalità $U \subseteq V$ che include solo vertici u tali che esista un cammino P_{iu} fra i e u

$$U = \left\{u \in V : \exists P_{iu} = \left(\left\{v_{\pi_0}, v_{\pi_1}\right\}, \ldots, \left\{v_{\pi_{k-1}}, v_{\pi_k}\right\}\right) \text{ with } v_{\pi_0} = i, v_{\pi_k} = u\right\}$$

È vero che ognuno è a sei passi di distanza da ogni altra persona del mondo attraverso una catena di conoscenze?

L'insieme dei vertici V include gli individui; l'insieme dei lati E le conoscenze

Si trovi per ogni individuo $v \in V$ il sottoinsieme di massima cardinalità $U_v^6 \subseteq V$ che include solo vertici u tali che esista un cammino P_{vu}^6 di al più 6 lati fra v e u

$$U_v^6 = \left\{u \in V : \exists P_{vu}^6 = \left(\left\{v_{\pi_0}, v_{\pi_1}\right\}, \ldots, \left\{v_{\pi_{k-1}}, v_{\pi_k}\right\}\right) \text{ with } v_{\pi_0} = v, v_{\pi_k} = u, k \leq 6\right\}$$

Se $U_v^6 = V$ per ogni $v \in V$, la proprietà dei "sei gradi di separazione" è valida

Figura 8.10: Modelli basati su grafi (2)

Si calcoli il numero di Erdős di un matematico

L'insieme dei vertici V include tutti i matematici (quello dato è u, Erdős è v); l'insieme dei lati E include tutte le coppie con un lavoro pubblicato insieme

si trovi il cammino di minima cardinalità P_{uv} fra u e v

$$\min |P_{uv}| \text{ such that } P_{uv} = \left(\left\{u, v_{\pi_1}\right\}, \ldots, \left\{v_{\pi_{k-1}}, v\right\}\right)$$

Un museo consiste di un insieme di corridoi, che si incrociano in sale.
Dove bisogna posizionare le guardie per averne una vicina ad ogni corridoio?
Quante guardie servono per controllare l'intero museo?

L'insieme dei vertici V include tutte le sale, l'insieme dei lati E tutti i corridoi

Si trovi il sottoinsieme di vertici di minima cardinalità $U \subseteq V$ tale che ogni lato del grafo sia adiacente ad almeno un vertice di U

$$\min |U| \text{ such that } X = \{\{u, v\} \in E : \{u, v\} \cap U \neq \emptyset\} = E$$

Figura 8.11: Modelli basati su grafi (3)

Quali linee ferroviarie bisogna bombardare per distruggere ogni collegamento fra un centro industriale nemico e il fronte?

L'insieme dei vertici V include tutte le stazioni (v è il centro industriale, V^* raccoglie le stazioni al fronte), l'insieme dei lati include tutte le linee ferroviarie

$$\begin{aligned} &\min |\Delta_U| \\ &\quad \Delta_U = \{\{u,v\} \in E : |\{u,v\} \cap U| = |\{u,v\} \cap (V \setminus U)| = 1\} \\ &\quad U \ni v \\ &\quad U \subseteq V \setminus V^* \end{aligned}$$

*Dato un insieme di possibili investimenti finanziari, il loro rendimento atteso (*ROI*) e la matrice di correlazione a coppie, qual è il sottoinsieme più redditizio di investimenti scorrelati a coppie?*

L'insieme dei vertici V include gli investimenti, il peso w_v fornisce il rendimento dell'investimento $v \in V$, l'insieme dei lati include tutte le coppie correlate

$$\max \sum_{v \in U} w_v \text{ such that } U \subseteq V \text{ e } E_U = \emptyset$$

Qual è la catena più corta di cambi di una lettera da `GATTO` *a* `PESCE`*?*

. . . fatelo da voi

Figura 8.12: Grafi orientati

Se la relazione binaria è asimmetrica, l'ordine degli elementi nelle coppie è significativo

Il modello è una coppia di insiemi $G = (N, A)$ detta grafo orientato

- un insieme di oggetti elementari detti nodi
- un insieme di coppie ordinate di oggetti detti archi

Un grafo orientato si rappresenta disegnando i nodi come punti (o cerchi) gli archi come linee e il loro orientamento con frecce

$$N = \{n_1, n_2, n_3, n_4, n_5\}$$

$$\begin{aligned} A = \{&(n_1,n_3),(n_1,n_4),(n_1,n_5),(n_2,n_1),(n_2,n_3),\\ &(n_3,n_2),(n_3,n_4),(n_4,n_1),(n_4,n_2),(n_4,n_5),\\ &(n_5,n_1),(n_5,n_3),(n_5,n_4)\} \end{aligned}$$

Si notino le parentesi rotonde: la coppia è ordinata

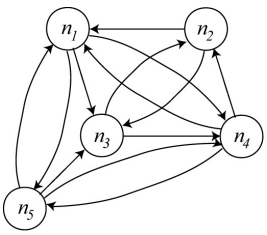


Figura 8.13: Cammini, cicli e tagli orientati

- i è la coda e j è la testa dell'arco (i, j)
- l'arco (i, j) è un arco uscente for i, un arco entrante per j
- il grado uscente δ_i^+ di un nodo $i \in N$ è il numero degli archi uscenti
- il grado entrante δ_i^- di un nodo $i \in N$ è il numero degli archi entranti
- un cammino orientato è una sequenza di archi la cui testa coincide con la coda del successivo (*tranne per l'ultimo arco*)

  $$P = \left((i_{\pi_0}, i_{\pi_1}), (i_{\pi_1}, i_{\pi_2}), \ldots, (i_{\pi_{k-1}}, i_{\pi_k})\right)$$

  I nodi i_{π_0} e i_{π_k} sono fortemente connessi e in un grafo fortemente connesso ogni coppia di nodi è fortemente connessa
- un ciclo orientato (circuito) è un cammino orientato il cui primo e ultimo nodo coincidono

  $$i_{\pi_k} = i_{\pi_0}$$

- dato un sottoinsieme di nodi $U \subset N$, la sezione uscente (entrante) Δ_U^+ (Δ_U^-) è il sottoinsieme di archi con coda (testa) in U e testa (coda) in $N \setminus U$

  $$\Delta_U^+ = \{(i, j) \in A : i \in U, j \in N \setminus U\}$$
  $$\Delta_U^- = \{(i, j) \in A : i \in N \setminus U, j \in U\}$$

Figura 8.14: Modelli basati sui grafi orientati

Alcune reti sociali considerano relazioni orientate fra gli utenti ("follower" e "leader")

In molti giochi, le posizioni evolvono in altre posizioni irreversibilmente (ad es., catture negli scacchi, le pedine nella dama, il tris...)

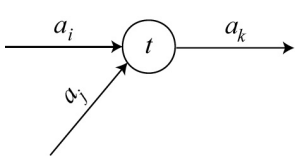
Nelle reti stradali urbane, molte strade sono a senso unico

Un progetto spesso è formato da attività soggette a una relazione binaria di precedenza, che richiede di terminare un'attività prima di cominciarne un'altra: $a_i \prec a_k$ *e* $a_j \prec a_k$

Modello activity-on-arc (AOA)

nodo $\leftrightarrow$ evento "milestone"

arco $\leftrightarrow$ attività



Modello activity-on-node (AON)

nodo $\leftrightarrow$ attività

arco $\leftrightarrow$ precedenza

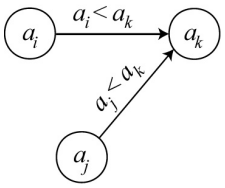


Figura 8.15: Grafi: operazioni

Ogni grafo non orientato corrisponde a un grafo orientato simmetrico

Sia $\mathcal{G}_N$ l'insieme di tutti i grafi orientati su un dato insieme di nodi N

I grafi orientati ammettono tipicamente le seguenti operazioni

- aggiunta di un arco: dato un grafo e una coppia ordinata di nodi, inserisce un arco fra i due nodi nel grafo dato

$$\text{insarco} : \mathcal{G}_N \times N \times N \to \mathcal{G}_N$$

E se l'arco esiste già?

- eliminazione di un arco: dato un grafo e una coppia ordinata di nodi, cancella dal grafo dato l'arco fra i due nodi

$$\text{cancarco} : \mathcal{G}_N \times N \times N \to \mathcal{G}_N$$

E se l'arco non esiste?

- verifica di esistenza: dato un grafo e una coppia ordinata di nodi, verifica se l'arco fra i due nodi esiste o no

$$\text{esistearco} : \mathcal{G}_N \times N \times N \to \mathbb{B} \qquad (\text{ovvero } \{0,1\})$$

Molte altre funzioni possono essere utilmente definite: lo faremo poi

Figura 8.16: Grafi: operazioni

In matematica basta definire un oggetto per crearlo

Nelle implementazioni concrete, però questo non sempre vale:
potrebbe occorrere qualche inizializzazione o allocazione dinamica

Per motivi tecnici, quindi è opportuno definire anche

- creazione: crea un grafo vuoto sull'insieme dei nodi N

$$\text{creagrafo} : N \to \mathcal{G}_N$$

- distruzione: distrugge un grafo

$$\text{distruggegrafo} : \mathcal{G}_N \to ()$$

Figura 8.17: Grafi: implementazioni

I nodi sono messi in corrispondenza biunivoca con numeri naturali

$$N \leftrightarrow \{1, \ldots, |N|\}$$

Le funzioni di peso sui nodi sono rappresentate da vettori/tabelle

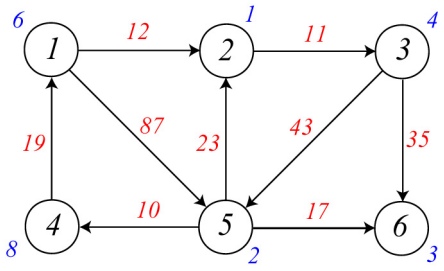
Per gli archi, ci sono tre rappresentazioni principali:

1. lista degli archi: una semplice lista/tabella che include tutti gli archi
2. matrice di adiacenza: una matrice quadrata le cui celle corrispondono a coppie di nodi
3. forward (backward) star: una lista/tabella che include per ogni nodo $i \in N$ una lista/tabella di archi uscenti (entranti)

Le funzioni di peso sugli archi si includono facilmente in ogni rappresentazione

Nei grafi non orientati, non occorrono forward e backward star (sono uguali!): ogni vertice ha una lista di incidenza

Figura 8.18: Lista degli archi



$N \to \{1, 2, 3, 4, 5, 6\}$

$w \to [\ 6\ \ 1\ \ 4\ \ 8\ \ 2\ \ 3\ ]$

$A \to \big((1,2), (1,5), (2,3), (3,5), (3,6), (4,1), (5,2), (5,4), (5,6)\big)$

$c \to [\ 12 \quad 87 \quad 11 \quad 43 \quad 35 \quad 19 \quad 23 \quad 10 \quad 17\ ]$

- Vantaggio: rappresentazione compatta ($\Theta(|A|)$)
- Svantaggio: ricerca inefficiente di un arco dato ($\Theta(|A|)$)

Figura 8.19: Lista degli archi: implementazione in C

```
typedef struct _grafo grafo;

struct _grafo
{
   int n;                                   (numero dei nodi)
   int m;                                   (numero degli archi)
   listaarchi L;                            (lista degli archi)
};
```

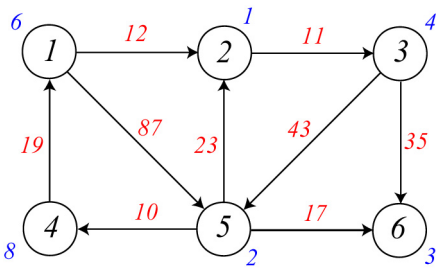
Ovviamente, occorre anche una libreria per gestire archi e liste di archi
L'esempio che segue usa l'implementazione a puntatori

```
typedef struct _arco arco;
typedef arco* listaarchi;
typedef arco* posarco;
typedef int nodo;

#define NO_ARC NULL
#define NO_NODE 0
```

```
struct _arco
{
   nodo orig, dest;
   posarco succ, pred;
};
```

Figura 8.20: Matrice di adiacenza



$$N \rightarrow \{1, 2, 3, 4, 5, 6\}$$
$$w \rightarrow [\,6\ \ 1\ \ 4\ \ 8\ \ 2\ \ 3\,]$$

$$(A, c) \rightarrow \begin{bmatrix} - & 12 & - & - & 87 & - \\ - & - & 11 & - & - & - \\ - & - & - & - & 43 & 35 \\ 19 & - & - & - & - & - \\ - & 23 & - & 10 & - & 17 \\ - & - & - & - & - & - \end{bmatrix}$$

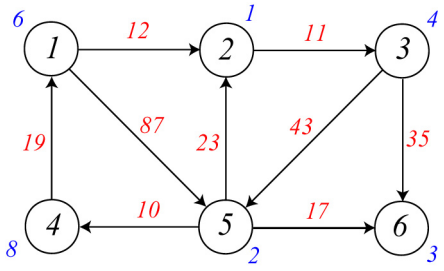
dove "-" è un valore numerico convenzionale per indicare "nessun arco"

- Vantaggio: ricerca molto efficiente di un arco dato ($\Theta(1)$)
- Svantaggio: occupazione di memoria enorme ($\Theta(|N|^2)$)

Figura 8.21: Matrice di adiacenza: implementazione in C

```
typedef struct _grafo grafo;
struct _grafo
{
   int n;                                  (numero dei nodi)
   int m;                                  (numero degli archi)
   boolean **MatriceAdiacenza;             (matrice di adiacenza)
};
```

Figura 8.22: Vettore delle liste di incidenza (*forward star*)



$$N \rightarrow \{1,\ 2,\ 3,\ 4,\ 5,\ 6\}$$
$$w \rightarrow [\ 6\ \ 1\ \ 4\ \ 8\ \ 2\ \ 3\ ]$$

$$(A,c) \rightarrow \begin{bmatrix} (2,12) \rightarrow (5,87) \perp \\ (3,11) \perp \\ (5,43) \rightarrow (6,35) \perp \\ (1,19) \perp \\ (2,23) \rightarrow (4,10) \rightarrow (6,17) \perp \\ \perp \end{bmatrix}$$

dove $\perp$ indica una lista vuota

- (S)Vantaggi: ricerca di efficienza intermedia di un arco dato $(\Theta(\delta_v^+))$
  occupazione di memoria intermedia $(\Theta(|N| + |A|))$
- Svantaggio: nessuna informazione sugli archi entranti
  (a meno che sia accompagnato dal vettore delle *backward star*)

Figura 8.23: Forward star: implementazione in C

```
typedef struct _grafo grafo;
struct _grafo
{
   int n;                                          (numero dei nodi)
   int m;                                          (numero degli archi)
   listaarchi *FS;                                 (vettore delle forward star)
   listaarchi *BS;                  (vettore delle backward star: può mancare)
};
```

Ovviamente, occorre anche una libreria per gestire archi e liste di archi
L'esempio che segue usa l'implementazione a puntatori

```
typedef struct _arco arco;          struct _arco
typedef arco* listaarchi;           {
typedef arco* posarco;                 nodo orig, dest;
typedef int nodo;                      posarco succ, pred;
                                    };
#define NO_ARC NULL
#define NO_NODE 0
```

- grafi rappresentati con la lista degli archi
- grafi rappresentati con il vettore delle liste di incidenza (*forward star*)

Ci concentriamo su grafi orientati, dato che quelli non orientati sfruttano le stesse rappresentazioni, con la convenzione che ad ogni lato (coppia non ordinata) corrispondono due archi opposti (cioè le due coppie ordinate composte dagli stessi elementi).

L'esercizio è anche dedicato all'analisi di complessità, dato che risolveremo lo stesso problema con diversi semplici algoritmi, adottando per ciascuno diverse implementazioni delle strutture dati astratte necessarie (anzi tutto, del grafo). Per ogni combinazione di algoritmo e implementazione, valuteremo la complessità.

Chi notasse eventuali incoerenze o errori, oppure avesse dubbi sul contenuto di queste pagine e dei codici, è pregato di segnalarmeli per contribuire a migliorare la qualità dei materiali del corso.

Problema

Si vuole scrivere un programma `sottografo.c` che riceva un grafo e un sottoinsieme di nodi, ed estragga dal primo il sottografo indotto dal secondo. Il programma riceve dalla linea di comando il nome di un file di testo che contiene il grafo e l'elenco degli indici dei nodi del sottografo. Deve poi stampare a video prima il grafo dato e poi il sottografo indotto in esso dai nodi elencati.

Formato di ingresso e di uscita Il grafo viene descritto descritto come un semplice elenco di archi. Ciascuno di questi è rappresentato come una coppia di

indici numerici separati da una virgola e racchiusi fra parentesi tonde. Gli indici corrispondono ai nodi del grafo e sono compresi fra 1 e il numero totale n dei nodi stessi. Per semplicità si suppone che non ci siano nodi isolati, per cui ogni nodo compare in almeno un arco dell'elenco. Per esempio, il grafo usato nei lucidi corrisponde a:

```
(1,2) (1,5) (2,3) (3,5) (3,6) (4,1) (5,2) (5,4) (5,6)
```

L'elenco dei nodi che individuano il sottografo verrà passato dall'utente attraverso la linea di comando, semplicemente sotto forma di una sequenza di numeri interi separati da spazi. Il sottografo indotto verrà stampato a video nello stesso formato usato per il file di ingresso.

Traccia della risoluzione

Prima di eseguire l'esercizio, conviene scorrere interamente i lucidi per acquisire la terminologia e le rappresentazioni fondamentali. Il punto di partenza dell'esercizio è il file `sottografo0.c`, che contiene le solite direttive, costanti simboliche e procedure di base. Inoltre, è fornito un abbozzo di libreria per la gestione di grafi orientati, costituito dai file `grafo-la.h` e `grafo-la0.c`. In essa sono previste inizialmente solo le funzioni:

- `creagrafo`, che alloca le strutture necessarie a rappresentare un grafo con un dato numero n di nodi[1];
- `distruggegrafo`, che dealloca tutte le strutture dinamiche;
- `insarco`, che aggiunge un nuovo arco, dati i nodi estremi;
- `cancarco`, che cancella un arco, dati i nodi estremi;
- `esistearco`, che indica se una data coppia di nodi corrisponde a un arco.

La libreria adotta l'implementazione con matrice di adiacenza, ma all'inizio le sue funzioni sono definite tutte vuote.

Prima fase (`sottografo1.c`) Nella prima fase, applichiamo come di consueto la modalità *top-down*, chiamando e dichiarando le funzioni:

- `InterpretaLineaComando`, che estrae dalla linea di comando il nome del file che contiene il grafo dato e l'elenco dei nodi da considerare;
- `CaricaGrafo`, che carica il grafo dal file di testo indicato nella linea di comando;
- `StampaGrafo`, che stampa il grafo a video;
- `StampaSottografoIndotto`, che determina gli archi del sottografo indotto e li stampa a video.

[1] Per semplicità, imponiamo la limitazione che il numero dei nodi sia dato all'inizio e invariabile. Una libreria più flessibile potrebbe partire da un grafo nullo (senza nodi) e includere funzioni per inserire e cancellare anche nodi, oltre che archi.

Per il momento, definiremo tre delle quattro funzioni, lasciando vuota quella di calcolo e stampa del sottografo indotto, che è l'argomento principale della lezione.

La funzione `InterpretaLineaComando` è leggermente più sofisticata del solito, dato che non deve restituire un numero di argomenti noto a priori, ma un elenco di interi di lunghezza incognita (oltre ad una stringa con il nome del file dei dati). Oltre alla solita verifica del numero di argomenti (questa volta, non fisso, ma inferiormente limitato) e all'acquisizione del nome del file che contiene il grafo, la procedura deve leggere un elenco di numeri interi. La cardinalità dell'elenco non è nota a priori, ma è determinata dal valore di `argc`: i primi due elementi della linea di comando, infatti, sono il nome del programma e il nome del file dei dati; gli altri dovrebbero essere gli indici dei nodi, e sono contenuti nel vettore `argv` dall'indice 2 all'indice `argc-1`. A questo punto, si può allocare un vettore dinamico di interidella lunghezza opportuna. Siccome gli elementi di `argv` non sono interi, ma stringhe, vanno convertiti prima di copiarli uno per uno nel vettore (consiglio: `sscanf` o `atoi`). Al termine, il vettore e la sua cardinalità vanno restituiti al `main`. Per farlo, si passano entrambi per indirizzo. Siccome si è definito il tipo `vint` come un puntatore a intero che rappresenta un vettore dinamico di interi, l'argomento sarà di tipo `vint *`. Sarebbe equivalente (ma meno chiaro) dichiararlo di tipo `int **`. Volendo semplificare la procedura, Le operazioni sul vettore dinamico si potrebbero fare nel `main` o addirittura nella successiva procedura di stampa del sottografo indotto, passandole il vettore `argv`, ma sarebbe decisamente inelegante: si tratterebbe di interpretare una parte della linea di comando fuori della funzione dedicata a tale compito, mescolando l'acquisizione dei dati con l'esecuzione dell'algoritmo. Un facile esercizio di programmazione sarebbe invece di caricare l'elenco dei nodi da un secondo file di testo, anziché leggerlo dalla linea di comando.

La funzione `CaricaGrafo` riceve il nome del file di testo e restituisce il grafo. Questo viene quindi passato per indirizzo. La funzione apre il file di testo, determina il numero dei nodi in modo da poter creare il grafo, legge gli archi uno alla volta e li aggiunge via via al grafo stesso. Questa decomposizione del problema consente di usare le funzioni fondamentali di gestione del grafo elencate più sopra. Siccome il file non fornisce il numero dei nodi del grafo, dobbiamo ricavarlo indirettamente. In generale, non sarebbe possibile, ma l'ipotesi che non ci siano nodi isolati permette di supporre che l'ultimo nodo sia quello di indice massimo che compare negli archi del grafo. Quindi, scorriamo il file una prima volta per leggere gli indici e conservare il massimo, e una seconda volta per aggiungere al grafo gli archi. Entrambi gli scorrimenti richiedono di leggere un formato non del tutto banale, con coppie di interi racchiusi fra parentesi e separati da una virgola. La funzione `fscanf` con la specifica `" (%d ,%d )"` risolve il problema[2].

Infine, definiamo la procedura che riceve il grafo e ne stampa a video gli archi. Contrariamente alla lezione sulle liste, questa volta includeremo la funzione nella libreria, perché sarebbe inefficiente se realizzata usando le poche funzioni di libreria indicate nella lezione e perché è potenzialmente utile per altre applicazioni (il formato di stampa è abbastanza naturale da supporre che non cambi). Fra l'altro, nel seguito scopriremo che le altre implementazioni del grafo suggeriranno l'introduzione di altre funzioni di libreria, che consentiranno implementazioni più efficienti della stampa, e che quindi suggerirebbero di modificare pesantemente la funzione stessa: meglio inserirla nella libreria fin dal principio. Chiameremo questa funzione `stampagrafo`, per coerenza coi nomi interamente minuscoli delle altre funzioni di

[2]Gli spazi prima delle parentesi e della virgola servono a rendere flessibile la lettura, cioè a evitare che si interrompa per non aver trovato le parentesi e la virgola esattamente al punto indicato: lo spazio indica che può esserci un numero qualsiasi, anche nullo, di separatori. Non occorrono spazi dopo le parentesi e la virgola perché anche la specifica `%d` consente la presenza di un numero qualsiasi di separatori.

libreria, e ne aggiungeremo la dichiarazione al `grafo-ma.h` e la definizione al file `grafo-ma.c`. A rigore, il grafo andrebbe passato alla funzione per valore, dato che non viene modificato, ma lo passiamo per indirizzo, perché questo riduce il passaggio di strutture di grandi dimensioni alla copia di un puntatore, rendendolo più efficiente.

Seconda fase (`grafo-ma.c`) Analogamente alla scorsa lezione, nella seconda fase passiamo alla modalità *bottom-up* per realizzare la libreria di gestione di grafi, in modo da avere subito dei risultati a video (la stampa del grafo) che ci consentano di verificare almeno in parte la correttezza dell'algoritmo. Come indicato nei lucidi, la libreria definisce:

- i nodi come numeri interi[3] da 1 a $n = |N|$;
- la relazione di adiacenza fra i nodi (dunque, gli archi) come una matrice quadrata di ordine n di valori logici (`TRUE`, cioè 1, se l'arco fra nodo di riga e di colonna esiste; `FALSE`, cioè 0, altrimenti); aggiungiamo anche il numero degli archi, anche se non strettamente necessario.

Cominciamo con le funzioni di creazione e distruzione, `creagrafo` e `distruggegrafo`. Queste funzioni consistono essenzialmente nell'allocazione e deallocazione della matrice di adiacenza, che è una matrice dinamica (all'occorrenza, si veda la pagina 16 dei lucidi della lezione 5).

Le funzioni di inserimento e cancellazione di un arco, `insarco` e `cancarco`, sono quasi banali: si tratta di inserire il valore `TRUE` o `FALSE` nella cella opportuna della matrice di adiacenza, e aggiornare il numero degli archi. Il tutto richiede tempo costante. Che succede se l'arco da inserire è già presente, o quello da cancellare manca? Si può scegliere di lasciare all'utente della libreria la responsabilità di eseguire l'operazione solo quando è lecita. In tal caso, conviene riportare questa indicazione nei commenti al prototipo della funzione, così che l'utente sia informato delle condizioni per l'uso corretto della funzione. Oppure si può scegliere di verificare la condizione e abbandonare l'esecuzione con un messaggio di errore se è violata (è una soluzione piuttosto drastica). Oppure si può decidere di verificare la condizione e, in caso di violazioni, non fare nulla, come succede nella teoria degli insiemi quando si aggiunge a un insieme un elemento che già contiene o se ne sottrae uno che non contiene. In questa implementazione, la verifica ha un costo molto basso, per cui si è scelta quest'ultima strada, ma nelle altre implementazioni il controllo appesantisce moltissimo la complessità. Sceglieremo quindi la strada dell'efficienza, assegnando all'utente della libreria la responsabilità di eseguire l'operazione solo quando è lecita. Conviene riportare questa indicazione nei commenti al prototipo della funzione, così che l'utente sia informato delle condizioni per l'uso corretto della funzione.

La funzione `esistearco`, che è l'ultima ancora da definire, consente proprio di valutare la condizione di appartenenza. La definizione è banale e richiede tempo costante: basta consultare la matrice. Ci si può chiedere che senso abbia chiamare una funzione per farlo: al solito, si tratta di mantenere separate la struttura dati astratta e la sua implementazione.

[3] In linea di principio, i nodi potrebbero avere molte informazioni associate, e si potrebbe trovare più conveniente definirli come puntatori a tali strutture. Volendo nascondere la loro definizione, si può ricorrere alla definizione di un tipo `nodo` con l'istruzione `typedef`. In realtà, in questo esercizio ci interessa solo la topologia del grafo e, anche se i nodi del grafo modellassero oggetti sofisticati, potremmo modellare le relative informazioni come funzioni di peso conservate in vettori e usare usare gli indici numerici dei nodi per accedervi. Che gli indici siano consecutivi semplifica la trattazione, perché consente di rappresentare l'intero insieme N semplicemente con la cardinalità n.

Terza fase (sottografo-ma.c) La fase successiva dell'esercizio definisce la funzione `StampaSottografoIndotto`, che, dato un grafo $G = (N, A)$ e un sottoinsieme di nodi $S \subseteq N$, determina gli archi del sottografo $G_S = (S, A_S)$ indotto da S su G e poi li stampa. Ci sono diversi algoritmi possibili per individuare gli archi del sottografo indotto. Ne analizziamo tre in maggior dettaglio.

Algoritmo 1 scorre tutte le coppie ordinate (o, d) di nodi di N e valuta per ciascuna tre condizioni: se ciascuno dei due nodi estremi appartenga al sottoinsieme S e se la coppia corrisponda a un arco di A. Se la coppia soddisfa tutte le condizioni, la stampa. Si può chiaramente descrivere questo algoritmo col seguente pseudocodice.

```
1: for all o ∈ N do
2:     for all d ∈ N do
3:         if o ∈ S and d ∈ S and (o, d) ∈ A then
4:             StampaArco(o, d);
5:         end if
6:     end for
7: end for
```

Per determinare se un dato nodo appartiene o no al sottoinsieme S, siccome il sottoinsieme è rappresentato con un vettore s (di oggetti di tipo `nodo`, che sono poi dei semplici interi), introdurremo una funzione `appartienetabella`[4], che riceverà l'elemento da cercare, il vettore s e la sua cardinalità $n_S = |S|$. La complessità temporale dell'Algoritmo 1 è quindi data da una sommatoria doppia, sugli indici dei nodi, del tempo richiesto dalla valutazione delle tre condizioni e dalla stampa:

$$T_1 = \sum_{o=1}^{n} \sum_{d=1}^{n} (2T_S + T_A + T_{\text{pr}}) \tag{8.1}$$

dove $T_S(\cdot)$ è il tempo richiesto per la valutazione dell'appartenenza di un elemento di N a S, $T_A(\cdot)$ quello per la valutazione dell'appartenenza di una coppia di elementi di N ad A e $T_{\text{pr}}(\cdot)$ il tempo per la stampa di un arco.

Se adottiamo l'implementazione del grafo con la matrice di adiacenza, la valutazione di esistenza dell'arco richiede tempo costante: $T_A \in \Theta(1)$. La valutazione di appartenenza a S di ciascuno dei due nodi richiede tempo $T_S \in \Theta(n_S)$, dato che corrisponde allo scorrimento del vettore s che elenca gli indici dei nodi[5]. Infine, la stampa dell'arco richiede tempo costante: $T_{\text{pr}} \in \Theta(1)$. Quindi, la complessità dell'Algoritmo 1 in questa implementazione è:

$$T_1(n, n_S) = \sum_{o=1}^{n} \sum_{d=1}^{n} (2\Theta(n_S) + \Theta(1) + \Theta(1)) \in \Theta\left(n^2 n_S\right) \tag{8.2}$$

Questa analisi dipende non da un parametro dimensionale, ma da due: il numero totale dei nodi e quello dei nodi del sottoinsieme S. Si può osservare che, nel caso pessimo, $n_S = n$, e quindi la complessità è in $\Theta\left(n^3\right)$. Questa analisi è corretta, ma più grossolana, e ignora il fatto che su alcune istanze (quelle relative a sottografi indotti da piccoli sottoinsiemi di nodi) l'algoritmo è più veloce che su altre.

[4] A rigore, si tratta di un vettore, dato che la sua dimensione non cambia, ma trattandosi di una parte del più ampio vettore che contiene tutti i nodi viene spontaneo pensarlo come una tabella. Ad ogni modo, non cambia nulla.

[5] E se il vettore S fosse ordinato e si usasse la ricerca binaria? Avremmo un costo aggiuntivo al principio per ordinare il vettore ($\Theta(n_S \log n_S)$) e una complessità inferiore durante l'algoritmo ($T_S \in \Theta(\log n_S)$). Non approfondiamo questa soluzione perché vedremo che gli algoritmi che spendono T_S sono in realtà tutti dominati.

La complessità spaziale dell'Algoritmo 1 è dominata dallo spazio occupato dal grafo: $S_1(n) \in \Theta(n^2)$.

Algoritmo 2 scorre tutti i nodi di N come potenziali origini, valuta per ciascuno se appartenga al sottoinsieme S; per quelli che vi appartengono, scorre tutti i nodi di N come potenziali destinazioni, valuta per ciascuno l'appartenenza al sottoinsieme S; infine, valuta se la coppia di nodi corrisponda a un arco di A e, in tal caso, la stampa.

```
1: for all o ∈ N do
2:     if o ∈ S then
3:         for all d ∈ N do
4:             if d ∈ S and (o, d) ∈ A then
5:                 StampaArco(o, d);
6:             end if
7:         end for
8:     end if
9: end for
```

È semplicemente un miglioramento dell'Algoritmo 1, basato sull'osservazione che la valutazione di appartenenza del primo nodo di ogni coppia al sottoinsieme può essere fatta nel ciclo esterno anziché in quello interno. In questo modo, se la valutazione fallisce, si evita di richiamare il ciclo interno. La complessità spaziale di questo algoritmo è ovviamente identica a quella del precedente. Quella temporale richiede un'analisi un po' più complessa, ma molto simile: il ciclo esterno ha n iterazioni che eseguono tutte la valutazione di appartenenza, ma solo n_S di queste iterazioni proseguono con l'esecuzione del ciclo interno, mentre le altre $n - n_S$ si arrestano subito:

$$T_2 = \sum_{o=1}^{n} T_S + \sum_{i_o=1}^{n_S} \sum_{d=1}^{n} (T_S + T_A + T_{\text{pr}}) \tag{8.3}$$

Adottando sempre l'implementazione del grafo con matrice di adiacenza, si ottiene:

$$T_2(n, n_S) = \Theta(nn_S) + n_S\, n\,(\Theta(n_S) + \Theta(1) + \Theta(1)) = \Theta\left(nn_S^2\right) \tag{8.4}$$

È facile vedere che la nuova complessità è identica alla vecchia per $n = n_S$, ma è molto inferiore per $n_S \ll n$.

Algoritmo 3 cambia completamente approccio, perché sfrutta il vettore che contiene i nodi del sottografo in modo da scorrere solamente loro. In dettaglio, l'algoritmo scorre tutte le coppie di posizioni nella tabella, ricava per ciascuna posizione il corrispondente nodo, valuta se la coppia di nodi corrisponda a un arco in A, e in tal caso lo stampa.

```
1: for i_o := 1 to |S| do
2:     o := s[i_o];
3:     for i_d := 1 to |S| do
4:         d := s[i_d];
5:         if (o, d) ∈ A then
6:             StampaArco(o, d);
7:         end if
8:     end for
9: end for
```

Quindi:

$$T_3 = \sum_{i_o=1}^{n_S} \left(T_{\text{find}} + \sum_{i_d=1}^{n_S} \left(T_{\text{find}} + T_A + T_{\text{pr}} \right) \right) \tag{8.5}$$

dove $T_{\text{find}}(\cdot)$ è il tempo richiesto per trovare il nodo in N dato un elemento del vettore che rappresenta S.

Adottando l'implementazione del grafo con matrice di adiacenza, e osservando che il vettore fornisce il nodo con un banale accesso in tempo costante ($T_{\text{find}} \in \Theta(1)$):

$$T_3(n, n_S) = \sum_{i_o=1}^{n_S} \left(\Theta(1) + \sum_{i_d=1}^{n_S} \left(\Theta(1) + \Theta(1) + \Theta(1) \right) \right) = \Theta\left(n_S^2\right) \tag{8.6}$$

La complessità temporale dell'Algoritmo 3 è molto migliore di quella dei precedenti, anche nel caso pessimo, mentre quella spaziale non cambia.

I tre algoritmi sono via via più efficienti. Questo non dipende in realtà dall'implementazione, anche se l'implementazione determina l'espressione precisa della loro complessità. Infatti, per qualsiasi espressione di $T_S(\cdot)$, $T_A(\cdot)$ e $T_{\text{pr}}(\cdot)$ il secondo algoritmo dominerebbe il primo, dato che fa le stesse cose, ma in alcuni casi evita il ciclo interno. Il terzo algoritmo domina il secondo purché T_{find} non sia più lenta di T_S, cosa che avviene in qualsiasi implementazione sensata. Nel seguito, quindi, abbandoneremo i primi due algoritmi.

D'altra parte, questi non sono i soli algoritmi possibili. Nel seguito ne introdurremo un quarto. Inoltre, discuteremo un'implementazione del sottoinsieme dei nodi S che consenta di ottenere un test di appartenenza più rapido. Ovviamente, è possibile combinare i quattro algoritmi con le tre implementazioni del grafo e le due implementazioni del sottoinsieme, ottenendo 24 casi, ma il modo in cui stiamo procedendo dovrebbe dare un'idea più chiara dei meccanismi all'opera nel progetto e nella valutazione di algoritmi, del ruolo che in essi giocano da un lato le strutture dati astratte e dall'altro le implementazioni. La trattazione dovrebbe anche dare un buon esempio del modo più scorrevole per descrivere un algoritmo e analizzare la sua complessità.

Quarta fase (grafo-la.c) A questo punto, procediamo a sostituire in blocco la libreria basata sulla matrice di adiacenza con una libreria basata sulla lista degli archi. Il file `grafo-la.h` riporta le definizioni di base, mentre occorre definire una per una le funzioni nel file `grafo-la0.c`, esattamente come si è fatto per l'implementazione con la matrice di adiacenza. Per semplificare l'esercizio, è già interamente definita una libreria `listaarchi` che gestisce le liste di archi, realizzandone un'implementazione a puntatori bidirezionale, circolare e con sentinella. I principi di fondo sono quelli discussi nella lezione sulle liste.

Ancora una volta, dobbiamo definire le funzioni di creazione, distruzione, aggiunta e cancellazione di archi e di valutazione dell'esistenza di un arco dati gli indici dei nodi estremi. Per farlo, sfrutteremo la disponibilità di una libreria per gestire liste di archi (`listaarchi.h` e `listaarchi.c`), che rende quasi banale l'implementazione di gran parte delle nuove funzioni. La creazione e la distruzione del grafo sfruttano la creazione e la distruzione della lista degli archi, anziché della matrice di adiacenza. L'aggiunta di un nuovo arco consiste nella sua aggiunta alla lista (per analogia con l'esercizio sulla biblioteca, aggiungeremo gli archi in coda, anche se qualsiasi posizione sarebbe corretta). La cancellazione di un arco è più complessa, perché la funzione di cancellazione dalla lista richiede di conoscere la sua posizione nella lista, mentre sono noti solo gli indici dei nodi estremi. Per trovare la posizione di un arco a partire dagli estremi, sarà necessario scorrere la lista. Questa posizione si potrà poi usare per cancellare l'arco. Infine, la funzione che valuta l'esistenza di un arco dati gli estremi esegue anch'essa uno scorrimento della lista. L'aspetto interessante è che, esattamente come nell'esercizio sulla gestione di una biblioteca, se si sostituisce la nuova libreria alla vecchia nelle direttive di inclusione e nella linea di comando della compilazione, il programma continua a funzionare correttamente senza modifiche.

Cambia però completamente la complessità dei tre algoritmi discussi più sopra. Anzi tutto, la complessità spaziale migliora in alcuni grafi, mentre peggiora in altri. Il grafo occupa infatti spazio $\Theta(m)$ e $m \leq n^2$ (in assenza di autoanelli, si ha $m \leq n(n-1)$). Questo significa che per i grafi sparsi l'occupazione è molto inferiore, mentre per quelli densi è asintoticamente la stessa, ma ha costanti moltiplicative più alte, per la presenza dei puntatori e della sentinella[6]. Quindi l'implementazione migliore dal punto di vista spaziale dipende dal tipo di istanze che probabilmente ci si troverà a risolvere.

Per quanto riguarda il tempo, abbiamo mostrato che i primi due algoritmi sono dominati dal terzo. Limitiamoci a discutere la complessità di quest'ultimo[7], e poi introduciamone un quarto.

Algoritmo 3 La ricerca di un nodo in N dato il suo indice in S e la stampa di un arco richiedono ancora tempo costante. La valutazione di esistenza di un arco dati gli estremi richiede invece tempo lineare nel numero degli archi ($T_A \in \Theta(m)$) se si adotta l'implementazione con lista degli archi[8]. Adattando l'espressione (8.5), si ottiene la stima:

$$T(n, m, n_S) = \sum_{i_o=1}^{n_S} \left(\Theta(1) + \sum_{i_d=1}^{n_S} (\Theta(1) + \Theta(m) + \Theta(1)) \right) \in \Theta\left(mn_S^2\right) \qquad (8.7)$$

che è peggiore di un fattore m di quella ottenuta con l'altra implementazione. D'altra parte, l'implementazione con lista di archi suggerisce un altro possibile algoritmo per determinare il sottografo indotto.

Algoritmo 4 Questo algoritmo è basato sull'idea di scorrere direttamente gli archi, anziché le coppie di nodi estremi potenziali. L'algoritmo scorre tutti gli archi del grafo e per ciascuno determina i nodi estremi e valuta se entrambi appartengano al sottoinsieme S, come nello pseudocodice seguente.

[6] Implementare la lista degli archi come un vettore risolverebbe questo problema.

[7] Per esercizio, si valutino le altre complessità: dovrebbero essere $\Theta\left(n^2(m+n_S)\right)$ per l'Algoritmo 1 e $\Theta(nn_S(m+n_S))$ per l'Algoritmo 2.

[8] Mettendoli in ordine lessicografico, cioè per valori crescenti dell'indice del primo estremo e poi del secondo, si potrebbe cercarli scorrendo la lista a partire dalla cima o dal fondo secondo che il nodo di origine dell'arco sia uno dei primi o degli ultimi. Questo probabilmente migliorerebbe le costanti moltiplicative nel caso medio, ma non la complessità asintotica nel caso pessimo. Tenerli in un vettore ordinato, invece, ridurrebbe il tempo a $O(m)$.

```
1: for all a ∈ A do
2:     (o, d) := LeggeEstremiArco(a);
3:     if o ∈ S and d ∈ S then
4:         StampaArco(o, d);
5:     end if
6: end for
```

Questo algoritmo richiede che la struttura dati astratta "grafo" consenta operazioni di scorrimento degli archi e di accesso ai loro estremi, di cui dovremo aggiungere alla libreria le dichiarazioni e le definizioni. Precisamente, aggiungeremo le seguenti quattro funzioni:

- `primoarco`, che restituisce la posizione del primo arco nella lista degli archi del grafo;
- `succarco`, che restituisce la posizione dell'arco successivo a quello dato nella lista degli archi del grafo;
- `finearchi`, che indica se la posizione corrente è esterna alla lista degli archi del grafo;
- `leggeestremiarco`, che determina i nodi estremi di un arco data la sua posizione nel grafo.

Per farlo, sfrutteremo la disponibilità di una libreria per gestire liste di archi (`listaarchi.h` e `listaarchi.c`), che rende l'implementazione delle nuove quattro funzioni quasi banale. Tutte richiedono tempo costante, come avevamo ipotizzato più sopra. In effetti, l'implementazione con lista degli archi è particolarmente adatta a realizzare queste operazioni, grazie all'organizzazione degli archi in una lista.

Se si ipotizza che lo scorrimento di A richieda tempo costante per ogni arco, la complessità dell'Algoritmo 4 si può stimare come:

$$T = \sum_{a=1}^{m} (2T_S + T_{\text{extr}} + T_{\text{pr}}) \tag{8.8}$$

dove $T_{\text{extr}}(\cdot)$ rappresenta la complessità di determinare gli estremi di un arco dato, che è costante se si suppone che tale informazione sia conservata nella struttura che descrive l'arco.

Si noti che nulla impedisce di aggiungere le funzioni di scorrimento degli archi all'implementazione con matrice di adiacenza: in generale, *un'implementazione non consente né impedisce l'esecuzione di un algoritmo; si limita ad accelerarla o rallentarla.* Però la cosa sarebbe tutt'altro che banale, e aprirebbe una quantità di problemi tecnici legati alla programmazione in C, che evitiamo di discutere[9]. Questo significa che la libreria `grafo-la` avrà più funzioni della libreria `grafo-ma`, e lo stesso succederà quando introdurremo la terza implementazione. Siccome nelle versioni successive dell'algoritmo utilizzeremo le funzioni aggiuntive, non sarà possibile usare una qualsiasi delle tre librerie con la versione finale dell'algoritmo, anche se concettualmente dovrebbe esserlo.

[9] Limitandosi a un accenno, bisognerebbe definire in qualche modo la posizione di un arco in A (ad es., con un puntatore di tipo `boolean *` alla cella della matrice) e implementare come funzioni le operazioni di ricerca del primo arco, di passaggio all'arco successivo e di verifica che lo scorrimento sia terminato (ad es., con cicli che saltino da una cella di valore `TRUE` alla successiva). Il tutto potrebbe essere un esercizio istruttivo, ma sarebbe una soluzione piuttosto barocca e inefficiente.

Infine, la disponibilità delle nuove quattro funzioni consente anche di implementare la funzione `stampagrafo` in modo più efficiente. In dettaglio, la vecchia funzione scorreva tutte le coppie di nodi, valutava l'appartenenza della coppia corrente all'insieme degli archi e, nel caso, la stampava, impiegando nel complesso tempo $\Theta\left(n^2\right)$. La nuova funzione scorre direttamente gli archi e li stampa, in tempo $\Theta\left(m\right)$. L'aver incorporato nella libreria la funzione di stampa consente di mantenere invariato il suo uso in tutte le versioni dell'algoritmo, evitando il problema descritto più sopra.

Quinta fase (`sottografo-la.c`) Ora possiamo implementare l'Algoritmo 4 per determinare il sottografo indotto. Con le nuove funzioni aggiunte alla libreria scorrere gli archi è molto semplice. L'operazione più costosa è invece il test di appartenenza di un generico nodo al sottoinsieme S. Esistono due principali implementazioni per rappresentare un sottoinsieme S di un dato insieme N:

1. una *tabella* o un vettore che contenga gli elementi del sottoinsieme stesso (tabella se il sottoinsieme è variabile, vettore se è costante;
2. un *vettore di incidenza*, cioè un vettore di $|N|$ valori logici, associati agli elementi dell'insieme N, ciascuno dei quali indica se il corrispondente elemento appartiene o no al sottoinsieme S.

Per generalità, nella prima implementazione parleremo di tabella anche se nel nostro caso l'insieme S non subisce modifiche, ed è quindi un semplice vettore.

Ovviamente, entrambe le strutture vanno costruite, e vanno aggiornate se il sottoinsieme viene modificato. La scelta tra le due implementazioni dipende dalle operazioni necessarie:

- scorrere il sottoinsieme implementato come tabella richiede tempo $\Theta\left(|S|\right)$, mentre scorrerlo implementato come vettore di incidenza richiede $\Theta\left(|N|\right)$;
- valutare l'appartenenza di un elemento all'insieme richiede $\Theta\left(|S|\right)$ con l'implementazione come tabella, $\Theta\left(1\right)$ con quella come vettore di incidenza;
- la complessità spaziale della tabella è $\Theta\left(|S|\right)$, mentre quella del vettore di incidenza è $\Theta\left(|N|\right)$.

Nell'Algoritmo 3, l'implementazione come vettore di incidenza non offre alcun vantaggio, perché l'operazione principale è lo scorrimento dell'insieme S e non si fanno test di appartenenza. Discutiamo invece l'Algoritmo 4.

Algoritmo 4 (S come tabella) Se il grafo è implementato come lista di archi e il sottoinsieme S come tabella, la complessità dell'espressione (8.8) diventa:

$$T\left(m, n_S\right) = \sum_{a=1}^{m} \left(\Theta\left(1\right) + 2\Theta\left(n_S\right) + \Theta\left(1\right)\right) \in \Theta\left(m n_S\right) \tag{8.9}$$

Algoritmo 4 (S come vettore di incidenza) Prima di eseguire l'algoritmo, bisogna costruire il vettore di incidenza V_S del sottoinsieme $S \subseteq N$, per cui lo pseudocodice diventa:

```
1: for i := 1 to n do
2:     V_S[i] := false;
3: end for
```

```
4: for all o ∈ S do
5:     V_S[o] := true;
6: end for
7: for all a ∈ A do
8:     (o, d) := LeggeEstremiArco(a);
9:     if V_S[o] = true and V_S[d] = true then
10:        StampaArco(o, d);
11:    end if
12: end for
```

e l'espressione (8.8) della complessità, modificata con due termini aggiuntivi che descrivono la costruzione e l'inizializzazione del vettore V_S, diventa:

$$T(m,n) = \Theta(n) + \Theta(n_S) + \sum_{a=1}^{m} (\Theta(1) + \Theta(1) + \Theta(1) + \Theta(1)) \in \Theta(n+m) \quad (8.10)$$

Notiamo, di passaggio, che la tabella s è sempre disponibile, per cui stiamo in effetti rappresentando l'insieme S in due modi diversi. In questo pseudocodice, però, la tabella serve solo a contenere i dati nel formato fornito dall'utente (se l'utente avesse fornito il vettore di incidenza, la tabella non ci sarebbe).

Ora ricordiamo che la complessità dell'Algoritmo 3 peggiora implementando il grafo come lista degli archi e non trae vantaggio dall'implementare il sottoinsieme S come vettore di incidenza. Ne deriva che la sua migliore implementazione (finora) richiede tempo $\Theta\left(n_S^2\right)$. Per l'Algoritmo 4, invece, l'implementazione di S come vettore di incidenza ($\Theta(m+n)$) domina quella come tabella ($\Theta(mn_S)$), a meno che n_S sia una costante molto piccola (e probabilmente nemmeno allora). Ma in tal caso, è chiaro che l'Algoritmo 3 sarebbe nettamente migliore. Restano quindi in lizza:

- l'Algoritmo 3 con l'implementazione del grafo come matrice di adiacenza ($\Theta\left(n_S^2\right)$) per istanze con sottoinsiemi piccoli e grafi densi;
- l'Algoritmo 4 con l'implementazione del grafo come lista di archi e del sottoinsieme come vettore di incidenza ($\Theta(m+n)$) per istanze con sottoinsiemi grandi e grafi sparsi.

Sesta fase (`grafo-fs.c` e `sottografo-fs.c`) Terminiamo l'esercizio realizzando l'implementazione del grafo con il vettore delle *forward star*, cioè delle liste degli archi uscenti da ciascun nodo. Bisogna quindi realizzare le funzioni `creagrafo` e `distruggegrafo`, le quali, rispettivamente, allocano e deallocano tale vettore. La funzione `esiste_arco` ha complessità intermedia fra quelle delle due precedenti implementazioni. Infatti, scorre solo gli archi uscenti dal nodo iniziale o dell'arco dato (o,d), in tempo $\Theta(\delta_o^+)$, dove δ_o^+ è il grado uscente del nodo o[10].

Teoricamente, bisognerebbe poi implementare le funzioni per lo scorrimento della lista globale degli archi, che non esiste. La funzione `primoarco` potrebbe scorrere i nodi per indice crescente fino a trovarne uno che abbia archi uscenti e restituire il primo di tali archi. La complessità sarebbe $\Theta(n)$ nel caso pessimo (ma molto probabilmente in media sarebbe $\Theta(1)$). Analogamente si potrebbe ragionare per le altre. Tuttavia, queste funzioni sono piuttosto inutili, perché si può scorrere l'intero insieme degli archi combinando uno scorrimento dei nodi con uno scorrimento degli

[10] Vale anche qui quanto detto per accelerare la ricerca nella lista degli archi ordinandoli lessicograficamente.

archi uscenti da ciascun nodo. Ci limiteremo quindi a sostituire in blocco quelle funzioni con altre che gestiscono la lista di archi uscenti da ciascun nodo. Queste funzioni sono facili da implementare usando la libreria `listaarchi` e richiedono tutte tempo costante.

Completata la libreria, possiamo reimplementare i due algoritmi rimasti in lizza e valutarne la complessità. Quella spaziale è dominata dal vettore delle *forward star*, che richiede spazio $\Theta(n+m)$, dato che occorrono n celle nel vettore per accedere a tutte le liste e ognuno degli m archi compare esattamente in una lista.

Algoritmo 3 La forma dell'algoritmo non cambia, dato che usa solo la funzione `esistearco`, che è ancora presente nella libreria, ma bisogna rivalutare la complessità (8.5):

$$T(n, n_S) = \sum_{i_o=1}^{n_S} \left(\Theta(1) + \sum_{i_d=1}^{n_S} \left(\Theta(1) + \Theta\left(\delta_o^+\right) + \Theta(1) \right) \right) \in \Theta\left(n_S^2 \delta_{\max}^+\right) \qquad (8.11)$$

dove $\delta_{\max S}^+$ rappresenta il massimo grado uscente[11] per i nodi di S. Ancora una volta, questa implementazione è peggiore di quella con la matrice di adiacenza.

Algoritmo 4 Questo algoritmo non cambia concettualmente, ma lo scorrimento della lista degli archi viene scomposto nello scorrimento dei nodi e, per ciascun nodo, della sua *forward star*.

```
1: for i := 1 to n do
2:     V_S[i] := false;
3: end for
4: for all o ∈ S do
5:     V_S[o] := true;
6: end for
7: for all o ∈ N do
8:     for all a ∈ Δ_o^+ do
9:         (o, d) := LeggeEstremiArco(a);
10:        if V_S[o] = true and V_S[d] = true then
11:            StampaArco(o, d);
12:        end if
13:    end for
14: end for
```

D'altra parte, la decomposizione del ciclo su tutti gli archi offre una preziosa occasione per ridurre ulteriormente la complessità temporale. Infatti, non è più necessario considerare tutti gli archi: basta considerare quelli uscenti da nodi del sottoinsieme S. Questo consente di risparmiare anche il test sul primo estremo dell'arco.

```
1: for i := 1 to n do
2:     V_S[i] := false;
3: end for
4: for all o ∈ S do
5:     V_S[o] := true;
6: end for
7: for all i_o = 1 to n_S do
8:     o := s[i_o];
9:     for all a ∈ Δ_o^+ do
10:        (o, d) := LeggeEstremiArco(a);
```

[11] La stima è migliorabile, ma non occorre, dato che scarteremo questa implementazione.

```
11:         if V_S[d] = true then
12:             StampaArco(o, d);
13:         end if
14:     end for
15: end for
```

Un'altra funzione sarà necessaria per determinare in ciascun arco il nodo di destinazione (se si vuole conservare l'astrazione ed evitare accedere direttamente alle liste e ai puntatori che le implementano). La funzione `leggeestremi`, infatti, presume che il grafo contenga una sola lista e non è direttamente applicabile in questo caso.

Si noti anche che questo pseudocodice usa entrambe le rappresentazioni dell'insieme S (tabella e vettore di incidenza) per eseguire vere e proprie operazioni dell'algoritmo: la tabella per identificare l'origine di ogni arco, il vettore di incidenza per valutare l'appartenenza della destinazione al sottoinsieme desiderato.

L'algorimo risultante ha complessità:

$$T(n, m, n_S) = \Theta(n) + \sum_{i_o=1}^{n_S} \sum_{a=1}^{\delta_o^+} \Theta(1) = \Theta\left(n + \sum_{i_o=1}^{n_S} \delta_o^+\right)$$

dove il termine $\sum_{i_o=1}^{n_S} \delta_o^+$ conta gli archi uscenti dai nodi di S, che sono certamente meno di m, eventualmente molto meno. Questo algoritmo è più efficiente di quello in $\Theta(n+m)$ ed è superato da quello in $\Theta\left(n_S^2\right)$ solo per sottoinsiemi molto piccoli e grafi molto densi.

Capitolo 9

Visita di grafi e componenti connesse

Questo capitolo è dedicato allo studio delle proprietà di connessione di un grafo e ai relativi algoritmi. Considereremo sia la connessione semplice (per grafi non orientati o per grafi orientati simmetrici) sia la connessione forte (per grafi orientati generici). Questi problemi sono strettamente legati a un problema più elementare, detto problema della *visita di un grafo*. Vedremo che esistono diversi algoritmi per risolvere questo problema (come ce ne sono diversi per ordinare un vettore).

9.1 Componenti connesse di un grafo

Abbiamo visto nello scorso capitolo che due vertici di un grafo non orientato sono connessi se esiste un cammino che li ammette come estremi. Questo implica che la connessione è una relazione binaria sui vertici del grafo. Tale relazione è considerata riflessiva per convenzione, perché ogni vertice si può vedere come legato a sé stesso da un cammino vuoto. È simmetrica perché nei grafi non orientati i lati sono simmetrici, cioè le coppie di estremi sono liberamente scambiabili fra loro. Infine, è una relazione transitiva perché se c'è un cammino da v_1 a v_2 e uno da v_2 a v_3, c'è anche un cammino da v_1 a v_3, costituito dal concatenamento dei primi due. Ne concludiamo che la relazione di connessione semplice in un grafo non orientato è una relazione di equivalenza. I sottoinsiemi massimali di vertici reciprocamente connessi formano classi di equivalenza, che si dicono *componenti connesse* del grafo. Nell'esempio della Figura 9.1 i vertici 1, 2, 4 e 5 sono fra loro raggiungibili. Per esempio, 1 raggiunge direttamente 2 e 4 e indirettamente 5 in vari modi: con il cammino $(1, 4, 5)$ o il cammino $(1, 2, 5)$, o il cammino $(1, 4, 2, 5)$, ecc.... I cammini sono in generale molti, e se ammettiamo anche quelli non elementari o non semplici, come $(1, 2, 4, 1, 2, 5)$, che comprendono cicli, eventualmente ripetuti più volte, cominciamo a intuire la potenziale difficoltà del problema di determinare le componenti connesse di un grafo. D'altra parte, i vertici 3 e 6 sono direttamente connessi fra loro, ma sconnessi dagli altri. Quindi, il grafo in figura ha due classi di equivalenza, che non sono in alcun modo collegabili fra loro.

A che cosa servono le componenti connesse? I grafi sono uno strumento modellistico onnipresente, e le componenti connesse spesso corrispondono a oggetti di grande importanza (vedi Figura 9.2). Per esempio, se il grafo descrive una rete stradale, le sue componenti connesse sono posizioni geografiche mutualmente raggiungibili per mezzo di strade: Milano e Oslo sono reciprocamente connesse, mentre

Su un grafo non orientato $G = (V, E)$, la relazione di connessione è

- riflessiva (per convenzione)
- transitiva
- simmetrica

Quindi i vertici raggiungibili formano classi di equivalenza
che sono dette componenti connesse

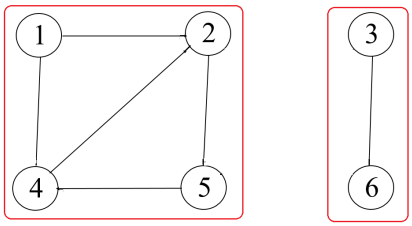


Figura 9.1: Componenti connesse di un grafo

Milano e New York non lo sono. Se il grafo descrive gli stati di funzionamento di un impianto, le componenti connesse sono sottoinsiemi di stati che si possono mutuamente trasformare l'uno nell'altro. Se il grafo descrive una rete sociale, le componenti connessi sono blocchi di individui in grado di comunicare fra loro, magari indirettamente. Se è un sistema di equazioni e i lati indicano che un'equazione ha variabili in comune con un'altra, blocchi di equazioni fra loro connessi dovranno essere risolti in qualche modo insieme, mentre blocchi indipendenti si potranno vedere come problemi separati. Un problema potrebbe essere enorme, ma costituito da tanti sottoproblemi di dimensioni molto più piccole. Se questo è il caso, la risoluzione ne è grandemente facilitata. Capita quindi spesso che problemi complessi contengano un sottoproblema nel quale occorre determinare le componenti connesse di un opportuno grafo.

Quali sono le difficoltà fondamentali poste da questo problema? La prima è che in genere si può io posso andare da un vertice a un altro in tanti modi diversi e nessuno è ovviamente meglio di un altro, finché il problema riguarda solo la topologia del grafo. Una seconda difficoltà sta nel fatto che i cammini potrebbero contenere cicli, e avere lunghezza qualsiasi, illimitata. Una terza possibile difficoltà sta nel fatto che non è garantito che la connessione può anche non essere soddisfatta: in genere esistono anche coppie di vertici non reciprocamente raggiungibili. Se consideriamo il problema delle componenti fortemente connesse, dovremo tener conto anche del fatto che gli archi non sono simmetrici, ma asimmetrici.

La strada maestra per determinare le componenti connesse di un grafo è osservare che è possibile ridurlo a un altro problema più semplice (vedi Figura 9.9), che ne costituisce il nocciolo duro. Quest'altro problema è detto problema della visita di un grafo ed è costituito da un grafo non orientato $G = (V, E)$ e uno specifico vertice s, detto *sorgente*. La soluzione del problema è il sottoinsieme dei vertici del grafo che sono raggiungibili dal vertice sorgente. Perché interessa questo sottoproblema?

Determinare le componenti connesse di un grafo è utile a individuare

- posizioni geografiche mutuamente raggiungibili
- stati di funzionamento di un sistema mutuamente trasformabili
- individui che possono comunicare fra loro
- blocchi di equazioni indipendenti tra loro
- ...

Insomma, è un sottoproblema abbastanza comune

Le principali difficoltà da affrontare nel risolvere questo problema sono

- la presenza di cammini alternativi fra gli stessi vertici
- l'esistenza di cicli, che tornano indietro a vertici già raggiunti
- l'esistenza di componenti separate, per cui non si può scorrere l'intero insieme dei vertici V passando attraverso i lati

Per le componenti fortemente connesse, c'è anche l'asimmetria degli archi

Figura 9.2: Determinare le componenti connesse

9.2 Determinazione delle componenti connesse

Perché qualsiasi algoritmo per risolvere il problema della visita automaticamente consentirebbe di risolvere il problema delle componenti connesse, procedendo nel modo indicato nella Figura 9.9. Per prima cosa definiamo l'insieme dei vertici già visitati dall'algoritmo, inizialmente vuoto. Quindi, estraiamo uno dei vertici non visitati. Possiamo usare questo vertice come sorgente per risolvere il problema della visita, con l'algoritmo che abbiamo ipotizzato di avere. Otterremo un sottoinsieme di vertici U_v. È semplice mostrare che U_v è una delle componenti connesse del grafo di partenza. Infatti, esiste esattamente una componente connessa del grafo che contiene il vertice v. Che relazione ha tale componente con U_v? La componente è contenuta in U_v, perché tutti i vertici che stanno nella stessa componente connessa di v sono raggiungibili da v, perché sono vicendevolmente raggiungibili l'uno dall'altro. D'altra parte l'insieme U_v è contenuto nella componente che stiamo cercando, perché l'insieme dei vertici raggiungibili da v è anche l'insieme dei vertici coraggiungibili da v, per la simmetria della relazione di connessione. Perciò qualsiasi coppia di vertici (v_1, v_2) in questo insieme è connesso a v e v è connesso a ciascuno di essi. Ma allora v_1 è connesso a v e v è connesso a v_2, e per transitività v_1 è connesso a v_2, cioè qualunque coppia di vertici in U_v è vicendevolmente connesso. Di conseguenza, U_v è contenuto nella componente connessa che contiene v.

Se prendiamo uno dei vertici non ancora visitati, lo assumiamo come sorgente, eseguiamo l'algoritmo di visita a partire da esso, costruiamo il sottoinsieme raggiunto e marchiamo i suoi vertici come visitati, possiamo poi proseguire cercando un altro vertice non visitato, dunque un'altra componente, e ripetere l'algoritmo finché esistono vertici non visitati. Questo è l'algoritmo di determinazione delle componenti connesse. Terminato l'algoritmo, avremo determinato un certo numero c di componenti connesse, che costituiscono una partizione dell'insieme V dei vertici

del grafo.

Assumiamo di aver un algoritmo di visita dal vertice sorgente s, cioè un algoritmo che enumera i vertici del grafo (V, E) raggiungibili da s

$$U_s \leftarrow \text{visita}\,(V, E, s) \qquad (\text{con } s \in U_s \subseteq V)$$

Per determinare le componenti connesse basta

1. definire un insieme dei vertici visitati, inizialmente vuoto
2. considerare un vertice v non visitato
   - visitare il grafo da v: U_v è una componente connessa, perché
     - sia C_v la componente connessa (unica) contenente v
     - $U_v \supseteq C_v$ perché i vertici di C_v sono raggiungibili da v
     - $U_v \subseteq C_v$ perché i vertici di U_v sono vicendevolmente raggiungibili
       passando per v e sfruttando la simmetria dei cammini
3. se esistono ancora vertici non visitati, tornare al punto 2

Al termine, i sottoinsiemi U_v ottenuti formano una partizione

Figura 9.3: Componenti connesse e visita di un grafo

9.2.1 Vettore di marcatura

Per rappresentare i singoli sottoinsiemi di questa partizione si potrebbero usare liste, tabelle o altre strutture già viste. In particolare, potremmo usare dei vettori di incidenza. Possiamo però sfruttare il fatto che le componenti sono tra loro disgiunte per evitare di avere un vettore per ciascuna, e risparmiare spazio e tempo. La rappresentazione che ne deriva prende spesso il nome di *vettore di marcatura* (vedi Figura 9.5). Si tratta di un vettore C che associa a ogni vertice v di V (cioè ad ogni indice intero fra 1 e $|V|$) non un valore binario (come sarebbe in un vettore di incidenza), ma l'indice c_v della componente cui v appartiene. Quindi, invece di avere i valori 0 e 1, abbiamo i valori interi da 1 al numero totale $c_{\max}$ delle componenti connesse del grafo, più lo 0 se ammettiamo vertici che (al momento, temporaneamente) non appartengono ad alcuna componente.

Con questa struttura dati, l'operazione di inserire un vertice in una componente (operazione richiesta dall'algoritmo appena discusso) costa $\Theta(1)$, perché si tratta solo di marcare un elemento del vettore, una sostituzione. D'altra parte, una volta risolto il problema, se si vuole sapere a quale componente appartiene un vertice, basta accedere con un'operazione di proiezione al corrispondente elemento del vettore, e il costo è ancora in $\Theta(1)$.

Altre operazioni potrebbero essere meno efficienti, ma la determinazione delle componenti non le richiede. L'uso di $c_{\max}$ vettori di incidenza avrebbe gli stessi vantaggi, ma richiederebbe molto più spazio. L'uso di liste o tabelle richiederebbe una ricerca nelle tabelle o liste per capire a quale componente un vertice appartiene, analogamente a quanto succedeva nel capitolo precedente per capire se un elemento appartiene o no a una tabella o a una lista. Il costo sarebbe $\Theta(n)$ per lo scorrimento

Figura 9.4: Rappresentazione delle componenti connesse di un grafo
Come rappresentare i sottoinsiemi U_v?
Il vettore di marcatura C indica la componente cui appartiene ogni $v \in V$
(con un indice c intero progressivo, con l'indice s della sorgente, o altro)

- inserire un vertice in una componente richiede $\Theta(1)$
- cercare la componente cui appartiene un vertice richiede $\Theta(1)$

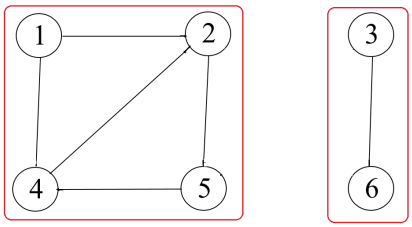
Altre operazioni sono meno efficienti, ma non sono richieste

Figura 9.5: Il vettore di marcatura

completo di liste o tabelle non ordinate, $\Theta(\log n)$ per la ricerca binaria in tabelle ordinate.

L'uso di molti vettori di incidenza rende anche più complicata la ricerca di un nuovo vertice non visitato, dato che richiede di verificare $c_{\max}$ elementi di vettori, anziché uno solo.

Figura 9.6: Esempio



- tutti i vertici sono marcati come non assegnati: `C = [ 0 0 0 0 0 0 ]`
- si parte dalla sorgente $s = 1$
- la visita da $s = 1$ restituisce $U_1 = \{1, 2, 4, 5\}$: `C = [ 1 1 0 1 1 0 ]`
- si salta la sorgente $s = 2$, che è già marcata ($C_2 \neq 0$)
- la visita da $s = 3$ restituisce $U_3 = \{3, 6\}$: `C = [ 1 1 2 1 1 2 ]`
- si saltano le sorgenti $s = 4$, 5 e 6, che sono già marcate ($C_4 = C_5 = C_6 \neq 0$)

Figura 9.7: Esempio di determinazione delle componenti connesse

La Figura 9.7 illustra un semplice esempio. Inizialmente, i vertici sono tutti marcati come non assegnati nel vettore di marcatura. Partiamo da una sorgente scelta arbitrariamente, per esempio il vertice 1, e visitiamo a partire da essa tutti i vertici raggiungibili. Non importa, per il momento, come lo facciamo: il problema è dato per risolto in partenza. Visitiamo i vertici 2, 4 e 5, ottenendo la prima componente, che contiene anche il vertice 1 perché per riflessività un vertice è sempre raggiungibile da sé stesso. Quindi, marchiamo questi elementi nel vettore di marcatura con un indice.

Ci sono due convenzioni standard per l'indice da usare nel vettore di marcatura. La prima è di usare un intero progressivo, come già visto. Siccome questa è la prima componente, marchiamo 1, 2, 4 e 5 con l'indice 1. La seconda convenzione comune è di usare un indice che ricordi la sorgente della visita. In questo caso, la marcatura sarebbe ancora 1, inteso però come indice della sorgente, non numero progressivo delle componenti identificate.

Al passo successivo, bisogna cercare un vertice non visitato. Viene spontaneo scorrerli tutti nel vettore fermandosi al primo con marcatura nulla (ovviamente senza ripetere la ricerca sui vertici già considerati). Scopriamo, in tempo costante, che il vertice 2 è già stato visitato, e quindi passiamo al vertice 3, che invece non è marcato. Quindi, il vertice 3 può fare da sorgente. Eseguiamo la visita partendo dal vertice 3 e otteniamo l'insieme $U_3 = \{3, 6\}$. Nelle posizioni corrispondenti del vettore C scriviamo l'indice progressivo 2 oppure l'indice della sorgente, cioè 3. La scelta della convenzione dipende dalle applicazioni.

Qui adottiamo la marcatura con l'indice progressivo c
(*al termine, fornisce il numero di componenti connesse*)

```
ComponentiConnesse(V,E)
{
  c := 0;
  for each s in V
    C[s] := 0;

  for each s in V
  {
    if (C[s] = 0)
    {
      c := c+1;
      visita(V,E,s,C,c);
    }
  }
  return C;
}
```

Figura 9.8: Algoritmo per la determinazione delle componenti connesse

La Figura 9.7 riporta lo pseudocodice dell'algoritmo. Questo riceve il grafo, inizializza a 0 il vettore di marcatura e scorre i vertici nell'ordine. Saltando quelli che sono già stati visitati, quando trova una sorgente incrementa l'indice progressivo delle componenti ed esegue una funzione di visita. Questa richiede il grafo, la sorgente, il vettore di marcatura e l'indice con il quale deve marcare i vertici raggiunti nel corso della visita. Al termine, la funzione restituisce il vettore di marcatura.

Prima di passare al codice, notiamo che il vettore di marcatura è una rappresentazione completa e corretta della soluzione, ma non univoca. Scambiando fra loro gli indici delle componenti, si ottiene un vettore diverso ($C = [\,2\,2\,1\,2\,2\,1\,]$), che però rappresenta la stessa partizione. È un problema? Non per quanto riguarda questo corso, ma potrebbe esserlo se volessimo enumerare le soluzioni o se le generassimo esaustivamente e volessimo evitare di generare più volte la stessa. Infatti, ci sono applicazioni nelle quali la presenza di $c_{\max}!$ permutazioni di indici è particolarmente

sgradevole.

9.3 Laboratorio

Nell'esercizio pratico realizzeremo per prima cosa un programma che determini le componenti connesse di un dato grafo non orientato, assumendo di disporre di un algoritmo per la visita del grafo stesso a partire da un dato vertice.

Il codice da cui partiremo consiste nel file `compconn0.c` e nelle librerie per gestire i grafi introdotte nel capitolo precedente. In particolare, rappresenteremo il grafo come un vettore di liste di adiacenza, perché approfondendo il problema della visita scopriremo che l'algoritmo di visita ha bisogno di conoscere per ogni vertice tutti i vertici adiacenti. La rappresentazione che fornisce questa informazione nel modo più efficiente possibile è il vettore delle liste di adiacenza.

```
int main (int argc, char *argv[])
{
  char filedati[ROW_LENGTH];
  grafo G;
  vint C;
  int nc;


  /* Legge da linea di comando il file che contiene il grafo */
  InterpretaLineaComando(argc,argv,filedati);

  /* Carica il grafo */
  CaricaGrafo(filedati,&G,FALSE);

  /* Stampa a video il grafo */
  printf("G = ");
  stampagrafo(&G);
  printf("\n");

  /* Crea una soluzione C vuota */
  nc = 0;
  C = (vint) calloc(G.n+1,sizeof(int));
  if (C == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione del vettore di marcatura
        C!\n");
    exit(EXIT_FAILURE);
  }

  /* Determina le nc componenti connesse del grafo G nel vettore C */
  ComponentiConnesse(&G,C,&nc);

  /* Stampa le componenti connesse */
  StampaComponenti(nc,C,G.n);

  /* Dealloca le strutture dati dinamiche */
  distruggegrafo(&G);
  free(C);

  return EXIT_SUCCESS;
}
```

Non c'è molto da commentare. La funzione di interpretazione della linea di comando si limita a leggere dalla linea di comando il nome del file dei dati. La funzione che carica il grafo è leggermente diversa da quella realizzata nello scorso

capitolo: ha un argomento in più, il valore logico `Orientato`, che serve ad applicare questo programma sia a grafi non orientati (che stiamo trattando ora) sia a grafi orientati.

Il file `grafo-no.txt`, infatti, ha un formato ambiguo: contiene un elenco di coppie, ma non specifica se si tratti di coppie ordinate o no, quindi di un grafo orientato o no. Potremmo aggiungere l'indicazione nel file. Potremmo anche distinguere le coppie ordinate da quelle non ordinate rappresentando le prime con parentesi tonde e le altre con parentesi graffe. Questa sarebbe un'ottima idea, ma è una convenzione che solo qualche libro di testo e qualche articolo particolarmente attento adotta, mentre nella maggior parte dei casi non si segue questa convenzione. Quindi, ipotizzeremo invece che l'utente aprendo il file sappia già se esso rappresenta un grafo orientato o no. Nel nostro caso, il grafo è non orientato.

```
(1,2) (1,4) (2,4) (2,5) (3,6) (4,5)
```

Siccome le procedure di gestione dei grafi sono adattate ai grafi orientati, per applicarle a un grafo non orientato basta convertire il grafo letto da file nel corrispondente grafo orientato simmetrico, sostituendo ciascun lato con una coppia di archi opposti. Per esempio, quando si legge la coppia $(1, 2)$ si aggiunge al grafo non solo l'arco $(1, 2)$ ma anche l'arco $(2, 1)$. La modifica è molto semplice: basta un'istruzione in più nel ciclo di inserimento degli archi.

```
  }

  strcpy(filedati,argv[1]);
}

/* Carica dal file filedati il grafo G, assumendo che sia orientato o
    no secondo il valore di Orientato */
```

Ovviamente, se abbiamo commesso un errore e il grafo è in realtà orientato, stamo aggiungendo degli archi in eccesso. Se poi il grafo nel file contenesse già coppie di archi opposti, per esempio $(1, 2)$ e $(2, 1)$, il grafo risulterebbe avere due volte ciascuno dei due archi. In pratica, gli algoritmi che consideriamo qui non ne verrebbero inficiati, ma in generale potrebbe essere un problema.

Caricato il grafo, il programma determina le componenti connesse del grafo `G` e conserva le componenti nel vettore di marcatura `C` e il loro numero in `nc`. Il vettore fornisce per ogni indice $v \in V$ l'indice intero progressivo C_v che identifica la componente a cui appartiene v. Come noto, si potrebbe anche usare l'indice del vertice usato come sorgente per individuare la componente. Scartiamo invece le idee di rappresentare la soluzione con `nc` tabelle o liste o vettori di incidenza binari (uno per ogni componente): tabelle e liste renderebbero inefficiente il test di appartenenza, i vettori di incidenza richiederebbero molto più spazio.

Quindi, il programma stampa le componenti e termina deallocando le strutture dati dinamiche. Vedremo che, in effetti, l'uso di tabelle o liste renderebbe invece più efficiente questa operazione. Come al solito, procediamo in modalità *top-down*.

9.3.1 Prima fase

Nella prima fase, dato che il caricamento del grafo è già disponibile, procediamo a realizzare la ricerca delle componenti connesse e la loro stampa a video.

Per la prima funzione, è necessario decidere dove allocare e dove inizializzare il vettore C. Potremmo costruire una funzione che alloca internamente il vettore e lo restituisce all'esterno, come abbiamo fatto in alcuni capitoli precedenti, oppure potremmo distinguere la gestione della soluzione del problema dal vero e proprio algoritmo che lo risolve e costruire una soluzione vuota (cioè allocare un vettore della lunghezza opportuna), per poi passarla a una funzione che determinare le componenti connesse e riempie la soluzione stessa. Seguiremo questa strada, dato che separa meglio il problema dai dettagli tecnici e consente di usare la funzione anche in casi in cui una struttura per la soluzione esiste già per qualche altro motivo. Contrariamente allo pseudocodice, assumeremo quindi che la soluzione sia inizializzata fuori dall'algoritmo che determina le componenti connesse. Questo permette anche, volendo, di completare soluzioni parziali, cioè in cui alcune componenti sono note e altre no. Volendo applicare il concetto di struttura dati astratta, si potrebbe infine costruire una libreria per rappresentare una partizione di un insieme. In questo caso, `C` e `nc` sarebbero dichiarata come campi di una partizione e sarebbe possibile avere funzioni per costruire, distruggere e stampare una partizione, nonché passare una partizione vuota alla funzione che determina le componenti connesse per ottenerne una partizione completa. Per brevità, non lo facciamo.

Il programma dovrà poi stampare le componenti connesse nel formato seguente. La prima riga indicherà il numero delle componenti, seguito dalla parole chiave `componenti`:

```
2 componenti
```

Quindi un'altra riga fornirà il vettore di marcatura racchiuso fra parentesi quadre e preceduto dalle parole chiave `C =`.

```
C = [ 1 1 2 1 1 2 ]
```

Infine, stamperemo tante righe quante sono le componenti, riportando gli indici dei vertici che ne fanno parte, preceduti dalle parole chiave `U[`c`] =`, dove c è l'indice progressivo della componente via via considerata.

```
U[1] = 1 2 4 5
U[2] = 3 6
```

Per quanto riguarda la soluzione, definiremo nel `main` un vettore dinamico di interi e un intero.

```
vint C;
int nc;
```

Il primo andrà esplicitamente allocato (e deallocato alla fine), il secondo andrà inizializzato a 0. Anche il vettore di marcatura va inizializzato, e il valore 0 fornito da `calloc` può essere una scelta del tutto accettabile per indicare che i vertici al momento non appartengono ad alcuna componente.

```
/* Carica il grafo */
CaricaGrafo(filedati,&G,FALSE);

/* Stampa a video il grafo */
printf("G = ");
stampagrafo(&G);
printf("\n");
```

La funzione `ComponentiConnesse` riceve il grafo, passato per indirizzo per efficienza (non perché venga modificato) e restituisce in uscita il vettore `C` e l'intero `nc`: solo il secondo viene passato per indirizzo perché il primo è già stato allocato e la procedura ne modificherà solo gli elementi.

```
/* Calcola le nc componenti connesse del grafo G e le salva in C */
ComponentiConnesse(&G,C,&nc);
```

La chiamata va accompagnata da una dichiarazione nella sezione dedicata ai prototipi: è una funzione `void` perché i risultati sono dichiarati nel sezione dei dati, riceve un vettore di interi e un puntatore a intero.

```
/* Calcola le nc componenti connesse del grafo G e le salva in C */
void ComponentiConnesse (grafo *pG, vint C, int *pnc);
```

A questa dichiarazione corrisponde una definizione con corpo vuoto nella sezione finale del codice, che riempiremo in seguito.

La stampa delle componenti connesse utilizza la funzione `StampaComponenti`, che andrà bene anche per stampare le componenti fortemente connesse. La procedura richiede il numero di componenti e il vettore di marcatura, di cui dobbiamo anche conoscere la lunghezza. Quindi i dati saranno `nc`, `C` e `G.n`. Ne deriva la chiamata:

```
/* Stampa le componenti connesse */
StampaComponenti(nc,C,G.n);
```

e il prototipo, in cui si specifica che la procedura non ha risultati e che il numero di componenti è intero, il vettore è formato da interi e la lunghezza è un intero.

```
/* Stampa le nc componenti descritte dal vettore di marcatura C di
   lunghezza n */
void StampaComponenti (int nc, vint C, int n);
```

Questa funzione stampa qualsiasi partizione di un insieme di numeri consecutivi compresi fra 1 e la lunghezza nel formato descritto sopra, il che la rende più generica e valida per usi diversi.

Ora possiamo realizzare il corpo delle due funzioni. La determinazione delle componenti connesse ripete fedelmente lo pseudocodice, che è già molto chiaro, salvo l'inizializzazione che è stata esternalizzata nel `main`. Rimane un ciclo che scorre tutte le potenziali sorgenti nell'insieme dei vertici, un test che valuta se il vertice corrente è già stato raggiunto, e quindi marcato, e una chiamata a una procedura di visita. L'indice `s` per le sorgenti scorre sugli indici dei vertici. Il test verifica che `C[s]` sia nullo. In caso positivo, si incrementa il numero di componenti `nc`, che va inizializzato a 0 prima del ciclo. Nel farlo, si badi alla precedenza fra operatori: le parentesi sono essenziali per evitare che l'incremento sia applicato al puntatore `pnc` anziché all'oggetto puntato. Sempre in caso positivo, chiamiamo l'algoritmo di visita (di cui realizzeremo tre versioni diverse, quindi per ora gli diamo un nome generico), il quale riceve la sorgente `s`, il vettore di marcatura `C` e l'indice da assegnare nel vettore di marcatura ai vertici visitati. Avendo scelto di usare un indice progressivo,

questo indice al momento è il numero corrente delle componenti, `*pnc`. Altrimenti, potrebbe essere l'indice `s` della sorgente. Lo pseudocodice suggerirebbe di restituire come risultato il vettore di marcatura (a rigore, anche il numero di componenti), ma sappiamo che in C si tende a trattare i risultati di grandi dimensioni come dati passati per argomento per poterli modificare all'interno della procedura.

```
/* Calcola le nc componenti connesse del grafo G e le salva in C */
void ComponentiConnesse (grafo *pG, vint C, int *pnc)
{
  nodo s;


  *pnc = 0;
  for (s = 1; s <= pG->n; s++)
    if (C[s] == 0)
    {
      (*pnc)++;
      visita(pG,s,C,*pnc);
    }
}
```

Ovviamente, occorre aggiungere il prototipo. La funzione `Visita` è di tipo `void`, riceve per indirizzo il grafo `G`, l'indice intero della sorgente `s`, marca nel vettore `C` i vertici visitati con l'indice `c`.

```
/* Visita il grafo G a partire dalla sorgente s marcando i vertici
    visitati nel vettore C con l'indice c */
void visita (grafo *pG, nodo s, vint C, int c);
```

Rimane la stampa delle componenti. La funzione `StampaComponenti` riceve il numero di componenti e il vettore di marcatura. Stampa subito il numero con la relativa parola chiave, poi scorre il vettore stampandone gli elementi nell'ordine, con un pochino di contesto per rendere più leggibile il tutto. Queste operazioni sono molto semplici ed è facile realizzarle in modo efficiente. Più complicato è ottenere la seconda parte della stampa, che non corrisponde esattamente alla rappresentazione usata. Per brevità useremo un approccio inefficiente, ma semplice, scorrendo il vettore C varie volte, una per ogni componente, e stampando ogni volta solo gli indici dei vertici della componente corrente. Anche qui aggiungeremo un po' di contesto per migliorare la leggibilità.

```
/* Stampa le nc componenti descritte dal vettore di marcatura C di
    lunghezza n */
void StampaComponenti (int nc, vint C, int n)
{
  int i, c;


  printf("%d componenti\n",nc);
  printf("C = [ ");
  for (i = 1; i <= n; i++)
    printf("%d ",C[i]);
  printf("]\n");

  for (c = 1; c <= nc; c++)
  {
    printf("U[%d] = ",c);
    for (i = 1; i <= n; i++)
      if (C[i] == c) printf("%d ",i);
```

```
    printf("\n");
  }
}
```

Siccome la procedura di visita à ancora vuota, il vettore rimane interamente nullo, ma possiamo valutare la correttezza sintattica del codice e vedere se i risultati sono coerenti con le premesse. Compilando

```
gcc -Wall -pedantic compconn1.c grafo-fs.c listaarchi.c -o compconn.exe
```

e lanciando l'eseguibile sul file di dati `grafo-no.txt` otterremo

```
0 componenti
C = [ 0 0 0 0 0 0 ]
U[1] =
U[2] =
U[3] =
U[4] =
U[5] =
U[6] =
```

che corrisponde al fatto di avere incrementato il numero di componenti ad ogni iterazione del ciclo che scorre i 6 vertici, di non avere modificato il vettore di marcatura inizializzato a 0 e di non avere alcun vertice assegnato a ciascuna delle 6 componenti fittizie.

Per poter procedere, dobbiamo risolvere il problema della visita di un grafo.

9.4 Visita di grafi

Il problema della visita di un grafo a partire da un vertice sorgente consiste nel determinare il sottoinsieme dei vertici che sono raggiungibili dal vertice sorgente stesso usando i lati del grafo. L'idea di base degli algoritmi di visita è riassunta nella Figura 9.9.

Si tratta di gestire due insiemi di vertici:

- l'insieme dei vertici visitati, cioè via via raggiunti, che alla fine costituirà la soluzione del problema;
- l'insieme di vertici che visitati, ma non ancora usati per visitarne altri.

Abbiamo quindi l'insieme complessivo V dei vertici, un sottoinsieme U di vertici visitati che va crescendo e un suo sottoinsieme Q, che fa da frontiera attiva e fa crescere U, come nelle piante. Il concetto fondamentale è che quando visitiamo un vertice lo annotiamo da parte (in Q, e quindi anche in U) e lo usiamo per visitare tutti i vertici adiacenti ad esso, attraverso la sua lista di adiacenza. Una volta che un vertice è stato visitato e anche usato per scorrere i suoi adiacenti (dunque è ancora in U, ma non è più in Q), non ha senso tenerlo ancora in considerazione, perché fornirà sempre lo stesso risultato. Insistere su quel vertice è inutile e dannoso, perché porterebbe a procedere indefinitamente. Quindi, tutti gli algoritmi di visita partono dal vertice sorgente dato e lo inseriscono nell'insieme dei vertici visitati e in quello dei vertici visitati e non ancora usati. Poi prendono uno di questi ultimi

L'idea fondamentale è di gestire due insiemi di vertici:

- l'insieme U dei vertici visitati (cioè raggiunti)
- l'insieme Q dei vertici visitati, ma non usati per visitarne altri (*ovviamente,* $Q \subseteq U \subseteq V$)

Ogni vertice deve essere visitato e usato per visitarne altri una sola volta
Usarli più volte è

- inutile, perché i vertici adiacenti sono gli stessi
- dannoso, perché si cicla indefinitamente sugli stessi vertici

Gli algoritmi di visita

- partono dalla sorgente: $U \leftarrow \{s\}$, $Q \leftarrow \{s\}$
- per ogni vertice $v \in Q$ visitato, ma non ancora usato
  - scorrono l'insieme `Adj(v)` dei vertici adiacenti a v
  - se un vertice adiacente w è non ancora visitato (test su U)
    - aggiungono w a U (*ora è visitato*)
    - aggiungono w a Q (*non è ancora stato usato per visitarne altri*)

Il grafo come vettore di *forward star* fornisce `Adj(v)` efficientemente

Figura 9.9: Visita di un grafo

vertici attivi, indicato con `v` e lo usano per scorrere tutti i suoi adiacenti. L'insieme dei vertici adiacenti viene qui indicato con `Adj(v)`. Per ognuno si va a vedere se è già stato visitato oppure no. Nel primo caso, non serve fare nulla: è già nell'insieme U, è già entrato nell'insieme Q ed eventualmente ne è già uscito. Se invece `v` non è ancora stato visitato, bisogna aggiungerlo a U e a Q.

L'operazione di scorrimento dei vertici adiacenti giustifica la scelta di rappresentare il grafo come vettore di liste di adiacenza, che finora avremmo potuto sostituire con altre implementazioni. Bisogna però ora ragionare sull'implementazione dell'insieme U e dell'insieme Q, in base alle operazioni che dobbiamo compiere su di essi (vedi Figura 9.10

Come rappresentare i due sottoinsiemi U e Q?

Al solito, dipende dalle operazioni richieste dall'algoritmo

L'insieme U dei vertici visitati richiede:

1. di aggiungere un elemento nuovo
2. di verificare l'appartenenza di un elemento dato

Con un vettore di incidenza entrambe le operazioni richiedono $\Theta(1)$

Per gestire diversi insiemi U, si può usare un vettore di marcatura C

L'insieme Q dei vertici visitati, ma non usati richiede:

1. di aggiungere un elemento nuovo
2. di estrarre un elemento qualsiasi

Con un vettore di incidenza, l'aggiunta richiede $\Theta(1)$, l'estrazione $\Theta(n)$!

Le liste consentono entrambe le operazioni in $\Theta(1)$
se si limitano l'aggiunta e l'estrazione alle posizioni estreme

Con questi vincoli, è possibile anche risparmiare spazio

Diverse strutture per Q danno luogo a diversi algoritmi di visita

Figura 9.10: Rappresentazione degli insiemi ausiliari nella visita di un grafo

Sull'insieme U si eseguono dei test di appartenenza, perché per ogni vertice `w` adiacente a un vertice `v` visitato bisogna sapere se è già stato visitato, e delle operazioni di aggiunta, quando il vertice è nuovo. La struttura dati più efficiente per queste due operazioni è il vettore di incidenza. L'insieme Q, invece, deve garantire la possibilità di aggiungere elementi velocemente, ma anche estrarre velocemente un elemento non noto a priori. Il vettore di incidenza è veloce per la prima operazione, ma non per la seconda, dato che richiede di scorrere il vettore finché non si trova un elemento dell'insieme. Useremo quindi una lista. Siccome non ci interessa estrarre un elemento particolare, ma qualsiasi elemento va bene, adotteremo delle liste che sono più efficienti in termini di spazio di quelle generali, ma limitano strettamente l'accesso agli elementi estremi della lista: il primo o l'ultimo. Queste liste si chiamano, rispettivamente, *pile* e *code*. Rappresenteremo quindi l'insieme Q con una pila oppure con una coda. Secondo che si usi l'una o l'altra, si otterranno due algoritmi di visita diversi.

9.4.1 Visita in ampiezza

La visita in ampiezza è conserva i vertici visitati, ma non usati, in una coda. La sua struttura è descritta nella Figura 9.11.

La visita in ampiezza conserva i vertici visitati non usati in una coda Q

```
BFS(V,E,s,C,c)
{
  Q := ∅;
  C[s] := c;
  Enqueue(s,Q);
  while not IsEmpty(Q) do
  {
    v := Front(Q);
    Dequeue(Q);
    for each w in Adj(v) do
      if (C[w] = 0)
      {
        C[w] := c;
        Enqueue(w,Q);
      }
  }
}
```

Figura 9.11: Visita in ampiezza

Si parte avendo inserendo l'indice `s` del vertice sorgente nell'insieme U e nell'insieme Q. Per l'insieme U non occorre nemmeno un vettore di incidenza, dato che il vettore di marcatura `C` svolge la stessa funzione. In effetti, `C` si può vedere come la combinazione di tanti vettori di incidenza binari in un solo vettore intero: quando `C[v]` vale 0, l'elemento `v` non appartiene ad alcuna componente; quando `C[v]` vale `c`, l'elemento `v` appartiene alla componente di indice `c` e a nessun'altra. Questo è consentito dal fatto che le componenti connesse formano una partizione, cioè sono disgiunte tra loro. Quindi, quando visitiamo un grafo a partire da una sorgente non usiamo l'intero vettore di marcatura, ma solo gli elementi della componente connessa visitata. Questi elementi sono sicuramente tutti marcati con uno 0 quando l'algoritmo di visita parte e non c'è alcun rischio di sovrascrivere elementi sbagliati. Per inserire un vertice in una componente, anziché fissare a 1 l'indice corrispondente nel vettore di incidenza della specifica componente, fissiamo a `c` l'indice nell'unico vettore di marcatura. Volendo, si può visualizzare `C` come $\sum_{c=1}^{c_{\max}} V_c c$, dove V_c sarebbe il vettore di incidenza della componente di indice c, per cui $V_{cv} = 1$ equivale a `C[v] = c`.

Per gestire la coda Q, invece, occorre definire un insieme di funzioni *ad hoc*. Per prima cosa occorre creare una coda vuota. Quindi, al principio inseriremo nella coda l'indice `s` del vertice sorgente e nel ciclo inseriremo in coda l'indice `w` del generico nuovo vertice raggiunto. Questa operazione in gergo viene definita `Enqueue` (accodamento). Dopo di che, finché esistono elementi nell'insieme Q, se ne estrae uno. Per costruzione, l'unico elemento raggiungibile è quello che sta in testa. L'operazione che lo determina si chiama in gergo `Front`, mentre l'operazione di estrarlo si chiama `Dequeue` (si noti che `Front` legge l'elemento senza estrarlo, mentre `Dequeue` lo estrae senza leggerlo: sono operazioni ridotte all'essenzialità).

Per ogni vertice adiacente a `v` che non sia in U, procediamo ad aggiungerlo ad U e a Q con le operazioni già descritte.

9.4.2 Implementazione delle code come vettori

A questo punto occorre descrivere l'implementazione fisica della coda. Ovviamente, si può usare una qualsiasi delle molte librerie per la gestione di liste che abbiamo già discusso. Però le rappresentazioni che abbiamo considerato non sono molto efficienti rispetto all'occupazione di memoria e non sono molto facili da usare. Anche le tabelle andrebbero bene, dato che gli elementi che dobbiamo gestire sono indici di vertici, e quindi conosciamo a priori il numero massimo di elementi che una coda potrebbe dover contenere, cioè il numero totale di vertici del grafo. Però la cancellazione e la ricerca di elementi non sono molto efficienti[1] Esiste però una rappresentazione delle code che ha tutti i vantaggi che possiamo desiderare. Si tratta di usare un semplice vettore di elementi. Questa rappresentazione non ha alcun *overhead* di memoria, non richiede puntatori e consente di realizzare tutte le operazioni di base delle code nella maniera più efficiente possibile in tempo costante.

La coda è una lista con inserimento dal fondo ed estrazione dalla cima
Si parla di gestione *FIFO*, ovvero *First-In First-Out*
(il primo elemento inserito sarà il primo estratto)

Nel caso della visita di grafi, l'algoritmo ha tre proprietà molto utili

1. la coda non conterrà mai più di $n = |V|$ elementi:
   si può gestire la coda come un vettore `V`, senza puntatori
2. ogni elemento entra dalla coda ed esce dalla testa:
   si eseguono inserimenti e cancellazioni con due semplici indici interi:
   - `tail` è la posizione in cui va inserito il prossimo elemento
     (dunque, la prima posizione libera)
   - `head` è la posizione da cui si estrae il prossimo elemento
     (dunque, la prima posizione occupata)
3. ogni elemento estratto dalla coda non vi rientrerà mai più

Figura 9.12: Implementazione delle code come vettori

L'idea fondamentale della coda è che si tratta di una lista gestita in modalità *first-in first-out* (*FIFO*): gli elementi entrano in una posizione ben determinata, che è quella di coda ed escono da una posizione ben determinata, che è quella di testa (vedi Figura 9.12). Questo permette di manipolare in maniera più semplice la struttura. Nel caso specifico che ci interessa, cioè la visita di grafi, abbiamo poi altri due vantaggi rispetto all'usdo generico di una coda. Il primo è che conosciamo il massimo numero di elementi che potrebbero mai essere contenuti nella coda: non ci saranno mai più di n elementi, dove n è il numero dei vertici del grafo. Questo vuol dire che possiamo definire l'occupazione massima possibile di memoria. Le liste a puntatori servono perché sono totalmente dinamiche e non pongono un limite massimo alla memoria. Qui non servono. Il secondo vantaggio è che gli elementi entrano nella coda al massimo una volta: se un elemento esce dalla coda non ci

[1] Su questo devo ragionare meglio: in fondo pile e code somigliano molto a delle tabelle.

rientra più. Questo si vede bene nella Figura 9.11: la sorgente `s` entra in coda al principio, il generico vertice `w` appena raggiunto vi entra nell'ultima istruzione del ciclo interno. In entrambi i casi, il vettore di marcatura viene immediatamente contrassegnato con l'indice positivo `c`. Questa marcatura non viene mai azzerata. D'altra parte, l'unico modo per entrare nella coda è superare il test che richiede di avere marcatura nulla. Quindi, un vertice che entra in coda non ci rientra mai più: si entra e si esce al massimo una volta. Vedremo che questo semplifica ulteriormente la gestione.

Una coda può essere facilmente rappresentata da un vettore che contiene l'informazione e da due indici numerici, che determinano le posizioni estreme. Convenzionalmente, si indica con `tail` la posizione della coda e con `head` quella della testa. Per essere precisi, `tail` non è l'ultima posizione occupata dalla coda stessa, ma la prima posizione libera, cioè la posizione in cui si andrà a inserire un eventuale nuovo elemento. Nella Figura 9.13 compare un vettore di 10 elementi, allocato per contenere la coda $(8, 6, 3, 4, 1)$. Questa coda ha 8 in testa e 1 in coda: 8 è l'elemento che è arrivato per primo e verrà estratto per primo (appunto, *first-in first-out*, mentre 1 è l'elemento arrivato e che verrà estratto per ultimo. L'indice `tail` vale 9, cioè un eventuale ulteriore arrivo verrà scritto nella posizione 9 del vettore. Nella parte inferiore della figura, si vede arrivare l'elemento 5, e questo viene scritto nella posizione di coda. Ovviamente, a questo punto, la posizione di coda si sposta in avanti e potenzialmente sfora la memoria allocata. D'altra parte, abbiamo notato che in un algoritmo di visita ogni vertice entra in coda e ne esce al massimo una volta e i vertici sono n. Ne consegue che non avremo mai più di n estrazioni e non sforeremo mai la memoria allocata. Vedremo poi come gestire questo problema in altre applicazioni, in cui è invece possibile fare più estrazioni del limite.

Se invece vogliamo conoscere l'elemento in cima alla coda, basta recuperare l'indice `head`, che vale 4, e leggere l'elemento del vettore in tale posizione. Se vogliamo estrarlo dalla coda, cioè cancellarlo, non faremo altro che incrementare `head` da 4 a 5 e automaticamente l'elemento non sarà più in coda, senza alcun bisogno di cancellarlo fisicamente. È piuttosto semplice gestire questa struttura, finché ci si vincola rigorosamente a fare solo inserimenti in coda e accessi ed estrazioni dalla testa.

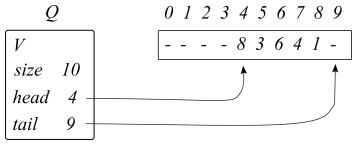
Una possibile implementazione in C è riportata nella Figura 9.14. Una coda di interi si può rappresentare con un vettore di interi, l'indicazione della sua dimensione allocata e due indici interi che determinano la testa (posizione occupata ed estratta per prima) e la coda (prima posizione libera, disponibile per il primo nuovo elemento).

9.4.3 Seconda fase: visita in ampiezza

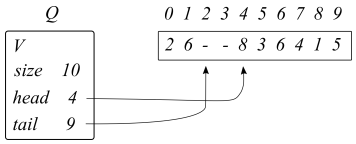
Procediamo ora a implementare la visita in ampiezza di un grafo, ipotizzando di avere la libreria `intqueue` che permette di definire e gestire code di interi. In particolare, la libreria fornisce due funzioni per creare e distruggere la coda, una funzione per valutare se la coda è vuota, una per leggere l'elemento in testa alla coda (senza estrarlo), una per estrarlo (senza leggerlo) e una per inserire in coda un nuovo elemento. Ovviamente, bisogna includere nel `main` il file di intestazione della libreria `intqueue.h`, per poter definire variabili di tipo coda e usare le relative funzioni.

Date queste funzioni, lo pseudocodice della Figura 9.11 è già estremamente vicino alla soluzione.

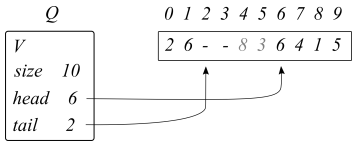
```
/* Visita in ampiezza il grafo G a partire dalla sorgente s marcando
   i vertici visitati in C con l'indice c */
```



`Front(Q)` restituisce `Q.V[Q.head]`, cioè 8
`IsEmpty(Q)` restituisce `(Q.tail == Q.head)`, cioè *false*



Inserimento di tre elementi: `Enqueue(5,Q)`, `Enqueue(2,Q)`, `Enqueue(6,Q)`



Cancellazione di due elementi: `Dequeue(Q)`, `Dequeue(Q)`

Figura 9.13: Esempi

```
void BFS (grafo *pG, nodo s, vint C, int c)
{
  intqueue Q;
  nodo v, w;
  posarco pa;


  /* Crea una coda vuota */
  CreateQueue(&Q,pG->n);

  /* Marca la sorgente s e la inserisce nella coda */
  C[s] = c;
  Enqueue(s,&Q);

  /* Finche' la coda non e' vuota */
  while (!IsEmptyQueue(&Q))
  {
    /* Estrae l'elemento in testa alla coda */
    v = Front(&Q);
    Dequeue(&Q);

    /* Scorre i vertici adiacenti a tale elemento */
    for (pa = primoarcoFS(pG,v); !finearchiFS(pG,v,pa); pa =
        succarcoFS(pG,v,pa))
    {
      /* Se l'elemento adiacente non e' ancora stato visitato */
      w = leggedestarco(pG,v,pa);
      if (C[w] == 0)
      {
        /* Marca l'elemento adiacente e lo inserisce nella coda */
        C[w] = c;
        Enqueue(w,&Q);
```

```
typedef struct _intqueue intqueue;
struct _intqueue
{
  int *V;
  int size; /* dimensione massima della coda */
  int head; /* indice dell'ultima posizione occupata */
  int tail; /* indice della prima posizione libera */
};
```

L'accesso in lettura al primo elemento è banale

```
Front(Q)
{
  Return Q.V[Q.head];
}
```

La coda è vuota quando la posizione in testa è la prima libera

```
IsEmpty(Q)
{
  Return (Q.tail == Q.head);
}
```

Figura 9.14: Implementazione delle code come vettori

```
      }
    }
  }

  DestroyQueue(&Q);
}
```

La generica procedura `Visita` è stata ribattezzata `BFS`, aggiornando dichiarazione, definizione e relativi commenti, nonché la chiamata entro la procedura per il calcolo delle componenti connesse.

La procedura di visita parte con una coda vuota. Come nel caso del vettore di marcatura, anche qui è possibile allocare e deallocare la struttura dinamica all'interno della procedura oppure farlo all'esterno e riceverla nell'elenco degli argomenti. Contrariamente al vettore di marcatura, teniamo la gestione interna. Questa è una scelta abbastanza arbitraria, fatta per semplicità. In molti casi potrebbe convenire la scelta opposta. Un motivo di portare la gestione della memoria dinamica all'esterno potrebbe essere di riutilizzare la coda più volte, allocandola e deallocandola coda una volta sola, anziché una volta per ogni componente connessa. Uno svantaggio è costringere l'utente della procedura di visita a gestire esplicitamente la memoria della struttura ausiliaria anche quando la procedura viene chiamata una volta sola. Ovviamente, la coda `Q` viene passata per indirizzo alla procedura `CreateQueue`, dato che questa ne deve modificare i campi. Bisogna passare anche la dimensione della coda, che sappiamo essere pari al numero di nodi del grafo, dato che certamente non occorre un numero maggiore di elementi. Al termine della procedura sarà necessario distruggere la coda.

Una volta allocata la coda `Q`, marchiamo la sorgente `s` nel vettore `C` con l'indice `c` e la inseriamo nella coda con la funzione di libreria `Enqueue`. Finché la coda non è vuota (circostanza verificata con la funzione di libreria `IsEmptyQueue`, leggiamo l'elemento in cima con `Front` e lo estraiamo con `Dequeue`. Tutte queste funzioni passano la coda per indirizzo, in alcuni casi perché ne modificano i campi, in altri per semplice efficienza. Partendo dal vertice estratto dalla coda, scorriamo tutti i vertici adiacenti usando la lista degli archi uscenti. Il cursore `pa` scorre gli archi della lista, e di ciascuno bisogna determinare la destinazione, come già fatto nel capitolo precedente. Quindi, si valuta se la destinazione è già stata visitata consultando il vettore di marcatura. In caso negativo, marchiamo il vertice adiacente e lo inseriamo nella coda, con le stesse operazioni eseguite sulla sorgente. Il codice è sintatticamente corretto, a patto che si compilino, oltre a `main.c`, tutte le librerie utilizzate, cioè `grafo-fs.c`, `listaarchi.c` e `intqueue.c`. Ovviamente, finché le funzioni per la gestione della coda sono vuote, il risultato non è corretto.

9.4.4 Terza fase: funzioni di libreria delle code

Procediamo ora a realizzare le 6 funzioni di libreria per gestire una coda di interi.

1:09:30 Cosa vuol dire creare una coda crea una coda vuol dire che qua dentro ci sono questi quattro campi che vanno riempiti coerentemente con l'idea che si tratta di una coda vuota abbiamo una certa dimensione abbiamo un vettore di interi perfetto il vettore PQ Reccia vi va al locato della Dimensione corretta con una funzione culloch sai se la dimensione allocata quindi qui non farò il solito trucchetto di scrivere sai il più uno perché non sto usando la coda come vettore da1n la coda viene usata come vettore dalla posizione ed alla posizione Tail è tutto interno è nascosto in questa tabella non mi occorre accedere alla posizione i-esima di questo vettore e quindi lo tratto diversamente io faccio sempre 6 + 1 n + 1 perché sono vettori A cui Accedo con un indice che ha un significato esterno utile all'utente vieni qui avrò semplicemente SAIS e i singoli elementi sono degli interi ancora una volta mi servirà un test di allocazione corretta ne avevamo fatta una che mi piacerebbe recuperare ma che non trovo più va bene La riscrivo Se PQ freccia B = Nulla succede giustamente un messaggio di errore andiamo a stampare errore nella locazione del della coda e a questo punto ci resta semplicemente da andare a determinare gli altri campi Allora il campo sai è banalmente il valore dell'argomento Size mentre Dobbiamo capire dove andare a settare la testa e la coda Allora È abbastanza evidente dalle esempio che abbiamo fatto in precedenza che Via via gli elementi verranno aggiunti in coda per indici crescenti e verranno estratti dalla testa quindi la testa tanto vale che sia il primissimo elemento disponibile cioè lo zero D'altra parte la coda sarà il primo elemento in cui inserire oggetti

1:12:08 Perché sia il primissimo elemento disponibile cioè lo zero D'altra parte la coda sarà il primo elemento in cui inserire oggetti quando avremo la possibilità di inserire oggetti e quindi sarà Zero anche lui per cui pdq testa Sarà inizialmente zero e pdq coda sarà anch'esso il valore Zero Come si fa a distruggere una coda è molto semplice si dealloca tutto quello che contiene quindi si dealloca il vettore p q freccia lì e si vanno a settare gli altri campi in maniera che sia abbastanza evidente che cosa è successo per ulteriore pulizia il puntatore pqv Che punta ancora la zona de allocata è bene che sia settato a nulla la dimensione allocata sarà 0 e in un certo senso la coda sarà come se fosse vuota e quindi Tutti questi elementi saranno tranquillamente Zero anche loro come si fa a sapere se una coda è vuota Questa è abbastanza interessante come era la coda vuota all'inizio all'inizio avevamo è detto Ale che puntano la stessa posizione e Cosa succederebbe se puntassero la stessa posizione non zero ma in un altro situazione è molto semplice vorrebbe dire che quella posizione è la posizione in cui se ci fosse qualcosa ci sarebbe il primo elemen-

to della coda D'altra parte anche la posizione in cui andavi inserire un elemento che diventerà il primo visto che la coda e vuota Conclusione quando testa e coda sono identiche è proprio il caso di dire che la coda È vuota la coda non è vuota Quando quando sono diversi perché quando sono diversi vuol dire che c'è un elemento in testa e che se voglio aggiungere qualcosa Devo andare a scriverlo altrove per cui la condizione il mtq è banalissima e diventa che head and Tail siano esattamente lo stesso numero intero Come si fa a determinare l'elemento che sta in testa Ovvio è l'elemento del vettore che sta in posizione PQ ed è quindi vado a prendere nel vettore V l'elemento che sta nell'indice PQ ed è questo è semplice come si fa ad accordare un nuovo elemento è semplice si tratta di fare una scrittura dove nella posizione Tail E cosa si va a scrivere nella posizione Tail l'elemento i che stiamo aggiungendo Quindi io mi vado a mettere nel vettore in posizione Tail e in quella posizione scrivo it ho finito No non ho finito Perché a questo punto se mi arrivano al progetto continuo a scriverlo sempre lì e non va bene Devo spostare in avanti la posizione PQ freccia Tail e quindi la devo incrementare Qui si pone un problemino il problemino è nel caso in questione non andremo mai a suonare il vettore ma in generale potremmo sforare il lettore E allora E allora discutiamo un attimo questo caso cioè Supponiamo che questo non sia un vettore di nodi di un grafo usato in una visita ma sia qualcos'altro sia un vettore di interi e quindi abbiamo una coda di interi e Supponiamo che a questa coda si debba inserire l'intero 5 poi li integro due poi l'intero sei quindi il primo sarà il 5 che viene scritto Come dice giustamente il l'indice tail in posizione 9 dopodiché si sposta e diventa 10 se arriva qualcos'altro in due io non posso scrivere nel posizione 10 perché esterna il vettore farei un danno è grave devo uscire Devo segnalare un errore si potrebbe farlo ma in realtà Qualcuno mi potrebbe far osservare che all'inizio del vettore o quattro posizioni libere le quali posizioni All'inizio dei tempi non erano libere sono state occupate ma probabilmente Ho estratto 4 dalla coda con delle operazioni di The q e queste posizioni qui Sono sfruttabili tranquillamente Cioè In effetti io potrei ancora dire che questo elemento due lo vado a mettere in posizione 0 che cosa mi serve Mi serve che quando inserisco l'elemento 5 nell'indice 9 anziché incrementare 9 A 10 anzi che fa passare te il da 9 a 10 lo faccio passare a zero e questa è aritmetica modulare è un orologio che dopo che sono passate le 11 va a finire sulle 12 e quando sono passate le 12 va a finire sul 1 anziché sulle 13 e quindi non è un grosso problema quindi l'idea 1:16:57 A finire sulle 12 e quando sono passate le 12 va a finire sul 1 anziché sulle 13 e quindi non è un grosso problema quindi l'idea è gestire Tail e poi in realtà anche ed in aritmetica modulare il che funziona fin tanto che evidentemente il numero di oggetti che stanno qua dentro non diventa eccessivo poi affronteremo questo problema per adesso ragioniamo un attimo su questa possibilità questa possibilità consiste nel dire abbiamo incrementato Tail Se però Tail Diventa troppo grosso lo aggiungo anche se in questa applicazione non sarebbe il caso ripeto e sto aggiungendo operazioni che costano Quindi potrebbe anche valere la pena di non farle se per caso Tail diventa maggiore o anche uguale alla sorgente ovviamente diventerà primo e poi Maggiore io ma non diventerà mai Maggiore Allora semplicemente non facciamo altro che sottrargli la dimensione stessa in maniera che si ritorni in aritmetica modulare 9 diventa 10 ma 10 non va bene sottraggo 10 che è la dimensione questi sono 10 Celle 10 meno 10 fa 0 e io vado a scrivere qua Tail è pronto sulla posizione Zero così quando arriva il 2 vado a scrivere in posizione 0 e te il diventa uno e quando arriva il 6 scrivo in posizione 1 e te il diventa 2 che è quello che succede nel passaggio dal secondo al terzo grafico Allora abbiamo finito così si ni Nel senso che ci vorrebbero dei test che mi impedissero di fare dei danni tutto sommato non soltanto test in piedi solo di fare dei danni cosa ci manca da implementare ci manca da implementare la estrazione dalla testa estrazione la fronte restituisce il valore che sta in testa se mi vogliamo cancellare quel valore per cancellarlo banalmente si tratta di andare a incrementare ed se voglio due volte cancellare elementi dalla

coda stessa cancellare Lotto cancellare il 3 senza neanche preoccuparmi di quanto valgono banalmente basta incrementare ed da 45 da 5 a 6 l'8 e il 3 sono ancora scritti sono segnati in grigino non sono raggiungibili e prima o poi verranno cancellati dalle successive aggiunte in coda alla coda stessa quindi molto banalmente si tratta di andare a incrementare ed è questo è tutto e che succede però se vuoi incrementare ed arrivo al 9 è la stessa cosa Cioè se la coda è sfuggita e ha fatto il giro dall'altra parte anche la testa non certo punto dovrà sfuggire e fare il giro dall'altra parte Quindi se ed supera la dimensione della coda stessa andiamo a sottrargli quella dimensione in maniera che tutto sia nuovamente corretto questo meccanismo con le due IV che strettamente Non sono necessarie per l'applicazione mi consente però di andare finalmente a implementare nel modo corretto nel modo corretto la mia visita Qui abbiamo Esatto tutti i file considerati possiamo andare a lanciare commscon sul grafo va bene Recuperiamo degli errori prima errore stavamo lavorando sul file in q0 questo non risolve comunque la questione c'è un secondo errore che ho fatto prima nella determinazione delle componenti connesse dove la ricerca la faccio non a partire da sorgenti già marcate ma partita da sorgenti non marcate E questo dovrebbe togliere un po' di cose ma probabilmente non ancora tutto Infatti parte un bel ciclo infinito dovuto a che cosa Traduci 1:21:03 Partirà da sorgenti non marcate E questo dovrebbe togliere un po' di cose ma probabilmente non ancora tutto Infatti parte un bel ciclo infinito dovuto a che cosa pronto un ciclo di stampa vengono effettivamente determinate due componenti e sono due componenti corrette Quindi il problema sta nella stampa che prima abbiamo testato ma abbiamo testato senza avere delle componenti vere e proprie e quindi Qui c'è un ciclo con M Più che un ca Tempest di un errore che abbiamo già visto in precedenza Perfetto forse adesso ci siamo e in effetti otteniamo le nostre componenti che stavamo cercando Allora andiamo avanti a questo punto abbiamo realizzato la visita in ampiezza Perché si chiama visita in ampiezza l'ho già sapete lo sapete già perché sostanzialmente partendo dal nodo uno si prendono tutti i nodi adiacenti che sono il 2 e il 4 Dopodiché partendo dal nodo 2 si prendono gli adiacenti Che sono l'uno il 45 l'uno non viene considerato il 4 neanche si prende il 5 ma poi si prosegue col quato quindi in sostanza si procede andando prima a visitare i nodi adiacenti al primo poi nodi adiacenti a quelli che erano adiacenti al primo nodo eccetera eccetera realizzando quindi una visita che si allarga in ampiezza prima di approfondire la visita andando lontano dalla sorgente e tra l'altro sapete anche che questo vi garantisce la possibilità di marcare prima tutti i nodi che stanno a un arco a un lato di distanza dal vertice di sorgente Poi tutti i vertici che sono a due lati di distanza poi tutti quelli che stanno a tre lati e quindi anche di determinare la distanza minima come numero di lati da un nodo sorgente quindi di risolvere un caso particolare del problema dei cammini minimi chiusa la parentesi a questo punto che cos'è che mi restava da fare Mi sono infilato in una parentesi più lunga del solito e l'ho persa quello che ci resta da fare E andare a capire quanto ci costa tutto questo fare un po' di analisi di complessità questa analisi complessità sarà abbastanza sofisticata ma piuttosto tutti va Allora l'analisi di complessità è condotta Sul documento traccia che avete in rete e è basata sullo pseudocodice che vengono scelte Come si fa ad analizzare questo pseudocodice vedere la solita è un algoritmo iterativo istruzioni hanno una loro complessità facilmente determinabile e i cicli corrispondono delle sommatorie però qui cicli sono un po' strani perché quantomeno il primo non è un ciclo conteggio il secondo questo for all si può vedere con un ciclo a conteggio sugli archi uscenti da un dato nodo Allora qui abbiamo una inizializzazione una creazione di coda vuota che evidentemente comprende un certo numero di operazioni che potremmo dire di tempo costante anche se la allocazione del vettore lungo n dovrebbe farci pensare altrimenti quindi prenderei dire che più un test adn nonostante io abbia scritto zd1 dopodiché abbiamo però una marcatura è un accodamento che sono sicuramente in tempo costante Dopodiché c'è questo ciclo for questo sito non è assolutamen-

te evidente a priori quanto non ciclo while non è assolutamente evidente a priori quanto costi per renderlo evidente dobbiamo osservare come abbiamo fatto prima che l'elemento che viene estratto dalla testa della coda in queste due operazioni poi non verrà mai più infilato nella coda stessa perché Perché elemento estratto dalla testa quando è entrato ho qua sopra o qui dentro nella coda è stato anche mercato e nessun elemento che sia stato marcato una volta verrà mai più va inserito nella coda stessa Quindi in sostanza questo Wine o ti senti noi Possiamo vederlo come un ciclo for su tutti i no anche a parte 1:25:36 Quindi in sostanza questo qua il no ti senti noi Possiamo vederlo come un ciclo for su tutti i nodi che appartengono sono appartenuti alla coda tutto tutti i nodi che sono stati quindi raggiunti visto che un nodo viene inserito in coda non appena viene raggiunto Ok per ognuno di questi nodi abbiamo operazioni in tempo costante che sono pronte le cure e poi un ciclo di scorrimento degli archi uscenti dal nodo stesso Questo ciclo avrà un'operazione in tempo costante ed eventualmente altre operazioni in tempo costante con i tutto il resto è tendenzialmente abbastanza tranquillo e si può scrivere in questa forma con un tetto di uno che probabilmente troverete corretto interviene nella versione che mettevano avete più una sommatoria su tutti i vertici che via via vengono raggiunti che sono i vertici della componente vs che stiamo costruendo di che cosa di operazioni in tempo costante che sono queste front ede-q più una sommatoria su tutti gli altri uscenti quindi sugli archi di Delta più del vertice generico vi di attenta di uno Quanto costa tutto questo se stiamo facendo la sommatoria su tutti i vertici di un sottoinsieme e su tutti gli altri che escono da quei vertici È abbastanza evidente che stiamo facendo una sommatoria su tutti gli archi che hanno il primo estremo in un sottoinsieme Ma questo sottoinsieme che è una componente connessa del grafo non ha solo il primo estremo nel sottoinsieme anche il secondo estremo perché per definizione la componente connessa se un elemento è un vertice sta in una componente connessa anche gli adiacenti stanno nella componente connessa e quindi questa sommatoria doppia pervinche steinways e a che sta in delta più di me diventa una sommatoria per sa che sta nell'insieme degli archi della componente connessa uds ignorando le operazioni che pesano un po' meno quello che riusciamo a ottenere complessivamente è una operazione che va a sommare su tutti i nodi della componente e su tutti gli archi della componente una complessità costante cioè un theta DNS più MS Questa è la complessità dell'algoritmo di visita prima di proseguire passando all'analisi della algoritmo delle componenti connesse torno un attimino indietro su questa cosa che vi ho detto prima riguardo il teta di uno che in realtà è un set adn è assolutamente vero che noi abbiamo allocato la coda azzerando la completamente con questa operazione di culloch rigorosamente parlando non c'era nessun bisogno di azzerare queste Celle perché una volta che la coda e vuota in quanto è detta e sono identiche lì dentro ci portare qualsiasi cosa per cui Se volessimo veramente essere efficienti qui dovremmo scrivere una operazione di malloch che ha tempo costante e l'analisi che stiamo facendo si riferisce a questa ipotesi non lo faccio per non confondere ulteriormente le acque passiamo a realizzare ad analizzare la procedura di componenti connesse Qui abbiamo un'altra strana ciclo che sembra molto più semplice perché è un ciclo su tutti i vertici qui non ciclo da 1 a Denny e apparentemente uno dovrebbe dire abbiamo una operazione iniziale di azzeramento c'è un azzeramento iniziale delle marcature che in realtà viene fatto fuori dalla funzione delle componenti connesse dopodiché abbiamo uno scorrimento di tutti i nodi c'è un test in tempo costante quando questo test scatta eseguiamo un'operazione di incremento di cinta in poco stante è un'operazione di visita e abbiamo detto che la visita ci costa n spmf c'è il numero dei Nodi e numero degli archi appartenenti alla componente di sorgente S apparentemente questa sommatoria non fa altro che moltiplicare perenne la complessità NS PMS la verità che però questo non è corretto non è corretto nel senso che è una stima per eccesso corretta che non tiene conto però di un fatto il fatto è che 1:30:23 Questo non è corretto non è corretto nel senso

che è una stima per eccesso corretta che non tiene conto però di un fatto il fatto è che questo test IV CDS uguale a zero viene soddisfatto abbastanza raramente cioè viene soddisfatto soltanto per cui vertici sono gente che non sono stati raggiunti in visite precedenti se noi teniamo conto invece di questa situazione ci vediamo conto di che cosa ci rendiamo conto del fatto che entreremo che eseguiremo il test ad ogni passo Quindi abbiamo un test adn complessivo che comunque va su Massi a quest'altro e quindi non conta ma andremo a entrare qui dentro soltanto tante volte Quante sono le componenti quindi soltanto NC volte allora la sommatoria può Ben essere scritta sommatoria anziché su tutti i vertici vi può essere scritta come una sommatoria su tutti i vertici di uno che è questo test qua più una sommatoria sulle componenti da 1 a n c di quel che succede qua dentro e qua dentro che cosa succede ignorato il tetto di uno succede che c'è una visita cioè una visita che costa sopra la Bianca ma tns più MS Adesso viene più ragionevole chiamarli ncpm ci sto cambiando la notazione in maniera molto informale ma cerchiamo di capirci perché la cambio così perché sto considerando il numero dei nodi il numero degli archi della componente C esima invece chiamarla componente associata La Sorgente S la chiamo componente di indice ci ma è la stessa cosa Questo cosa ci dice ci dice che stiamo a parte il termine tenta di n complessivo che mo sommando i nodi della componente C su tutte le componenti e gli archi della componente C su tutte le componenti Ma siccome le componenti non hanno in comune i nodi e non hanno in comune gli archi stiamo sommando tutti i nodi stiamo andando tutti gli archi da cui si deduce che la ricerca delle componenti connesse in un grafo con la visita in ampiezza ci ha una complessità totale keta dnpm cioè proporzionale lineare nel numero dei Nodi e nel numero degli Archi del grafo stesso e a questo punto faccia una pausa Devi trovarti allora abbiamo visto nella prima parte di questa lezione un algoritmo di calcolo delle componenti connesse basato su una generica visita e poi abbiamo sviluppato un algoritmo di visita in ampiezza di cui abbiamo anche fatto l'analisi di complessità oltre a un'implementazione portata fino al dettaglio Adesso possiamo vedere un altro algoritmo di visita che è basato banalmente su l'idea di descrivere l'insieme Q che fa da frontiere Vanno insieme uday dei vertici visitati Cioè è l'insieme dei vertici che sono visitati ma non sono ancora stati usati per visitarmi degli altri l'idea di rappresentare questo insieme con un con una coda ma con una pila cioè come un altro insieme che consenta di inserire ed eliminare elementi in tempo costante sfruttando il fatto che lista le posizioni di accesso sono strettamente delimitate in particolare Ti ricordo che la lista la pila è definita come una lista nella quale sia gli inserimenti sia le cancellazioni avvengono allo stesso estremo nel primo elemento in quella che viene definita cima della pila Allora in cosa consiste la visita in profondità esattamente nella stessa e serie di operazioni fatta per la visita in ampiezza Infatti qui vedete costate la depth first Search è la visita che prima di tutto va in profondità alla breakfast La visita che prima di tutto va in ampiezza con le operazioni corrispondenti come vedete si parte non con una coda Kuma con uno Stack ovvero pila S vuoto poi si va ad assegnare alla sorgente come indice di marcatura l'indice ci della nuova componente che stiamo costruendo si va a aggiungere alla pila anziché alla coda la sorgente e qui tecnicamente il gergo dice di usare una operazione di Push Dopodiché Finché la coda la pila non è vuota che andremo a determinare l'elemento che sta in cima anziché in testa e questo viene fatto con l'operazione di Top sempre in gergo si va a estrarre questo elemento 1:35:05 Speak now.

Sì Ma anziché in testa e questo viene fatto con l'operazione di Top sempre in gergo si va a estrarre questo elemento dalla pila stessa con Un'operazione che in gergo definito di pop e poi per ogni nodo per ogni vertice adiacenti a quello dato andiamo a verificare se è mancato no e se non è mancato e solo Allora lo marchiamo e lo andiamo a inserire nella pila e l'inserimento in pila avviene con un operazione

di Push Cosa vuol dire tutto questo che con una certa semplicità Noi dovremmo poter tornare al nostro codice e invece di realizzare un algoritmo di componenti connesse che utilizzi la breakfast dovremmo poterla commentare via e sostituirla con un algoritmo che invece realizzi la depth-first Search la dfsc quindi aggiungeremo esattamente con gli stessi argomenti andremo a dichiarare definire quindi ci interesserà un algoritmo dfs che visita in profondità il grafo g a partire dalla sorgente Macan di vertici visitati e nel vettore C con indice ci piccolo questo algoritmo noi potremmo tranquillamente andare a definirlo con un corpo tempo veramente vuoto e quindi la fase successiva del nostro esercizio Sara riempire questo algoritmo chiaramente ispirandoci molto pesantemente a quello in ampiezza non useremo più la coda di interi ma andremo invece utilizzare un'altra libreria che alta libreria chiudiamo questi due file e andiamo semplicemente a vedere che è già fornita una libreria in Stack punto H con un inizio di implementazione vuoto in Stack Zero e lo speck è costituito da che cosa è Allora dunque la pila si può anch'essa rappresentare con un vettore Esattamente come la coda soltanto che la politica di gestione è lifo last in first out anziché Fifo first in first out E questa implementazione significa che abbiamo sempre bisogno di un vettore dinamico con una dimensione allocata e siccome però inserimenti e cancellazioni avvengono nello stesso. avremmo bisogno di un solo indice intero che mi se vogliamo è leggerissimamente più efficiente dal punto di vista dell'occupazione di spazio qui vediamo vediamo un esempio Siccome gli oggetti entrano ed escono nella posizione finale o un po' come se fosse una tabella rigore una pila è molto simile a una tabella quello che la distingue dalla tabella il fatto che la tabella consente l'accesso diretto agli elementi interni mentre la pila consente solo l'accesso in lettura e scrittura l'ultimo elemento bene allora qui abbiamo la solito solito vettore di 10 elementi allocato e abbiamo un indice top che ci dice che l'elemento di indice 4 che è il quinto useremo indicizzazione da zero visto che non ci interessa entrare nel vettore in una data posizione e quindi non non ha senso indicizzarlo da1n per aderire al per comunicare correttamente semplicemente col mondo esterno visto che Comunicheremo con operazioni pre definite da funzioni Quindi se elemento in cima a indice 4 Ecco che 8 questo elemento vd4 cioè uguale 8 è quello che possiamo leggere o in cui eventualmente possiamo scrivere top dall'ultima posizione occupata se vogliamo aggiungere qualcosa dobbiamo andare a incrementare top per poter poi scrivere però questo andremo poi nel dettaglio a vederlo in seguito quello che ci interessa per ora sono definite come sono dichiarate queste grandezze e quali sono le funzioni di default creazione distruzione un test di pezza e un operazione per accedere all'elemento che sta in cima e due operazioni per aggiungere un nuovo elemento è per cancellare senza neanche leggerlo l'elemento che c'è attualmente luci Ma potete abbastanza facilmente immaginare come si implementino ma lo vedremo in seguito quello che vi invito a fare adesso è a respirando vi abbastanza alla BFF e ispirando via lo pseudocodice che sostanzialmente dice già tutto quello che ci serve andare a implementare la funzione di FS Dopodiché andremo uno per uno a riempire queste 6 funzioni Allora tfs che cosa deve fare

1:39:42 Dopodiché andremo uno per uno a riempire queste 6 funzioni Allora tfs che cosa deve fare deve Per prima cosa costruire questo Stack vuoto lo farà chiamando la funzione Clear Stack e lo farà lavorando su una coda di una coda su una pila di interi che per non aver fantasia chiameremo Esse abbiamo questa pila di interi che inizialmente va creata che inizialmente va creata vuota e che quindi direi passata per indirizzo con una dimensione è la dimensione al solito è il numero dei nodi anche qui vedremo che la pila ha una serie di vantaggi te lo rivediamo dopo tanto dobbiamo solo costruire la struttura dell'Arco ritmo stesso quindi come abbiamo fatto prima in sostanza decomponiamo l'algoritmo nella creazione di una pila vuota seguita da che cosa seguita dalla inserimento in questa pila che dalla marcatura nella sorgente del elemento adesso gente stessa che mi mancheremo la

sorgente e la inseriremo nella pila inserimento in pila tecnicamente viene indicato come Push anziché come enqueue e l'inserimento avviene nella posizione S dopodiché cosa faremo e Procederemo Finché la pila non è vuota e questa operazione consisterà nella chiamata di una funzione is Empty Stack anziché i 20 kW che poi è il motivo per cui non ho chiamata i20 semplicemente perché volevo poter tenere nello stesso codice 1 is Empty Stack è una esenti Q senza che il compilatore si lamentasse Dopodiché andremo a estrarre l'elemento non l'elemento che c'hai in testa alla coda ma andremo a estrarre l'elemento che c'è in cima alla fila Allora allenamento che è in cima alla pila si chiama top della pila stessa sarà il terzo è generico Digli che sta in cima alla pila s e per eliminarlo andremo a chiamare la funzione Pop che modifica la pila S fatta questa operazione per ogni adiacente al vertice V correremo E déterminée mo me termineremo i vertici adiacenti quindi per ogni adiacente al vertice V ovviamente ci serve definire questo vertice V come ci serve definire un posizione per gli altri che adesso andremo a scorrere qui non cambia nulla perché stiamo insieme intanto sempre come vettori di fuoco e starà E per ognuno di questi vertici lui questi Archi uscenti andremo a determinare intanto Chi è il vertice adiacente e poi faremo questo bel test che ci dice se dobbiamo effettivamente marcare e aggiungere il nuovo elemento come abbiamo fatto in precedenza Quindi se l'elemento adiacente non è ancora stato visitato intanto lo determiniamo dopodiché se non è stato ancora visitato possiamo andare a marcarlo e inserirlo nella pila anziché nella coda e ovviamente l'inserimento non sarà una Thank you ma sarà una push che lavora sul vertice W Ok A questo punto abbiamo la nostra funzione dfs che se abbiamo incluso in Stack potrebbe anche funzionare ovviamente non fa nulla perché perché le funzioni sono tutte vuote però Proviamo a vedere se è quantomeno sintatticamente corretta stiamo lavando su interspec di zero che è l'unica cosa che cambia in tutto questo e abbiamo un po' di indicazioni di terrore dovute sostanzialmente al fatto che non avevamo ancora incluso leather quindi dobbiamo includere Leather in stack vedere come va a finire mi scopi e che volta dobbiamo includere entrambi Perché altrimenti perdiamo 1:44:33 Finire e scoprire che in realtà dobbiamo includere entrambi Perché altrimenti perdiamo la correttezza di tutte le funzioni che avevamo introdotto in precedenza Perfetto Adesso abbiamo i veri errori vedevo io sono semplicemente il fatto che c'è un q e un w non definiti dovevo doppio È il nodo ed effettivamente andava dichiarato e qui c'è ancora un q che è sbagliato e ti fa anche capire che è bene dare dei nomi alle cose perché in questa maniera il compilatore che non capisce di che cosa si sta parlando però ci può dare una mano in varie situazioni a cosa sono dovuti questi problemi ha fatto che anche nella compilazione dobbiamo continuare a includere la libreria per le code a meno che si voglia totalmente cancellare la funzione bfs Ma non è quello che vogliamo ora Se lanciamo con Consul solito grafo non orientato siccome le funzioni che stiamo usando sono vuote ci dovremmo aspettare di ottenere un risultato vuoto è il motivo per cui non sembra che io faccio apposta È che non ho cambiato il nome alla funzione che stavo chiamando stiamo chiamando la funzione di visita in profondità dfs stiamo compilando tutte queste cose qua e a questo punto mi aspetto di ottenere un risultato che non è corretto perché abbiamo ottenuto questo risultato cerchiamo di capire un attimo perché è diverso leggermente da quello che abbiamo ottenuto in precedenza che era un tuttozero Qui ci sta dicendo che abbiamo 6 componenti isolate Il motivo è che prima tenevamo sempre Zeri perché avevamo realizzato la funzione Eccola qua la funzione di FS l'avevamo realizzato ma non l'avevamo riempita è sostanzialmente quello che stiamo facendo qui è creare uno Stack in realtà vuoto stiamo marcando sorgente S con un indice progressivo Dopodiché tutto quello che segue non fa assolutamente nulla la funzione wireless te che restituisce vero quindi salta fuori immediatamente e di conseguenza questa funzione dfs ad ogni sua chiamata qui va a mancare la sorgente e manca la sua gente con un indice progressivo che poi viene continuamente incrementato incrementato incrementato di conseguenza

il risultato è questa questa questa stampa sbagliata ma giustificata da quello che effettivamente stiamo facendo per ottenere il risultato corretto dobbiamo andare a riempire le 6 funzioni della libreria Allora procediamo a riempire queste funzioni tanto però prima un'occhiata ai lucidi che ci spiegano come funziona questa libreria e li faccio le osservazioni che avevo preannunciato esattamente come nel caso della coda questa pila non conterrà mai più di n elementi dove n e il numero di vertici del vettore del grafo di conseguenza sappiamo quanto al locale Questa è un bel risultato il secondo punto è che ciascuno dei Questi elementi entra ed esce solo dalla cima e di conseguenza non abbiamo bisogno di grandi complicazioni per gestire inserimenti e cancellazioni ci basta andare a giocare su questo indice top Dopodiché l'altro servazione che possiamo fare è che ogni elemento che viene tirato fuori dalla pila non ci rientrerà mai più questa osservazione in fondo non è importante come è stata nel caso della coda perché qui non abbiamo problemi di vettori come abbiamo visto in precedenza dove gli oggetti si tendono a spostarsi Comunque sia verso destra e quindi un certo punto potrebbero spawnare nel caso della pila Vedete che gli oggetti vengono aggiunti a destra ma poi vengono anche cancellati facendo regredire la pila verso sinistra e quindi non c'è un problema di aritmetica modulare in questo caso per esempio siamo in questa situazione iniziale con i soliti oggetti 8364 uno che sono stati inseriti nell'ordine prima uno poi 4 poi 6 Poi 3 porto e di conseguenza che cosa succede che l'elemento che sta in cima Cioè in posizione 4 come indicato è l'elemento 8 la funzione top ci deve restituire 8 la funzione is Empty ci restituirà falso perché perché evidentemente Siccome il top è la posizione 4 non è 1:49:17 Devo restituire 8 la funzione is Empty ci restituirà falso perché perché evidentemente Siccome il top è la posizione quattro e non non è sicuramente quella non è sicuramente quella di una pila vuota come dovrebbe essere per una pila vuota setup forse Zero vorrebbe dire che comunque nel elemento Zero c'è qualcosa quindi toccherebbe essere meno 1 Dopo di che cosa andiamo a vedere andiamo a vedere che se dobbiamo inserire gli oggetti 526 nella pila stessa andiamo a scrivere l'elemento 5 non in posizione top ma invasione topi uno e poi incrementiamo o meglio prima incrementiamo e poi scriviamo nella posizione top inseriamo l'elemento due increment eremo top che da quanto era salito a 5 non puoi mettere mo A6 è in posizione 6 Scriveremo l'intero due che va aggiunto e infine per un ulteriore Pooh sinceramente le moto da 6 a 7 in posizione elemento sei Se vogliamo cancellare un paio di elementi ci basta fare un paio di decrementi di Coop dal 17 al 6 e poi al 5 e vedete che gli elementi che stavano nella pila rimangono ancora nella pila ma sono inesistenti qualunque qualunque successiva aggiunta andrà a sovrascrivere e a cancellarli e questo più o meno tutto quello che ci serve sapere Quindi possiamo tornare a implementare la nostra libreria della pila Come si fa a creare uno speck bene come sempre andiamo a prenderci la lista della spesa Cioè nella dichiarazione vediamo di che cosa è costituito uno Stack e andiamo ad aggiornare inizializzare i valori di questi tre campi Quindi abbiamo un PS freccia vi che è il vettore di interi che ci interessa è in quanto vettore di interi lo allungheremo della Dimensione corretta come vi ho separè in precedenza Se volessimo essere veramente efficienti dovremmo scrivere malloch io scriverò Carlo che per coerenza con gli anni scorsi ma in effetti è un insufficienza del tutto inutile perché riempire di Zeri questa fila non fa altro che andare a cancellare delle posizioni che sono comunque non visibili e quindi non c'è vantaggio vero e proprio dobbiamo andare ad assegnare dal locale Size posizioni ciascuna della dimensione di un intero dobbiamo andare a verificare se la locazione è andata a buon fine confrontando il puntatore col solito puntatore nulla e dobbiamo andare a stampare i suoi tuo messaggio di errore nel caso in cui questo non abbia funzionato Quindi che ci potrebbe essere stata un errore nella locazione di una fila e questo sistema il caso a questo. dobbiamo andare a inizializzare la dimensione che è banalmente il il l'argomento sai che c'è stato passato da fuori e a inizializzare la posizione top Qual è la posizione top la posizione t'appelles l'ultima la posizione che sta in cima l'elemento che sta in

cima alla pila stessa ma siccome di elementi non ce ne sono inizializzare mo almeno uno è altrettanto possibile pensare a pile nelle quali la posizione top punti in realtà non l'ultimo elemento che c'è ma il primo elemento che non c'è cioè la posizione in cui andava inserire un po' come succede nella coda in tal caso ovviamente PS top sarebbe inizializzato a zero e cambierebbero tutte le funzioni successive basta essere coerenti con se stessi dall'esterno nessuno si rende conto di niente perché la struttura dati astratta e le si accede soltanto con le funzioni e non c'è più da preoccuparsi dell'implementazione Ovviamente il costo è chiamare funzioni e quindi essere un po' meno efficienti per distruggere uno che procediamo come prima Cioè andiamo a deallocare il vettore B stesso se vogliamo essere molto chiari assegniamo nulla al puntatori maniera che non vada comunque appuntare delle aree che non sono disponibili e andiamo a segnare a 0 la dimensione e andiamo e a segnare il top a che cosa potrebbe dire a 0 perché tanto comunque non c'è problema lo stai Come va eventualmente vi creato ci potrebbe dire a uno per sottolineare il fatto che è un po' come se fosse vuoto Non è così importante fate un po' quello che ti pare dal punto di vista della essere vuoto Non vuota abbiamo già detto uno Stack è vuoto quando la posizione top è pari a me 1:54:10 Posizione top è pari a meno 1 con la convenzione che abbiamo assunto Qual è l'elemento che sta in cima l'elemento che sta in cima elemento che appartiene al vettore V E che sta nella posizione top e basta Come si fa a inserire un elemento Allora devi inserire un elemento noi dobbiamo spostare la posizione top in avanti di un passo quindi andare incrementarla e poi possiamo andare nella posizione che al momento Adesso è top e scrivere il nuovo elemento da aggiungere infine se uno volesse cancellare un elemento tutto quello che ha da fare è diminuire di 1 la posizione top e basta Non ci sono altre particolari tecnicalità a questo punto se andiamo a compilare quanto fatto ci troviamo con un errore di battitura per cui abbiamo nel test abbiamo assegnato nulla anziché confrontare Per fortuna il compilatore segnala queste stranezze perché è strano che ci sia un assegnamento all'interno di un IV è possibile ma è strano e di conseguenza abbiamo evitato un errore e possiamo provare a lanciare il nostro programma Ottenendo effettivamente lo stesso risultato di prima che cosa si può dire a questo punto si può fare un'analisi di complessità Ma l'analisi di complessità In sostanza coincide con quella che abbiamo già fatto perché Perché per quanto riguarda le componenti connesse non è cambiato nulla potrebbe cambiare la complessità della visita perfetto la complessità di creare uno Stack vuoto è identica a quella di di creare una coda vuota cioè costante se non fate la callo come la murloc la complessità di marcare eta di 1 quella di aggiungere un elemento con una puspi tosto che non Q è unitaria il test di Tezza è unitario top Pop eccetera son tutte unitari non cambia assolutamente nulla quindi la stessa analisi che abbiamo fatto prima si ripete qui e questo significa in sostanza che possiamo concludere che la visita di una componente a partire da un dato vertice a una conquista che accetta DNS più MS dove NS il numero dei nodi della componente che ha sorgenti in sms il numero degli lati della componente che ha sorgenti nes mentre nel complesso il calcolo delle componenti connesse fatto usando la visita in profondità a complessità tetta dnpm dove ndm il numero dei numero dei vertici e numero dei lati del grafo dato sarebbe tutto se non che esiste una altra variante della visita in profondità che è la implementazione di questa visita in maniera ricorsiva perché è possibile implementare la visita in profondità in maniera ricorsiva perché in realtà noi potremmo anche dire vediamolo su un esempio io parto dal vertice 1 e a questo punto visito tutti gli adiacenti che sono i due il quarto per ogni adiacente che cosa faccio e potrai visitare gli adiacenti Che sono l'uno il 4 il 5 devo evitare di rivisitare quelli che ho già visitato Ma se mi porto dietro un vettore che al solito vettore di marcature ci che mi dice Chi è già stato visitato nel momento in cui visito il 2 so che lui nel 4 sono già stati visitati non ho bisogno di farli e posso proseguire soltanto con gli adiacenti al 4 questa cosa mi dà luogo a una espressione particolarmente semplice nella l'algoritmo che consiste nel dire

trendy il tuo Avete visto comparire miracolo 1:58:16 L'algoritmo che consiste nel dire prendi il tuo Avete visto comparire miracolosamente una riga perché mancava nel testo originale Allora la versione ricorsiva della visita in profondità consiste nel dire marca intanto La Sorgente che chiaramente è stata visitata Dopodiché Guarda tutte le tutti i vertici che sono adiacenti alla sorgente stessa e per ciascuno Verifica se per caso è già stato mancato sennò lo Marchi e riparti con la visita a partire da quel vertice li vedete che l'espressione iterativa in realtà è facilmente mappabile su questa espressione ricorsiva Che cos'è che cambia in più sostanzialmente cambia che la versione iterativa gestisce esplicitamente una pila Cioè a una pila inizialmente vuota ci mette la sorgente Finché la pila non è vuota ne estrae l'elemento primo e poi che cosa fa va a vedere gli adiacenti e li mette in cima alla pila ripete la operazioni sulla pila stessa Ma se noi Esegui Simo invece la versione ricorsiva che cos'è che succederebbe succederebbe che noi avremmo La Sorgente guarderemo ciascuno dei vertici adiacenti Prenderemo un adiacente siccome avevo similmente non è marcato lo mancheremo e poi a partire da esso chiameremo la ricerca in profondità che vuol dire vuol dire sostanzialmente marcarlo tra parentesi questo credo esima dalla necessità di avere questa marcatura qui che quindi potrei togliere e permetterebbe poi di passare ad adiacenti degli adiacenti e quindi ho un ulteriore chiamata Che cosa costituiscono tutte queste chiamate nel sistema operativo è una serie di record di attivazione che sono messi su una fila su uno speck e qui si vede chiaramente che la pila come struttura dati all'interno di un programma e la pila come strumento per la gestione delle chiamate di funzione sono la stessa cosa e le successive pile che si vanno ad accumulare i successi record di attivazione che si vanno ad accumulare sulla pila di sistema da cosa sono costituiti da questi cinque oggetti il numero del l'insieme dei vertici l'insieme dei lati che per noi saranno una struttura grafo In sostanza un puntatore natura grafo Umberti cfw un vettore C è un indice intero ci di indice progressivo di componente che va copiato Quindi ogni chiamata all'Oca 5 oggetti che diventano 4 nella nostra implementazione in C i quali sono i G C grande e ci piccolo sono sempre gli stessi perché il geografo è sempre lo stesso il vettore di marcatura è sempre lo stesso e l'indice di marcatura ci l'indice di componente è sempre lo stesso volta in volta noi continuiamo ad allocare sullo Stack di sistema che cosa un campo o doppio che cambia in continuazione che tutte le volte è un modo adiacente al precedente e questa cosa in realtà è la stessa cosa che succede nella pila del definizione iterativa della ricerca in profondità Con la differenza che li abbiamo una pila di interi costituita soltanto dai vertici che vengono via via esplorati Monte Qui abbiamo una pila di ambienti ognuno costituito da 4 getti che sono un grafo un indice di vertice convettore puntatore alla prima posizione di marcatura è un indice di componente quindi il meccanismo è perfettamente identico però è più corposo D'altra parte il vantaggio che è molto più compatto nella scrittura specialmente se poi si cancella questa CV doppio cosa che sono in questo momento praticamente sicuro che sia possibile allora proviamo a implementare questa cosa e vedremo che c'è in realtà una piccola differenza anche se non ce ne accorgeremo veramente ma ve la sottolineerò dobbiamo tornare al nostro codice e dobbiamo andare a far che cosa a definire una terza possibile implementazione dell'algoritmo di visita che chiameremo dfs ricorsiva che ovviamente avrà gli stessi parametri gli stessi parametri che già conosciamo e che però richiedeva una dichiarazione è una definizione come sempre la dichiarazione farà una visita in profondità 2:02:59 Una dichiarazione è una definizione come sempre la dichiarazione fra una visita in profondità in modo ricorsivo visita in profondità in modo ricorsivo in profondità e grafologia partire da eccetera eccetera eccetera Questo ci consente di fs-per ricorsiva ci consente di andare a scrivere più sotto il corpo inizialmente vuoto della nostra funzione questo corpo vuoto Noi andremo a riempirlo ispirandoci al codice ma in maniera molto semplice dovremmo cioè andare a Eccoci qua a marcare La Sorgente senza inserire in nessuna fila visto che non c'è una pila mancheremo La Sorgente S con il l'indice

ci Dopodiché andremo a scorrere tutti i vertici adiacenti alla sorgente Scooby vertici adiacenti alla sorgente ovviamente dobbiamo tener conto del fatto che qui la sorgente S non è generico vertice V A questo punto se determiniamo chi è l'elemento adiacente Questo vuol anche dire che ci occorre un nodo doppio e ci occorre un l'arco Pia e comunque qualche campo anche in questa chiamata Ci occorre Dopodiché andremo a determinare Chi è il destinazione sempre partendo dalla sorgente valuteremo se il nodo è stato marcato o no E a questo punto non faremo altro che richiamare ricorsivamente la funzione stessa Su quale nodo la richiameremo può andare diretti richiameremo al sul grafologiche esattamente lo stesso di prima sul nodo doppio con il vettore di marcatura ci con l'indice ci piccolo e direi che è decisamente evidente quanto questa espressione sia più compatta e più elegante della precedente Adesso vediamo se rimane anche cometta Vi faccio notare che non c'è più bisogno in realtà né dello Stack né della coda e sto tenendo Perché esistono ancora le funzioni dfs bfs nel mio codice non vorrei doverle cancellare però stiamo questa volta chiamando all'interno dell'algoritmo delle componenti connesso e stiamo chiamando la dfs ricorsiva vediamo se l'ha chiamata a successo sempre sul grafo non orientato e direi che successo ce l'ha a questo punto dovremmo discutere la complessità ma è abbastanza evidente che le operazioni sono più o meno le stesse di prima quindi la complessità sarà sempre tetta DNS PMS per la visita e teta dnpm per intera determinazione delle componenti connesse che cosa vi avevo anticipato vi avevo anticipato vi avevo detto però di mettere da parte il fatto che qui nella definizione iterativa di ricerca in profondità le varie i vari pertici adiacenti al vertice corrente vengono uno per uno messi sulla pila quindi per esempio se guardiamo il caso del vertice uno che ha come adiacenti il 2 e il 4 si va a mettere sulla pila il 2 e poi sopra si va a mettere il 4 Che cosa significa Questo significa che nel momento in cui poi si esaurito questo ciclo si torna a Monte si estrae un elemento dalla pila l'elemento che viene estratto non è il 2 è il 4 e quindi si analizzano e si Marcano i nodi adiacenti vertici adiacenti al 4 nella versione ricorsiva invece se seguiamo lo stesso ordine nella sequenza degli archi abbiamo che visitiamo il vertice due e lo andiamo direttamente ad andiamo direttamente a chiamare ricorsivamente la funzione 2:07:46 O direttamente ad andiamo direttamente a chiamare ricorsivamente la funzione sul vertice due quindi lo chiamo direttamente e poi andiamo a vedere gli accenti del 2 quindi c'è una differenza Riguardo all'ordine in cui i nodi adiacenti vengono visitati se si vuole veramente ottenere lo stesso meccanismo di visita con la dfs iterativa piuttosto che ricorsiva bisogna correre questo ciclo sui nodi adiacenti nell'ordine opposto cioè dall'ultimo al primo anziché dal primo all'ultimo perché perché questo meccanismo di inserimento su una Pira esplicita inverte l'ordine dei nodi adiacenti come risultato non cambia assolutamente nulla se ci interessa l'insieme dei nodi visitati ma vedremo che ci sono delle varianti del pane della vita in cui invece il risultato potrebbe cambiare producendo un risultato equivalente nulla di nulla di grave direi che quello che ci manca effettivamente è per quanto riguarda le componenti connesse l'ultimo aspetto è il seguente allora nel costruire lì sotto insieme dei vertici raggiunti a partire da una certa sorgente In realtà noi stiamo sostanzialmente seguendo un albero perché è un albero perché da ogni vertice raggiungiamo uno o più vertici facciamo il caso della visita in profondità dal nodo uno eseguita in maniera ricorsiva dal nodo uno visitiamo in modo due è dannoso due immediatamente visitiamo gli adiacenti e adiacenti e l'uno che però è già marcato quindi non lo guardiamo visitiamo il 4 e poi danno 24 guardiamo gli adiacenti che sono Roomba 25 immediatamente visiteremo il 5 lì tra l'altro diversamente da come indicato qui la visita in profondità ricorsiva seguirebbe l'ordine 12244550 poi visita il 4 e dal 4 visita il 5 quindi l'albero che otterremo sarebbe costituito dal lato 12 dall'arco 12 dall'arco 14 e poi dall'arco 456 infine consideriamo la visita in ampiezza iterativa Che cosa fa questa visita prende il nodo uno vertice uno mette in coda il 2 e il 4 e finalmente estrae first in first out dalla coda al vertice due dalle 2 va a visitare

gli adiacenti e chi c'è di adiacente non ancora visitato quindi in conclusione Ognuno di questi algoritmi sostanzialmente se si va a vedere in un solo Quali sono i vertici visitati ma anche da che parte ci sono stati raggiunti produce non un insieme di vertici di un insieme di archi un albero che ha chiamato albero di visita e algoritmi di visita diversi danno alberi di visita diversi questo che ha rappresentato qui in realtà è un po' infelice perché non corrisponde a nessuno dei tre casi probabilmente l'anno prossimo lo modificherò Ad ogni modo quello che mi interessa sottolineare adesso Che cos'è È che sei qualcuno è interessato anche a sapere come sono stati ottenuti questi insiemi di vertice in alcuni applicazioni può essere il caso può rappresentare questo albero in qualche modo come si rappresenta un albero allora un albero è un grafo e di conseguenza può essere rappresentato con matrice di adiacenza lista di archi e vettore delle oltre a tutte le altre modalità che magari avete visto in teoria che comunque ricadono sostanzialmente in queste però non ha senso usare la matrice di adiacenza perché un albero per definizione estremamente sparso e usare una banale liste di Archi si ha senso però ci fa un pochino perdere la struttura di questo albero e la visita detto delle forme Star non è particolarmente interessante perché è un po' una via di mezzo a cosa sto puntando è una rappresentazione che è estremamente compatta e al tempo stesso presenta un po' di vantaggi dal punto di vista computazionale questa la prenotazione è la rappresentazione a vettore padre questo vettore F F sta per fare fornisce per ognuno dei vertici con odi possiamo parlare anche di nodi visto che la visita comunque ha un orientamento anche in un grafo non orientato questo vettore qui dicevo a per ogni nodo indicazioni dell'indice del nodo padre per esempio se prendiamo il nodo sei che è stato raggiunto dal 3 in posizione 2:12:35 Ogni nodo l'indicazione dell'indice del nodo padre per esempio se prendiamo il nodo sei che è stato raggiunto dal 3 in posizione 6 fd6 o quella tre Analogamente anche fd5 ha come padre il 2 sd4 come padre 2F di due è uno Dove sono le situazioni in frigo rete critiche nelle radici di Questi alberi che non hanno un padre Allora non avendo un padre o gli si mette una marca di qualche genere per esempio 0 per indicare che appunto sono da dici Oppure si fa una specie di auto nello si dice che una radice è padre di se stessa e quindi F di $1 = 1$ e ft3 $= 3$ che è la strada che sia seguita in questo caso era un esercizio interessante da aggiungere a quello che abbiamo fatto fino adesso sarebbe dati Codice che abbiamo scritto in uscita oltre a spedire il vettore di marcature o invece del vettore di marcature restituita il vettore F che va inizializzato ovviamente tutto a zero Dopodiché invece si può entrare marcare eccetera eccetera e quello che bisogna fare che sostanzialmente ogni volta che un nodo viene marcato ogni volta che un nodo W viene marcato ci viene anche mancavano con l'indice del suo modo padre e che hanno da padre e no doppio è la stessa cosa si può fare nella chiamata ricorsiva si può fare nella zona chiamata ricorsiva un pochino più delicato e lascio aperto il l'esercizio è si può fare nella visita in ampiezza Eccola qua Perché qui W come padre di le domande a mail o messaggi sul forum le limitiamo procediamo Su l'ultimo argomento della giornata l'argomento è quello di determinare le componenti fortemente connesse di un grafo Allora la relazione di connessione forte è diversa da quella di connessione debole Nel senso che obbedisce alla ordinamento degli archi intanto si parla di un grafo orientato n a non di un grafo non orientato vie e poi gli archi hanno un ordinamento che va rispettato cosa succede succede che se si guarda la connessione forte questa in generale non è una relazione simmetrica scusate vi prendiamo Allora il la relazione di connessione semplice su un grafo orientato non è simmetrica per il semplice motivo che riparto da capo Allora se prendiamo anziché un grafo non orientato un grafo orientato obbedire all'indicazione degli archi ci permette di dire che un dato nodo è connesso a un altro ma non ci permette immediatamente di dire che il secondo note connesso al primo in generale non lo è perché Lia che hanno un orientamento Questo vuol dire che delle proprietà che davamo per scontato e nei grafi non orientati la riflessività la tranquillità e la simmetria questa terza in generale non vale per

ottenere la simmetria bisogna introdurre la connessione forte cioè bisogna richiedere esplicitamente che ci siano un cammino fra due nodi ma un cammino orientato è che ci sia un cammino rientrato che va È un cammino rientrato che torna questo cosa significa in sostanza che la connessione forte code di tutte e tre le proprietà e quindi ha una relazione di equivalenza e ci permette di definire delle classi valenza che sono le cosiddette componenti fortemente connesse Mentre se guardassimo la connessione semplice o ignoriamo l'orientamento degli altri E allora tutto va bene oppure se lo teniamo presente non abbiamo una relazione di equivalenza e quindi non abbiamo il concetto di componente Supponiamo di considerare un grafo entrato e Supponiamo di considerare la relazione di connessione forte domanda Come si fa a trovare le componenti fortemente connesse nel grafo che vediamo qua sotto abbiamo che il nodo uno in realtà è connesso ai nodi 24 e indirettamente anche al 5 ma nessuno di questi nodi lo può raggiungere per cui in realtà il nodo uno da solo è connesso fortemente solo a se stesso costituisce una classe di equivalenza che è semplicemente un singoletto il 2 e il 4 il 5 essendoci un circuito orientato 254 sono tutti vicendevolmente raggiungibili e quindi costituiscono un'altra classe di equivalenza 2:17:23 Essendoci un circuito orientato 254 sono tutti che scende volmente raggiungibili e quindi costituiscono un'altra classe di equivalenza il 36 sono fra loro e raggiungibili e quindi sono una classe di equivalenza ma non possono non non hanno altri nodi con i quali ci sia questa connessione forte Si può osservare che ciascuna classe di equivalenza la caratteristica di essere o sconnessa dalle altre del tutto oppure di avere una connessione in un verso ma non nell'altro la classe 1 rispetto alla classe 245 può raggiungerla ma non può essere raggiunta la classe 245 poi se va aggiunta dalla classe 36 ma non può raggiungerla Quindi in poche parole se considerassimo ogni classe come se fosse un nodo quello che otterremo sarebbe un grafo orientato è privo completamente di circuiti come sarebbe un grafo orientato e aciclico in sostanza sarebbe un albero o albero orientato arborescenza Lasciamo perdere quella è una proprietà di fondo che comunque non utilizzeremo andiamo procediamo a vedere in che maniera si determinano le componenti fortemente connesse Per fortuna ci viene ancora buono il problema della visita però prima di vedere questo potere della visita dobbiamo introdurre una definizione che quella che combina che deriva dalla nozione di raggiungibilità la nozione di raggiungibilità che è semplicemente il suo complemento dato un nodo vi i nodi da cui vi è raggiungibile si chiamano No dico raggiungibili Sì quindi ci sono partendo da un nodo dato vi ci sono i nodi che possono essere raggiunti da lì che ci sono i nodi tra cui vi può essere raggiunto e questi due insiemi non coincidono in generale ora la proprietà fondamentale che ci permette di realizzare algoritmi per determinare le componenti fortemente connesse e il fatto che se noi conosciamo l'insieme dei nodi da cui si vede i nodi raggiungibili da che potremmo chiamare più di vi per il solito più che indicarti uscenti è l'insieme dei nodi da cui vi è raggiungibile Quindi potremmo chiamarlo meno e l'intersezione di questi due i sottoinsiemi Eh guarda caso la componente fortemente connessa che contiene vi Perché domanda allora molto semplice perché se uno prende questa intersezione fra i nodi che sono raggiungibili e i nodi che sono coraggio un GB li davi e sicuramente questi nodi godono di proprietà che la componente fortemente connessa Ah cioè quelli di essere raggiungibili Eco raggiungibili almeno da uno di questi nodi cioè di conseguenza che cosa ne risulta né risulta che l'intersezione di questi due insiemi contiene la componente fortemente connessa sto facendo lo stesso ragionamento in maniera un po' più complicata evidentemente lo stesso ragionamento che ho fatto per quanto riguarda la connessione semplice viceversa però se noi prendiamo la componente fortemente connessa che contiene lì e se è contenuta all'interno di questa famosa intersezione perché è contenuta Perché tutte le coppie dei suoi nodi sono vicendevolmente raggiungibili e guarda caso prendendo è sicuramente possibile partire da uno e raggiungere lì ed è possibile partire da di raggiungere quell'altro di conseguenza abbiamo un possibile potenziale algoritmo

Se abbiamo un algoritmo di visita di un grafo orientato che rispetti l'ordinamento degli archi l'orientamento degli archi abbiamo anche un algoritmo per determinare le componenti fortemente connesse si tratta di prendere un nodo V di visitare il grafo a partire da lì e quindi ottenere una delle due dei due sottoinsiemi che dicevo Dopodiché bisognerebbe fare una visita andando all' indietro come si fa a fare una visita andando indietro un modo possibile quello di costruire il grafo che ha tutti gli archi invertiti cioè grafo cosidetto trasporto trasporto perché si può pensare che la matrice di adiacenza sia trasporto scambi righe e colonne scambio origine destinazione quando si visita il rapporto sport si ottiene un altro insieme di nodi che sono raggiungibili da via del Grappa trasporto Dunque con raggiungibili nel gruppo di partenza se ne fa l'intersezione si ottiene un sottoinsieme di nodi fatto questo tutto insieme di nodi messo da parte si fa a cercare un altro nodo che non sia ancora stato nei raggiunto neco raggiunto e si fa la stessa cosa quindi una una generiamo le componenti fortemente connesse 2:22:10 Nodo che non sia ancora stato nei raggiunto neco raggiunto e si fa la stessa cosa quindi una una generiamo le componenti fortemente connesse tutto questo si può vedere nell'esempio qua sopra partiamo con un insieme ci completamente non mercato partiamo da una sorgente per esempio il nodo uno e generiamo il sottoinsieme dei nodi raggiungibili da uno che sono 2 4 e 5 Oltre ovviamente a uno quindi il sottoinsieme C1 che ci dice Chi è raggiungibile davi contiene 1245 viceversa se noi consideriamo il trasporto oppure se risaliamo gli altri all'indietro possiamo partire da uno e all'indietro non andiamo da nessuna parte riusciamo a raggiungere soltanto uno di conseguenza dobbiamo avere un insieme o meno di uno che contiene soltanto uno se interseca insiemi otteniamo ancora un insieme che contiene soltanto 1 e quindi imbarcheremo la posizione 1 del vettore di marcatura con un valore 1 vuoi perché è la prima componente vuoi perché la sorgente di questa componente è il nodo uno a questo punto cercheremo un secondo vertici nodo che non sia ancora stato mercato e sarà il nodo 2 Allora danno due non possiamo raggiungere il 5 e poi il 4 che mi abbia Hanno due allora danno due non possiamo raggiungere il 5 e poi il 4 Quindi abbiamo il 24 e il 5 questo è più di due se risaliamo gli occhiali indietro da due possiamo raggiungere l'uno il 4 i sei in direttamente il 5 indirettamente il 3 quindi li aggiungiamo tutti un Meno di 2 contiene tutti i nodi Inter Se chiamo 245 con 1 2 3 4 5 6 e otteniamo 245 Questo significa che mancheremo col valore due vuoi perché è il secondo sottoinsieme vuoi perché la sorgente e due mancheremo i nodi 2 4 e 5 passiamo a un altro sottoinsieme primo nodo Non ancora mercato è il 3 Il 3 permette di raggiungere il 6 il 5 e il 2 Il 4 quindi 2 3 4 5 6 mentre si può raggiungere il nodo 3 partendo da dove partendo dal 6 e partendo dalle 3 e basta Allora intersecando ho più di 3 con meno di 3 si ottiene 36 e questi due nodi possono essere marcati con la componente 3 A questo punto se cerchiamo nodi non ancora mancati non li troviamo perché 456 sono già mercati e quindi ci fermiamo questo grafico a tre componenti fortemente connesse la componente uno la componente 245 e la componente 3:06 che cosa dobbiamo fare per implementare un algoritmo di questo genere partiamo dal file componenti fortemente connesse zero Quindi Comp Ford con di zero nel quale che cosa troviamo cose molto simili quello che avevamo inizialmente la interpretazione della linea di comando il caricamento del grappo attenzione nel caricamento del grafo invece di avere l'argomento buhler pavimentato falso ce l'abbiamo vero Perché noi vogliamo interpretare il file lo vogliamo interpretare come grafo orientato Infatti il file che caricheremo che è questo grafo Oh Eccolo qui a gli archi con l'orientamento da osservare c'è l'arco 36 Ma c'è anche l'arco 63 Ok fatto questo dovremmo stampare il grafo per essere sicuri di averlo letto bene e costruire una soluzione vuota ottenuto lo stesso codice di prima è un vettore di marcatura ho detto venitevi all'inizio ci sono 0 componenti e il vettore l'ho inizializzato tutto a zero chiameremo una funzione componenti fortemente connesse che al momento è vuota Eccola qua e poi stamperemo le componenti fortemente connesse che abbiamo calcolato E que-

sta stampa in realtà è perfettamente identica la stampa delle componenti connesse perché è una partizione di nodi quindi ricicliamo il codice già scritto per fortuna Dopodiché ci liberiamo del grafo e ci liberiamo del vettore c e va tutto bene Stiamo includendo la libreria grafo FS anche qui quello che conta è la adiacenza quindi va benissimo bevete fuoco bastardi tutti i nodi il vettore del Fuoco stadi tutti i nodi e a questo punto Si tratta di andare implementare codice che cosa dobbiamo fare in componenti fortemente connesse qualcosa di un po' diverso da quello che abbiamo fatto per implementare le componenti connesse Che cosa c'è di diverso Dunque nelle componenti connesse torno indietro noi ci limita Vamo a volte inizializzare questo che lo facciamo nel Maine come prima scorrere tutti i nodi finché ne trovavamo uno non mercato aggiornare il codice e lanciare una visita qui dobbiamo fare due visite una sul grafo è una scusa ha trasposto non solo la visita lavorava su un vettore che era il vettore di marcatura direttamente questa cosa noi non la possiamo più fare Vi faccio notare che quando Abbiamo esplorato per esempio la componente che partiva da 1 da 1 a noi riusciamo a marcare 245 e all'indietro che non riusciamo Mercalli Dopodiché dobbiamo fare l'intersezione di queste due insiemi e marcare solo il nodo uno Quindi ciascuna delle due visite può mancare dei nodi in più rispetto a quelli che alla fine Devono risultare marcati e questo non è lecito andremo a sporcare dei dati che in effetti non vanno sporcati Questo vuol dire In sostanza che il vettore che possiamo usare qui la visita Se vogliamo utilizzare un meccanismo si 2:28:20 Non vanno sporcati Questo vuol dire In sostanza che il vettore che possiamo usare qui nella visita Se vogliamo utilizzare un meccanismo simile a quello precedente il vettore che possiamo usare qui nella visita del primo grafo è nella visita già fatto esposto non sono in generale il vettore cima sono altri vettori E questo potrebbe essere un problema anche perché tali vettori giustamente devono essere ripuliti non ci può bastare ripulirli una volta per tutti all'inizio quindi facciamo un implementazione non particolarmente efficiente ma cerchiamo di andare sul sicuro useremo tre vettori uno per la visita del grafo dato uno per la visita del grafo trasporto che dovremo costruire e poi useremo il vettore C che risulta dall'intersezione dei due vettori costruiti precedentemente e quello lì in effetti verrà usato una volta per tutte come possiamo procedere dovremmo correre Giustamente i nodi Quindi avremo un nodo sorgente il quale nodo sorgente procedeva da 1 fino al totale dei nodi che solito PG freccia n ad ogni nodo andremo a verificare se per caso è stato marcato o non è stato mercato e se non è stato mercato decideremo di fare una visita partendo da lì fin qui tutto identico adesso cominciano le differenze cosa faremo prima cosa visiteremo il grafo dato la visita e qui dobbiamo scegliere una procedura di visita io sceglierò la procedura di visita ricorsiva per il semplice motivo che è la più corta e quella più veloce da scrivere e quindi ce la caviamo ragionevolmente bene non è la procedura più consigliabile perché perché tiene occupato lo speck molto più di quello che si fa con la procedura di profondità iterativa e quindi in generale è bella ma non è consigliabile almeno credo sia piccolino però qui possiamo farlo Quindi dobbiamo visitare, Foggini che si chiama PG a partire dalla sorgente F dati CNC ma abbiamo detto che ci serve un altro vettore di incidenza quindi costruiamoci un vettore di incidenza uno sarebbe lui più quello in uscita e ovviamente questo vettore andrà in qualche maniera all'avvocato della Dimensione corretta è la dimensione corretta sarà pgn più 1 e i singoli elementi saranno degli interi e dovremmo andare a verificare che la locazione sia andata a buon fine errore nella locazione del vettore 1 Allora questa cosa ci permetterà di visitare il crafond mancando uno con che cosa Per esempio con l'indice cima in realtà siccome ad ogni passo noi visiteremo usando uno non è così importante ma cosa importante Attenzione è che dobbiamo essere sicuri che uno sia pulito è vero che uno qui è stato allocato con la culloch ma ad ogni passo ogni volta che viene utilizzato è sporco dall'utilizzo precedente per cui prima di fare questa cosa noi dovremmo scorrere e quindi ci serviva un Alto indice dovremmo scorrere per V che va da 1 a n il vettore 1 e 3 pulirlo scrivendoci degli

zeri potrebbe anche valere la pena di una funzione a questo punto che lo ripulisca ma va bene così Allora a questo punto dovremo poi costruire la foto la sposto e ci vorrà una funzione ci voleva una funzione che dato il grappo G costruisce costruisce il grafo trasposto parte Stagi e costruisce GT dove GTI sarà un altro grafo e sarà il grafo trasposto 2:33:07 Speak now.

G e costruisce GT dove GT sarà un'altro grafo e sarà il grafo trasposto domanda lo facciamo qua dentro o il grafo trasporto si può fare una volta per tutte ovviamente il grafo trasporto si può fare una volta per tutte e decisamente conviene farlo una volta per tutte questo Pigi questo è Pigi Il primo è passato per indirizzo per motivi di efficienza Il secondo è passato per l'indirizzo perché va fisicamente costruito quindi avendo fatto questo possiamo procedere a fare la visita del grafo trasposto il quale verrà a utilizzare un lettore di incidenza V2 che ovviamente va anch'esso allocato e che ovviamente va anche se ho ripulito prima dell'uso dopodiché faremo questa visita hutu e lavorando sul grafo trasposto fatto ciò Noi abbiamo in uno vertici visitati Nella visita diretta e in U2 i vertici visitati Nella visita inversa che cosa si può fare si può fare una bella intersezione di uno con U2 che sono tutte e due vertici della stessa dimensione per produrre che cosa per produrre il famoso verdetto veci tutti questi vertici hanno la stessa lunghezza che n possiamo metterli insieme Questa funzione a prendere gli indici che sono diversi da zero in uno che in U2 In tali indici all'interno del vettore ci va a scrivere che cosa va scrivere l'indice ci piccolo quindi ci sarà anche quello scriviamo un bel commento se si cui cosa stiamo facendo interseca i vettori di incidenza chiamo Vittoria di incidenza perché contengono 80 c1e U2 e copia in C I loro valori comuni questo mi fa in effetti pensare che ci non sia necessario perché sia uno sia U2 contengono C oppure 0 per cui ci risparmiamo questo passaggio sempre bene fare i commenti Allora a questo punto che si tratta di fare si tratta di andare a Ricordarsi che una o due vanno le allocati alla fine perché se no li lasciamo impegnati Dopo di che si tratta di andare a ipotizzare il fatto che anche il grafo trasposto andrà a sua volta vengo per locato E dov'è che si tratta di osservare che Abbiamo ipotizzato di aveva disposizione la funzione per costruire il grafo trasposto è una funzione per l'intersezione nonché una funzione per la dfs ricorsiva queste tre funzioni vanno realizzate grazie al cielo la funzione per la dfs ricorsiva ce l'abbiamo già quindi non facciamo altro che copiare ce l'ha dall'altro codice andare a inserirla qua sotto e ovviamente andare a dichiararla queralto e ci mancano due funzioni che sono il calcolo dell' intersezione di due vettori e la costruzione di un grafo trasposto andiamo a dichiarare le funzioni stesse e andiamo a definire le con un corpo vuoto costruisce il trafo GTI trasporto del grappo G gli abbiamo il grafo puntatore Pigi e abbiamo il risultato che il grapo puntatore PGT questi vengono copiati sotto con un corpo che per il momento è vuoto e poi ci serve l'intersezione di due vettori 15 serve la funzione intersezione

2:37:30 Questi vengono copiati sotto con un corpo che per il momento è vuoto e poi ci serve l'intersezione di due vettori 15 serve la funzione intersezione potremmo chiamare intersezione che Tori ma adesso sto procedendo velocemente che interseca il 21 il vettore 2 e produce in uscita un vettore a questo punto potremmo anche chiamare lì una o due copie in v e loro valori comuni che è la dimensione di tutti questi vettori di Lunghezza n Ok Allora dobbiamo riempire queste due funzioni al solito il giochino richiederebbe di passare la palla voi lasciarvi provare e poi vedere la mia soluzione prima di farlo sinceri Diamoci di avere scritto perlomeno sintatticamente in modo corretto tutto quello che ci serviva quindi stiamo lavorando su Comp fortemente con zero su la rappresentazione del grafo Affogo Star basta perché no la lista degli archi ovviamente basta perché non usiamo code e pile in questa implementazione Costruiamo il file componenti fortemente connessi eseguibile scopriamo che c'è da te questo è un residuo di una versione precedente in cui stampa grafo non era incluso nella libreria Ma adesso è qui c'è un NC che manca della dichiarazione di

tipo componenti fortemente connesse nella Liga 158 Eccoci qua pnc perché questo non l'avevo copiato pnc va determinato perché viene ad essere inizialmente Nullo e di volta in volta ad ogni passo si incrementa la componente connessa corrente e ovviamente Questa è la l'indice ci che qui stavo dando per scontato di conoscere questo dovrebbe essere corretto c'è 12 non dichiarato canale si tratta di salire e aggiungerlo e poi forse il tutto funziona lavorando su grafo orientato quello che mi aspetto semplicemente di leggere il grafo nient'altro forse è corretto Diciamo che è difficile saperlo visto che non stiamo ancora facendo nulla di interessante possiamo però costruire questo famoso grafo trasporto Come si fa a costruire il grafo trasporto Allora per costruire il geografo trasporto ci si ispira a Come si fa a costruire un grafo come abbiamo fatto costruire Il Grappolo Abbiamo caricato creandolo dato che conosciamo la dimensione e poi andando a inserire uno alla volta gli archi dato che li leggevamo Qui come si fa ha visto che abbiamo un grafo creiamo il grafo della Dimensione opportuna e poi inseriamo uno che uno gli archi opposti a quelli che leggiamo nel gruppo di partenza Quindi in sostanza prima che era il drappo con lo stesso crea un grafo con lo stesso numero di nodi di Dopodiché andremo a correre correre gli archi di G e aggiunge a GP di archi opposti molto molto semplice come si fa a creare il grafo funzioni 2:42:30 Molto molto semplice come si fa a creare il grafo funzioni che dobbiamo creare un grafo di dimensione pari a Pigi freccia.in e questo grafo lo chiamiamo PGT adesso abbiamo il grafo senza Archi Come si fa ad avere gli altri bisogna soccorrere Gli Archi del grappo G15 serve un po' d'Arco siccome G è rappresentato a vettore delle star abbiamo anche bisogno del nodo che potrebbe essere il solito nodo di origine quindi percorrere gli altri Se vi ricordate dalla stampa in giù quello che si fa a scorrere i no di dpg da 1 a n e per ogni Arpioni No ad andare guardare gli archi qua sopra abbiamo un simpatico ciclo che fa esattamente quello che ci serve Ovviamente la sorgente non si chiama S Qui si chiama Oh e a questo punto siamo in grado di far che cosa e siamo in grado di determinare gli estremi di questo arco il primo estremo lo conosciamo e si chiama o il secondo estremo ti chiamerà di quindi andiamo a metterci un bel b bel dì vedremo e di non è altro che la destinazione dell'Arco è nel grafo PG parte dal nodo può E sa come posizione Pia no tipo e di non abbiamo da far altro che aggiungere al l'altro grafo aggiungere al grafo PGT quello trasporto non il lago o di malaco di o abbiamo fatto giusto chi lo sa un buon modo per saperlo visto che abbiamo la funzione di stampa di un grafo e usarla e usarla immediatamente sotto la chiamata alla costruzione del grafo trasposto quindi possiamo fare una bella stampa del grafo GT a video e vediamo un attimo se la cosa ha successo Allora cosa succede Non è molto facile vederlo perché abbiamo stampato in vigoroso sequenza ingrato il grafo trasposto quindi facciamo una bella stampa dgt per esempio così riusciamo a distinguere questo grafo da quell'altro andiamo anche a capo alla fine è probabilmente qui ci manca Esatto ci manca una capo anche in questa Stampa possiamo compilare rilanciare e quello che vediamo e che ha fatto asporto contiene 21 al porto di 12 24 al posto di 4226 al porto di 6236 al posto di 6341 al porto di 14 e via dicendo potete andare avanti a controllare dovrebbe veramente essere il grafo trasposto Allora se questo è corretto possiamo togliere la stampa del grafo trasposto e proseguire col resto la dff già funziona dovevo fare l'intersezione di due vettori Come si fa a fare l'intersezione di due vettori Allora I lettori sono tutti puliti dimessi stiamo dando per scontato che queste intersezioni non vadano a sovrascrivere dove non devono quindi non ci sono grandi i controlli da fare l'unica cosa da fare e scorrere I due vettori entrambi con un bel cursore intero i e di conseguenza 2:46:35 Entrambi con un bel cursore intero i e di conseguenza andremo a vedere che cosa per i che va da 1 fino a n Dobbiamo verificare che entrambi i vettori siano diversi da zero se uno di I am a giovedì zero.eu tu e d è maggiore di zero Ecco che possiamo andare a inserire in Rigoni Che cosa è qui è un problema perché noi sappiamo che nel nostro uso uno o due siccome sono stati azzerati poi sono stati riempiti da una funzione di

visita con un ben preciso indice che lo stesso per tutte e due Uno di e U2 di saranno identici qui ebbe capire come vogliamo implementare questa funzione Se vogliamo riutilizzarla Diciamo che non ci interessa Allora se non ci interessa scelgo io e dico che invii con iscriverò il valore di U con uno di E questo dovrebbe chiudere il gioco Nel senso che salvando e andando a ricompilare ci troviamo finalmente ad avere le tre componenti che stavamo aspettando ci il che dovrebbe Chiudere questo bel tour de force e qualsiasi domanda vi rimando al email e al Forum e vi saluto

Figura 9.15: Implementazione delle code come vettori

L'inserimento usa la prima posizione libera, che scorre un passo in avanti

```
Enqueue(x,Q)
{
  Q.V[Q.tail] = x;
  Q.tail = (Q.tail+1) % Q.size;
}
```

Quando la coda eccede il vettore si usano le celle liberate in testa con l'aritmetica modulo `Q.size` (dimensione allocata)

Lo spazio è esaurito quando `Q.head == Q.tail+1`

La cancellazione sposta un passo in avanti la prima posizione occupata

```
Dequeue(Q)
{
  Q.head = (Q.head+1) % Q.size;
}
```

La visita in ampiezza non richiede l'uso dell'aritmetica modulare, perché ogni nodo entra nella coda al massimo una volta

Lezione 11

Traccia della risoluzione

Seconda fase (`compconn2.c` e `intqueue.c`) L'algoritmo di visita in ampiezza (*BFS*) rappresenta l'insieme dei vertici raggiunti, ma non ancora usati per estendere la visita con una coda. Tale struttura astratta può essere implementate come:

- una tabella;
- una lista a puntatori (in tutte le otto varianti viste nella lezione dedicata);
- una lista con vettori e indici (in tutte le otto varianti viste nella lezione dedicata;
- un semplice vettore con due indici che marcano le posizioni di aggiunta ed eliminazione.

Figura 9.16: Visita in profondità (versione iterativa)
La visita in profondità conserva i vertici visitati non usati in una pila S

```
DFS(V,E,s,C,c)
{
  S := ∅;
  C[s] := c;
  Push(s,S);
  while not IsEmpty(S) do
  {
    v := Top(S);
    Pop(S);
    for each w in Adj(v) do
      if (C[w] = 0)
      {
        C[w] := c;
        Push(w,S);
      }
  }
}
```

```
BFS(V,E,s,C,c)
{
  Q := ∅;
  C[s] := c;
  Enqueue(s,Q);
  while not IsEmpty(Q) do
  {
    v := Front(Q);
    Dequeue(Q);
    for each w in Adj(v) do
      if (C[w] = 0)
      {
        C[w] := c;
        Enqueue(w,Q);
      }
  }
}
```

Figura 9.17: Implementazione delle pile come vettori
La pila è una lista con inserimento ed estrazione dalla testa
Si parla di gestione *LIFO*, ovvero *Last-In First-Out*
(l'ultimo elemento inserito è il primo estratto)

L'algoritmo ha le stesse tre proprietà molto utili della *BFS*:

1. la pila non conterrà mai più di $n = |N|$ elementi:
   si può gestire la pila come un vettore `V`, senza puntatori
2. ogni elemento entra ed esce dalla cima:
   si eseguono inserimenti e cancellazioni con un semplice indice intero:
   - `top` è la posizione in cui è stato inserito l'ultimo elemento
     (dunque, l'ultima posizione occupata)
3. ogni elemento estratto dalla pila non vi rientrerà mai più

Figura 9.18: Implementazione delle pile come vettori

```
typedef struct _intstack intstack;
struct _intstack
{
  int *V;
  int size; /* dimensione massima della pila */
  int top; /* indice dell'ultima posizione occupata */
};
```

L'accesso in lettura al primo elemento è banale

```
Top(S)
{
  Return S.V[S.top];
}
```

La pila è vuota quando la posizione in cima è fuori del vettore

```
IsEmpty(Q)
{
  Return (S.top == -1);
}
```

Figura 9.19: Implementazione delle pile come vettori

L'inserimento usa la posizione dopo la cima, che scorre un passo avanti

```
Push(x,S)
{
  S.top = S.top+1;
  S.V[S.top] = x;
}
```

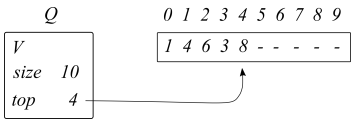
Lo spazio è esaurito quando `S.top == S.size`

La cancellazione sposta un passo indietro l'ultima posizione occupata

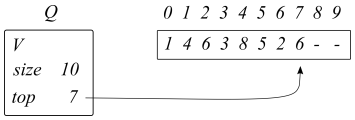
```
Pop(S)
{
  S.top = S.top-1;
}
```

La gestione della pila non richiede l'uso dell'aritmetica modulare, perché inserimenti ed estrazioni avvengono dalla stessa parte del vettore

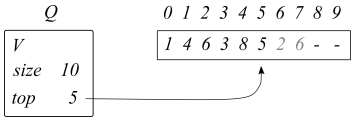
Figura 9.20: Esempio



`Top(S)` restituisce `S.V[S.top]`, cioè 8
`IsEmpty(S)` restituisce `(S.top == -1)`, cioè *false*



Inserimento di tre elementi: `Push(5,S)`, `Push(2,S)`, `Push(6,S)`



Cancellazione di due elementi: `Pop(S)`, `Pop(S)`

Figura 9.21: Visita in profondità (versione ricorsiva)

Esiste una versione ricorsiva della visita in profondità
che usa la pila di sistema anziché una pila esplicita

```
DFS(V,E,s,C,c)
{

  C[s] := c;




  for each w in Adj(s) do
    if (C[w] = 0)
    {

      DFS(V,E,w,C,c);
    }
}
```

```
DFS(V,E,s,C,c)
{
  S := ∅;
  C[s] := c;
  Push(s,S);
  while not IsEmpty(S) do
  {
    v := Top(S);
    Pop(S);
    for each w in Adj(v) do
      if (C[w] = 0)
      {
        C[w] := c;
        Push(w,S);
      }
  }
}
```

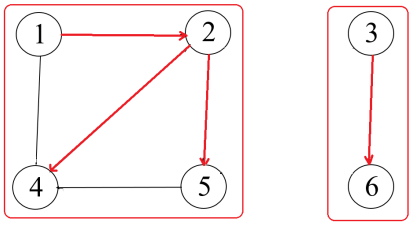
È equivalente alla precedente purché si scorra Adj(v) in ordine inverso

È molto più semplice ed elegante, ma un po' meno efficiente

Figura 9.22: Alberi di visita

La soluzione è descritta dal vettore di marcatura C, ma è anche possibile annotare per ogni vertice il lato che lo raggiunge e avere l'albero di visita

- sottografo del grafo di partenza $G = (V, E)$
- orientato in base all'ordine di visita
- rappresentabile con un vettore f che indica per ogni vertice v il vertice f_v da cui si è raggiunto v (vertice padre)



$f = [\ 1\ \ 1\ \ 3\ \ 2\ \ 2\ \ 3\ ]$

Se vi sono più componenti connesse, è una foresta (un albero ciascuna)

Nelle dispense l'albero è descritto dalla lista dei lati, nei codici non è descritto

Figura 9.23: Componenti fortemente connesse di un grafo

Su un grafo orientato $G = (N, A)$, la relazione di connessione è

- riflessiva (per convenzione)
- transitiva
- in generale non simmetrica

La relazione di connessione forte è anche simmetrica (vale nei due versi)

I vertici reciprocamente raggiungibili formano classi di equivalenza che sono dette componenti fortemente connesse

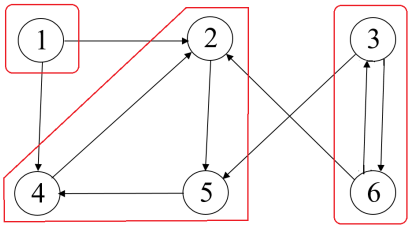


Figura 9.24: Componenti fortemente connesse e visita di un grafo
Definiamo nodi co-raggiungibili da v i nodi da cui v è raggiungibile

La componente fortemente connessa che contiene v è
l'intersezione fra nodi raggiungibili e nodi co-raggiungibili da v:

- tutti i nodi della componente son raggiungibili e co-raggiungibili da v
- tutti i nodi raggiungibili e co-raggiungibili da v sono reciprocamente raggiungibili passando per v, e quindi appartengono alla componente

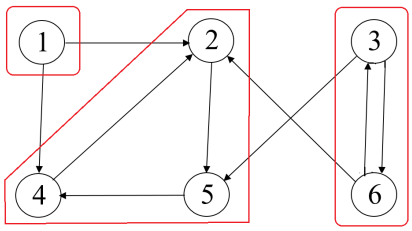
In un grafo orientato, la visita determina solo i nodi raggiungibili da v, ma invertendo il verso degli archi, la visita determina i nodi co-raggiungibili

Quindi, per determinare le componenti fortemente connesse basta scorrere i nodi come visto prima e per ciascun nodo v

- visitare il grafo da v
- visitare il grafo trasposto da v (nodi identici e archi invertiti)
- calcolare l'intersezione dei due sottoinsiemi di nodi visitati

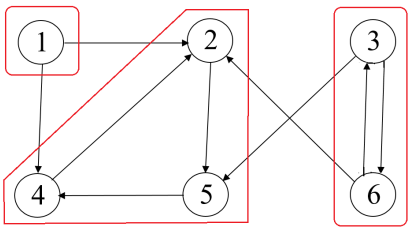
Perciò nel seguito discutiamo solo le componenti connesse

Figura 9.25: Esempio



- tutti i vertici sono marcati come non assegnati: `C = [0 0 0 0 0 0]`
- si parte dalla sorgente $s = 1$
- la visita diretta da $v = 1$ restituisce $U_1^+ = \{1, 2, 4, 5\}$:
  `C1 = [1 1 0 1 1 0]`
- la visita inversa da $v = 1$ restituisce $U_1^- = \{1\}$: `C2 = [1 0 0 0 0 0]`
- l'intersezione è $U_1 = \{1\}$: `C = [1 0 0 0 0 0]`

Figura 9.26: Esempio



- la visita diretta da $v = 2$ restituisce $U_2^+ = \{2, 4, 5\}$: `C1 = [0 2 0 2 2 0]`
- la visita inversa da $v = 2$ restituisce $U_2^- = \{1, 2, 3, 4, 5, 6\}$: `C2 = [2 2 2 2 2 2]`
- l'intersezione è $U_2 = \{2, 4, 5\}$: `C = [1 2 0 2 2 0]`
- la visita diretta da $v = 3$ restituisce $U_3^+ = \{2, 3, 4, 5, 6\}$: `C1 = [0 3 3 3 3 3]`
- la visita inversa da $v = 3$ restituisce $U_3^- = \{3, 6\}$: `C2 = [0 0 3 0 0 3]`
- l'intersezione è il sottoinsieme $U_3 = \{3, 6\}$: `C = [1 2 3 2 2 3]`
- le sorgenti 4, 5 e 6 sono già marcate

Quest'ultima implementazione è di gran lunga la più efficiente sia rispetto allo spazio sia rispetto al tempo. Seguendo sempre l'approccio delle strutture dati astratte, costruiremo librerie indipendenti dal programma in modo da nascondere l'implementazione. Questo consente di cambiare implementazione semplicemente sostituendo la libreria senza modificare il programma. Inoltre, dal punto di vista didattico il codice diventa molto leggibile e chiaro. Come già discusso, c'è uno svantaggio in termini di efficienza, dato che operazioni banali come la cancellazione di un elemento o la verifica di vuotezza (singole istruzioni) diventano chiamate di funzione, con tutto il corollario di valutazione degli argomenti, allocazione di spazio sulla pila di sistema, copia dei parametri, e deallocazione finale dello spazio dalla pila di sistema.

La seconda fase dell'esercizio comporta quindi la realizzazione della funzione `BFS` per la visita in ampiezza, sulla falsariga dello pseudocodice riportato nei lucidi. È disponibile una libreria `intqueue.h`, che implementa una coda di interi. I concetti di base di questa implementazione sono richiamati nei lucidi alle pagine 12 e 13. Nel file `intqueue0.c`, le definizioni delle funzioni sono inizialmente vuote: usando la libreria, la visita crea una componente diversa per ogni vertice del grafo (il risultato è scorretto, ma si sta avvicinando a quello finale). Si deve quindi procedere a riempire le funzioni, completando la libreria `intqueue.c`.

Vale la pena di osservare che, rispetto alla definizione astratta di coda riportata nelle dispense del modulo di teoria (e rispetto alla definizione generale di lista adottata in una delle precedenti lezioni di laboratorio), le funzioni fondamentali che manipolano una coda la ricevono per indirizzo e non restituiscono alcun risultato. Questo è banalmente dovuto al fatto che, in genere, nel linguaggio C si preferisce passare alle funzioni le strutture complesse per indirizzo, per limitare le operazioni di copia dei dati. Analogamente, e per lo stesso motivo, si preferisce restituire un risultato di tipo struttura con il passaggio per indirizzo, anziché con l'istruzione

`return`. Lo stesso succederà in seguito per la libreria di gestione delle pile.

Analisi di complessità dell'algoritmo *BFS* L'algoritmo di visita in ampiezza si può descrivere con il seguente pseudocodice.

```
 1: procedure BFS(V, E, s, C, c)
 2:     Q := ∅;
 3:     C[s] := c;
 4:     Enqueue(s, Q);
 5:     while not IsEmpty(Q) do
 6:         v := Front(Q);
 7:         Dequeue(Q);
 8:         for all w ∈ Adj(v) do
 9:             if C[w] = 0 then
10:                 C[w] := c;
11:                 Enqueue(w, Q);
12:             end if
13:         end for
14:     end while
15: end procedure
```

La creazione di una coda vuota, la marcatura della sorgente e il suo inserimento in coda richiedono tempo costante, $\Theta(1)$. Il ciclo alle righe $5-14$ è abbastanza complicato da valutare, perché non indica chiaramente il numero di iterazioni: bisogna ricavarlo con un ragionamento. Ad ogni iterazione, si estrae un vertice dalla coda. Questo vertice era entrato in coda o prima del ciclo (se è la sorgente) o dentro il ciclo soddisfacendo la condizione alla riga 9 (`if (C[w] == 0)`). In entrambi i casi, si tratta di vertici appena raggiunti dalla visita. Appena un vertice viene inserito nella coda il corrispondente indice nel vettore C viene marcato. Quindi un vertice uscito dalla coda è marcato, non viene mai smarcato e non rientrerà mai più nella coda grazie alla condizione alla riga 9. Questo significa che ogni iterazione del ciclo $5-14$ è associata a un diverso vertice raggiunto dalla visita. I vertici raggiunti dalla visita sono tutti e soli quelli della componente connessa, per cui la complessità temporale del ciclo corrisponde a una sommatoria con indice $v \in U_s$. Dentro il ciclo, l'estrazione del vertice dalla coda (righe 6 e 7) richiede tempo costante, mentre il ciclo più interno (righe 8–13) scorre i vertici adiacenti, per mezzo della lista degli archi uscenti. Le operazioni interne a questo ciclo (test di assenza della marcatura, marcatura e inserimento nella coda) richiedono tempo costante, $\Theta(1)$. Riassumendo:

$$T = \Theta(1) + \sum_{v \in U_s} \left(\Theta(1) + \sum_{a \in \Delta_v^+} \Theta(1) \right)$$

Se consideriamo per ogni vertice della componente tutti gli archi uscenti, stiamo sostanzialmente considerando tutti gli archi della componente, dato che non solo l'origine, ma anche la destinazione di tali archi appartiene alla componente:

$$\sum_{v \in U_s} \sum_{a \in \Delta_v^+} 1 = \sum_{a \in A(U_s)} 1 = |A(U_s)|$$

da cui

$$T = \Theta(1) + \Theta(n_s) + \Theta(m_s) = \Theta(n_s + m_s)$$

dove indichiamo per brevità con $n_s = |U_s|$ e $m_s = |A(U_s)|$ il numero di nodi e il numero degli archi della componente che include il nodo s.

Analisi di complessità per la ricerca delle componenti connesse L'algoritmo di ricerca delle componenti connesse si può invece descrivere con il seguente pseudocodice.

```
1: procedure ComponentiConnesse(V, E)
2:     c := 0;
3:     for all s ∈ V do
4:         if C[s] = 0 then
5:             c := c + 1;
6:             Visita(V, E, s, C, c);
7:         end if
8:     end for
9: end procedure
```

L'azzeramento iniziale delle marcature richiede tempo lineare, $\Theta(n)$. Il successivo ciclo scorre i vertici, che sono n. Nel caso pessimo, esegue (apparentemente) un test di assenza di marcatura, l'incremento dell'indice della componente e una visita. In totale, si conclude che:

$$T = \Theta(n) + \sum_{s \in V} (\Theta(1) + \Theta(n_s + m_s))$$

che si può maggiorare un po' brutalmente osservando che $n_s \leq n$ e $m_s \leq m$ per ogni $s \in V$ per ottenere $T \in O\left(n^2 + nm\right)$. D'altra parte, $T \notin \Omega\left(n^2 + nm\right)$. Infatti, ogni volta che si esegue una visita, tutti i vertici della componente vengono marcati, e quindi il test di assenza di marcatura viene soddisfatto solo una volta per ogni componente.

$$T = \Theta(n) + \sum_{s \in V} \Theta(1) + \sum_{c=1}^{nc} \Theta(n_c + m_c)$$

dove si è indicato con nc il numero di componenti e con n_c e m_c il numero di vertici e di archi della componente di indice c (per comodità, abbiamo cambiato notazione rispetto all'analisi dell'algoritmo di visita). È ovvio che la somma di tali due numeri su tutte le componenti dà, rispettivamente, il numero totale di vertici e di lati del grafo.

$$T = \Theta(n) + \Theta(n) + \Theta(n + m) = \Theta(n + m)$$

che è chiaramente molto meglio della rozza stima per eccesso da cui si era partiti.

Terza fase (`compconn3.c` e `intstack.c`) Nella terza fase dell'esercizio, sostituiamo la `BFS` con la visita in profondità (`DFS`), descritta nelle pagine $14 - -16$ dei lucidi. Tale algoritmo è praticamente identico alla `BFS`: confrontando i relativi pseudocodici, ci si rende conto che cambiano solo la struttura dati usata per conservare l'insieme dei vertici raggiunti e ancora non usati per proseguire la visita, e i nomi delle funzioni usate per:

- aggiungere elementi all'insieme (`Enqueue` anziché `Push`)
- togliere elementi dall'insieme (`Dequeue` anziché `Pop`)
- accedere all'unico elemento leggibile dell'insieme (`Front` anziché `Top`)

È disponibile la libreria `intstack.h`, sempre con un file `intstack0.c` di definizioni vuote (che genera lo stesso risultato sbagliato di prima). Il file va completato in base ai meccanismi descritti alle pagine 17–18 dei lucidi.

Si può condurre un'analisi di complessità per la visita in profondità *DFS* e per la ricerca delle componenti connesse che danno luogo allo stesso risultato ottenuto più sopra, ancora una volta perché le funzioni che manipolano la pila sono molto simili a quelle usare per manipolare la coda, e hanno anch'esse complessità costante.

Quarta fase (`compconn4.c` Si può quindi realizzare la versione ricorsiva della visita in profondità (`DFSricorsiva`), che ha forti analogie con quella iterativa, come illustrato nei lucidi a pagina 19. A questo proposito è interessante disegnare passo per passo su un piccolo esempio quanto avviene nella pila di sistema, paragonando il comportamento della funzione ricorsiva con quello della funzione iterativa corrispondente. Questa ha una pila di numeri interi esplicitamente dichiarata nell'area di memoria riservata alla funzione `DFS`, mentre la funzione ricorsiva ha solo una cella intera `s`, dedicata alla sorgente. Però le chiamate ricorsive della funzione stessa creano sulla pila di sistema una sequenza di blocchi di memoria identici, che contengono le sei informazioni `grafo *pG`, `int s`, `vint C`, `int c`, `int w` e `posarco pa`. In particolare, la cella `s` si trova ripetuta a intervalli regolari lungo la pila di sistema e contiene esattamente gli stessi indici interi contenuti nella pila esplicita quando si esegue la funzione iterativa. La versione ricorsiva ha il vantaggio dell'eleganza ed essenzialità. Ovviamente, occupa molta più memoria sulla pila di sistema e richiede più tempo, a causa del meccanismo di allocazione, copia e deallocazione implicito nelle chiamate. La complessità asintotica rimane, tuttavia, la stessa.

Tutti e tre gli algoritmi possono essere modificati per restituire (invece del vettore marcatura C o in aggiunta ad esso) l'albero di visita, cioè l'albero formato dai soli archi utilizzati nel corso della visita. Tale albero è descritto alla pagina 20 dei lucidi, ed è un buon esercizio aggiungerne la costruzione ai codici realizzati in precedenza.

Quinta fase (`compfortconn1.c`) Consideriamo ora la ricerca delle componenti fortemente connesse su un grafo orientato. Anche questo problema è riducibile al problema della visita, ma questa volta si richiede di calcolare sia l'insieme U_v^+ dei nodi raggiungibili da v sia l'insieme U_v^- dei nodi co-raggiungibili da v. Infatti, la componente fortemente connessa che contiene v coincide con l'intersezione di tali due insiemi, dato che la relazione di connessione forte fra nodi è simmetrica e transitiva e:

- la componente fortemente connessa è contenuta in $U_v^+ \cap U_v^-$ perché tutti i suoi nodi sono per definizione raggiungibili e co-raggiungibili da v;
- $U_v^+ \cap U_v^-$ è contenuto nella componente connessa perché
  - tutti i vertici dell'intersezione sono raggiungibili e co-raggiungibili da v;
  - per transitività, tutte le coppie di vertici sono vicendevolmente raggiungibili e co-raggiungibili passando per v.

Per realizzare l'algoritmo, possiamo riutilizzare il codice scritto per implementare la visita del grafo, ma la visita va ripetuta due volte: sul grafo dato e sul grafo trasposto. Occorre quindi definire una funzione che costruisca il grafo trasposto a partire da quello dato. Inoltre, la visita applicata sui due grafi deve restituire due vettori diversi, di cui poi bisogna calcolare l'intersezione per marcare gli elementi di C. Siccome i due vettori sono in generale più grandi rispetto alla componente, vanno ripuliti ogni volta prima di cominciare la visita (contrariamente al caso

delle componenti connesse, in cui una visita restituisce esattamente la componente cercata).

Sarebbe anche possibile non costruire fisicamente il grafo trasposto, realizzando una procedura di visita che usi gli archi all'indietro anziché in avanti per la seconda delle due chiamate. Affinché questa procedura sia efficiente, però il grafo deve essere rappresentato come vettore di *backward star*, oltre che come vettore di *forward star*. Bisogna quindi aggiornare la libreria che descrive il grafo, aumentando l'occupazione di memoria in misura analoga a quanto abbiamo fatto costruendo un secondo grafo a partire dal primo.

Analisi di complessità per la ricerca delle componenti fortemente connesse L'algoritmo di ricerca delle componenti fortemente connesse si può descrivere con il seguente pseudocodice.

```
 1: procedure ComponentiFortementeConnesse(N, A)
 2:     A^T := GrafoTrasposto(A);
 3:     c := 0;
 4:     for all s ∈ N do
 5:         if C[s] = 0 then
 6:             c := c + 1;
 7:             Visita(N, A, s, U^+, c);
 8:             Visita(N, A^T, s, U^-, c);
 9:             C := C + (U^+ ∩ U^-);
10:         end if
11:     end for
12: end procedure
```

Purtroppo, in questa analisi non si può ripetere esattamente quanto detto per le componenti connesse. Per prima cosa, occorre costruire il grafo trasposto. Questo comporta di:

1. creare e riempire un vettore di *forward star* orientate in modo opposto (dalla destinazione all'origine), in tempo $\Theta(n+m)$;
2. per ogni nodo non ancora marcato, visitare sia il grafo dato sia il grafo trasposto
3. intersecare i sottoinsiemi ottenuti con le due visite.

Gli insiemi generati dalle due visite contengono la componente fortemente connessa che include il nodo corrente, ma non coincidono con essa: solo la loro intersezione coincide con essa. Non è quindi più vero che si esegue una sola visita per ogni componente. Rimane la stima grezza $O\left(n^2+nm\right)$[2]. Fortunatamente, esistono algoritmi per il calcolo delle componenti fortemente connesse che riescono a raggiungere la complessità $\Theta(n+m)$, ma non vengono trattati in questo corso.

9.5 Esercizi

Realizzare una versione efficiente della procedura di stampa, che scorra una volta sola il vettore C, usando un'opportuna struttura dati ausiliaria.

[2] A rigore, si può migliorare la stima considerando che per ogni nodo le due visite toccano solo nodi e archi della componente (non fortemente, ma semplicemente) connessa che lo contiene, per esempio ottenendo $\Theta\left(n \max_{s \in V}(n_s + m_s)\right)$.

Modificare la gestione della coda per la visita in ampiezza e della pila per la visita in profondità portando le istruzioni di allocazione e deallocazione all'esterno della procedura per il calcolo delle componenti connesse, in modo da riutilizzare la stessa struttura senza doverla allocare e deallocare ogni volta.

Capitolo 10

Alberi binari

Questo capitolo è dedicato agli alberi, e in particolare agli alberi binari. Gli alberi sono grafi. Per cominciare, parliamo di grafi non orientati, quindi un insieme di vertici e un insieme di lati. Hanno la caratteristica fondamentale di essere:

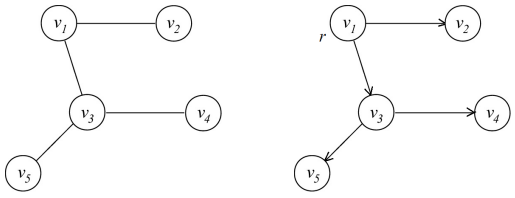
1. *connessi*: ogni coppia di vertici del grafo stesso è legata da un cammino;
2. *aciclici*: nessuno dei cammini costituiti dai lati del grafo si richiude su sé stesso ciclicamente.

Questo automaticamente significa che qualunque coppia di vertici si prenda, il cammino che li lega è unico. Se ce ne fossero due, potremmo usarne uno per andare e uno per tornare, e avremmo chiuso un ciclo.

Un albero $T = (V, E)$ è un grafo

- connesso: ogni coppia di vertici è legata da un cammino
- aciclico: nessun cammino si richiude su sé stesso

Quindi, ogni coppia di vertici è legata esattamente da un cammino
Albero radicato è un albero con un vertice r marcato come radice
È orientato dalla radice attraverso i nodi interni sino alle foglie



Albero ordinato ha una relazione di ordine totale sui figli di ciascun nodo

Albero binario ha una al massimo due figli per ciascun nodo

Figura 10.1: Alberi

Si può considerare anche una definizione orientata di albero. Un modo molto semplice di ottenere tale definizione è di *radicare* un albero, cioè di assegnare a uno

dei vertici il nome di *radice* e orientare tutti i cammini che connettono quel vertice (a questo punto, nodo) agli altri procedendo dalla radice al nodo, oppure dal nodo alla radice. Così facendo, otteniamo un grafo orientato. In alcuni testi di teoria dei grafi, questi alberi orientati radicati si chiamano *arborescenze*, ma in questo corso, e in genere nei testi di base di algoritmi, questa nomenclatura non si usa. Una volta stabilita una radice, i nodi dell'albero stesso che non hanno archi uscenti (cioè i nodi terminali dei vari cammini) si definiscono *foglie* e quelli che ne hanno si definiscono *nodi interni*. In un albero radicato ogni nodo, tranne la radice, ha un arco entrante che lo collega al cosiddetto *nodo padre*, e ha zero o più archi uscenti, che lo collegano ai *nodi figli*.

La Figura 10 riporta a sinistra un albero (qualunque coppia di vertici è collegata da uno e un solo cammino) e a destra l'albero orientato ottenuto radicando il primo nel vertice (ora nodo) v_1. I nodi interni sono v_1 e v_3, le foglie sono v_2, v_4 e v_5. Il nodo v_1 è padre dei nodi v_2 e v_3.

10.1 Alberi ordinati

Introduciamo ora qualche concetto ulteriore che non ha più strettamente a che fare con gli alberi in quanto grafi, esulando dalla pura topologia, cioè dalla semplice relazione binaria fra oggetti elementari. Consideriamo ciascun nodo dell'albero e ordiniamo totalmente gli archi che ne escono. Nella Figura 10, per esempio, diciamo che l'arco (v_1, v_3) precede l'arco (v, v_2), e l'arco (v_3, v_5) precede l'arco (v_3, v_4). Allora, avremo un *albero ordinato*. Un albero ordinato impone una relazione di ordine totale, senza *ex aequo*, sugli archi che escono da ciascun nodo, e quindi sui nodi figli di ciascun nodo.

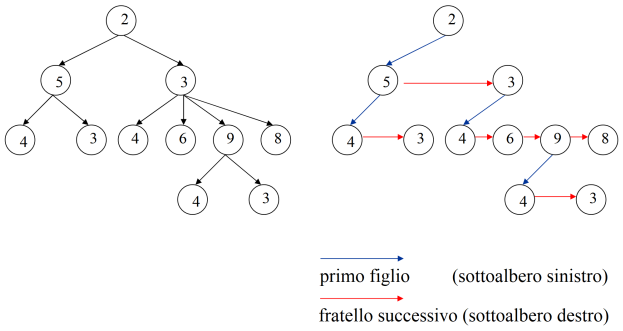
Se fissiamo un limite massimo al numero degli archi uscenti da ogni nodo, fissiamo la cosiddetta *arità* dell'albero (orrido anglicismo, etimologicamente insensato). L'unico aspetto che ci interessa a riguardo è il concetto di *albero binario*. Un albero binario è un albero che ha al massimo due figli per ciascun nodo (ovviamente, può averne anche uno solo o nessuno). In un albero binario ordinato tipicamente un nodo ha un *figlio sinistro* (il primo nell'ordine) e un *figlio destro* (il secondo), ma in effetti un albero può avere anche solo un figlio destro (il motivo risulterà chiaro fra poco).

Gli alberi radicati e ordinati in generale non vengono rappresentati come grafi, ma per livelli, con la radice in alto, gli archi uscenti rivolti verso il basso, il primo figlio di un nodo sulla sinistra e nel livello successivo al nodo, e il secondo figlio sulla destra nel livello successivo.

Dato un generico albero ordinato, si può automaticamente costruire un corrispondente albero binario. Questa costruzione è rappresentato nella Figura 10.2: si parte da un albero ordinato generico, con nodi che hanno qualsiasi numero di figli, e conserviamo per ogni nodo il primo arco uscente, cancelliamo tutti gli altri successivi, ma li sostituiamo con una catena di archi (un cammino orientato) che collega tutti i figli dello stesso padre nel loro ordine. Per esempio, anziché far uscire dal nodo 2 gli archi $(2, 5)$ e $(2, 3)$, facciamo uscire solo il primo arco, $(2, 5)$ e poi colleghiamo i nodi 5 e 3 nell'ordine con un arco $(5, 3)$. Ancora, invece di avere gli archi $(3, 4)$, $(3, 6)$, $(3, 9)$ e $(3, 8)$, facciamo uscire dal nodo 3 soltanto il primo arco, $(3, 4)$, e poi colleghiamo 4, 6, 9 e 8 con una catena di tre archi consecutivi. In questo modo, otteniamo un albero binario che corrisponde esattamente (e biunivocamente!) all'albero di partenza, e che consente di eseguire qualunque algoritmo sull'albero stesso, fingendo che sia binario. Questo genere di trucchi ormai ci è familiare, poiché abbiamo visto molti esempi di strutture che ne simulano altre: terne di vettori che simulano liste, matrici quadrate che simulano grafi, ecc... L'importante

Un generico albero ordinato si può rappresentare con un albero binario:

- il sottoalbero sinistro di un nodo nell'albero binario rappresenta il primo figlio nell'albero ordinato
- il sottoalbero destro di un nodo nell'albero binario rappresenta il fratello successivo nell'albero ordinato



Applicazioni tipiche sono gli alberi genealogici e i sistemi di classificazione

Figura 10.2: Alberi ordinati

è che ci siano delle funzioni che consentano di eseguire su un processore le operazioni che idealmente compieremmo su un oggetto astratto.

Nel seguito, ignoreremo qualunque categoria di alberi che non siano gli alberi binari. Qualunque problema richieda di gestire alberi ordinati o no con un'arietà qualsiasi può essere ridotto a un problema, e quindi ad algoritmi, su alberi binari. È per questo che siamo particolarmente interessati a trattarli.

10.2 Alberi binari

Introduciamo ora, come di consueto, la struttura astratta albero binario. Questa struttura ha molto in comune con le liste, in particolare la definizione ricorsiva e la struttura totalmente dinamica.

Diremo *albero binario su un insieme base U* un oggetto che ha una definizione ricorsiva, come le liste. Può essere

- un insieme vuoto, e questo è il caso base;
- una terna, costituita da
  1. un elemento r dell'insieme base, tipicamente chiamato *radice*;
  2. un albero binario T_s detto *sottoalbero sinistro*;
  3. un albero binario T_d detto *sottoalbero destro*.

Ancora una volta, questa non è una definizione tautologica, sebbene definiamo un albero binario in termini di alberi binari, perché c'è il caso base: prima o poi arriveremo al caso base e arresteremo la ricorsione (senza indicare quando, dunque con un limite indefinito ma non infinito).

Un albero binario T su un insieme U ha una definizione ricorsiva: è

- un insieme vuoto (caso base) oppure
- un elemento (nodo) radice $a_r \in U$ associato a due alberi binari
  1. sottoalbero sinistro T_s
  2. sottoalbero destro T_d

La definizione non è tautologica perché il caso base arresta la ricorsione, ma il numero dei nodi non è soggetto ad alcun limite

Le operazioni di proiezione e sostituzione usano un insieme di posizioni P per accedere ai nodi dell'albero

- dato un albero T, solo la posizione della radice r è nota
- dato un albero T e una posizione p si può ricavare direttamente solo
  - l'informazione associata al nodo in posizione p
  - le posizioni delle radici dei suoi due sottoalberi

La situazione è del tutto analoga a quella delle liste (*vedi Lezione* 7)

Figura 10.3: Albero binario: struttura dati astratta

Esattamente come nelle liste, non si può accedere a piacere a qualunque posizione dell'albero. Occorre un concetto astratto di *posizione* che consente di accedere a un elemento dell'albero, e solo alcune posizioni privilegiate sono direttamente accessibili (in particolare, la radice). Quindi tutte le operazioni che hanno bisogno di accedere all'albero in lettura, scrittura o modifica avranno bisogno di ricevere l'albero, ma anche una posizione (qualunque cosa essa sia). Dato un albero, l'unica posizione direttamente accessibile è quella della radice. Infine, se abbiamo un albero e una posizione, non possiamo saltare da quella posizione a qualunque altra, ma solo ricavare la posizione delle radici dei due sottoalberi dell'elemento nella posizione data. Un'estensione piuttosto frequente è la possibilità di accedere alla posizione del nodo padre, ammesso che un nodo padre esista. Siccome in generale, un nodo dell'albero può avere sottoalberi vuoti, oppure può non avere un nodo padre (è il caso della radice), occorre definire una posizione fittizia che è al solito rappresentata dal simbolo $\perp$, e che gestiremo come nel caso delle liste.

A questo punto possiamo passare in rassegna le operazioni eseguibili sugli alberi binari su un dato insieme base. Sono più o meno le solite operazioni. Avremo un'operazione di *proiezione* `leggenodo`, che richiede un albero e una posizione e fornisce in uscita un elemento dell'insieme base, e consiste nel leggere l'informazione contenuta in una data posizione di un dato albero. La *sostituzione* corrisponde alla scrittura e la chiameremo `scrivenodo`: riceve in ingresso un albero, una posizione e un elemento dell'insieme base e in uscita restituisce un albero che è lo stesso albero di partenza, ma modificato ponendo nella posizione data la nuova informazione. Avremo un'operazione di *verifica di vuotezza*, cioè una funzione `alberovuoto`, che riceve un albero e in uscita restituisce un valore booleano (vero o falso, 1 o 0), secondo che l'albero sia vuoto oppure non vuoto. Alla funzione `primolista` delle liste, che consentiva di ottenere la posizione iniziale di una lista data, corrisponde una funzione `radice`, che, dato un albero, fornisce la posizione della sua radice.

Sia $\mathcal{T}$ l'insieme di tutti i possibili alberi binari su U

Gli alberi binari ammettono tipicamente le seguenti operazioni

- proiezione: dato un albero e una posizione, fornisce il nodo corrispondente
$$\text{leggenodo} : \mathcal{T} \times P \to U$$
- sostituzione: dato un albero, una posizione e un nodo inserisce il nodo nell'albero sostituendo quello puntato dalla posizione
$$\text{scrivenodo} : \mathcal{T} \times P \times U \to \mathcal{T}$$
- verifica di vuotezza: dato un albero, indica se è vuoto
$$\text{alberovuoto} : \mathcal{T} \to \mathbb{B} \qquad (\text{ovvero } \{0,1\})$$
- accesso alla radice: dato un albero, fornisce la posizione del nodo radice
$$\text{radice} : \mathcal{T} \to P$$
Se l'albero è vuoto, restituisce la posizione fittizia $\perp$

Figura 10.4: Alberi binari: operazioni

Nel caso di un albero vuoto, tale posizione non esiste, e la funzione restituirà la posizione fittizia $\perp$.

Qui cominciano le differenze sostanziali con le liste. Invece di avere una posizione successiva, avremo due posizioni "figlie", restituite dalle funzioni `figliosinistro` e `figliodestro`. Queste funzioni ricevono un albero e una posizione e restituiscono un'altra posizione nell'albero stesso. Se non esiste un figlio sinistro o destro, perché il corrispondente sottoalbero è vuoto, il risultato sarà la posizione fittizia $\perp$. Altrimenti, sarà la posizione della radice del sottoalbero desiderato. Infine, se l'implementazione la ammette, potremo avere una funzione `padre`, che dato un albero e una posizione restituisce la posizione del nodo padre, che esiste sempre, tranne nel caso della radice. Queste sono le funzioni per muoversi all'interno di un albero.

Occorrono poi funzioni per gestire dinamicamente queste strutture, cioè ingrandirle e rimpicciolirle a piacere (fino a svuotarle). Nelle liste, avevamo operazioni di inserimento e cancellazione, e potremmo pensare di introdurle anche negli alberi. Qui però la situazione è più sofisticata. Data una lista e una posizione, se si vuole inserire un elemento nella posizione, basta inserirlo e spostare in avanti tutti gli elementi da quella posizione in poi. Abbiamo studiato i problemi pratici di implementare questa operazione tecnologicamente, ma dal punto di vista astratto è tutto lì. Cerchiamo di fare qualcosa di simile su un albero binario: dato un albero binario e una posizione, vogliamo inserire un nuovo dato in quella posizione. Tutto il sottoalbero appeso a quella posizione deve in qualche maniera spostarsi, ma dove? Non c'è una direzione ovvia ("in avanti"). Si potrebbe scendere a sinistra oppure a destra, e a priori ciascuna delle due soluzioni potrebbe essere giusta in situazioni diverse imprevedibili a priori. Ci si potrebbe rassegnare a definire due funzioni di inserimento: una che cala il sottoalbero a sinistra e una che lo cala a destra. Per la cancellazione, però, il problema è davvero insolubile, e questo spinge a cambiare la definizione, introducendone una più adatta alla struttura degli alberi.

Sia $\mathcal{T}$ l'insieme di tutti i possibili alberi binari su U

Gli alberi binari ammettono tipicamente le seguenti operazioni

- figlio sinistro: dato un albero e una posizione, fornisce la posizione della radice del figlio sinistro del nodo puntato dalla posizione

$$\text{figliosinistro} : \mathcal{T} \times P \to P$$

Se non esiste un sottoalbero sinistro, restituisce $\perp$

- figlio destro: dato un albero e una posizione, fornisce la posizione della radice del figlio destro del nodo puntato dalla posizione

$$\text{figliodestro} : \mathcal{T} \times P \to P$$

Se non esiste un sottoalbero destro, restituisce $\perp$

- padre: dato un albero e una posizione, fornisce la posizione del nodo padre

$$\text{padre} : \mathcal{T} \times P \to P$$

Per il nodo radice, che non ha padre, restituisce $\perp$

Figura 10.5: Alberi binari: operazioni

Sia $\mathcal{T}$ l'insieme di tutti i possibili alberi binari su U

Inserimento e cancellazione di elementi per un albero binario
sono piuttosto diversi dalle analoghe operazioni per le liste:

- costruzione: dato un nodo e due alberi binari, restituisce un albero che ammette il nodo come radice, il primo albero come figlio sinistro e il secondo come figlio destro

$$\text{costruiscealbero} : U \times \mathcal{T} \times \mathcal{T} \to \mathcal{T}$$

- cancsottoalbero: dato un albero e una posizione, cancella dall'albero il sottoalbero che ha come radice il nodo nella posizione data

$$\text{cancsottoalbero} : \mathcal{T} \times P \to \mathcal{T}$$

Questa differenza è dovuta alla struttura gerarchica!

Figura 10.6: Alberi binari: operazioni

Perché il problema della cancellazione è peggiore? Consideriamo un albero e una posizione supponiamo di voler cancellare il nodo che sta in quella posizione. In una lista, ovviamente, tutte le posizioni successive risalgono all'indietro e la lista è intatta. Nel caso dell'albero, cancellare un nodo lascia due sottoalberi figli. Quale dei due sale al livello del padre? E che cosa succede all'altro sottoalbero? Non c'è una vera e propria soluzione. Per questo motivo, inserimenti e cancellazioni di singoli elementi vengono sostituite da altre operazioni più adatte alla natura astratta degli alberi. Queste due operazioni sono dette *costruzione* e *cancellazione*. La funzione `costruiscealbero` riceve un elemento dell'insieme base e due alberi, e restituisce un nuovo albero che ha come radice l'elemento dell'insieme base che gli è stato passato, e come figli di sinistra e di destra i due alberi che gli sono stati passati. Se usiamo questa operazione di base, l'unico modo per costruire alberi è fondere fra loro sottoalberi, appendendoli a un radice che contiene un nuovo elemento. Questo procedimento può essere applicato anche ad alberi vuoti, costruendo un semplice alberello con un unico nodo, oppure a un albero vuoto e uno no, ricadendo in un certo senso nell'idea di "appendere" un albero a un nuovo nodo, come quando si inserisce un elemento in una lista e quel che segue viene fatto scivolare in avanti.

La cancellazione è molto più rude: dato un albero e una posizione, la funzione `cancsottoalbero` restituisce un altro albero che cancella completamente non solo il nodo nella posizione indicata, ma anche tutti quelli dei due sottoalberi ad esso appesi.

Figura 10.7: Alberi binari: operazioni

In matematica basta definire un oggetto per crearlo

Nelle implementazioni concrete, questo in genere non vale
Quindi è opportuno definire

- creazione: crea un albero binario vuoto

$$\text{creaalbero} : () \to \mathcal{T}$$

- distruzione: distrugge un albero

$$\text{distruggealbero} : \mathcal{T} \to ()$$

Figura 10.8: Alberi binari: operazioni

Rimangono come sempre le procedure che vengono a patti con l'implementazione concreta, cioè quelle di *creazione* e *distruzione*, che gestiscono in pratica le strutture in memoria. La funzione `creaalbero` crea un albero vuoto, mentre la funzione `distruggealbero` distrugge completamente un albero dato.

Come per le liste, esistono molte implementazioni diverse. Intanto, essendo un albero anche un grafo, si possono usare tutte le implementazioni descritte per i grafi. Poi, si può usare il vettore padre, descritto per gli alberi radicati nel capitolo sulle visite di grafi[1]. Altre implementazioni, però sono specificamente pensate per gli alberi radicati e ordinati. Qui approfondiremo una sola implementazione, basata su puntatori.

Esattamente come per le liste, l'idea è che le posizioni siano indirizzi di memoria. Precisamente, un albero è visto come un puntatore all'elemento radice (come

[1] Trovare la sezione

L'idea base è di rappresentare le posizioni con indirizzi di memoria

- l'albero corrisponde allora alla posizione della radice
- ogni elemento dell'albero corrisponde a una struttura con
  - il dato $a \in U$
  - la posizione della radice del sottoalbero sinistro ($\perp$ se non esiste)
  - la posizione della radice del sottoalbero destro ($\perp$ se non esiste)
  - la posizione del nodo padre ($\perp$ se non esiste)

```
#define NO_TREE NULL                          (albero vuoto)
#define NO_NODE NULL                          (posizione esterna all'albero)

typedef nodo *alberobinario;          (l'albero è l'indirizzo della radice)
typedef nodo *posizione;              (la posizione del nodo è il suo indirizzo)

typedef struct _nodo nodo;
struct _nodo {
   U a;                                       (U è il tipo del nodo generico)
   posizione Ts;              (posizione della radice del sottoalbero sinistro)
   posizione Td;              (posizione della radice del sottoalbero destro)
   posizione padre;                          (posizione del nodo padre)
};
```

Figura 10.9: Alberi binari: implementazione con puntatori

nelle liste lo è al primo elemento o alla sentinella). Gli elementi sono strutture, che contengono un campo informazione, di tipo U, e dei campi puntatore, di tipo posizione, per collegare l'elemento alle radici dei due sottoalberi, che sono gli indirizzi in memoria delle strutture contenenti gli elementi radice. Se vogliamo una rappresentazione completa, avremo anche un campo posizione per il nodo padre. Ciascuna delle tre posizioni può eventualmente essere il puntatore `NULL`, a rappresentare il fatto che uno dei sottoalberi è vuoto oppure che il nodo è la radice, e quindi non ha padre. Questo corrisponde alla circostanza per cui nelle liste lineari senza sentinella il campo `succ` dell'ultimo elemento e il campo `pred` del primo contengono il valore `NULL`. Quindi, un albero e una posizione sono entrambi puntatori a nodo. Un nodo è una struttura, che contiene un campo informazione e tre campi posizione, che puntano il sottoalbero sinistro, il sottoalbero destro e il nodo padre.

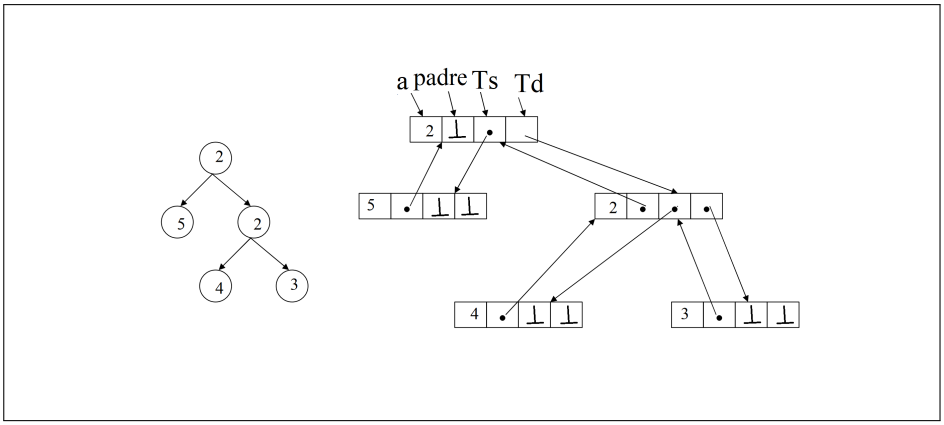


Figura 10.10: Alberi binari: implementazione con puntatori

A questo punto siamo in grado di costruire e gestire alberi binari. La Figura 10.10 offre un semplice esempio con 5 nodi che contengono come informazione un numero intero. Ciascuno dei 5 nodi è una quaterna di oggetti: un semplice numero intero, che è l'informazione, e tre puntatori. È ovvio che c'è un certo *overhead* di occupazione di memoria, ma la struttura è asintoticamente efficiente: rappresenta n informazioni in spazio $\Theta(n)$, e non potrebbe avere complessità inferiore. La costante moltiplicativa può non essere delle migliori, a causa dell'occupazione dovuta ai puntatori.

10.3 Laboratorio

considereremo un esercizio abbastanza corposo che ci mostrerà una possibile applicazione degli alberi binari per rappresentare delle strutture gerarchiche che cosa si tratta effettivamente quello che vogliamo realizzare è una specie di calcolatore aritmetico Che cos'è Un calcolatore aritmetico Un calcolatore aritmetico è un programma che riceve in ingresso un'espressione aritmetica cioè un'espressione costituita da numeri e da operatori che sono le 4 classiche operazioni che si imparano le scuole elementari la somma La differenza è il prodotto è la divisione combinate in vario modo ovviamente in modo molto più sofisticato di quello che si vede alle elementari diciamo nel modo che si vede alle scuole medie con le parentesi questa espressione andava caricata rappresentata in qualche modo all'interno del nostro programma Dopodiché il programma ci chiede sostanzialmente tre cose la prima è di Calcolare il valore di questa espressione E stamparlo a video la seconda sera di

convertire l'espressione dal formato iniziale che poi discutiamo in nel classico forma 0:24:05 Video la seconda sera di convertire l'espressione dal formato iniziale che poi discutiamo in nel classico formato quello in cui gli operatori sono delle dei singoli caratteri il più il per il diviso il meno e le espressioni sono dei numeri e alla fine della fiera si calcola il risultato finale e infine vogliamo anche determinare quella che è la profondità dell'espressione stessa dandole un'opportuna definizione che Chiama in causa ovviamente un albero e quindi la vedremo in seguito Allora per cominciare in che formato sono descritte queste espressioni non sono descritte in un formato abbastanza dal sapore abbastanza informatico che quello che vedete qui questo esempio prod di (Somma di 5,3, divisione di 10 e della differenza di 4 e due che è una appunto ispirata alle chiamate di funzione ma che in realtà somiglia moltissimo alla cosiddetta notazione prefissa ho notazione polacca che è comunemente usata in matematica è quella nella quale gli operatori precedono gli operandi la differenza fondamentale che la notazione polacca Intanto usa simboli di operatori più piccoli più elementari di questi e poi non usa ha fatto le parentesi ne aperte linee chiuse nelle virgole perché sto rettamente Non sono necessarie nelle abbiamo aggiunte per dare un tono più informatica anche per rendere un po' più leggibile In realtà ha un essere umano Questa scrittura perché le (e chiusa e ci permettono di capire dove cominciano dove finiscono gli operandi Ok nel file input 01 trovate alcune di queste espressioni e soprattutto è possibile cominciare a ragionare su che cosa sostanzialmente come sostanzialmente si possono descrivere Che cosa sono da un punto di vista matematico sono delle strutture intrinsecamente gerarchiche perché sono gerarchiche perché per definizione questa prima struttura che abbiamo preso a esempio richiede di fare il prodotto di che cosa di due numeri questi due numeri sono i due numeri separati da virgole all'interno delle parentesi tonde e però non sono numeri sono alla volta delle espressioni Quindi in effetti la definizione che stiamo adottando è una definizione ricorsiva Tanto per cambiare nella quale un'espressione può essere o semplice E allora sarà un numero diciamo un numero reale in genere Oppure un'espressione composta ed è composta da tre termini da una Terna in cui il primo termine che precede le pare è l'operatore e deve essere una di queste quattro stringhe perché ci interessano solo le espressioni aritmetiche Mentre gli altri due termini dell'espressione sono le operando uno lo prendo due due operandi e questi operandi possono essere espressioni aritmetiche semplici o potrebbe fare una volta essere composte Nel caso specifico i due operandi dell'operatore prod sono l'espressione composta somma di 5 e 3 e l'espressione composta divisione rapporto di 10 e differenza di 4 e due quindi siamo ulteriore livello ancora di composizione se la somma è applicata due operandi che finalmente sono semplici 53 la divisione applicata a un operando semplice che il 10 è un operando comporto che è la differenza di 4 e due ed è Quest'ultimo che finalmente applicato a due operandi semplici 4:02 e poteva in realtà andare avanti a piacere senza un limite predefinito la struttura si può come un albero è abbastanza intuitivo nel quale i nodi interni corrispondono degli operatori e le foglie corrispondono invece ha degli operandi semplici cioè dei numeri in sostanza a questo punto il calcolo di questa espressione quindi è già interessante che ci sia una corrispondenza 11 fra le espressioni aritmetiche in notazione prefissa perlomeno in realtà poi in qualunque in qualunque notazione e gli alberi binari perché sono binari perché gli operandi che abbiamo preso in considerazione gli operatori che abbiamo preso in considerazione sono tutti e quattro operatori binari Dopodiché dicevamo in questo albero così costruito abbiamo la simpatica sorpresa di notare che se scorriamo l'espressione Nell'ordine in cui viene scritta da sinistra a destra sostanzialmente stiamo compiendo una ben precisa visita dell'albero stesso abbiamo il prodotto e siamo qua poi abbiamo la somma e scendiamo nel figlio di sinistra poi abbiamo il 5 e scendiamo nel figlio di sinistra a questo punto non ci sono più figli saliamo e scendiamo altre che il figlio di destra la parente 0:28:50 5 scendiamo nel figlio di sinistra a questo punto non ci sono più figli saliamo e scendiamo altre che il figlio

di destra la (ci dice che è terminata quest'espressione la, si dice di passare al figlio destro della espressione padre e a questo punto abbiamo la divisione di che cosa di 10 e di un ulteriore sotto albero che è costituito dal nodo difc con i due figli 4:02 quindi c'è una netta direi somiglianza fra la espressione Letta Da sinistra a destra e quella che sarebbe una visita dell'albero fatta come fate in profondità Scusate fatta in preordine Perché preordine Perché Prima leggiamo il prodotto poi leggiamo i due figli e il figlio di sinistre somma dei due figli 53 a somma di 5 di 3 segue il figlio di destra della radice cioè divisione con i suoi due figli 10 e la differenza di 4 e 2 e la differenza si stampa scrivendo prima la differenza prima l'operatore e poi due operandi Quindi in effetti c'è una chiarissima corrispondenza tra espressioni alberi e in particolare fra la forma scritta dell'espressione e la visita in preordine dell'albero stesso Cosa vuol dire vuol dire che stiamo facendo una visita in preordine no vuol dire che se avessimo l'albero che volessimo stamparlo in questo formato faremo una visita in preordine ma noi non abbiamo l'albero abbiamo le espressioni quello che invece dovremmo fare se scegliamo di modellare le espressioni con un albero vede Chiaramente la scelta che faremo sarà costruire in qualche modo questo albero volta che abbiamo costruito Noi dovremmo torniamo indietro Noi dovremmo valutare l'espressione come si fa a valutare questa espressione e Qui l'idea è che per valutare un'espressione se è semplice perché è un numero e quindi il valore del numero stesso è il valore dell'espressione Ma se invece si tratta di un'espressione composta bisogna intanto capire quanto valgono i due operandi e poi applicare l'operatore corrispondente al nodo stesso per esempio se stiamo considerando l'espressione differenza di 4 e due prima valuteremo 4:02 e poi applicheremo la differenza a 3:58 questo in un qualche modo corrispondeva ancora una visita nella quale però prima bisognerà ragionare manipolare operare sui figli e poi sul padre e questo è tipico della visita in post ordine vedete che prima di poter calcolare l'intero espressione che è questo prodotto Dobbiamo sapere quanto vale questa somma famosa e Quanto vale questa divisione e la somma è la somma di 53 quindi alle 8 mentre per sapere quanto vale la divisione dobbiamo prima calcolare la differenza di 4 Ok due poi faremo la divisione di 10 meno 2 meno 10 / 25 e poi finalmente potremmo fare il prodotto di 8 per 540 a nessuno sarà sfuggito che questo corrisponde alle parentesi tonde o quadre che ho aggiunto tanto per avvicinarmi alla votazione tipica delle scuole Scuole medie e per poter in questo modo arrivare a calcolare il tutto applicando le classiche regole che si imparano a scuola il secondo punto dell'articolo del esercizio consiste esattamente nello scrivere questa espressione che è quella standard che è nota come in notazione infissa perché in fissa perché l'operatore viene messo in mezzo fra i due operandi più viene messo fra il 5 e il 3 il pelo viene messo fra la somma di 53 e il rapporto di 10 e la differenza di 4 e due eccetera eccetera Come si fa a ottenere questa espressione Beh non dovrebbe sfuggire il fatto che se riusciamo a stampare il tutto In notazione polacca con una visita in preordine probabilmente riusciremo a stampare tutto nella notazione classica anche lì con una visita di qualche genere ma ne parliamo poi dopo e analogamente la profondità che definiremmo come la profondità dell' albero sarà strettamente legata alla struttura dell'albero Qui tutto questo ci fa chiaramente supporre che il nostro modello del problema che abbiamo un problema pratico dovendo pensare a un ipotetica relazione Abbiamo un problema pratico che dovrà essere descritto per capire che cosa ci ci vuole poi abbiamo una fase di modello Ecco la fase di modello consiste proprio nel dire che siccome le espressioni aritmetiche una struttura gerarchica viene spontaneo descriverle come alberi e siccome si tratta di espressioni binarie saranno alberi binari e siccome sono espressioni 0:33:40 Spontaneo descriverle come alberi e siccome si tratta di espressioni binarie saranno alberi binari e siccome sono espressioni non simmetriche saranno alberi ordinati non si mette Che vuol dire che se io faccio quattro differenza di 4 ore 2 faccio 3:58 Non faccio 2 meno 4 Anche se alcuni dei degli operatori invece hanno la proprietà commutativa quindi non è né interamente simmetrica né intera-

mente asimmetrica Che significa che è asimmetrica conclusione ci serve un albero binario ordinato che è proprio quello che abbiamo fra le mani Cosa vogliamo fare un programma che legga una per una queste righe e in uscita cosa faccia primo stampi il valore dell'espressione secondo stampi l'espressione in notazione infissa nella classica forma e terzo determina la profondità dell'albero Questi sono i tre nostri compiti ok Quindi possiamo cominciare a ragionare sulla idea di andare a prendere il nostro codice iniziale Calcola Zero che come al solito contiene un pochino di funzioni già preparate per risparmiare tempo e questa volta ho costruito già preparato l'interpretazione della linea di comando che consiste semplicemente nel leggere il file il nome del file delle espressioni e poi avremo invece una funzione che la poteva le espressioni per inciso Uno potrebbe anche dire Ma perché non facciamo un qualche cosa che apre questo file di scarica dati che apre questo file di espressioni e mi mette queste che sono tutto sommato delle stringhe in un vettore di stringhe in maniera che poi io possa lavorare su un vettore è il file che ho aperto lo posso mediatamente chiudere questo sarebbe probabilmente anche meglio però siccome l'esercizio è già abbastanza sofisticato Ho preferito andare più semplicemente non è poi del tutto sbagliato L'unico problema è che si apre un file che rimane aperto durante tutta l'esecuzione cosa che non sarà poi molto pesante in realtà Allora a questo punto cosa dobbiamo fare Dobbiamo andare a riempire la funzione elabora espressioni e ho già fatto qualcosa anche da questo punto di vista ho aperto il file è in lettura e l'ho chiuso e in mezzo ho costruito un ciclo che ci permette di scorrere una dopo l'altra le espressioni stesse allora domanda Che cosa dobbiamo spettivamente fare per leggere le espressioni una alla volta domanda risposta finora per legge una riga di un file abbiamo sempre usato la funzione F Get S che può andare benissimo Ha il noto problema che carica anche l'a capo finale del quale forse dovremmo disfarsi forse no non è chiaro sostanzialmente lo scopriremo in seguito implementando il resto qui tanto per mostrare un'alternativa possibile ho scelto un'altra modalità di caricare una riga alla volta che ha il vantaggio se vantaggio è che mi ha detto che lo sia di non caricare la capo In cosa consiste consiste nella classica funzione AF Scan f nella quale aprendo il file AFP si applica una stringa di formato piuttosto complessa e poi si va a scrivere ciò che è stato eletto nel solito metto vedi caratteri espressioni che ha una lunghezza sufficiente il testo che dice che le espressioni sono al massimo di 256 caratteri Quindi più o meno ci si può giocare sul fatto che ci debba essere lo spazio per il terminatore per la capo oppure no Quindi Però diciamo che presumibilmente li dovremmo starci quello che mi interessa adesso e discutere la stringa di formato Cioè è insolito per 100 che preannuncia il fatto che dobbiamo interpretare qualcosa e andarlo a scrivere in una variabile Dopodiché invece di avere DSC qualcosa F abbiamo delle parentesi quadre Ecco le parentesi quadre dicono che tutto ciò che c'è dentro Individua i caratteri che ci piacciono o non ci piacciono nella stringa che stiamo cercando di individuare Cioè per 100's dice caricami tutte le sequenze di caratteri alfanumerici fermandoti al primo Blank al primo separatore c'è spazio tabulazione o questa espressione Qui invece dice caricami tutti i caratteri che trovi fermandoti quindi sono tutti accettabili tranne e questo accento circonflesso significa tranne la capo quindi se ho questa espressione di formato vuol dire che io parto all'inizio della riga trovo una p mi piace Sì perché non è una capo Euro 1 e una o una di una parentesi Tutto questo mi piace 0:38:30 Che io parto all'inizio della riga trovo una p mi piace Sì perché non è una capo Euro 1 e una o una di una parentesi Tutto questo mi piace fino a che ho caricato la (Adesso trovo una capo non mi piace siccome non mi piace mi fermo e tutto ciò che ho caricato fino a quel momento va in espressione Dopodiché leggo anche la capo quindi non mi trovo qui ma leggo la capo e prosegue la riga successiva questo probabilmente non è strettamente necessario per il semplice motivo che alla successiva integrazione il percento indica di saltare tutti i separatori quindi salterà La capo salterà gli spazi tutto quello che ci interessa comunque questa è un altro possibile modo di leggere

intere righe Qualcuno potrebbe chiedersi ma non potevo fare per 100's riporta Sì se siete sicuri che l'espressione è tutta compatta senza spazi Tab un azioni e a capo potete tranquillamente scrivere per 100s però mi sembrava un pochino più pericoloso meno generale E comunque mi premeva dimostrare anche questa possibilità a questo punto io direi potete tranquillamente procedere col primo passo che è molto semplice e consiste nel creare le scatole vuote corrispondenti a questi passi che adesso leggiamo definendo anche tutte le variabili e le inclusioni che occorrono quindi dovremmo caricare l'espressione corrente che è stata Letta qui la dobbiamo caricare da questa stringa in un albero e che cos'è un albero bella domanda cos'è un albero un albero è una struttura dati abbastanza sofisticata per la quale fortunatamente abbiamo a disposizione una libreria albero. h nella quale abbiamo esattamente la definizione che abbiamo visto sui lucidi con L'unica modifica che invece di avere un campo informazioni ne abbiamo due informazione sono l'operatore El operando E perché ci sono due campi informazione perché le nostre strutture possono avere al loro interno o un operatore o un numero e sono due cose diverse in sostanza allora per non saper né leggere né scrivere io mi preparo in un nodo ospitare Entrambe le cose Una delle due va sprecata e memoria in più ma a questo punto sono tranquillamente in grado di gestire entrambi i casi sono alternative molto sofisticate esiste un tipo Union in C A tutto sommato poi se aveste della un meccanismo per gestire una cosa di cui non si sa esattamente il tipo dovreste comunque in qualche modo avere un test che vi dica ma questo nodo che ho in mano è un operatore o è un operando cioè un valore numerico perderesti in tempo quello che guadagnata in Fazio e non lo facciamo è un dettaglio di programmazione va benissimo questo piccolo spreco che comunque costante per ciascun nodo Ok detto ciò che cos'è un operatore montatore ha un carattere perché è un carattere e perché semplicemente Siccome poi dovremmo stampare le espressioni in questo modo tanto vale avere il più al posto della somma Il l'asterisco o se preferite l'aics però ho tenuto la convenzione C al posto del prodotto eccetera eccetera quindi gli operatori sono caratteri mentre gli operandi quelli semplici sono dei Double sono dei numeri reali a doppia precisione tanto per essere pignoli perfetto per la Questa è la nostra struttura L'albero è incrementato a puntatori quindi un albero è un puntatore a un nodo e anche una posizione un puntatore a un nodo Quindi questi tre campi puntano al nodo padre alla radice del figlio di sinistra del sottoalbero di sinistra era felice sotto l'albero di destra e che cos'è un nodo 12 a quel rottura con tutti i campi relativi c'è alto ci sono le classiche funzioni che abbiamo introdotto nella parte di teoria queste funzioni nel file albero Zero punto ci sono dichiarate e definite come corpi vuoti Ma poi dobbiamo andare a riempire esattamente come abbia fatto in tanti altri esercizi Quindi vai adesso potete mettere in pausa riempire tutto questo sapendo che esiste una libreria e per poter usare questa libreria dovete includerla ho detto c'è il caricamento dell'espressione poi dovremmo Calcolare il valore dell' albero dell'espressione che è rappresentata dall'albero poi dobbiamo stampare la espressione stessa in notazione infissa poi dovremmo calcolare la profondità dell'albero e infine potremmo distruggerlo visto che è qua dentro nel caricamento l'abbiamo creato 0:43:25 Lento nel caricamento l'abbiamo creato Allora prima fase dobbiamo caricare l'espressione corrente dalla stringa espressione a un albero abbiamo bisogno di un albero che cos'è un albero come Intanto come si chiama un albero il tipo albero Eccolo qua è già definito in questa libreria quindi avremmo bisogno di un albero che chiamerò tutti e per poter usare il tipo albero dovremmo includere oltre le classiche librerie di sistema anche questa libreria scritta da noi albero. h A questo punto noi siamo in grado di assegnare all'albero ti io risultato di una funzione che potrei chiamare converte espressione la quale è la Quale funzione prende in ingresso una espressione sotto forma di stringa e la converte in un albero ci serve altro No il file è già stato aperto abbiamo già caricato l'espressione non c'è niente di male a questo punto possiamo solito meccanismo dichiarazione chiamata dichiarazione definizione possiamo

andare a Calcola caricare l'espressione rappresentata dalla stringa espressione nell'albero di quindi il risultato di questa funzione è un albero e la espressione stessa la funzione ha come dato una puntatore a carattere possiamo scendere e definire il corpo vuoto di questa funzione stesso giochino lo facciamo dopo Adesso dobbiamo Calcolare il valore di questa espressione probabilmente dovremmo anche stamparlo cominciamo a calcolare il valore Allora il valore che cosa sarà sarà un Double e potrebbe chiamarsi per esempio vi Che cos'è vi è il risultato del valore dell'espressione rappresentata dall'albero Ti una volta che ho vi possiamo tranquillamente andare a stampare a video in qualche modo in qualche formato Io sto seguendo un formato che mi sono inventato io e questo valore sarà un numero reale Dopodiché andremo a capo è il numero reale Sara Affi nuova funzione che abbiamo chiamato possiamo andare su a dichiararne il prototipo questo è una funzione che ha risultato Double e che prende come dato un albero e calcola il valore delle espressioni rappresentata dall'albero Ti possiamo scendere e avere una seconda funzione vuota poi dovremmo stampare che cosa l'espressione rappresentata dall'albero in notazione infissa E potremmo anche scrivere stampa espressione ti preferite specificare stampa espressioni in fissa stesso discorso e basta La Stampa fa già una stampa però Io aggiungerò anche una capo giusto per separare una dall'altra tutti questi risultati semplicemente perché non vorrei dovermi ricordare alla fine della stampa di andare lo scrivo qua c'è altro da dire andiamo su a dichiarare il la funzione che abbiamo aggiunto la funzione Stamp espressione non ha risultato quindi una Void e ha come dato un albero e queste due righe si possono finalmente aggiungere al nostro codice Ci siamo quasi dobbiamo calcolare la profondità dell'albero la profondità è ovviamente un numero intero è il valore di questo numero è il risultato di una funzione applicata ad un albero ancora una volta dovremmo stampare a video il risultato qui potremmo avere profondità qua ho cercato di tenere le stampe anche belle in linea tanto perché sia divertente e gradevole tutto il core Diamoci che è un numero intero la profondità e quindi cambia la stringa di formato per 100 di a questo punto abbiamo una ulteriore chiamata quindi un'ulteriore dichiarazione calcola la profondità 0:48:13 Una ulteriore chiamata quindi un'ulteriore dichiarazione calcola la profondità dell'albero ti che restituisce un intero rendendo in ingresso un albero e possiamo costruire l'ultima funzione sempre con corpo vuoto converrà per avere la correttezza sintattica che sia Calcola valore sia profondità avesti tui Scanu dei valori ragionevoli diciamo Allora per la profondità se io ho un albero non so che albero è potrei dire che è vuoto e dire che la profondità e zero tanto questi valori li cancelleremo alla fine ma è per avere la correttezza sintattica totale e per il valore Allora per il valore in effetti vi faccio salvare che in questa libreria esistono anche delle Costantino hop e nuova al A cosa servono al fatto che siccome il nodo contiene due campi ma uno solo dei due usato nei nodi interni ho usato il nodo operatore e è un valore delle foglie usato il nodo operando e non c'è un operatore in ciascuno dei due l'altro termine per pulizia è occupato da uno di questi oggetti e allora tanto vale sfruttare Non vale E dire che un albero che non sa non sa niente al suo interno o che non è neanche un albero ha valore nuova al Zero convenzionale potete cambiarlo assolutamente a piacere Va bene allora direi che tutto sommato questo c'è anche qua sopra un'altra cosa qui è una espressione una funzione che restituisce un albero Cosa succede se un altro in realtà non esiste per esempio che è questo non essendo un albero un puntatore può valere la pena di far restituire un puntatore nulla questa funzione Dopodiché ci rimane una sola funzione che è una volta e quindi siamo questa volta a posto allora essendo a posto dovremmo essere in grado di compilare con i soliti con i soliti opzioni attente la funzione i file Calcola Zero dobbiamo includere albero zero perché stiamo includendo delle funzioni dei tipi che sono in quella libreria e restituire in uscita Calcola punto.exe Vediamo se tutto funziona in teoria funziona L'unica cosa che fa questo programma e segnalare che il nome del file va fornito Dopodiché non succede nulla nel senso che ci sono

queste quattro stampe che corrispondono in sostanza nel main alla ripetizione per 4 volte di una valore Nullo perché calcola valore restituisce innoval una stampa che è un semplice a capo una profondità nulla perché abbiamo detto che la profondità è zero ci siamo dimenticati di andare a distruggere l'albero con questa funzione già che ci siamo la chiamiamo non cambia nulla perché per il momento la funzione è vuota Però giustamente abbiamo detto di completare la prima fase è come vedete L'albero è passato per indirizzo perché viene fisicamente distrutto a questo punto Siamo in effetti a posto Se volessimo fare un ulteriore incontro prova potremmo farci ristampare anche l'espressione ci faremo stampare una stringa espressione e poi andremo questo ci garantisce sul fatto che quantomeno la lettura è andata a buon fine e Infatti ciascuno dei quattro blocchi è preceduto dalla espressione che abbiamo letto quindi questa è un operazione di debugging è andata a buon fine la fase successiva dell'esercizio vabbè potrebbe consistere nel andare un po' più a fondo top Town avrete sicuramente notato che in tutte le versioni precedenti in cui abbiamo maneggiato una libreria Mi sono quasi subito precipitato a riempire queste funzioni vuote e lo scopo è duplice uno di avere il più possibile subito dei risultati per poter ragionare sul fatto che il codice sia corretto due il fatto che siamo freschi dalla prima parte della lezione dove abbiamo introdotto tutte queste funzioni e loro significato quindi insistevo in questo modo Fermo restando che Uno potrebbe anche andare invece avanti rigorosamente in modalità top-down Allora dobbiamo riempire queste funzioni come facciamo E sostanzialmente Dobbiamo ricordarci di come è fatto è che cosa consiste un albero binario qui è un disegnino che ci dà l'idea di Come è fatto un albero binario allora come si fa a creare un albero creare una 0:53:06 Un albero binario allora come si fa a creare un albero creare un albero vuoto Ovviamente è un albero vuoto è un albero che non ha nodi Quindi in effetti non ha nemmeno questa radice dobbiamo tornare indietro nell'espressione e osservare che siccome un albero è un puntatore a nodo becco se l'albero è vuoto la cosa più ragionevole che non ci sia un modo Quindi il puntatore sia a nulla Questo significa che invece che invece che continue che effettivamente effettivamente questo valore iniziale che era stato dato tanto per è corretto cioè creare un albero vuoto significa restituire i nostri restituire nulla per la distruzione rimandiamo un attimo perché abbiamo anche già osservato nella parte teorica che distrugge un albero o cancellare completamente un sottoalbero sono compiti molto simili si tratta semplicemente di Sotto l'albero a partire dalla radice che è l'intero albero quindi proseguiamo Come facciamo a sapere che un albero è vuoto questo non è sempre vero Quindi questa risposta di default è sbagliata Come si fa a sapere se un albero vuoto semplice di puntatore ti è esattamente uguale a nulla o per essere più corretti a nutri in questo caso L'albero è vuoto Altrimenti se si punta un nodo l'albero non è vuoto Dopodiché dobbiamo andare a leggere un nodo Come si fa a leggere un nodo Torniamo qua leggere un nodo comporta di conoscere l'albero di conoscere la posizione di quel nodo che sarà un puntatore hanno lo stesso dopodiché noi abbiamo un puntatore che ci dà un intero blocco tettino nodo dobbiamo entrare nel campo giusto è una situazione molto simile a quella delle liste in cui se si voleva leggere un elemento si prendeva il puntatore a quell'elemento si entrava nel campo specifico e si aveva tutto quello che era necessario quindi in effetti la posizione P è in effetti un puntatore è l'oggetto puntato è il nodo e ha al suo interno e campi e questi campi Come si chiamano questi campi Eccoli qua ti chiamano Coop Val e poi ci sono questi campi che per il momento non ci interessano Io voglio leggere lo prendo o l'operatore PAL o app voi a leggerli tutti e due perché non abbiamo distinto due funzioni diverse di conseguenza si potrebbe farlo Ovviamente ho voluto rimanere nel caso più generale di conseguenza quello che dovremmo fare Sarà andare a leggere il campo hop e il campo Val Esatto uno e l'altro e assegnare ciascuno dei due alla variabile all'argomento passato dall'esterno passato per indirizzo quindi dobbiamo assegnarlo all'oggetto puntato classica operazione di passaggio per l'indirizzo di un tatuaggio

finto che in realtà è il risultato della funzione Si può osservare come nelle liste che non compare T in questa espressione e la domanda è perché perché ci abbiamo dei puntatori Non è necessario conoscere l'intero albero ci basta indirizzo del nodo che stiamo trattando E allora Perché teniamo tutti per lo stesso motivo per cui l'abbiamo tenuto nelle liste perché se vogliamo una implementazione indipendente ti nasconda la struttura effettiva e fornisca la spesa Istituto parati astratta Allora dobbiamo pensare che l'albero potrebbe anche essere un vettore con degli indici la posizione sarebbe un numero intero e se qua dentro ne abbiamo un numero intero e basta Non abbiamo modo di sapere come è l'operatore come l'opera andò avremmo bisogno di conoscere il vettore che fa da albero oltre al numero intero di conseguenza ci servono entrambi Ci serve ti e ci serve più allora per coerenza lo teniamo anche nel caso della implementazione attentatori Come si fa a scrivere estremamente analogo Cioè dobbiamo andare a scrivere in più app e a scrivere in peval che cosa i valori passati da fuori cioè dobbiamo andarci a scrivere hop e Val rispettivamente quindi ancora più semplice nel caso della lista dei libri erano delle sto incopy altro vogliamo a questo punto la posizione va dice se qualcuno mi dà un albero Io voglio un puntatore al nodo la Dice esattamente la stessa cosa Se io punto un albero o un puntatore a questo blocco in questo blocco e la radice e quindi in realtà questa funzione consiste banalissima mente Nel restituire ti si può chiedere che senso abbia avevano funziona in cui si prende un oggetto e si pulisce lo stesso Oggetto la risposta è la solita tutto velati astratte Qualcuno potrebbe essere tale che ti è un albero ma l'uscita di radice 0:58:00 Tutto velati astratte Qualcuno potrebbe osservare che ti ha un albero ma l'uscita di radice dovrebbe essere una posizione Allora è corretto scrivere qualcosa del genere c'è un cast che prende ti e lo converte in una posizione Ma è anche vero che sono entrambi poi puntatori a nodo e quindi la conversione consiste nel non fare niente per cui direi che siccome questa è una libreria e sta a bassissimo livello Non ci preoccupiamo neanche di scrivere cose del genere Cosa si fa per il figlio sinistro il figlio destro molto semplice cos'è il figlio sinistro il figlio adesso sono questi due puntatori qualcuno mi ha dato una posizione P Che punta a un blocco E io voglio andare a prendere il campo TS il campo di di di questo blocco quindi sostanzialmente il figlio sinistro non è altro che il contenuto del campo TS del nodo puntato d'api e figlio destro è il contenuto del ancora una volta stiamo passando ti anche se non serve se avessimo un vettore con degli numeri interi ci servirebbe è come il padre il padre visto che l'abbiamo chiamato qui e Ammetto che sto facendo un po' di confusione Scusate non l'abbiamo chiamato P abbiamo chiamato padre quello che un pochino fa confusione è che in questo disegno l'ho chiamato P in realtà questo Chiama padre e però il campo padre del nodo puntato dalla posizione p e ciò che va restituito dalla funzione padre Bon Arriviamo alle due funzioni corpose e interessanti finalmente Come si fa a costruire un albero torniamo alla nostra definizione astratta gli ho detto che per costruire un albero Lunedì chiediamo un campo in formazione operatore e ho valore uno dei due tele adesso li prendiamo tutti e due uno dei due sarà inutile l'altro sarà utile Poi abbiamo bisogno di due alberi che saranno il sottoalbero sinistro il sottoalbero destro Eccoli qua il risultato sarà un albero complessivo albero ottenuto come costruendo un nuovo nodo in cui inseriremo linformazione.eu e appendendo questo nodo a sinistra il primo il secondo tutto sommato non particolarmente complicato Che cos'è che dobbiamo di conseguenza fare sicuramente dobbiamo allocare la memoria per un nodo 15 servirà un nodo che potrei chiamare R tanto perché poi diventerà una radice era questo nodo attribuiamo della memoria allocata Quanta memoria allora tanta quanta ne serve per un nodo sai già un nodo prima domanda perché usato malloch tante volte quasi sempre usato culloch nel corso perché culloch è particolarmente adatta per i vettori dato che qui si scrive il numero di elementi e poi si scrive quanto è grosso ogni elemento per usare culloch io qui dove scrivere culloch 1,3 è un po' buffo È uno solo non solo a Carlo che ha il vantaggio in generale che riempie di Zeri il l'area

locata abbiamo anche visto che non è sempre un vantaggio e la scorsa lezione Mi evito un punto in cui gli ho detto qui forse converrebbe fare ma l'ok perché culloch appesantisce la complessità della funzione è necessario qui azzerare tutti i campi mi ha risposto È che in realtà quello che adesso ci accingiamo a fare e a prendere uno per uno questi campi e riempirli noi a mano per cui azzerarli per poi dover riempire non è una cosa molto furba allora Maalox è meglio fatto questo possiamo andare dobbiamo andare a testare se per caso la locazione è andata a buon fine e in caso contrario andare a stampare il consueto messaggio di errore nell'allocazione di un nodo nodo in cui volendo potremmo anche andare a scrivere Che cosa le due informazioni operatore e valore abbiamo detto che l'operatore è un carattere mentre il valore è un numero reale Quindi cui possiamo passare hop e Val se proprio vogliamo essere pignoli Allora a questo punto quello che possiamo effettivamente fare è riempire i dati e quindi andare a dire che R Freeshop 1:02:50 Freeshop Che cosa contiene l'operatore che viene passato da fuori che R freccia Ball contiene il valore passato da fuori che a questo punto ci sono da riempire il padre di Sotto l'albero di sinistra e sotto albero di destra domanda se io ho un nodo radice questo Chi è il padre non c'è è la T rovesciata il nulla quindi il padre di R sarà in sostanza non Hood perché ho usato NOD perché è convenzionalmente il la maschera che null mette per dire posizione assurda Ok c'è altro che apparentemente quello che dovremmo fare assegnare TS al figlio sinistro etd al figlio destro e saremmo contenti così in realtà non è giusto Qui c'è un piccolo dettaglio ma importante e dettaglio il seguente nel momento in cui io decido che ho questo albero di quest'altro Aldo il 5 e il 2 con appesi il 4 e 3 e gli aggiungo il nuovo nodo 2 è vero che due deve puntare come padre ha niente e con i due figli al 5 e all'altro due ma è anche vero che il 5 e l'altro due Hanno smesso di essere delle radici e quindi non è più vero che hanno come padre nulla devono avere il nodo che li abbiamo appena passato quindi Qui c'è un possibilità di errore dovuto al fatto che tstd devono avere padre che non è più quello che hanno in questo momento in particolare Ti esce padre deve diventare la radice e anche te li padre deve diventare lavatrice è tutto fatto basta Questo no perché perché nessuno ha detto che TS abbia un campo padre cioè tsa un campo padre se è un albero dotato di nodi ma potrebbe essere un albero vuoto e in tal caso non esiste TS padre per il semplice motivo che TS null e quindi la freccia non mi porta da nessuna parte quindi questa è un'operazione che è corretta ma è corretta solo e unicamente se ti esce padre è diverso da un albero vuoto mentre quella sotto è corretta solo e unicamente se ti di padre è diverso da un altro vuoto e adesso effettivamente ci siamo passiamo alla cancellazione Come si fa a cancellare un sottoalbero mi faccio notare che stiamo passando l'albero Ma stai passando un puntatore alla posizione perché un puntatore alla posizione perché se io voglio cancellare questo nodo con tutto ciò che gli ho appeso e ho visto è passando un p che tutto questo venga dell'avvocato e io poi abbia in giro nella funzione chiamante 1 puntatore P che continua a puntare quest'aria di memoria che però è già locata allora è bene che questo puntatore diventi Nullo in maniera tale che poi non ha non ci siano più pasticci ed è per questo che lo passiamo per indirizzo ed è lo stesso motivo per cui passavamo per indirizzo la posizione nella cancellazione di elementi da una lista ne abbiamo discusso Allora come si fa a cancellare un intero sotto albero Il problema è che non si tratta di cancellare un nodo Ma si tratta anche di cancellare tutto ciò che sta sotto e questo sembra un compito immane ma il qui entra in gioco la potenza della ricorsione Supponiamo che io sia capace di cancellare questo sotto albero è che sia capace di cancellare questo sotto Albero come potrei fare chiamando la funzione Cancella sotto albero che non ho ancora scritto ma che do per scontato di essere capace di scrivere a questo punto diventa facile scrivere la funzione Cancella sotto albero se ipotizzo di saperla scrivere per i due sotto Albe perché si tratta semplicemente di del locale questo nodo è poco d'altro Quindi in sostanza che cos'è che mi tocca fare è quello che mi tocca fare è cancellare

sotto l'albero ho finito cancellare sottoalbero destro poi chiamare una procedura di liberazione del nodo 1 Free e poi siccome questo. lo voglio nulla metterlo a nulla quattro operazioni in croce con un'ulteriore considerazione che bisogna sempre Ricordarsi che quando si affronta una operazione su albero in particolare quando eri così va dobbiamo sempre ricordarci che c'è il caso di albero vuoto l'abbia già visto prima pensiamoci meglio ad 1:07:40 Quando eri così va dobbiamo sempre ricordarci che c'è il caso di albero vuoto l'abbiamo già visto prima pensiamoci meglio adesso Se un albero è vuoto Come si fa a cancellarlo risposta non si fa niente però è importante tenerne conto Allora dobbiamo dire che se ti non è un albero vuoto è soltanto Allora Scusate no qui sto ho leggermente sbagliato stiamo cancellando il sottoalbero puntato dalla posizione P quindi non sto chiedendomi se P è un albero vuoto mi sto chiedendo se l'oggetto puntato d'api è un albero vuoto oppure no e siccome P non è rigorosamente Un albero Ma è un puntatore un nodo c'è una posizione mi sto chiedendo se l'oggetto puntato d'api è o non è è una posizione ragionevole oppure no Se è una posizione ragionevole Ecco che finalmente possiamo fare i nostri quattro passaggi che abbiamo detto essere primo la chiamata ricorsiva sul sottoalbero di sinistra che è di solito albero di prima ma la posizione che posizione è è la posizione del nodo * pipì campo Ps Questo qui è un nodo Questo è un nodo all'interno di questa notte c'è il campo TS Questo è il sottoalbero sinistro la posizione del sottoalbero sinistro la posizione della radice del sottoalbero sinistro e noi dobbiamo passare l'indirizzo ti faccio notare anche le parentesi tonde dovute al fatto che altrimenti tutte le precedenze rischiano di andare a Patrasso perché la freccia viene applicata pipì prevalendo sulla * che invece deve essere eseguito per primo Ok stesso giochino sul sottoalbero destro per continuare con la cancellazione del nodo puntato da Pepi e infine per assegnare api il valore null o se preferiamo non UD vi faccio sembrare che questa cosa somiglia molto a una visita dell'albero del sottoalbero il che ovvio se io voglio cancellare un sottoalbero dovrò correrlo tutto in che maniera lo scorso abbiamo detto che prima cancello il sottoalbero sinistro poi cancello il sottoalbero destro e poi cancello il nodo radice e questa è una visita Imposta ordine che infatti corrisponde ricorsivamente a due chiamate ricorsive sul sottoalbero sinistro e destro seguite dalle operazioni sul nodo dato a questo punto cosa ci manca la funzione distrugge albero che avevamo lasciato ancora aperta e vi avevo preannunciato che quella funzione In sostanza è un'applicazione speciale della Cancella sotto albero c'è una cosa da dire ancora giusto la funzione Cancella sotto albero e la funzione costruisce albero Questo è un piccolo e vuole che stavo commettendo restituiscono entrambe un albero è vero che l'albero viene già modificato e quindi non c'è bisogno di passarlo anche in uscita però questa cosa è stata fatta per adattarsi il più possibile alla definizione astratta di operazioni su alberi che hanno degli Tati Ma hanno anche un risultato esattamente come abbiamo fatto nelle liste per cui bisogna però i risultati siano corretti Allora quando io Costruisco un albero prendendo le due Aggiungendo di una radice che cos'è che restituisco come albero istituisco la radice Mentre qua sotto Quando io cancello interamente un sotto albero dell'albero ti Che cos'è che ottengo in uscita ottengo il sottoalbero ti privato del suo sotto albero che è stato cancellato e su questo è sistemato possiamo tornare sopra e andare a distruggere l'albero domanda Come si fa a distruggere un albero si cancella a partire dalla radice quindi si cancella l'albero stesso che è l'albero da PT visto che Pitti è l'indirizzo dell' albero e a questo punto abbiamo bisogno di conoscere la posizione del nodo da cancellare Ma questo modo che cos'è la radice e la radicetti è l'indirizzo di ti Che cos'è indirizzo di tripiti per cui molto banalmente se vogliamo cancellare distruggere un albero quello che possiamo fare è lanciare cancia sotto albero sull'albero stesso a partire dalla posizione che punta all'albero stesso un poco 1:12:30 Sotto albero sull'albero stesso a partire dalla posizione che punta all'albero stesso un po' contorto ma tutto sommato dovrebbe essere tutto assolutamente chiaro sulla base di come abbiamo definito le nostre cose

e consentirci qua dentro nel nostro programma di non pensare più al mondo degli alberi e dei puntatori ma soltanto usare le funzioni che ci consentono di lavorarci Ok Proviamo a vedere se tutto questo ha avuto un esito corretto quantomeno sintatticamente che tutto funzioni non cambia assolutamente nulla perché le nostre funzioni non fanno praticamente nulla non fanno nulla in realtà adesso possiamo andare a riempire le 1-2-3-4 visto che distrugge albero è già pronta e Note 4 funzioni partiremo ovviamente da quella che converte l'espressione stringa in un albero Perché le altre dipendono dal fatto di avere un albero ok Come si fa a convertire l'espressione in un albero torniamo a vedere queste espressioni Come sono fatte le espressioni vediamo di metterle Ok di fronte alla struttura questa espressioni sono fatto in questo modo abbiamo una prima parte che è l'operatore poi abbiamo una parentesi (poi abbiamo il primo operando una virgola il secondo operando è una parentesi) la struttura è molto rigorosa ed è ricorsiva perché il primo operando a sua volta è costituito da un operatore una (un operando una, un altro operando una parentesi) e lo stesso succede in tutti i casi Tranne per le espressioni semplici Allora quando dobbiamo convertire una stringa che rappresenta un'espressione ed è un'espressione semplice intanto non è difficile riconoscerlo perché si apre con un numero reale Invece di aprirsi Con prod somma di odif e poi è molto facile costruire l'albero relativo altrimenti la cosa diventa più complicata perché bisogna prendere la nostra stringhe dividerla in tre parti e poi però non è difficilissimo perché la prima parte ci consente di determinare la radice dell'albero e le altre due parti sono delle espressioni che ci consentono se le sappiamo trasformare in alberi di costruirlo sottoalbero sinistro è un sottoalbero destro è una volta che abbiamo la radice sotto l'albero sinistro sotto l'albero destro abbiamo la funzione che ci rimette insieme quindi Tutto sta nel dividere questa stringa in tre parti nel primo di conoscere se l'espressione semplice o composta e semplice risolverla secondo se invece l'espressione composta dividerla in tre parti costruire un nodo con la prima parte così un albero con la seconda ma questo si fa ricorsivamente perché è un'espressione e terzo costruire un albero con la terza parte ma questa è ancora la chiamata ricorsiva e quindi potete facilmente immaginare che anche questa funzione che sostanzialmente converte le espressioni avrà una tipica struttura ricorsiva e lavorerà su alberi tutte le funzioni che lavorano su alberi hanno Tendono ad avere una espressione ricorsiva è vero che tutte le espressioni di corsive si possono tutte le funzioni del Consiglio si possono trasformare iniziative è vero che conviene farlo perché risparmiamo tutta la meccanica delle chiamate e della locazione dei record azione sullo Stack Ma è anche vero che se la che questa cosa è facile da fare nel caso di ricorsione terminale E qui la ricorsione non può in genere essere terminale per il semplice motivo che le chiamate lavorano su entrambi sotto alberi e quindi uno dei due sottoalberi non è terminale non è chiamata terminale Quindi in effetti bisogna trasformare una di queste funzioni ricorsive in iterative bisogna fare una serie di operazioni sofisticate che richiedono una gestione esplicita di una pila e sono cose che sono al di là dei limiti diciamo di questo corso quindi non le facciamo in cui oggi vedremo solo espressioni puramente ricorsive Allora adesso prima di invitarvi a fare voi tu studiamo uno schema per la funzione converte espressione che cosa deve fare questa 1:17:22 Fare voi tu studiamo uno schema per la funzione converte espressione che cosa deve fare questa funzione quello che abbiamo più o meno ha detto prima parole una prima cosa che dobbiamo fare e riconoscere se l'espressione è semplice o composta Quindi se espressione è semplice che cosa dobbiamo fare saluta il numero reale e costruisce un albero pensiamo un attimo Scusate prima mettiamo la condizione se l'espressione è semplice dobbiamo fare un paio di cose che consistono nel valutare il numero reale e costruire un albero con un solo nodo che contiene il numero Altrimenti se invece è composta E cos'è che dovremmo fare se invece composta dovevo andare a spezzare espressione in tre sottostringhe relative operando operatore 1 operatore 2 Una volta fatto questo andremo a identificare lo provando andremo a costruire il sottoalbero

relativo all'operatore 1 poi costruiremo il sottoalbero relativo all'operatore 2 infine costruire l'albero componendo operando Scusate ho sbagliato scambiato operando l'operatore era operatore è la identifica l'operatore Dopodiché determina il sottoalbero relativo operando una lettera 22 e costruisce l'albero componendo l'operatore con i due grandi Questo è quanto tieni a questo punto dovreste essere in grado di mettere in piedi da soli la funzione evidentemente chiamando delle sotto funzioni che implementeremo in seguito per i compiti più complicati allora di che si tratta come si fa a sapere che un'espressione semplice si tratta semplicemente di dire Ma se io la leggessi e cercassi di trovare un numero reale in essa c'è se facessi un f-scan F di che cosa una s-scan F perché sto leggendo la stringa espressione è nella stringa espressioni cercassi di determinare un numero tabol in caso lo determinarsi lo mettessi da qualche parte quindi potrei andare anche a costruire un valore vale se questa cosa è successo cioè se effettivamente trovo un numero reale Cosa vuol dire vuol dire che effettivamente la mia stringa comincia con un numero reale Questo non con non funziona in questa Riga prod non funziona Naviga Dip neanche nella Liga somma ma funziona qua in 2.5 quindi diciamo è un modo corretto di andare identificare quello che sta succedendo in questo caso andremo a valutare il numero reale stesso ma l'abbiamo già fatto e quindi possiamo costruire un albero che contenga un solo nodo il quale nodo è fatto in quella maniera lì come si fa a costruire un albero che è costituito da un solo nodo quindi con due sotto alberi vuoti avente un valore abbiamo in mano la nostra funzione la nostra funzione si chiama costruisce albero che guarda caso serve Noto un operatore o un operando Noi abbiamo operando noti due alberi vuoti vanno benissimo costruisce un albero quindi possiamo andare generare un albero ti è questo albero tiri Che cos'è È semplicemente il risultato di costruisce albero 1:22:07 Vuoti vanno benissimo costruisce un albero quindi possiamo andare generare un albero ti è questo albero ti Che cos'è È semplicemente il risultato di costruisce albero applicato a quale operatore non c'è un operatore No A quale valore è il valore C'è schiuma Val A quale sottoalbero sinistro not lì non c'è nessun sottoalbero sinistro e a quale sottoalbero destro nessun sottoalbero destro banalissimo Altrimenti se tutto questo non ha avuto successo vuol dire che l'espressione è composta diamo per scontato che non si siano pasticci perché insomma il le stringhe in ingresso siano ben formate Allora noi dobbiamo aspettare la stringa espressione in tre sottostringhe e questo Evidentemente è un compito abbastanza sofisticato per cui direi che è il momento di piazzare una funzione che potrebbe essere la seguente Io voglio scomporre un espressione contenuta nella stringa di espressione e voglio Scomporre in tre parti queste tre parti sono tre stringhe quindi devo andare a costruirmi tre stringhe relative al operando l'operatore uno l'operatore due c'è anche male in qualche modo io potrei chiamare S ovviamente di lunghezza Wow length anche se probabilmente basta molto meno perché se veramente è un operando e sarà al massimo lungo 5 caratteri Somma più su più 1 per il terminatore ma non stiamo a risparmiare e poi abbiamo S Sperando uno e s operando due vogliamo quindi scomporre espressione in s.s. operando 1S operando tue Questo è quello che vogliamo fare ipotizziamo di farcela va bene per farlo dobbiamo avere in alto la nostra funzione dichiarata e poi sotto definita Quindi questa funzione scompone la stringa espressione in tre sottostringhe relative a operando un operatore 1 e un operatore 2 è una funzione Void Perché i tre risultati essendo String sono passati per indirizzo abbiamo un dato puntatore a carattere espressione con risultato puntatore a carattere S1 risultato puntatore carattere se uno è un altro sepoc due tutte queste cose vanno aggiunte qua sotto Come funzione con un corpo vuoto e poi la riempiremo in qualche modo ho fatto dovevo andare identificare l'operatore operatore cosa sarà sarà una streghina div somma prod o si potrebbe fare direttamente però già che ci siamo Diciamo che scomponiamo e quindi faremo ancora una volta un funzione che identifica l'operatore e siccome abbiamo stabilito che un operatore Che cos'è un operatore abbiamo scritto qua sopra è un carattere allora noi possiamo definire

una variabile di tipo carattere pop e questa variabile op va a prendere il risultato di una funzione che identifica operatore che legge la stringa S questo è ciò che va fatto sarà molto facile realizzarla andiamo a catturare e costruire la nostra dichiarazione è la nostra definizione Ovviamente questa funzione restituisce un operatore e ha come dato una stringa identifica l'operatore descritto dalla stringa S e siamo ancora a posto questo non identifica quindi andrà cambiato anche 1:26:50 E siamo ancora a posto questo non identifica quindi andrà cambiato anche nella dichiarazione e andrà cambiato nella chiamata dovunque essa sia qua siamo a posto dobbiamo costruire questi due sotto alberi come si fa a costruire sotto alberi relativi a un operando che è una stringa semplice abbiamo già la nostra funzione che si chiama converte espressione a Quale funzione ci crea un albero Allora noi abbiamo due alberi quindi come abbiamo l'albero ti abbiamo l'albero TS l'albero TD e Questi alberi li Costruiamo a partire dalla funzione converte espressione che non lavora sull'intera String ma lavora sulla stringa S op-1 Analogamente costruiremo con la stessa funzione applicata sop2 il sottoalbero di destra e infine costruiremo l'albero componendoli l'intero albero ti verrà costruito componendo Con quale operatore l'operatore Opi Con quale valore nessun valore quest'anno da interno quale albero di sinistra TS quale albero di destra ti dico abbiamo finito No dobbiamo ricordarci di restituire l'albero che abbiamo costruito altrimenti rimane tutto in secco siamo a posto in teoria dal punto di vista sintattico sì perché abbiamo definito tutte le nostre funzioni dichiarate definito Qui c'è un control reaches End of non-void function giustamente perché la funzione identifica operatore deve restituire un operatore per esempio il solito no Coop giusto per scrivergli qualcosa e quindi sistemiamo anche questo avvertimento Ovviamente per adesso non stiamo facendo nulla vediamo se riusciamo a fare Finalmente qualche cosa per fare Finalmente qualche cosa è abbastanza chiaro che si potremmo riempire la stampa espressione ma l'espressione non abbiamo ancora costituita Quindi tutte queste funzioni che stanno sotto non è il caso di considerare le quelle che contano e che finalmente ci daranno il risultato sono la composizione dell'espressione e l'identificazione dell'operatore adesso io Vi suggerisco di ragionarci un attimo l'identificazione è molto semplice è una sequenza di confronti che dice ma la stringa S per caso è prod è somma è di Edif e se è una di queste restituisco in uscita il carattere corrispondente quindi molto semplice compone pressione decisamente più complicata perché vuol dire Prendi questa stringa E dividila in tre parti in cui la prima è un operatore La seconda è ciò che viene prima della, e la terza e ciò che viene dopo la virgola però Qui cominciamo ad avere il punto chiave in realtà che cos'è se la stringa è stata scritta bene Che cos'è il l'opera operatore operatore è ciò che viene prima della (qui mi sto mettendo in un piccolo ginepraio Nel senso che se uno mi scrive una cosa del genere Eh allora io leggerò prod spazio e però ti faccio non è un operatore quindi sto per dando per scontato che in realtà non ci siano spazi balordi e cose varie ma tutto questo si possono fare altre varianti che per allungherebbero l'esercizio inutilmente Allora davo per scontato che l'operatore sia tutto ciò che sta prima della (Non è difficile scrivere una funzione che trova questa questo pezzo questa sottostringa Dopodiché saltiamo la (Che cos'è l' operando El operando è tutto ciò che va fino alla, Ma attenzione se io dico fino alla, rischio di fermarmi alla prima, quindi evito di prendere somma (5 e questo non va bene perché non va bene la quella, perché bisogna prendere la seconda perché questa, è all'interno di una (quindi devo Sì cercare di , ma devo cercare di raggiungere la, contando le parentesi che sono state aperte e quando le parentesi sono bilanciate da parentesi chiuse Allora la prima, segna la fine della espressione che sto cercando Finita l'espressione salto la, e posso chiedermi come è costituita la terza parte è costituita da tutto ciò che va fino a parentesi chiusa ma non la prima parentesi chiusa e neanche la seconda la terza Perché Perché qui c'è una 1:31:40 IVA fino alla parentesi chiusa ma non la prima parentesi chiusa neanche la seconda la terza Perché Perché qui c'è una prima parentesi una seconda la seconda viene

chiusa la prima viene chiusa e finalmente ho la parentesi che mi interrompe l'espressione Quindi in realtà questi tre aspetti che sembrano diversi sembrano tre compiti diversi e Uno potrebbe affrontare come tra i compiti diversi in realtà sono lo stesso compito che compito è il compito è Prendimi tutto ciò che va fino a un carattere di fine che ti dico io è la prima volta una (la seconda volta è una virgola e la terza volta una parentesi chiusa è un carattere che ti dico io ma questo carattere me lo devi guardare solo se il conteggio di parentesi aperte chiuse è azzerato detto così non è poi così complicato potremmo vedere ti suggerisco di farlo con una funzione che prende la mia il mio vettore è trova il carattere un dato carattere purché non annidato fra parentesi e qui dico provateci quantomeno fare le identifica operatore Allora partiamo con l'identificazione l'operatore visto che è semplice e l'identificazione dell'operatore In sostanza è una cascata di confronti confronti di String in cui si prende la string.se si dice Ma per caso la stringa S è una somma e se il con perda risultato Nullo vuol dire che ho letto una somma Ma se avete una somma il carattere che devo restituire all'esterno è un più Help se non lo è lei non serve Evidentemente perché avete mi fa uscire però preferisco dare l'idea che sia una cascata in questo caso andremo a vedere se per caso la stringa S è una differenza è in tal caso restituiremo un meno e poi faremo delle cose simili per gli altri due operatori Cioè se per caso la stringa è un prodotto restituiremo un asterisco e se per caso è un div restituiremo un altro non dovrebbe succedere assolutamente altro Se procediamo in questo modo No va bene così direi lasciamo quest'ultima possibilità Giusto per non farci dire dal compilatore che c'è un bel superiore nel quale la funzione non vestito esce nulla evidentemente Questa sarebbe una situazione di errore ma noi Supponiamo che non si verifichi mai detto ciò passiamo alla parte colposa è complicata come si fa a scomporre un'espressione Allora vi avevo detto ipotizziamo di avere una funzione trova carattere non anni dato che fa la ricerca di che cosa inizialmente della (e va alla ricerca di questa parentesi nel vettore espressione cosa potrei farmi restituire E direi l'indice a Che carattere corrisponde qui c'è il carattere Zero che la p123 vorrei farmi dire che quattro è il carattere incriminato perché 4 perché in questo modo e sono molto contento perché ho il mio e mi è venuta tato allora trova il carattere non annidato vorrei la posizione di questo carattere che potrebbe essere per esempio un intero intanto per fare una volta che ho questa posizione Qui si potrebbero fare un po' di cose Uno potrebbe dire Allora adesso io mi copio qua dentro in esse tutto ciò che è in espressione fino alla posizione chiesi Ma se così in effetti si può fare e si può fare in questo modo poi cambierò la soluzione in realtà ma così ne vediamo due alternative O i vantaggi e gli svantaggi potevi fare una String Copy npn copia una stringa ma non interamente copia solo un massimo di n caratteri e noi abbiamo un massimo di n caratteri Quindi quello che possiamo effettivamente dire che noi vorremmo copiare in esse la nostra stringa espressione e vogliamo copiare i caratteri se ne compriamo in realtà dovremmo copiare i meno 1 di carattere perché dovremmo solo questi però 1:36:30 Copiare i caratteri se Copiamo in realtà dovremmo copiare i meno 1 di carattere perché dovremmo solo questi però poi vogliamo anche mettere un uno spazio uno spazio un terminatore alla fine Scusatemi Sono stato poco chiaro direi dobbiamo intenderci su Che cos'è questo i che viene restituito quello lì è la posizione l'indice della del carattere proibito che stiamo cercando quindi in quella cella lì Noi abbiamo un carattere che non vogliamo copiare di conseguenza se mi Copiamo i andremo da zero Aimen uno e giustamente copiare mo copiare Mo in esse i caratteri che ci interessano il problema A questo punto è che la stringa S non ha un terminatore e quindi bisogna andare nella posizione Jesi ma della stringa s a scrivere esplicitamente un terminatore l'ho detto benissimo ma spero che una volta eletto risulti un pochino più chiaro si può fare così sicuramente si può fare così c'è un altro modo che magari non mi azzardo a fare adesso ma che è altrettanto valido ammesso che sia lecito sporcare la stringa espressione in fondo non è la volta che l'abbiamo letta la portiamo via quello che si potrebbe fare proprio andare qua dentro e trac

cancellare questo questa parentesi non cancellando la con lo spazio a cancellandola con un terminatore perché ci piacerebbe Perché a questo punto la stringa espressione diventa la stringa S che noi stavamo cercando e quindi può anche essere usata in seguito Questa è una seconda possibilità che non per Seguirò nel seguito Ma che forse un pochino più addirittura un pochino più veloce ed elegante per intenderci quello che sto dicendo è che probabilmente non l'ho testato si potrebbe anche scrivere questo è il tutto dovrebbe anche funzionare Io continuerò invece a seguire questa strada Allora questo. abbiamo sistemato presumibilmente in S l'operatore dobbiamo continuare Cosa vuol dire continuare vuol dire che non ci interessa più l'intera stringa espressione interessa solo la stringa che parte dalla posizione di più uno come si fa ad avere la stringa che parte dalla posizione di più uno qui sfruttiamo l'equivalenza tra Vittorio e quindi il fatto che io posso usare un puntatore lo posso usare come se fosse un sotto vettore di un vettore dato e cosa vuol dire tutta questa cosa vuol dire che se io prendo la stringa espressione vado nella posizione Jesi ma anzi nella posizione i più uno esimo e vado a prenderne l'indirizzo questa cosa qui è in realtà un puntatore e lo posso vedere come un vettore questo vettori comincia qua quindi questo vettore è la stringa che contiene il primo e secondo operatore ed è una stringa che io in effetti posso tranquillamente andare a passare a trova carattere non annidato cioè io posso dire Fammi cercare quale carattere il carattere, non ne l'intera stringa espressione Ma nella stringa è commerciale espressione di più 1 cioè la stringa che comincia il carattere di più 1 di espressione il risultato sarà il carattere i attenzione questo carattere i è relativo a partire da questa posizione qua quindi è meglio se andiamo a prenderci un secondo puntatore un secondo indice J e poi ce ne servirà un terzo K e ci ricordiamo che J è un indice che va bene se andiamo a prendere questo vettore qui allora probabilmente la cosa più sensata è avere una stringa e due e avere una stringa e tre e assegnare a queste stringhe questa espressioni cioè quando si comincia a perdere la sensazione di che cosa si sta facendo la cosa migliore cominciare a dare dei nomi alle unità agli oggetti che stiamo usando in maniera che assumano un significato un pochino più chiaro A quel punto in effetti non è più stretta è necessario avere degli indici secondari Possiamo tranquillamente riutilizzarli e possiamo dire Perfetto andiamo a prendere la nel vettore e due andiamo 1:41:10 Finalmente riutilizzarli e possiamo dire Perfetto andiamo a prendere la nel vettore che due andiamo a prendere tutti i caratteri fino al carattere iesimo escludo e li andiamo a copiare in S O P 1 e in esse op-1 andiamo ad aggiungere il terminatore e infine la stessa precisa identica cosa faremo per l'altra espressione si partirà Questa volta non da espressione di più 1 ma da e due di più uno esattamente per lo stesso motivo di prima Cioè siamo partiti qui siamo andati a cercare la virgola l'abbiamo cassata e da i più uno in poi questa è la stringa che ci interessa andremo a cercare il carattere non annidato in e3 il carattere che si interessa e la parentesi chiusa andremo a copiare e 3 fino alla posizione i.s.o. pd2e concluderemo sop2 con un terminatore questo è decisamente sofisticato Ma come vedete anche piuttosto snello Probabilmente ti può rendere ancora più snello andando a mettere i terminatori direttamente sul vettore espressione ma non mi azzardo a farlo per il momento almeno Allora vediamo se la cosa compila non ci ci sono un po' di confusione dovuta al fatto che trova carattere non annidato non è stato come dovrebbe non è stato dichiarato ed è finito andiamo a far che cosa scriviamo anche un bel commento trova questa funzione che cosa fa trova nella stringa espressione il carattere Diamogli un nome CH quindi trova seguiamo anche l'ordine degli argomenti trova il carattere CH nella stringa che espressione evitando Allora trova la prima occorrenza non annidata tra parentesi eccociqua del carattere CH nella stringa espressione e questo la dice tutta e finalmente è chiaro abbiamo un risultato intero Questi sono i dati e Possiamo aggiungere la nostra nuova funzione che andremo presto a implementare sarà la prossima prima di implementare la vediamo se il codice compila abbiamo un controllo che raggiunge la fine della funzione perché qui nessuno ha stabilito Qual è

il valore di default diciamo Zero mi fermo subito in Che sistema il tutto non vale la pena di lanciare il codice perché ciò che abbiamo aggiunto e ancora una volta una funzione che in realtà non fa nulla per poter vedere qualcosa dobbiamo arrivare a riempire questa carattere non annidato fatto quello dovremmo finalmente riuscire ad avere implementato fino al livello più basso la funzione converte espressione che ancora non ci dice nulla ma ci mancherà poco per poter fare invece le altre funzioni che sono più semplici Allora come si fa a trovare il carattere non hanno dato questa cosa non è complicatissima stiamo lascio ovviamente invito ovviamente a mettere in pausa la registrazione a procedere ma fondamentalmente quello che dobbiamo fare scorrere la stringa tenendo traccia di quante sono le parentesi aperte e chiuse quindi sommando ogni volta che c'è una (sottraendo ogni volta che c'è una parentesi chiusa e tutte le volte che troviamo il carattere CH e il conteggio delle parentesi è nullo siamo a posto Ci fermiamo istituiamo una posizione corrente ovviamente può darsi che avevamo in fondo a quel punto dovremmo capire cosa fare direi che restituiamo la posizione finale ci fidiamo del fatto che i dati siano scritti Nel formato corretto Ok procediamo Quindi abbiamo un cursore che scuola e l'intera stringa Come si scorge una stringa si parte dal carattere zero e si procede fintanto che espressione di chi è diverso dal carattere 1:46:00 Scuola e l'intera stringa Come si scorge una stringa si parte dal carattere zero e si procede fintanto che espressione di i è diverso dal carattere separatore al carattere terminatore Questa è la classica scorrimento di una stringa non si scrive I minore di String len della stringa stessa perché Perché string-length non fa altro ad ogni passo che scorre Se l'intera stringa per vedere dove si finisce questo vorrebbe dire per n volte scorse l'intera stringa per vedere finita è finita è finita è finita Non ha senso invece ti va a vedere il carattere corrente per vedere se la stringa è terminata sono quelle piccoline efficienze che si nascondono nel codice quando non si è capito che cosa fanno le funzioni che si chiamano Allora qui stiamo correndo carattere per carattere e vi ricordo stiamo cercando di tenere un conteggio delle parentesi Quindi potremmo avere una variabile con che fa da contatore che inizialmente azzerata Ricordatevi sempre di inizializzare le variabili Quando si entra poi dici chi A questo punto ad ogni carattere andiamo a vedere ma per caso espressione di quel Bisogna capire se dobbiamo prima fare il conteggio delle parentesi e poi vedere che cosa sta succedendo o viceversa non dovrebbe essere molto diverso ma direi che preoccupiamoci dell'aspetto fondamentale espressione di è il carattere in questione si bene potrebbe essere che ci dobbiamo fermare ma per fermarci deve anche essere vero che il conteggio delle parentesi è fidanzato a questo punto usciamo restituendo i altrimenti qui non farò kiehl's per compattezza altrimenti che cosa succede dobbiamo vedere se per caso espressione di sé una (perché sei una (e questo sì che dobbiamo incrementare il conteggio e infine se è una parentesi chiusa dobbiamo decrementare il conteggio E questo dovrebbe essere quanto alla fine non restituiamo Zero cioè se siamo arrivati Qui invece di restituire zero e stituire mo Comunque il valore i che ovviamente questo punto corrispondeva con l'indice finale della stringa E questo può avere senso vuol dire che comunque la stringa sbagliata ma ad ogni modo ha senso dire che non ho trovato il carattere non annidato e quindi restituisco l'intera stringa Vediamo se questa cosa Compila la punto di vista sintattico è corretta non prevedo che ci siano risposte perché non abbiamo ancora fatto niente che meriti di essere stampato Ma possiamo finalmente andare a dedicarci alle funzioni ricorsive su albero quelle che risolvono il piede proprio problema Calcola valore stampe espressione profondità le faremo una dopo l'altra e per ciascuna Ovviamente vi invito a fare un tentativo voi e poi a vedere la mia soluzione ok Come funziona il calcolo del valore abbiamo visto in questi alberelli qui per calcolare il valore di una espressione Dobbiamo distinguere Come sempre sono tutte funzioni ricorsive il caso base al caso generico Che cos'è il caso base attenzione Di solito il caso base quando si maneggia un albero è il caso di albero vuoto che non va mai dimenticato e spesso è l'unico caso base quindi teniamo conto

del fatto che potrebbe esserci un albero vuoto che in effetti è l'unico già gestito ha detto innoval Però in realtà c'è un secondo caso base questo non è tipico ma succede e secondo caso base è quella in cui l'albero contiene solo il nodo radice e perché è un caso base perché in questo caso l'espressione facile da valutare si legge il operando Dopodiché c'è il caso generico di corsivo che quello in cui c'è un operatore invece Quindi come facciamo a distinguere il caso base di albero vuoto è facile ma il caso base di albero con una sola ditta dal caso generico quello si distingue in base al fatto che ci sia un operatore piuttosto che un operando se c'è un operatore vuol dire che l'espressione è composta e quindi Bisogna prepararsi a calcolare i due sotto alberi come con chiamate ricorsive e poi metterli insieme i due valori così ottenuti con l'operatore stesso il quale operatore ovviamente andrà come dire gestito sarà un più un per e quindi in base al carattere che abbiamo dovrà provocare un risultato diverso quindi Adesso dovreste essere in grado di fare il tutto sostanzialmente che cosa stiamo dicendo Sto dicendo che se l'albero è vuoto 1:50:50 Sostanzialmente che cosa stiamo dicendo Sto dicendo che se l'albero è vuoto ha valore lo Val e questo è molto semplice come si fa a sapere che l'albero è vuoto c'è una funzione albero vuoto che viene chiamata su ti è che tra risultato tu solo in questo caso altrimenti altrimenti Può darsi comunque che ci sia un caso base Cioè può darsi che se l'albero ha solo il nodo radice il suo valore è lo perando della radice Come si fa a sapere che un nodo ha soltanto il nodo Felice è piuttosto semplice bisogna intanto leggere il nodo a questo punto possiamo confrontare l'operatore El operando con 990 per la prima cosa da fare Sara andare a leggere il nodo come funzionava la funzione legge nodo Allora la funzione leggi nodo aveva questi quattro argomenti che erano l'albero che ovviamente ti la posizione che ci interessa è questa posizione che posizione è la radice quindi è la radice di tutti e due operatori passati per indirizzo e quindi giustamente qua ci servono dei dati ci servono un operatore hop e un operando fall quindi andremo a recuperare hop e andremo a recuperare Val dalla radice dell'albero e avendoli letti potremmo andare a capire questo vale sempre Comunque potremmo andare a capire se per caso l'albero ha solo il nodo radice Come si fa a saperlo si sa perché in Val c'è un vero e proprio un vero e proprio operando O meglio ancora in Coop Non c'è un operatore Quindi se Coop è uguale a questo è più sicuro perché abbiamo scelto come 90 e non vorrei che magari ci fosse invece effettivamente un operando = 0 è una scelta un po' difficile quella del nuval non ci sono scelte veramente valide Comunque se per caso invece questo è sicuro l'operatore non c'è vuol dire che abbiamo solo il nodo radice di conseguenza possiamo avere stituire in uscita come valore Val altrimenti invece abbiamo il caso ricorsivo il caso V corsivo è quello in cui l'espressione è composta quindi se l'espressione è composta che cosa si fa Valuta il valore dei due sotto alberi Per prima cosa e applica l'operatore hai due valori Questo è quello che dobbiamo fare come si fa a 1:54:20 Questo è quello che dobbiamo fare come si fa a calcolare i due valori Beh direi che possiamo chiamarli in qualche modo possiamo chiamarli valore sinistro e valore destro e possiamo anche ritrovarli con delle chiamate ricorsive della funzione che stiamo scrivendo precisamente li applicheremo a che cosa li applicheremo al figlio sinistro e al figlio destro all'albero figlio sinistro diti e all'albero figlio destro di tipo questo ci fornisce i due valori che cercavamo e a questo punto Si tratta semplicemente di applicare l'operatore a seconda dei casi l'operatore potrebbe essere 4 casi potrebbe essere un più e in tal caso restituiremo come risultato valesse più Valdi oppure potrebbe essere un meno in tal caso vestito in uova lesse meno Val di potrebbe essere un prodotto un rapporto è in ciascuno di questi casi andremo a eseguire l'operazione che è richiesta Nel caso specifico qui rimane appeso un'ulteriore else ma direi che non è il caso di trattarlo Nel senso che sicuramente se diamo per scontata che la struttura dell'albero sia corretta non abbiamo altre possibilità è tutto discutibile ma semplifichiamo ok Questo dovrebbe di per sé risolvere il problema del calcolo del valore vediamo che ci sono errori perché la funzione figlio sinistro in realtà data un po' per scontata

richiedeva un albero ma anche una posizione la posizione ovviamente è la posizione della radice dell'albero ti interessa non il figlio sinistro di un generico nodo il figlio sinistro della radice dopodiché Scusatemi un attimo ho sbagliato Sì ho trasferito Calcola valore richiede semplicemente un albero e nient'altro Esatto il pasticcio è che Calcola radice è richiesta come parametro di figlio sinistro invece l'oppio Pata a Calcola valore Ok quindi sono figlio sinistra figlio destro che hanno come secondo parametro radice Calcola valori non ha un secondo parametro e di conseguenza ci rimane solo questo control che raggiunge un valore finale di una funzione non-void perché siamo rimasti nel caso in cui in effetti usciamo da entrambe queste situazioni cosa sostanzialmente impossibile di conseguenza però per non avere avvertimenti Diciamo che se per caso uscissimo da tutti questi casi qui allora restituiremo un Noval e questo sistema la cosa adesso bisogna incrociare le dita e vedere se il tutto funziona ovviamente no risolviamo il problema allora la prima cosa che cerchiamo di valutare è se abbiamo spezzato correttamente le espressioni quindi proviamo a farci stampare le tre stringhe ss81 so2 in maniera che siano abbastanza ben distinte fra loro e quello che vediamo è qui andiamo a capo ci facciamo stampare SS op 1 S op 2 compiliamo ed eseguiamo 1:59:10 Compiliamo ed eseguiamo e vediamo che in effetti la composizione in forma 53 e dive andata a buon fine è la scomposizione di Somma 53 anche Dopodiché qualcosa Eh forse è successo potrebbe essere successo a livello della vuol dire che siamo scesi nell'albero correttamente e poi qualcosa è successo qua dentro che è più probabile In effetti Allora a questo punto proviamo a valutare la conversione delle espressioni quindi nel Convertire le espressioni nel calcolo del valore No è ancora una commissione dell'espressione va bene Diciamo che l'ho trovato nel senso che è un errore che purtroppo abbiamo commesso diverso tempo fa nell'implementare la libreria albero quando abbiamo costruito il nodo ho fatto un bel Ca Tempest e ho detto una cosa e ne ho fatta un'altra che cosa ho detto ho detto che il sottoalbero sinistro sottoalbero destro ammettono come padre la radice che noi stiamo aggiungendo Ma questo solo unicamente se esistono se non sono vuoti ma questa non è la condizione di sotto albero vuoto a condizioni sotto albero motore TS diverso da notare e TDT verso danno Trip questa cosa produceva in realtà un errore di accesso è l'errore di accesso prove provocava tutti i pasticci che abbiamo visto prima per quell'altra il programma si interrompeva non è che stampasse risultati sbagliati e a questo punto fortunatamente invece la cosa funziona possiamo verificare se riesco a trascinare Qua possiamo verificare che se riusciamo a trascinare dall'altra parte riusciamo a verificare che il prima espressione Vale 40 abbiamo fatto il conto all'inizio della lezione la seconda Evidentemente ha 10 / 3 cioè 3,3 fattoriale la terza una bella somma di serie 5 11 e 4 15 e 318 E 220 e 121 Eccoci qua è l'ultima e 2.5 quindi a livello calcolo dei valori perlomeno su queste quattro distanze il nostro algoritmo funziona lo diamo per buono ci rimangono le ultime due funzioni che sono Fortunatamente più semplici come funziona la stampa della espressione in notazione infissa Qual è la notazione infissa quella a cui siamo abituati dalla scuola cioè quella per la quale Il * il + il diviso il meno vengono messi in mezzo fra i due Fra operandi Allora cosa si tratta di fare e credo sia abbastanza evidente che se uno ha questa struttura si tratta di stampare prima di espressione di sinistra poi stampare il prodotto e poi stampare l'espressione di destra cioè di manipolare prima il sottoalbero sinistro poi il nodo radice poi sottoalbero destro eseguendo una visita questa volta in ordine allora come dobbiamo anche fare è consigliabile aggiungere le parentesi e per non andare a considerare le parentesi tonde quadre e eccetera eccetera Diciamo che le parentesi sono tutte tonde si pone un problema vedremo il problema Sarà che cosa fare con le espressioni semplici per farla semplice Io dirò che anche le espressioni semplici saranno racchiuse tra parentesi Verrà fuori una scrittura un po' barocca diversa da questa che è quella a cui siamo abituati in cui anche il 5 e il il 10 42 avranno delle parentesi e tutto quanto nel complesso avrà una parentesi è chiaro che si può ragionare un pochino per

semplificare la cosa Ma questo complica la forma della funzione che poi è quello che sostanzialmente mi interessa di sottolineare quindi andiamo in stampa espressione chiediamoci che cosa bisogna fare per stampare in notazione standard questo questo albero ti che noi abbiamo e non sarà particolarmente difficile sarà una scrittura ricorsiva vi invito come sempre tener conto del caso base caso base di albero vuoto volendo potreste distinguere un caso base di albero vuoto è un secondo caso base di espressione sempre 2:04:00 Volendo potreste distinguere un caso base di albero vuoto è un secondo caso base di espressione semplice Magari poi verremo a farlo però non è che questo non è dato nelle soluzioni che trovate in rete Allora come si fa a distinguere semplicemente se l'albero è vuoto l'albero ti è vuoto che cos'è che dobbiamo stampare e direi che non sappiamo praticamente nulla e quindi in realtà questo è un if che non mi dà nulla come risultato poi ci sarebbe Lens quando abbiamo uniche non dà risultato abbiamo un is not sostituiamo con notte e qui ci sarebbe un if albero vuoto non fare assolutamente nulla altrimenti fai queste cose però è un caso base talmente semplice che esattamente come Vedremo nelle prossime elezioni sugli algoritmi di ordinamento quando dovete ordinare un vettore vuoto non si fa nulla Anche qui quando abbiamo un albero vuoto su cui fare qualcosa non si fa nulla e quindi questo questo questo blocco di operazioni non viene neanche scritto tanto te vuoto alla se l'albero non è vuoto finalmente cosa dobbiamo fare semplicemente andare a stampare qualche cosa che cos'è che dobbiamo andare a stampare e bisogna capire Intanto se siamo in una espressione semplice o composta Perché nel caso di espressione semplice stamperemo un numero nel caso di espressione composta invece stamperemo una espressione composta con le parentesi Quindi come si fa a recuperare i dati l'abbiamo già fatto una volta siamo andati a leggere i dati del nodo siamo andati a leggere i dati del nodo stesso abbiamo letto l'albero ti la posizione che posizione è la posizione della radice di ti molto semplicemente Dopodiché siamo andati a caricare in opportune variabili che saranno un operatore op è un operando Ball andremo a caricare i due dati uno dei quali è sensato mentre l'altro è sporco È un valore di default È già abbiamo visto che ci conviene testare l'operatore per vedere se è un WhatsApp in questo caso vuol dire che non siamo in un nodo interno siamo in una foglia e quindi ci possiamo limitare a stampare banalmente il nostro valore reale Layton possiamo fare un Print F di un numero reale facciamo degli spazi per essere sicuro che la scrittura non sia tutta appiccicata ne faccio uno più uno dopo tanto per stare largo a questo punto Si tratta di andare a trattare il vero e proprio caso ricorsivo deve proprio caso ricorsivo Elsa e quello in cui noi abbiamo un operando e quindi ci converrà mettere delle parentesi giusto per racchiudere il tutto e rendere leggibile quello che stiamo facendo anche qui a ponder o in spazi tanto per poi si può sempre fare a tempo a toglierli dopo e andremo a stampare intanto l'espressione del albero sinistro quindi andremo a stampare l'espressione del figlio sinistro dell'albero ti rispetto alla posizione radice Vi ricordo che figlio finito riceve anche l'indicazione della posizione di cui noi vogliamo conoscere la radice ci manca una terza parentesi da chiudere Dopodiché andremo a stampare il l'operatore è l'operatore Fortunatamente è un semplice carattere Ok poi andremo a stampare il figlio destro infine chiuderemo la nostra parentesi in questo sistema la stampa dell'espressione testo possiamo dare un'occhiata a vedere come risulta abbiamo un hop mentre invece dobbiamo stampare il valore vedete il messaggio Dice che il formato X 100 F si aspetta argomenti di tipo Double ma noi abbiamo passato un argomento di tipo int E questo deve far rizzare le orecchie 2:08:55 Napoli ma noi abbiamo passato un argomento di tipo int E questo deve far rifare le orecchie argomenti alta non era di tipo int era di tipo operatore L'operatore ha un carattere i caratteri in ci sono gli interi quindi il solito messaggio in leggibile se uno non sa capire di che cosa si tratta però è comunque un segnale che c'è qualcosa che non va è stato molto utile e a questo punto dovremmo essere in grado di lanciare il tutte avere una stampa Da quale stampa non è il massimo della leggibilità però in effetti si può verificare che è

abbastanza corretta Nel senso che abbiamo una somma di 53 che viene moltiplicata per che cosa per il rapporto di 10 / un rapporto una differenza tra 4 e 2 e le altre sono analoghe Se volessimo rendere tutto un po' più leggibile possiamo probabilmente andare a togliere un po' di spazi possiamo andare a diminuire il numero di cifre decimali dopo la, per esempio potremmo in questo caso potremmo anche non mettere cifre decimali visto che sappiamo che sono numeri interi c'è un solo numero Questo qui Se solo questo 2.5 quindi diciamo che mettiamo una cifra dopo la virgola e questo diventa molto più ragionevole è leggibile Però evidentemente dipende poi dalla scelta che abbiamo fatto come si fa a calcolare la profondità dell'albero Questo è un esercizio dirittura banale potrebbe essere un classico esercizio da esame per esempio all'orale in teoria e In che cosa consiste la profondità dell'albero consiste nel numero massimo di nodi che sono compresi in un cammino della radice a una delle foglie Come si fa a calcolare la profondità di un albero algoritmo ricorsivo al solito avremo un caso base che è l'albero vuoto e Dopodiché andremo a capire data la profondità del sottoalbero di Sinis profondità e sotto l'albero di destra come si fa a trovare la profondità dell'albero complessivo vi lascio un minimo di tempo per farlo ma la soluzione rata è molto semplice ed è la seguente Allora al solito ci interroghiamo sul caso di albero vuoto se ti è un albero vuoto in questo caso abbiamo una risposta ed è che la profondità è zero distinguiamo il casi albero vuoto da quello di albero che ha una sola radice lì la profondità è uno è il caso di fare un secondo caso base No molto spesso non è necessario fare un secondo caso base abbiamo visto situazioni in cui è stato necessario perché avevamo da distinguere fra operatori operandi Ma se il vostro albero è fatto di oggetti omogenei Di solito il caso base albero vuote più che sufficiente Allora a questo punto se il nostro albero non è vuoto è chiaro che si tratta piuttosto banalmente di andare a determinare le 2 profondità e s20 didattico dei nomi pspd a queste due profondità vado a calcolare la profondità dell'albero di sinistra chiamando ricorsivamente il mio algoritmo su cosa sul figlio sinistro dell'albero ti rispetto alla posizione radice dell'albero ti vado a calcolarmi la posizione approfondita del sottoalbero di destra chiamando la stessa funzione su sottoalbero destro sempre rispetto alla radice e a questo punto Tutto dipende dal fatto che sia Maggiore PS o Maggiore PD e io posso anche dire molto banalmente che se per caso usiamo il costrutto if a tre argomenti se per caso PS maggiore di PD Ecco che il risultato finale sarà uno più PS perché uno più PS perché vuol dire che dei due sotto alberi profondo il sinistro alla sua profondità devo raggiungere devo aggiungere il nodo radice Analogamente se succede quell'altro per cui in realtà ancora più compattamente il mio risultato Potrebbe essere uno più p.s. se è maggiore di PDP di Se invece è il contrario questo è la profondità che io voglio restituire al mondo esterno mia scrittura molto compatta che il programmatore C Tendono ad amare non so se sia particolarmente leggibile Ad ogni modo questo ci dovrebbe fornire il risultato abbiamo un po' di errori vediamo un attimo di che cosa si tratta Allora si tratta di avere puoi mettere dei punti e, malmessi abbiamo una (che manca a Riga 229 qui sono altre 2:13:45 Abbiamo una (che manca a Riga 229 qui sono altre parentesi e anche questo è questo il Balestra interpretazione di tutto il codice è risolto questo dovremmo essere a posto Se lo lanciamo scopriamo che la profondità del primo dei nostri quattro esempi e quattro ed effettivamente abbiamo 1 2 3 e 4 Tutto su sottoalbero destro La seconda è 2 su entrambi la terza e sei tutto sul sottoalbero destro e la quarta è uno e siamo a posto così Allora a questo punto L'unica cosa che avete avrebbe da analizzare ancora eh E se cancellassi mo tutto e volessimo fare un implementazione a Vittoria indici Ecco questa volta non la facciamo nel testo che accompagna la lezione Ci sono alcuni ragionamenti Io stesso non ho in questa modifica questa implementazione non ho fatto questa implementazione Ho il sospetto sulla base dei ragionamenti che ho fatto che qualche piccola modifica alla interfaccia della libreria Cioè anche al file h sia effettivamente necessaria e però questo non è ampiamente Al di là del dei limiti del della lezione

che quindi si conclude qui

La lezione è centrata su un esercizio per impratichirsi sui seguenti argomenti:

- alberi binari rappresentati con puntatori
- alberi binari rappresentati con cursori

Chi notasse eventuali incoerenze o errori, oppure avesse dubbi sul contenuto di queste pagine e dei codici, è pregato di segnalarmeli per contribuire a migliorare la qualità dei materiali del corso.

Problema

Si vuole scrivere un programma `calcola.c` che elabora espressioni aritmetiche. Il programma acquisisce le espressioni da un file di testo, che contiene un'espressione in ogni riga, con una lunghezza non superiore a 256 caratteri. Le espressioni sono codificate come se fossero espressioni scritte in un linguaggio di programmazione, vale a dire nel seguente formato:

- espressione semplice: un numero reale;
- espressione composta: *operatore*`(`*operando1*`,`*operando2*`)`, dove
  1. *operatore* è il nome di un operatore aritmetico, cioè una stringa qualsiasi fra `somma`, `diff`, `prod`, `div`
  2. *operando1* e *operando2* sono due espressioni aritmetiche, semplici o composte.

Questo formato richiama da vicino la così detta *notazione prefissa* (o *polacca*), nella quale gli operatori precedono gli operandi. La differenza è che nella notazione polacca le parentesi non vengono usate, dato che sono ridondanti. Noi conserviamo le parentesi per maggiore leggibilità. Il file `input01.txt` riporta il seguente esempio di espressione composta:

```
prod(somma(5,3),div(10,diff(4,2))
```

il cui valore è

$$(5+3)*[10/(4-2)] = 8*(10/2) = 8*5 = 40$$

Il programma deve:

1. calcolare il valore dell'espressione e stamparlo a video;
2. convertire l'espressione nella classica notazione infissa (con i simboli $+$, $-$, $*$ e / inseriti fra gli operandi) e stamparla a video;
3. determinare la *profondità* dell'albero, definita come il numero massimo di nodi lungo un cammino dalla radice alle foglie dell'albero.

Traccia della risoluzione

Prima di eseguire l'esercizio, si possono scorrere interamente i lucidi per acquisire il concetto di albero binario e le basi delle implementazioni di alberi binari con puntatori.

Quindi si può ragionare un po' sul modello del problema. Un'espressione aritmetica composta ha una struttura gerarchica e ricorsiva che si presta benissimo ad essere rappresentata con un albero. Le operazioni aritmetiche qui considerate sono tutte binarie, per cui l'albero può essere binario, anche senza introdurre la convenzione dei puntatori al primo figlio e al fratello successivo. Inoltre, la notazione prefissa consente di costruire l'albero identificando facilmente l'operatore, che corrisponde alla radice di un albero, e gli operandi, che corrispondono ai due sottoalberi sinistro (il primo operando) e destro (il secondo). Le parentesi e le virgole servono solo al lettore umano: il programma le ignora. Il programma dovrà costruire l'albero binario che rappresenta ogni espressione sfruttando il fatto che la notazione prefissa presenta alla lettura prima il nodo padre e poi i suoi due figli, facilitando la costruzione dell'albero stesso. Merita osservare che, quindi, la rappresentazione di partenza dell'espressione corrisponde esattamente a una visita in pre-ordine dell'albero corrispondente.

Il file di partenza, `calcola0.c`, contiene, al solito, le direttive per simulare il tipo `boolean`, la costante simbolica `ROW_LENGTH` per gestire le righe in lettura e scrittura, la funzione `InterpretaLineaComando` che recupera il nome del file dalla linea di comando e lo copia in un'apposita variabile. Contiene anche uno scheletro della funzione principale, `ElaboraEspressioni`, la quale apre il file e lo scorre con un ciclo, caricando ciascuna riga in una stringa in modo da poterla elaborare.

Prima fase (`calcola1.c`) La prima fase dell'esercizio richiede quindi di completare lo scheletro della funzione principale, come sempre in modalità *top-down*, cioè di:

1. leggere l'espressione corrente, che corrisponde a una riga del file (consiglio: `fgets` oppure `fscanf(fp,"%[^\n]\n",Espressione)`: la semplice specifica `"%s"` è valida solo se l'espressione non contiene spazi bianchi);
2. convertire l'espressione corrente da una stringa a un albero binario;
3. calcolare e stampare il valore dell'espressione rappresentata dall'albero;
4. stampare a video in notazione infissa l'espressione rappresentata dall'albero;
5. calcolare la profondità dell'albero;
6. distruggere l'albero.

Lo faremo assumendo di avere a disposizione una libreria per la gestione di alberi binari, della quale è disponibile il file *header* `albero.h`, che adotta un'implementazione a puntatori:

- un `operatore` è dichiarato come un carattere;
- un `operando` è dichiarato come un numero reale;
- un `albero` è dichiarato come un puntatore a un `nodo`;
- una `posizione` è dichiarata come un puntatore a un `nodo`;

- un `nodo` è dichiarato come una struttura contenente un operatore e un operando (questo è uno spreco, perché uno solo dei due campi sarà utilizzato, ma semplifica le cose) e tre posizioni: quella del padre e quelle delle radici del sottoalbero sinistro e destro.

Sono anche definite delle costanti simboliche per rappresentare operatori fittizii (`NO_OP`), operandi fittizii (`NO_VAL`), alberi vuoti (`NO_TREE`) e posizioni fittizie (`NO_NODE`). Il file `albero0.c` fornisce solo i corpi vuoti delle funzioni; quelle che devono restituire un valore ne restituiscono uno fittizio convenzionale.

Seconda fase (`albero.c`) Per poter completare la costruzione dell'albero, occorre definire le funzioni della libreria che gestisce gli alberi binari:

- `albero creaalbero ()` crea e restituisce un albero vuoto;
- `boolean alberovuoto (albero T)` determina se l'albero dato è vuoto o no;
- `void leggenodo (albero T, posizione p, Operatore *pop, Operando *pval)` restituisce l'informazione (operando e operatore) associata ad una data posizione di un dato albero;
- `void scrivenodo (albero T, posizione p, Operatore op, Operando val)` assegna un operando e un operatore a una data posizione di un dato albero;
- `posizione radice (albero T)` restituisce la posizione alla radice dell'albero;
- `posizione figliosinistro (albero T, posizione p)` restituisce la posizione alla radice del sottoalbero sinistro del nodo in una data posizione di un dato albero;
- `posizione figliodestro (albero T, posizione p)` restituisce la posizione alla radice del sottoalbero destro del nodo in una data posizione di un dato albero;
- `posizione padre (albero T, posizione p)` restituisce la posizione del nodo padre di quello in una data posizione di un dato albero;
- `albero costruiscealbero (Operatore op, Operando val, albero Ts, albero Td)` costruisce un albero inserendo nella sua radice una data informazione (operatore e operando) e appendendogli due alberi dati come sottoalbero sinistro e destro;
- `albero cancsottoalbero (albero T, posizione *pp)` distrugge l'intero sottoalbero appeso a una data posizione di un dato albero;
- `void distruggealbero (albero *pT)` distrugge un intero albero.

Le funzioni sono quasi tutte banali. È utile osservare che la distruzione di un intero albero è un caso particolare della cancellazione di un sottoalbero (basta applicarla alla radice), e che per cancellare un sottoalbero bisogna visitarlo rigorosamente in post-ordine, dato che bisogna deallocare prima i sottoalberi e poi il nodo radice; altrimenti, i sottoalberi non sarebbero più raggiungibili. Di conseguenza, tutte le operazioni fondamentali sono in tempo costante $\Theta(1)$ (sempre che le informazioni associate ai nodi abbiano dimensione costante), tranne la cancellazione e la distruzione, che richiedono tempo lineare nel numero n degli elementi dell'albero ($\Theta(n)$).

Terza fase (`calcola2.c`) Nella fase seguente dell'esercizio proseguiamo in modalità *top-down* convertendo l'espressione letta da file in un albero binario con la funzione `ConverteEspressione`. Siccome l'espressione è ricorsiva, conviene che anche la funzione che la manipola lo sia. Avremo quindi, come sempre, due casi:

- *caso base*: se l'espressione è semplice, cioè ridotta a un numero reale (per verificarlo, si può usare `sscanf`), basta costruire un albero con la radice contenente come operando il numero reale e un operatore fittizio e due sottoalberi vuoti;
- *caso ricorsivo*: se l'espressione è composta, occorre dividerla nelle sue tre stringhe componenti:
  1. operatore: la stringa iniziale sino alla prima parentesi aperta, esclusa;
  2. primo operando: la stringa compresa fra la prima parentesi aperta, esclusa, e la virgola che separa i due operandi (che non è in genere la prima virgola!) esclusa;
  3. secondo operando: la stringa finale che parte dalla virgola separatrice, esclusa, fino alla parentesi conclusiva, esclusa.

  Siccome è un'operazione sofisticata, se ne occuperà un'opportuna funzione `ScomponeEspressione`, da realizzare in seguito.

Una volta scomposta l'espressione, l'operatore va tradotto da una stringa (`somma`, `diff`, `prod` o `div`) nel carattere corrispondente (`'+'`, `'-'`, `'*'` o `'/'`), con un'opportuna funzione `IdentificaOperatore`. I due operandi, invece, vanno analizzati ricorsivamente per convertirli in alberi, ancora una volta grazie alla funzione `ConverteEspressione`. Otterremo così un operatore e due alberi, che possiamo fondere in un albero con la funzione di libreria `costruiscealbero`.

Quarta fase (`calcola3.c`) Procediamo quindi a realizzare le funzioni:

- `IdentificaOperatore`, che è una banale serie di confronti in cascata (`if ... else if ... else`) condotti con la funzione `strcmp`;
- `ScomponeEspressione`, che scorre la stringa data copiandone via via i caratteri in tre stringhe ausiliarie.

È possibile fare questo in modo abbastanza elegante, riconoscendo che i tre compiti sono molto simili fra loro: la prima stringa (operatore) termina alla prima parentesi aperta; la seconda (primo operando) termina con una virgola; la terza (secondo operando) termina con una parentesi chiusa. La virgola e la parentesi chiusa che indicano la fine dei due operandi non sono però le prime incontrate lungo la stringa: bisogna ignorare le virgole e le parentesi "annidate" in altre parentesi di livello inferiore. Per esempio, in `prod(somma(5,3),div(10,diff(4,2))` l'operatore è `prod`, mentre i due operandi non sono `somma(5` e `3, bensì somma(5,3)` e `div(10,diff(3,2)`. Per risolvere questa difficoltà, è sufficiente trovare la prima occorrenza di un carattere separatore che non sia annidata all'interno di una coppia di parentesi. Dunque, si tratta di scorrere la stringa tenendo conto del numero di parentesi aperte che non siano ancora state chiuse: il separatore va accettato solo se questo numero è nullo. La procedura `TrovaCarattereNonAnnidato` fornisce la posizione del separatore. Questa posizione può essere poi usata per dividere l'espressione originale in due parti. La prima parte va copiata (con una funzione `strncpy`, dato che non stiamo copiando un'intera stringa, ma solo una sua porzione)

nella stringa di destinazione. La seconda parte deve diventare la nuova stringa da suddividere. Infatti, la ricerca del separatore successivo non può ripartire dall'inizio dell'espressione, ma deve partire dal carattere successivo al separatore precedente. Un modo un po' criptico, ma molto efficace di fare questo è sostituire `Espressione` con `&Espressione[i+1]`, cioè con il vettore (la stringa) che parte dalla posizione successiva al separatore precedente. Questo è reso possibile dall'equivalenza fra puntatori e vettori e dal fatto che la variabile `Espressione` non è in realtà il vettore statico allocato e riempito nella funzione `ElaboraEspressioni`, ma semplicemente un puntatore nello spazio di memoria locale della funzione `ConverteEspressione`, e quindi si può tranquillamente modificarlo senza alterare il vettore statico originale (che, in quanto vettore statico, non può stare a sinistra in un assegnamento).

Siccome le funzioni qui realizzate si applicano a stringhe con una lunghezza massima predefinita, la loro complessità è in $\Theta(1)$. Questo ci permette di valutare la complessità della conversione di un'espressione in un albero binario. Essa coincide con quella di una visita dell'albero stesso, cioè è proporzionale al numero n di elementi dell'albero ($\Theta(n)$), che è il numero totale di operatori e operandi che compaiono nell'espressione.

Quinta fase (`calcola4.c`) A questo punto, si può procedere a realizzare la funzione `CalcolaValore`, che valuta l'espressione rappresentata dall'albero. Ancora una volta, il modo più semplice è procedere ricorsivamente, avendo:

- un *caso base* corrispondente all'albero vuoto, per il quale si restituisce il valore fittizio `NO_VAL`;
- un secondo *caso base* nel quale la radice non contiene un operatore (cioè contiene l'operatore fittizio `NO_OP`), ma solo un operando, che va restituito;
- un *caso ricorsivo* nel quale occorre prima valutare le due espressioni figlie e poi applicare l'operatore ai loro valori.

Si noti che tutto questo equivale a una visita in post-ordine dell'albero stesso, dato che abbiamo prima valutato i nodi figli e poi ricavato da loro il valore del padre. Avendo realizzato tutte le operazioni su alberi binari (salvo la distruzione) in tempo costante, qualsiasi algoritmo sia riducibile a una visita dell'albero richiede tempo lineare nel numero dei suoi nodi, cioè $\Theta(n)$. Questo vale, ad esempio per la valutazione dell'espressione.

Sesta fase (`calcola5.c`) Passiamo quindi a realizzare la funzione `StampaEspressione` che stampa l'espressione rappresentata dall'albero nella classica notazione infissa, con l'operatore intermedio fra i due operandi. Per evitare ambiguità, dobbiamo aggiungere le parentesi tonde, che nella notazione prefissa di partenza erano presenti, anche se non necessarie: qui invece lo sono. Stamperemo quindi una parentesi aperta al principio dell'intera espressione e una chiusa al termine. Possiamo interpretare le parentesi come istruzioni di navigazione nell'albero: la parentesi aperta fa scendere al livello successivo, quella chiusa fa risalire al precedente. Naturalmente, l'implementazione sarà ancora ricorsiva, con due casi base (albero vuoto e albero ridotto alla sola radice) e un caso ricorsivo, nel quale l'elaborazione del figlio sinistro precede quella dell'operatore nella radice, che precede quella del figlio destro. Il caso di albero vuoto non comporta di fare nulla (quindi, nel codice apparentemente non compare). Il caso di albero ridotto alla sola radice comporta la stampa del valore conservato nella radice, che è un'espressione semplice. Si tratta quindi di una visita in in-ordine. Di conseguenza, la complessità è ancora $\Theta(n)$.

Settima fase (`calcola6.c`) L'ultima fase dell'esercizio realizza il calcolo della profondità dell'albero, anch'esso in modo ricorsivo:

- nel *caso base*, la profondità è per convenzione nulla;
- nel *caso ricorsivo*, la profondità è la maggiore fra le profondità dei due sottoalberi, incrementata di 1 per tener conto del livello aggiuntivo introdotto dalla radice.

Si tratta ancora di una visita, ma in effetti l'elaborazione del nodo radice è talmente banale (sommare 1 al massimo) che non ha molto senso cercare di classificare il tipo di visita. Comunque sia, la complessità è $\Theta(n)$ come in tutti i casi precedenti.

Nota finale: l'implementazione a vettori e indici A questo punto, si potrebbe procedere a creare una nuova libreria di gestione, basata sull'implementazione con vettori e indici interi. La differenze fondamentali sono:[2]

- l'albero è ospitato in un vettore allocato una volta per tutte dalla funzione `creaalbero`; la dimensione non varia durante l'esecuzione (se è insufficiente, si esce con un messaggio di errore) e il suo valore `TREE_SIZE+1` tiene conto degli elementi massimi ospitabili e della testa di una *lista libera*, che ospita gli elementi del vettore non impiegati nell'albero corrente;
- i puntatori sono sostituiti da indici numerici; per accedere ai nodi, bisogna quindi conoscere sia la posizione sia l'albero: il nodo in posizione `p` dell'albero `T` è banalmente `T[p]`
- ogni elemento del vettore contiene l'informazione associata a un nodo e gli indici numerici nel vettore stesso dei nodi radice dei due sottoalberi e del nodo padre;
- la radice è l'elemento del vettore corrispondente a un indice speciale (`RADICE`);
- gli elementi del vettore correntemente non impiegati nell'albero formano una lista libera, la cui testa corrisponde a un indice speciale;
- inserimenti e cancellazioni corrispondono a spostamenti fra albero e lista libera.

Pur essendo molto diversa dalla precedente, la nuova implementazione non richiede nessuna modifica al file `calcola.c`: potenza delle strutture dati astratte.

10.4 Esercizi

10.4.1 Esercizio

Si modifichi l'esercizio svolto in laboratorio in modo che le stringhe che rappresentano l'operatore e gli operandi non siano esplicitamente rappresentate da vettori di caratteri, ma implicitamente dall'espressione letta in ciascuna riga e dagli indici iniziale e finale dei caratteri che vi appartengono. Per esempio, quando `Espressione` vale `prod(somma(5,3),div(10,diff(4,2)))` l'operatore è rappresentato dagli indici 0 e 3, il primo operando dagli indici 5 e 14 e il secondo dagli indici 16 e 32.

[2] Quanto segue è stato riadattato dalla lezione sulle liste senza procedere all'effettiva implementazione: si prega di segnalare eventuali errori o incongruenze.

Capitolo 11

Alberi binari di ricerca

Questo capitolo è dedicato a una struttura dati astratta nota come *dizionario*, e in particolare a una delle sue implementazioni principali, il così detto *albero binario di ricerca*. Il nome stesso suggerisce un forte legame con gli alberi binari descritti nel capitolo precedente. In effetti, c'è molto in comune nella struttura, ma lo scopo di questa struttura dati è completamente diverso da quello degli alberi binari visti precedentemente. Quelli rappresentavano strutture gerarchiche in sé (per esempio, le espressioni aritmetiche). Gli alberi binari di ricerca, invece, servono a descrivere insiemi privi di una struttura gerarchica, nei quali la gerarchia è inserita "a forza" per favorire la soluzione del problema della ricerca.

Che cos'è il problema della ricerca? Vi abbiamo già accennato in alcune lezioni di teoria parlando del problema dell'ordinamento, dato che ricerca e ordinamento sono due grandi problemi, spesso appaiati, ai quali si dedicano intere enciclopedie. Il problema della ricerca riguarda un *insieme universo* U, sul quale supponiamo sia definita una relazione di ordinamento totale. Si vuole rappresentare un sottoinsieme di U in modo da poter rispondere a domande come:

- questo elemento dell'insieme universo fa parte del sottoinsieme rappresentato?

Insomma, si vuole realizzare un test di appartenenza il più efficiente possibile. Abbiamo visto una situazione molto simile nell'esercizio del sottografo indotto. Nelle ultime fasi dell'esercizio, molti degli algoritmi implementati valutavano se un dato nodo appartenesse o no al sottoinsieme S che andava a indurre nel grafo dato il sottografo da determinare. Questo problema andava risolto nella maniera più efficiente possibile. Inizialmente, lo avevamo con una *ricerca sequenziale* su una tabella, cioè semplicemente scorrendo gli elementi del sottoinsieme (raccolti in una tabella di interi che conteneva gli indici dei nodi del sottoinsieme) e restituendo `vero` quando l'elemento corrente coincideva con l'elemento cercato, `falso` se si arrivava in fondo alla tabella senza trovarlo. Questo algoritmo costa ovviamente $\Theta(n_S)$ in tempo, e la rappresentazione del sottoinsieme costa $\Theta(n_S)$ in spazio. Dal punto di vista dello spazio, è ottimale; dal punto di vista del tempo, lascia un po' a desiderare. Ci eravamo anche baloccati con l'idea di applicare una *ricerca binaria* dopo aver introdotto un ordinamento per indici crescenti sugli elementi dell'insieme S, cioè sugli indici dei nodi. In lezioni di teoria si è visto che determinare l'appartenenza o meno di un elemento a una tabella ordinata costa tempo $\Theta(\log n)$, che è decisamente meglio. Non avevamo però applicato questo algoritmo perché avevamo trovato un'ulteriore possibilità: il *vettore di incidenza*, cioè un vettore di valori logici lungo come il numero dei nodi del grafo, in corrispondenza a ogni elemento del quale si ha il valore 1 (`vero`) o 0 (`falso`) secondo che quell'elemento appartenenesse o no al sottoinsieme. Questa implementazione permette di rispondere istantaneamente

Un dizionario T su un insieme universo U totalmente ordinato

- rappresenta un sottoinsieme finito di U: $\mathcal{T} \subseteq 2^U$
- consente di eseguire operazioni di ricerca, cioè di indicare se un dato elemento di U appartiene a T oppure no

Altre strutture già trattate possono svolgere queste funzioni:

- tabelle
- tabelle ordinate
- liste
- vettori di incidenza

ma presentano forti svantaggi:

- hanno scarsa efficienza temporale per alcune operazioni
  - ricerca in strutture non ordinate
  - inserimenti e cancellazioni in strutture ordinate
- hanno scarsa efficienza spaziale per insiemi universo U molto grandi (eventualmente, infiniti)

Gli alberi binari di ricerca (ABR) e le *tabelle hash* cercano di limitarli

Parleremo solo dei primi

Figura 11.1: Dizionari

alla domanda: basta leggere il valore del vettore di incidenza nella cella di indice corrispondente.

Sembrerebbe quindi che il problema della ricerca sia stato risolto nel modo migliore possibile, in $\Theta(1)$. Qual è il problema? Il problema è che questa soluzione ha un'efficienza spaziale scarsa nel momento in cui l'insieme universo è molto grande, perché il vettore occupa $\Theta(n)$ celle, tante quanti sono i nodi del grafo. Se l'insieme universo non fosse l'insieme dei nodi di un grafo, ma l'insieme dei numeri interi, sarebbe un insieme infinito. Potrebbe anche essere finito, ma non rappresentabile attraverso un vettore esplicito con la memoria disponibile. Questa strada ci viene preclusa. Rimangono le altre strade: la tabella, la tabella ordinata, la lista, che hanno un'occupazione di spazio ridotta all'essenziale ($\Theta(n_S)$). Però, il problema della ricerca viene risolto in tempo $\Theta(\log n_S)$ per la tabella ordinata, $\Theta(n_S)$ per la tabella e la lista. Non sono certamente complessità ideali, quanto meno le ultime due.

Inoltre, nel momento in cui si vogliono aggiungere e togliere elementi dal sottoinsieme, cioè si vuole renderlo dinamico (cosa che nel caso del grafo non succedeva), sorgono nuovi problemi. Le liste sono dinamiche, ma molto inefficienti temporalmente. Le tabelle, ordinate o no, sono dinamiche solo fino a un certo limite massimo, che potrebbe non essere chiaro a priori e potrebbe essere molto grande. Si veda il Capitolo XXX: la tabella era stata definita come come una struttura contenente un vettore dinamico e due interi, corrispondenti alla dimensione allocata e alla dimensione effettivamente usata. La differenza tra dimensione allocata e usata consentiva di variare le dimensioni della tabella, aggiungendo e cancellando elementi. L'aggiunta consisteva semplicemente nel incrementare il numero di elementi e scrivere nel nuovo posto l'elemento aggiunto. La cancellazione consisteva nel sovrascrivere l'elemento da togliere con l'ultimo e ridurre la dimensione della tabella stessa. Tutto questo funziona benissimo, salvo il fatto che in questo modo la tabella non è ordinata. Quindi, questa rappresentazione è inefficiente. Dal punto di vista della flessibilità spaziale, abbiamo un limite: quando il numero di elementi tocca la dimensione allocata, non si può aggiungere nulla, e abbiamo visto nel Capitolo YYY come affrontarlo. Dal punto di vista temporale, se si fanno inserimenti e cancellazioni, o si fanno scalando gli elementi (e allora hanno costo temporale, potenzialmente ingente) o si fanno inserendo elementi in fondo alla tabella e sostituendo gli elementi cancellati con l'ultimo (e allora, si perde l'ordinamento).

Quello che cerchiamo, quindi, è di realizzare una struttura che sia pienamente dinamica e conservi un test di appartenenza efficiente anche a seguito di inserimenti e cancellazioni. Una possibile soluzione, che introduce un compromesso differente fra occupazione spaziale e complessità temporale di inserimenti, cancellazioni e test di appartenenza è l'albero binario di ricerca. Un albero binario di ricerca è un'ipotetica struttura dati astratta che permette di fare queste operazioni. L'operazione fondamentale è la *ricerca*, che indicheremo con `member`. Rispetto alle dispense, ho aggiunto il suffisso `ABR` a tutti i nomi delle funzioni per evitare confusioni con gli alberi binari già discussi[1] La funzione `member` riceve un elemento dell'insieme base e un albero binario di ricerca e restituisce `vero` o `falso` secondo che l'elemento appartenga all'albero o no. La seconda operazione è il test di *vuotezza*, che determina se un dato albero binario di ricerca è vuoto oppure no. Questo sarà fatto con la funzione `ABRvuoto`. Dopo di che, avremo l'inserimento: la funzione `insertABR` riceve un elemento dell'insieme universo e un *ABR* e restituisce un *ABR* uguale a quello di partenza, ma con in più l'elemento aggiunto. Notate che, contrariamente al caso delle liste e degli alberi binari, non c'è un'indicazione di posizione: stiamo inserendo elementi in una struttura dati astratta che non ha una struttura interna

[1] Questo forse andrà rivisto: contraddice un po' l'assunto delle strutture dati astratte: le funzioni potrebbero essere implementate come tabelle hash.

Sia $\mathcal{T}$ l'insieme di tutti i possibili dizionari su U

I dizionari ammettono tipicamente le seguenti operazioni

- ricerca: dato un elemento e un dizionario, indica se l'elemento fa parte del dizionario

  $$\text{member} : U \times \mathcal{T} \to \mathbb{B}$$

  È l'operazione fondamentale di questa struttura dati

- verifica di vuotezza: dato un dizionario, indica se è vuoto

  $$\text{vuoto} : \mathcal{T} \to \mathbb{B} \qquad (\text{ovvero } \{0,1\})$$

- inserimento: dato un elemento e un dizionario, inserisce l'elemento nel dizionario

  $$\text{insert} : U \times \mathcal{T} \to \mathcal{T}$$

  Non c'è controllo sulla posizione dell'elemento

- cancellazione: dato un elemento e un dizionario, cancella l'elemento dal dizionario

  $$\text{delete} : U \times \mathcal{T} \to \mathcal{T}$$

Figura 11.2: Dizionari: operazioni

controllabile dall'esterno. L'operazione di *cancellazione* è eseguita dalla funzione `deleteABR`, che riceve un elemento dell'insieme base e un albero binario di ricerca, e restituisce un albero binario identico a quello di partenza, ma privo dell'elemento che è stato indicato. Anche qui, non viene fornita la posizione dell'elemento, ma direttamente l'elemento stesso.

Sorge spontanea una domanda: che succede se si cerca di inserire un elemento già presente nell'albero? E che succede se si cerca di cancellare un elemento che non appartiene all'albero? La risposta sarà parzialmente arbitraria, come al solito: si può decidere se stampare un messaggio di errore e terminare l'algoritmo, o comportarsi come se si trattasse di operazioni su insiemi matematici, e quindi in entrambi i casi lasciare il sottoinsieme invariato. Questa è la strada che abbiamo scelto e che percorreremo nell'esercizio pratico.

21:31 ci sono due operazioni che sfruttano il fatto che la albero binario di ricerca è definito tipicamente per insieme Universo ordinati gli ho detto E poi ho lasciato cadere il il tema che sull'universo sono insieme Universo tipicamente si richiede che esista una relazione di ordine Diciamo pure una relazione di ordine totale perché perché questa relazione di ordine permette di rendere la ricerca più semplice se state cercando una parola in un elenco confuso dovete scorrere tutto l'elenco non c'è niente da fare quindi applicare la ricerca sequenziale Ma se l'insieme Universo ordinato Cioè se le parole sono potenzialmente ordinabili in modo lessicografico bar alfabetico Allora abbiamo in mano la possibilità di fare una ricerca binaria cioè di trovare la parola sul dizionario senza leggere tutto il dizionario che evidentemente è folle Allora io vi ho detto all'inizio l'insieme universo non è detto che sia ordinato Però se non è ordinato l'unica ricerca possibile quella sequenziale quindi abbandoniamo ogni idea di strutture dati sofisticate quindi ipotizzare mo che è l'insieme Universo Si ha ordinato e questa struttura che noi andremo a imporre consentiva di determinare di rispondere alla domanda members elemento c'è o no E anche lì

cancellare elementi dalle insieme Spesso in maniera più efficiente ovviamente siccome insieme ordinato anche i sottoinsiemi sono ordinati quindi qualsiasi dizionario te qualunque albero binario di ricerca ti è un insieme totalmente ordinato e in questo modo è possibile definire delle funzioni minime massimo le quali ricevono un albero con le restituiscono in uscita un elemento dell'insieme Universo dell'evento minimo piuttosto che l'elemento massimo al solito possiamo chiederci cosa succeda nel caso particolare in cui laddove vuoto se la due volte Evidentemente non c'è un minimo ma si può restituire un valore fittizio che indicheremo come più infinito per dire il minimo peggiore che ci possa essere come Analogamente nel caso del massimo se l'albero è vuoto restituiremo un meno infinito convenzionale è tutto qui No ovviamente Siccome stiamo parlando di 0:24:10 Analogamente nel caso del massimo se l'albero è vuoto a Venturi Remo un meno infinito convenzionale è tutto qui No ovviamente Siccome stiamo parlando di strutture dati che poi vanno calate nella realtà di un processore avremmo bisogno di funzioni per la creazione e la distruzione di alberi binari vuoti di alberi binari di ricerca la creazione nei creolo vuoto la distruzione invece mi distrugge uno qualsiasi vuoto Non vuoto sottolinea ancora il fatto che negli alberi binari di ricerca contrariamente a vettori tabelle liste alberi binari non esiste un concetto di posizione nei rappresentato da un indice numerico non è rappresentato da un puntatore E allora la domanda è ma quando io dico di inserire o di togliere un elemento doveva quell'elemento e la risposta è che va in una posizione che è in tenuta nascosta all'utente e che è sulla base dell'ordine totale del insieme Universo u e quindi di riflesso dell'ordine totale che viene proiettato sul sottoinsieme che stiamo rappresentando con il lotto albero binario di ricerca veniamo allora Apriamo un pochino alla struttura dati astratta rimanendo abbastanza strati ancora ma capendo Perché si parla di albero binario di ricerca L'idea è di usare un albero binario Per rappresentare un sottoinsieme In che modo ogni nodo è un elemento di sottoinsieme come sempre useremo i numeri interi come rappresentanti di questo insieme così abbiamo un insieme totalmente ordinato in partenza allora un albero binario di ricerca è un albero binario Quindi è un albero nel quale ogni nodo ha due figli un figlio si inizia un figlio detto che sono alberi binari anch'essi A meno che l'altro non sia vuoto e poi c'è però una proprietà la proprietà 1 ep tutti i nodi sono associate informazioni diverse fra loro non ci sono doppioni Contrariamente alle liste hai detto e le tabelle gli alberi binari tutti i nodi sono diversi perché Perché l'unica cosa che noi vogliamo fare su un albero binario di ricerca è conoscere il minimo e il massimo e inserire cancellare e della volta la bruttezza ma soprattutto rispondere a test di appartenenza e non ha senso In questo caso avere dei doppioni semplicemente occupano spazio per niente ed è il motivo per cui quando inseriamo qualcosa che c'è già in realtà non lo inseriamo quindi Diversamente dai gli alberi binari in quelli finali di ricerca non ci sono doppioni ma proprietà due la più importante l'informazione associata un nodo deve essere strettamente successiva a tutte le informazioni associate ai nodo del sottoalbero di sinistra è strettamente precedente a tutte le informazioni associate in ordine sotto albero di destra è strettamente maggiore del 2 del 1 del 4 del 3 ed è strettamente minore del Lotto così il 2 è strettamente maggiori dell'uno ed è strettamente minore del 4 e del 3 il 4 è strettamente maggiore del 3 ed è strettamente minore di qualunque nodo stia nel sottoalbero destro che però è vuoto Analogamente l'uno e l'otto che sono foglie e non hanno sotto alberi sono strettamente maggiori strettamente minori dei loro sotto alberi infiniti arresti che sono vuoti e quindi si può dire qualunque cosa Ok Allora questa è la struttura e come vedete abbastanza facilmente istituisce una forte struttura sulla sul sul insieme quindi l'insieme apparentemente è un insieme qualunque contenente i numeri 6 24138 ma in realtà questi numeri non sono messi a casaccio sono messi in un modo tale da rispettare delle regole piuttosto strette queste regole vengono sfruttate per rispondere ai test di appartenenza per calcolare il minimo per calcolare il massimo in modo abbastanza intuitivi che indagheremo

un po' meglio poi nella seconda parte della lezione prima di passare a questa però discutiamo un attimino l'implementazione Come si fa implementare ovviamente trattandosi di un albero binario con L'unica differenza che c'è un'informazione associata ai nodi che non è una differenza perché succedeva anche per gli alberi binari la differenza sta nei vincoli che questa informazione deve rispettare Allora trattandosi di un albero binario tutte le implementazioni degli alberi binari sono valide sia quella a puntatori sia quella a Bettoni e indici noi vedremo solo l'implementazione a puntatori prenotazione appuntato riprese vede come albero un puntatore al nodo radice quindi una PR è un puntatore a nodo lo chiamiamo ABR per specificare che un albero binario di ricerca non un albero binario 0:29:00 Lo dice quindi una PR è un puntatore a nodo lo chiamiamo ABR per specificare che un albero binario di ricerca e non un albero binario generico e non c'è un non esiste un concetto di posizione o non esiste un tipo posizione non esiste una costante simbolica non Hood mette c'è la costante nostri che rappresenta l'albero vuoto Dopodiché un nodo è una struttura che contiene l'informazione nel campo a informazione Daddy un tipo u che deve però ammettere una ordinamento totale e tre puntatori a nodo che sono di sotto albero il puntatore alla radice del sottoalbero sinistro quello lavatrice il sottoalbero destro e il puntatore al nodo padre quindi esattamente identica a prima e a questo punto in realtà possiamo chiederci Come utilizzare questa struttura e Come realizzare le 24 e 8 che abbiamo visto in precedenza e questo è quello che vedremo nella seconda parte della lezione Spegni trovati nella seconda parte di questa tredicesima lezione dedicata agli alberi binari di ricerca andiamo ad affrontare un problema pratico che usa Questi alberi in particolare nella implementazione a puntatori che abbiamo introdotto nella nella prima parte di che cosa si tratta Allora si tratta In sostanza di realizzare una specie di gestore di insiemi di sottoinsiemi di numeri interi l'idea Qual è di avere un programma che chiameremo ricerca nel quale il quale andrà a caricare da file di testo una serie di numeri per esempio qui sotto abbiamo banale esempio la serie di numeri interi 95777 scritti in fila ho scritti in colonna non importa tanto la lettura esattamente la stessa e Dopodiché una volta caricati questi file andare a manipolare questo sotto insieme dei numeri interi in in quello delle possibili operazioni Quindi è un programma interattivo il programma si aspetta che l'utente digiti da tastiera delle operazioni e poi eseguire queste operazioni sul insieme di che operazioni si tratta la prima operazione è l'operazione member seguita da un numero intero è questa operazione chiede al programma di indicare se quel numero sta nel insieme oppure non sta nel insieme l'idea è che il programma dovrà rispondere stampando ho i doveri è il numero intero che è stato richiesto dal utente oppure i non in te nel caso in cui il numero non appartenga a quel insieme che stiamo gestendo dopodiché abbiamo delle operazioni che rendono l'insieme dinamico altrimenti banalmente si tratterebbe di mettere i numeri in un vettore e applicare al vettore la ricerca sequenziale o magari binaria dopo aver ordinato il vettore quindi le operazioni che non sono prevedibili a priori vengono fornite da tastiera dall'utente sono le operazioni di in Sert di un dato numero intero i.o.d. d'elite del dato numero intero dove lei chiede il programma di aggiungere questo numero al insieme e la Denise chiede di toglierlo Qui si apre la domanda Ma che succede se il numero che stiamo cercando di inserire già appartiene all'insieme e che succede se il numero che stiamo cercando di togliere non appartiene all insieme si può scegliere uno potrebbe ipotizzare una risposta con un errore addirittura uscita dal programma Si potrebbe ipotizzare invece che il programma segnali l'errore ma non esca Oppure si potrebbe ipotizzare che non si faccia nulla e che semplicemente si applichi la si applicheranno le regole le i principi della insiemistica per cui quando si aggiunge un elemento l'insieme A l'insieme rimane lo stesso e quando si toglie da un insieme che non un elemento da un insieme che non lo contiene l'insieme nello stesso e di questa la strada che seguiremo per la semplicità poi abbiamo una procedura di stampa la procedura di stampa chiede al programma di stampare gli

insieme non in un formato qualunque ma per valori crescenti li faccio notare che a causa di questa ipotesi che abbiamo fatto prima su l'inserimento di elementi che già ci sono lì insieme non ama i doppioni è un vero e proprio sotto insieme non è un sottoinsieme con ripetizioni come invece potrebbe succedere se la presentassimo con una lista con un tabella o con qualsiasi altra le strutture che abbiamo visto finora vogliamo elementi singoli Perché Perché come potete vedere le operazioni che dobbiamo eseguire su questi elementi non traggono nessuno 0:33:50 Di singoli Perché Perché come potete vedere le operazioni che dobbiamo eseguire su questi elementi non traggono nessun vantaggio dal fatto di averli i multipli quindi dobbiamo stampare poter stampare gli elementi nello stesso formato che abbiamo detto in precedenza per esempio tutti su una riga in realtà riformato d'ingresso gli ho detto è libero formato di uscita ovviamente deve essere codificato e diciamo che stamperemo tutto su una riga con spazi singoli fa un numero e l'altro mentre il comando enti chiedeva il programma di dire se l'insieme correntemente è vuoto oppure no nel programma dovrà semplicemente stampare a video ti è vuoto oppure ti non è vuoto ti è questo nome convenzionale è legato al fatto che useremo un albero binario di ricerca e l'albero inglese si dice three che altro altre due funzioni si chiederanno che danno il programma di fornire il minimo o il massimo del insieme come si tratta di insiemi sottoinsiemi finiti dell'insieme dei numeri interi hanno Certamente Un minimo è un Massimo e questo minimo massimo lo stamperemo Nel formato min ti fa parentesi = un al valore che viene effettivamente restituito oppure Max di ti fa parentesi uguale al valore dell'elemento massimo Infine c'è il comando Exit che chiede Finalmente il programma di uscire dobbiamo poi decidere che cosa fare nel caso in cui l'utente digiti un comando sbagliato eccetera eccetera Anche qui si potrebbe uscire quello che invece noi faremo sarà di segnalare all'utente che il comando non è stato riconosciuto detto ciò possiamo andare a vedere che cosa hai già disponibile dal punto di vista del codice È già disponibile un file ricerca Zero punto ci che fornisce le solite direttive generiche per poter gestire gli input output le relazioni con operativo e le stringhe c'è un questo contavo di aggiungerlo dopo c'è un la solita definizione della lunghezza di una stringa convenzionale di una riga e c'è un main con già contenente la funzione di interpretazione della riga di comando che è il più comune che abbiamo usato in questo corso cioè quella che va a leggersi il nome del file dei dati e lo inserisce in un opportuna variabile fai da te Dopodiché c'è scritto una traccia di quello che dobbiamo fare vieni a questo punto come potete facilmente immaginare L'idea sarebbe di mettere in pausa la registrazione riempire questa traccia col solita principio col solito approccio top-down e consiste nel definire delle funzioni che fanno delle cose ben definite dichiararle qua sopra e definire le ruote di sotto Dopodiché dobbiamo andare a inserire qualche cosa queste funzioni vuote C'è da dire che queste funzioni richiedono l'uso di un albero binario di ricerca e Questo significa che abbiamo bisogno di una libreria se non di funzioni definite qua dentro ma non sarebbe molto modulare di una libreria per gestire alberi binari di ricerca ora questa libreria è già fornita c'è un file di intestazione AVR. h che fornisce la definizione che abbiamo visto nei lucidi e un certo numero di funzioni che sono più o meno quelle che abbiamo visto nei lucidi dico più o meno perché in aggiunta abbiamo una funzione di stampa che nell'elenco dei lucidi non avevamo D'altra parte Ognuno di queste funzioni ha nel file ABR 0.5 un corpo vuoto o Tutt'al più ridotto alla restituzione di un valore di default che serve semplicemente per marcare il posto ed evitare che il compilatore dia dei messaggi di avvertimento nel momento in cui pompilia in tutto questo già compila perfettamente solo gli dicevo il primo passo consiste nell'andare a costruire questi corpi vuoti credo che sia una questione di pochi minuti dopodiché vi darò la mia soluzione riempiremo queste funzioni allora la prima cosa da fare Sostanzialmente è andare a recuperare dalle file dei dati i numeri e inserirli in un albero binario di ricerca questo suggerisce abbastanza agevolmente di andare da inserire in un albero te che cosa Sostanzialmente i dati del problema

raccogliendo Read All file stesso affinché questa funzione abbia un senso è necessario definire una variabile T la quale sarà Nient'altro che un albero 0:38:30 I dati del problema raccogliendo lì dal file stesso affinché questa funzione abbia un senso è necessario definire una variabile ti la quale sarà Nient'altro che un albero binario di ricerca e tipo ABR Ovviamente per avere un senso deve essere da qualche parte deve essere definito è definito nella libreria br. h e questo comporta l'inclusione di tale libreria fusione di avviare. h questo punto abbiamo tutte le definizioni che ci interessano Abbiamo anche già alcune funzioni e abbiamo anche i corpi vuoti di queste funzioni però dobbiamo compilare tutto quanto insieme cosa significa Significa che possiamo finalmente procedere a dichiarare il prototipo di questa funzione dicendo che la funzione caricabile restituisce un ABR che prende in ingresso una stringa di caratteri possiamo anche definire dichiarare definire qui sotto la funzione carica br vuota e a questo punto siamo tornati non situazione di programma che compila senza problemi terminiamo il main dobbiamo poi eseguire questi comandi Cosa vuol dire eseguire comandi e vuol dire chiamare una funzione esegue comandi che applica al albero ti che cosa i comandi che arrivano da tastiera e quindi non è un altro ingresso li prenderemo da tastiera e dopodiché ci possiamo chiedere se questa funzione che restituisca qualcosa linea di principio Visto che stiamo uscendo non dice nulla ma potremmo prepararsi all'idea che al termine magari dobbiamo fare qualche altra elaborazione Futura che non faremo in questo esercizio e quindi potremmo restituire d'albero stesso dopodiché si può discutere sul fatto che ti in realtà essendo un puntatore noi lo sappiamo Eh sì passata ingresso Ma viene anche modificato e quindi gli ho tenuto in uscita per cui queste più la scrittura astratta che non ciò che effettivamente avviene però diciamo possiamo permetterci di farlo e tutto sommato rende più chiaro il codice quindi possiamo procedere a dichiarare la nuova funzione questo esegue sul abrt i comandi forniti da tastiera Quindi vestiti chiamo 1 br eseguendo i comandi su un dato ABR e a questo punto possiamo riportare in basso il l'intestazione fornendo alla funzione un corpo vuoto di un terminato no come tutte le strutture dati dinamiche che sono stata in qualche modo create all'inizio alla fine dobbiamo liberarsene distruggendo Lo è la funzione già esiste Si chiama distrugge ADR e andiamo a distruggerla passando la questa volta per indirizzo detto che viene modificata Dalla funzione stessa Questo sostanzialmente chiude la prima parte del programma tant'è che possiamo anche fare un tentativo di compilazione del codice con tutte le opzioni di sicurezza e dobbiamo compilare ricerca zero.com 0.5 per produrre il nostro file eseguibile abbiamo due avvertimenti di quale ci ricordano che le due funzioni che abbiamo introdotto in realtà devono restituire un albero e allora bisogna inventarsi se non vogliamo vedere con gli avvertimenti un albero fittizio da restituire e nel primo caso potrebbe essere un albero vuoto visto che non stiamo facendo nulla mentre nel secondo potrebbe essere un albero vuoto ma forse più ragionevole restituire l'albero attività che ti entra e non succede niente dentro la funzione quindi a questo punto vi compilando abbiamo un Percorso netto possiamo anche andare a lanciare il programma che evidentemente se gli diamo il numero sbagliato di momenti segnal volevo le Ma se invece gli diamo gli argomenti Giusti non fa nulla perché i tutte le funzioni sono vuote A questo punto è ragionevole procedere implementare queste due funzioni che questa è la seconda fase del esercizio nell'implementazione procediamo al solito ragionando e facendo un 0:43:10 E questa è la seconda fase del esercizio nell'implementazione procediamo al solito ragionando e facendo un po' di commenti che danno lo schema del della funzione stessa dopodiché vi lascerò la possibilità di risolverlo per i fatti vostri In sostanza che cosa deve fare la funzione carica VR funzione carica avviare deve Per prima cosa aprire il file fidati questo è abbastanza ovvio e siccome Apri file evidentemente l'ultima cosa che deve fare è chiuderlo e questo lo segnamo così non ce lo dimentichiamo Una volta aperto il file dovrà andare a leggere questo file tanto per darci per avere un'idea apriamo questo file di esempio così vediamo come è fatto è

una banale sequenza di numeri interi quindi non sa niente di difficile ma capite che si tratterà di eseguire un ciclo di lettura Quindi avremo un qualche cosa del genere finché ci sono nuovi interi Che cos'è che dobbiamo fare qui dentro il nostro ciclo Lo devo andare a In sostanza leggere l'intero corrente e andare a inserirlo nella nell'albero stesso quindi inserisce l'intero corrente nel ABR altro ma sostanzialmente No salgo il fatto che per poter inserire un intero in una br evidentemente dobbiamo crearlo e ci siamo dimenticati di farlo ma l'albero andrà creato prima del ciclo quindi la prima cosa da fare Sara creare un ABR vuoto questo è effettivamente la logica del programma a questo punto direi che potete proseguire per i fatti vostri e io fra un attimo vi do la mia soluzione questa soluzione consiste nel ovviamente dichiarare una variabile puntatore a file utilizzarla aprendo i file dei dati in modalità lettura e verificando che l'apertura sia andata a buon fine se per caso l'apertura non è andata a buon fine è un buon momento per interrompersi e segnalare qualcosa l'utente per capire che cosa è successo È stato un errore nell'apertura di un dato file e segnalare il nome del file aiuta l'utente a cercare di capire che cosa è successo già che ci siamo possiamo procedere immediatamente alla chiusura del file in uscita poi come si fa a creare una birre vuoto per fortuna esiste una funzione che lo fa per noi quindi direttamente ce la Copiamo questa funzione restituisce un ABR qui diamo un nome che potrebbe essere ti e ovviamente questo APR va dichiarato Perfetto Non ci sono argomenti perché tanto la birre vuoto come facciamo a leggere una sequenza di numeri interi ci sarà Ovviamente un Wild ci sarà ovviamente una funzione che legge e a seconda di quello che ha detto potremmo decidere se interrompere o non interrompere il nostro file e la funzione di lettura principe e la funzione f-scan F contiene fresca F applicata al file AFP Siccome dobbiamo leggere i numeri interi viene buona la specifica per 100 di dove andiamo a scrivere questi numeri e Diamogli un nome Allora questi numeri chiamiamo noi per esempio andiamo a leggere nella variabile e la f-scan FC restituisce il numero di oggetti che ha riconosciuto e assegnato a variabili fintanto che questo numero è uno va Tutto bene Quando il numero è 011 Reo che ne fai c'era qualcosa che non è stato riconosciuto come numero intero per esempio una parola o invece vuol dire che sia arrivati alla fine del file e quindi bisogna effettivamente uscire in entrambi i casi vale la pena di uscire e basta quindi questo è la condizione di permanenza nel ciclo fatto ciò bisogna inserire questo numero in nella br Come si fa a inserire un numero in una br con la funzione insert ABR questa funzione riceve in te voi che riceve l'albero ti restituisce l'albero e quindi abbiamo il nostro 0:48:00 E riceve l'albero ti e restituisce l'albero ti e quindi abbiamo il nostro programma E questo dovrebbe risolvere la funzione carica ABR possiamo anche provare a valutare la correttezza sintattica per vedere se tutto è andato bene e sembrerebbe che sintatticamente vado a posto ovviamente non succede nulla perché tutte queste funzioni sono vuote L'unica cosa che abbiamo aggiunto apertura e chiusura del file che quantomeno esiste se Infatti avessimo scritto sbagliato il nome del file Ecco che ci sarebbe stata se la segnalazione di errore Invia un buon test Comunque ora dobbiamo eseguire i comandi Come si fa a eseguire i comandi Questo è un pochino più sofisticato Nel senso che ciò che dobbiamo fare è leggere continuamente da tastiera ciò che l'utente scrive e interpretare i comandi e riconoscere i comandi e poi eseguirli evidentemente un ciclo è un ciclo interattivo con l'utente Come si può fare a impostare questa situazione e allora procediamo al solito per commenti e poi vi lascerò la possibilità di risolvere il programma l'esercizio e poi vedremo la soluzione Allora in sostanza che cosa si tratta si tratta di dire che finché l'utente Inserisci comandi o meglio finché l'utente non indica di terminare e questo lo farà scrivendo The Great finché questo va avanti che succede abbiamo un ciclo in questo ciclo che cosa faremo la prima cosa che dobbiamo fare è leggere il comando poi dobbiamo interpretarlo e poi dovremmo eseguirlo tutto qua allora è una buona norma dovendo interagire con l'utente comunicargli che siamo pronti a eseguire i suoi comandi cioè non c'è niente di più frustrante che vedeva il

programma fermo e non si capisce se è fermo perché sei bloccato sei fermo perché sta lavorando facendo una cosa complicata o se è fermo perché è in attesa di un qualche qualche istruzione da parte della tastiera Allora probabilmente la cosa più sensata più umana e quella di stampare un messaggio che dica Dammi questo comando è il modo più semplice di stampare il cosiddetto pront Come dice il gergo informatico che è quello che da terminale è sostanzialmente questo Maggiore qui che mi dice siamo sono pronto a eseguire i tuoi comandi Dopodiché possiamo effettivamente procedere a leggere il comando dell'utente in qualche modo Dopodiché dovremmo identificare il comando dell'utente e eseguirlo possiamo anche vedere se sono due istituzioni separate ho la stessa adesso questa Evidentemente è tentativo questa serie di commenti esegue il comando quindi la mia indicazione ora è di mettere in pausa e provare a realizzare questo pezzo di codice è abbastanza semplice risolvendo i problemi che affronta e troverete Riavvia la mia soluzione e poi la seguente Come si fa a sapere che l'utente ha detto di terminare la condizione piuttosto sofisticata vuol dire che sia letto un qualcosa che questo qualcosa Exit il ciclo a condizione iniziale condizione finale sembrerebbe condizione finale più ragionevole però tanto per variare visto che comunque i cicli sono mutuamente intercambiabili si può sempre trasformare in maniera meccanica un ciclo condizione finale in una condizione iniziale eccetera eccetera faccio una implementazione condizione iniziale tanto per fare questo lo ottengo con una variabile logica fine che ovviamente all'inizio sarà falsa perché non voglio terminare Dopodiché mi dirò ma finché questa variabile rimane falsa a Possiamo e dobbiamo procedere a eseguire questo ciclo di operazioni Quindi qua dentro dobbiamo poi gestire la variabile fine mettendo la Vero inopportune situazioni la prima cosa che bisognava fare è stampare il pronto quindi dovremmo stampare a video questo famoso Maggiore con lo spazio giusto per non cominciare a scrivere attaccati a questo punto dovremmo leggere un comando Che cos'è un comando comando e quello che l'utente batte prima di scrivere invio Quindi è un qualcosa che terminato da un invio è una riga verrebbe spontaneo usare la funzione Get S che effettivamente la funzione corretta te lo diciamo che si consiglia da più parti di usare una funzione un pochino più sicura 0:52:50 Da più parti di usare una funzione un pochino più sicura che la funzione F Get S perché FTS più sicura è che vi ricordo che questa funzione che in realtà lavora su un file e la domanda è che fai stiamo usando poi ci rispondiamo a questa domanda come secondo parametro ha il numero massimo di caratteri da leggere il problema qui è il fatto che sei comando è molto lungo potrebbe sforare la dimensione che non gli assegniamo al comando quindi dobbiamo anche ricordarci del comando sarà una stringa di un'opportuna lunghezza E io continuo a usare la stessa lunghezza voulent convenzionale Quindi non dobbiamo leggere più di questo numero di caratteri questo garantisce l'uso delle FS con questo parametro garantisce che sia anche l'utente scrive 1000 caratteri in realtà non ne vengono letti più di 256 quindi non si va a sforare la stringa comando ho fatto andiamo a scrivere il risultato nella stringa stessa ora possiamo rispondere alla domanda ma il file che file è non è un file di testo No però dovete ricordarvi che l'input la out e lo standard è vero standard input standard output standard è vero sono tutti streaming stream Scusate esattamente come i file e quindi funzionano alla stessa maniera Tanto è vero che quando stampiamo un messaggio a video in errore stiamo stampando ancora print-f ma con la F Print F diretta sullo standard ero Analogamente questa FTS che riceve dallo standard input e questo sistema le cose a questo punto noi dobbiamo riconoscere il comando Come si fa a riconoscere il comando è qui il comando non è uno sono tanti ci sono alcuni comandi che hanno un argomento e quindi sono costituiti da due parole La prima è una parola e la seconda è un numero intero altri sono costituiti semplicemente da parole questo Cosa significa significa che noi potremmo fare varie cose lavorando sulla stringa comando Vi propongo una possibilità ci sono diverse implementazioni questa implementazione che Vi propongo divide le tre i tre le tre istituzioni che hanno un parametro dalle 5

che non ne hanno e quindi si divide ciò che contiene le stringhe che contengono due oggetti da quelle che ne contengono uno abbia mente da quelle che ne contengono 0 o tre o quattro che sono chiaramente dei risultati sbagliati dei comandi sbagliati a cui risponderemo diventi 100 che il comando non è stato riconosciuto quindi la prima cosa da fare è andare a estrarre da comando il contenuto Come possiamo farlo con una funzione s-scan F che lavora su una stringa comando e che nella stringa da cercare sicuramente 10:57:36 Vediamo i vari casi e sottocasi potrebbero essere due potrebbe essere una potrebbe essere un numero diverso ha stretto di rigor di logica più di due non possono esserne perché qui abbiamo per 100s percento.di quindi potrebbe essere 0 o potrebbe esserci una condizione di errore Tutto questo non ci interessa qualunque cosa succeda che non è due o uno è chiaramente un errore E allora Che cosa succede se c'è un errore che segnaliamo l'errore c'è stampiamo il fatto che il comando non è stato riconosciuto e andiamo a capo niente! questa volta allora Cosa succede se invece abbiamo riconosciuto due cose correttamente e succede che possiamo eseguire il comando per quello dicevo in realtà qua dentro sia in un caso sia nell'altro abbiamo questo pezzo di questa funzione è in realtà non è una funzione su tante funzioni la seconda la seconda dello specifico comando Come si fa a riconoscere il comando ce l'abbiamo in hop e tutte le volte dobbiamo riconoscere una stringa viene buona la funzione screen con Peppe possiamo comprare hop per esempio con member se il risultato è zero l'abbiamo riconosciuta e possiamo fare un certo insieme di cose poi vediamo che cosa Altrimenti potrebbe anche essere sempre nel caso con = 2 potrebbe essere che abbiamo determinato la funzione insert oppure che abbiamo determinato la funzione del it quindi Qui abbiamo il cavo insert e poi abbiamo il caso delitto tre casi mutuamente esclusivi poi cosa succede e può anche darsi che abbiamo scritto qualcos'altro che non c'entra niente ma in questo caso già sappiamo che dobbiamo restituire comando non riconosciuto Cosa succede se abbiamo letto member succede che abbiamo letto member che abbiamo un numero intero dobbiamo semplicemente eseguire l'operazione eseguire l'operazione significa testare questa condizione qui Dopodiché il testo dell'esercizio dice che a seconda che Nember sia vero falsa Scriveremo in Ti ho poi non finti Sulla base del risultato di member come si può fare tutto questo ci vogliamo essere molto compatti proviamo potremmo dire che bisogna stampare allora che si tratta di stampare a video una stringa la quale stringa è sto facendo una cosa abbastanza involuta ma è una occasione di vedere un costrutto che ogni tanto si usa in Cina a seconda che la funzione member sia vero falsa quindi stiamo testando member ABR applicato alla numero e all'albero ti la seconda che mi sembra sia vero o falso Ecco che sei vero Scriveremo No ho sbagliato scusate mi sono lanciato in una variante che mi sembrava promettente Ma qui abbiamo bisogno di scrivere un numero che non conosciamo quindi facciamo la nella maniera standard nella maniera standard si tratta semplicemente di dire che a seconda che membro sia vero o falso se member è vero andremo a stampare la stringa andremo stampare la stringa numero in t dov'è Il numero è i altrimenti invece andremo a stampare l'altra stringa per cui il numero non è in perfetto ho cercato di fare tutto in un'unica operazione ma spesso non è il caso cosa succede nel caso in cui la operazione fosse una in Sert ancora più semplice perché se l'operazione è una incerta non dobbiamo stampare nulla dobbiamo semplicemente andare a scrivere dobbiamo semplicemente andare ad applicare l'operazione di insert' a quell'operazione restituisce l'albero ti ricevendo ingresso il numero lì è l'albero ti è la stessa cosa in effetti succede nel caso della operazione d'elite ABR Quindi questi due casi è estremamente semplice Ci rimangono le altre cinque operazioni la struttura sarà molto semplice con 1:02:20

Nel caso della operazione d'elite ABR Quindi questi due casi è estremamente semplice Ci rimangono le altre cinque operazioni la struttura sarà molto semplice con solita cascata di sto in compair riusciamo a distinguere tutti i vari casi nell'or-

dine abbiamo come primo caso Print poi abbiamo qui vedremo che cosa succede poi abbiamo il caso Hell's con la funzione MT e poi avremo le varie funzioni min Max min Max in ciascuna dobbiamo reagire nel modo appropriato che cosa si fa quando c'è il comando Print molto semplice Esiste la funzione stampa br qui dobbiamo fare una piccola ipotesi l'ipotesi è capire se la stampa se la funzione Print ABR che non abbiamo ancora realizzato stampa effettivamente tutto l'albero e poi va a capo oppure semplicemente stampa l'albero e non va a capo nella mia soluzione ho ipotizzato che non andasse a capo alla fine e quindi per avere una scritta pulita a video andremo a capo noi esplicitamente cosa succede nel caso ehm ti succede una cosa molto simile al caso member Cioè a seconda del risultato della funzione avviare vuoto Come stampare una cosa piuttosto che un'altra quindi mi ispiro pesantemente a questa funzione è quello che faremo e andare a vedere se la br vuoto e ovviamente AVR lavora solo su t non lavora su insuti in quel caso diremo che ti è vuoto e ovviamente non abbiamo un argomento gli mentre nell'altro caso dovremmo dire nel caso opposto dovevo dire che ti non è vuoto Abbiamo sistemato anche la MT minimo e massimo Euro 1000 Massimo sono molto semplici è una singola stampa nella quale bisogna scrivere quindi ti ho macity uguale a un numero quindi andremo a dire come funziona il Print F la chiamata Print F che Minh city è uguale a un dato numero intero questo numero Che cos'è questo numero ehm in ABR dt è assolutamente analoga è la situazione di Max BT solo che chiamiamo la funzione Max AB rdt infinite Exit cosa dice dice di terminare e noi abbiamo già visto che per terminare Bisogna che la variabile fine diventi vera quindi molto semplicemente in questo caso ci limiteremo a dire che fine è uguale a tu questa è una delle possibili implementazioni di tutto il procedimento Allora se adesso noi andiamo a compilare questo codice abbiamo già dato numero di errori per esempio abbiamo un nella riga 113 ho invertito le posizioni della stringa e del file questo perché in realtà è un po' incoerente rispetto alla fnf dove la prima cosa è il file e poi c'è la stringa quindi esattamente il contrario qui possiamo ricompilare ed era sostanzialmente solo quelle nuove lì perché c'era l'argomento 1 poi c'era l'argomento 362 argomenti sono stati scambiati E qui c'è una questione di tipo di argomenti cosa succede Se lanciamo il programma poco o nulla compare il nostro pront possiamo scrivere varie cose e però non abbiamo nessun sostanziale risultato so se lo scriviamo nulla il conteggio degli argomenti a zero e quindi il comando effettivamente non è riconosciuto D'altra parte se andiamo a scrivere qualche cosa come Print 3 il comando non è riconosciuto perché ha due argomenti e print anni dovrebbe avere una sola provincia da sola non fa assolutamente nulla e Exit si fa uscire e quindi siamo abbastanza soddisfatti tutto sommato non è quello che vogliamo ottenere ma ci stiamo muovendo nella direzione giusta 1:07:10 Pizza ci fa uscire e quindi siamo abbastanza soddisfatti tutto sommato non è quello che vogliamo ottenere ma ci stiamo muovendo nella direzione giusta allora questo punto che cosa sostanzialmente bisogna fare procedendo in modalità top down bisogna andare a riempire tutte le funzioni che abbiamo chiamato che sono tutte funzioni di libreria in sostanza Quindi abbiamo abbiamo messo tanto abbiamo finito il nostro programma se la libreria fosse completa il nostro programma di funzionerebbe tranquillamente passiamo all'implementazione della libreria La libreria contiene 9 funzioni 24689 che andremo a incrementare per parti non tutte insieme da Cosa Partiamo ci sono tre funzioni che sono la creazione la distruzione e abbiamo è vuoto Le quali sono assolutamente topologiche cioè consistono banalmente nel fatto di andare a manipolare gli elementi del dell' albero binario di ricerca senza avere nessuna impatto senza dipendere in alcun modo dal informazione contenuta quindi sono assolutamente identiche a quelle dell' albero binario normale per cui questa parte mi dovrebbe essere particolarmente semplice mi faccio pena notare che mentre che aveva brbm vuoto sono davvero identiche alle funzioni di albero binario la distruzione è un po' diversa per il semplice motivo che nel caso dell' albero binario andiamo a vederselo nel caso dell' albero binario nel caso dell' albero

binario noi avevamo questo è albero sì che quello che effettivamente ci serve Noi avevamo precedente avevamo la distruzione vi tocca ricondotta alla cancellazione di un sotto albero e qui non abbiamo operazione di cancellazione del sottoalbero Per cui dobbiamo fare un pochino di rimaneggiamento per andare a recuperare tutto ciò che ci serve ma la struttura esattamente la stessa allora passiamo Che cosa vuol dire creare una br vuoto semplice vuol dire creare un altro vuoto e l'altro è vuoto si chiama nutri per cui non c'è da fare niente questa questo comando che era stato dato di default È già buono che cosa vuol dire controllare che una breve sia vuoto o no qui abbiamo scritto Trudi default È una breve non è sempre vuoto Come si fa a sapere se un albero binario di ricerca è vuoto esattamente come si fa a sapere se un albero binario è vuoto cioè si va a vedere se la radice è il puntatore nulla ovvero il puntatore il valore nutri e quindi la implementazione è rigorosamente la stessa nel caso degli alberi binari si tratta semplicemente di valutare se ti è uguale oppure no passiamo finalmente alla questione della distruzione Allora d'istruzioni consisterebbe nel cancellare il sottoalbero dell'albero ti che parte dalla posizione PT Cioè dalla posizione * Pitti Cioè dalla radice Nel nostro caso possiamo andare direttamente qua sotto e dire come si fa a cancellare un albero cancellare un albero Allora prima di tutto la se l'albero è vuoto Non c'è niente da fare e se non è vuoto se non è vuoto la cancellazione veniva fatta con la visita in postordine cancellando il sottoalbero sinistro poi sottoalbero destro e poi del locando il nodo radice e settando il puntatore a 0 questa cosa può essere tranquillamente ripetuta Quindi ancora una volta quello che possiamo fare dire che se per caso L'albero è vuoto Aldo vuoto potremmo chiamare questa funzione ma non ha molto senso siamo dentro nella libreria abbiamo accesso a tutte le i dettagli e le tecnicalità di implementazione quindi direttamente Scriveremo che se per caso l'albero ti che è* PT è un albero vuoto Non dobbiamo fare nulla Se invece non è vuoto Ecco che dobbiamo distruggere i due sotto alberi e quindi possiamo passare a distruggere la br Qual è l'ampiezza è di sinistra Allora come si chiama il sotto albero di sinistra di ti si chiama * PT fra parentesi tonde perché poi la freccia prevarrebbe su l'asterisco e non vogliamo che prevalga quindi dobbiamo fare tondi * PT per prendere l'albero ti freccia per andare a prendere l'oggetto puntato in campo dell'oggetto puntato e ti interessa In particolare il campo TS Dopodiché di questo sotto albero ci interessa non sotto albero ma l'indirizzo deve essere passato per indirizzo e quindi aggiungiamo 1 e è molto con dire contorto ma è assolutamente la stessa cosa che facevamo 1:12:00 Tanto per indirizzo e quindi aggiungiamo 1 e è molto dire contorto ma è assolutamente la stessa cosa che facevamo nel caso dell' albero binario Quindi se l'avete capitali l'avete capita anche qua stessa cosa facciamo sull'albero destro e poi possiamo procedere ad e al locale che cosa la radice che è l'oggetto puntato da PT infine possiamo andare a dire che * pipì è uguale annulla o hanno Tree in maniera tale che effettivamente anche in uscita l'albero risulti completamente distrutto risulti un puntatore a 0 altrimenti il puntatore continuerebbe a puntare al nodo radice che pure stato del locato questa cosa si sistema le tre operazioni relative alla gestione topologica del nostro albero quindi possiamo limitarci a valutare se sintatticamente è tutto corretto Dopodiché non dovrebbe cambiare nulla nel senso che in sostanza abbiamo solo aggiunto creazioni che non fa nulla una distruzione che opera su un albero vuoto quindi non fa nulla e dei test che davano sempre risultato positivo perché ovviamente L'albero è vuoto per cui non val la pena neanche di provare di nuovo a lanciare il programma quello che vale effettivamente la pena di fare invece è andare a esplorare la successiva funzione Quale funzione si potrebbe pensare di lanciarsi sulla stampa o full member eccetera Ma è chiaro che finché l'albero non è riempito non vedremo nulla quindi forse conviene concentrarsi sulla funzione di inserimento anche se è una funzione vela attivamente sofistik e questo è quello che andiamo effettivamente a fare andiamo a lavorare sulla funzione insert ABR Come funziona l'inserimento e qui non possiamo ispirarci alla albero binario nell' albero binario

noi non avevamo un inserimento abbastanza particolare che consisteva Eccolo qua nella costruzione di un albero a partire da due sotto alberi dati con l'aggiunta di una nuova radice quindi c'è bisogno di due sotto alberi che venivano messi insieme aggiungendo una radice nel caso dell' albero binario di ricerca questa cosa è assolutamente impossibile perché perché la posizione del nodo che stiamo aggiungendo non è sotto il nostro controllo non possiamo decidere che il l'elemento i che stiamo aggiungendo vada in radice l'elemento gli deve andare dove deve andare a seconda della sua del suo valore in maniera da conservare la proprietà fondamentale degli alberi binari di ricerca secondo cui radice ogni modo interno il nodo dell'albero è successivo strettamente ai nodi del sottoalbero sinistro e precedente strettamente ai nodi del sottoalbero destro e allora come si fa a inserire un nuovo elemento in un albero binario di ricerca vedremo che in qualche modo questa costruisce albero ci viene buono ancora qui a fare tutto un discorso allora andiamo un attimo a vedere nel nei lucidi ispiriamoci a questo esempio come si farebbe a introdurre un nuovo numero per esempio 5 oppure 10 in questo albero è chiaro che 5 A meno che si voglia ristrutturare l'intero albero non può andare in radice l'unico modo che per cui si può appendere aggiungere un nuovo elemento a un albero così costituito e appendendolo a uno dei nodi che non ha due figli quindi si può prendono le foglie 138 Oppure si potrebbe appenderlo al mondo quato come figlio di destra effettivamente questa è la posizione unica è la posizione valida Perché Perché 5 successiva quattro e quindi deve stare nel suo sotto l'albero di destra cioè se ci si ragiona un attimo ci si rende conto che in realtà qualunque numero entri esiste una sola posizione che è ragionevole per quel numero a meno che non si vogliano rimescolare e spostare le posizioni dei vari nodi Perché Perché il numero che entra nel momento in cui entra e si può vedere se è precedente successivo o caso particolare coincidente con la radice cioè coincidente vuol dire che il numero già c'è quindi siamo a posto Se precedente vuol dire che va a sinistra se è successivo vuol dire che va a destra e questa cosa non vale solo per la radice Ma vale in tutti i nodi dell'albero stesso fino a che si arrivi a una posizione in cui uno di questi sotto alberi sinistro destro è vuoto non uno il sottoalbero in cui bisogna andare inserire nuovo nodo è vuoto ma allora il nodo stesso si candida a diventare sotto l'albero c'è in sostanza l'inserimento avviene con la creazione di un nuovo sotto albero che è fatto di un solo nodo quel modo lì che stiamo aggiungendo e con il suo è il con il suo apprendimento col fatto di appenderlo in un 1:16:50 Stiamo aggiungendo e come suo è il con il suo apprendimento col fatto di appenderlo in un'opportuna posizione come figlio di sinistra o come il figlio di destra di un opportuno nodo dell'albero dato e l'opportuno deriva dalla relazione di ordine totale sugli elementi dell'insieme Universo in particolare su un elemento che stiamo aggiungendo sugli elementi del dizionario quindi notate che c'è anche qui la creazione di un nuovo albero molto semplice in realtà un albero che ha una solo nodo non un albero costituito da due sotto alberi Appesi a un solo nodo ma abbiamo già usato questa funzione costruisce albero funzione Qui abbiamo già usata nel caso di albero costituito da una sola radice è sufficiente che i due sotto alberi sinistro adesso siano vuoti quindi questa è un pochino l'idea c'è andare a trovare la posizione giusta costruire il nuovo albero e appenderlo nella posizione giusta prima di farlo vi faccio un caffè una premessa che vale per tutte le funzioni successive la premessa è che generalmente Nei laboratori Io cerco di adeguarmi al massimo possibile alla notazione alla nomenclatura della delle dispense di teoria questa cosa non è sempre possibile per esempio nel caso delle liste abbiamo visto che siccome le dispense di teoria ipotizzano un linguaggio in cui i parametri vengono passati per indirizzo 20-inch vengono passati per valore ci sono dei punti un po' critici e in particolare l'inserimento in una lista e quando io Inserisco una posizione in una lista il puntatore a quella posizione è l'elemento nuovo e quello vecchio a maggior ragione della cancellazione abbiamo abbiamo già discusso queste cose e gli ho già detto che a volte non si riesce a essere del tutto coerenti e di conseguenza ci si scolla

dalle dispense di teoria questa cosa succede in particolare in questa lezione questa la premessa Anche perché ci sono due aspetti ed è che le dispense di teoria perlomeno nella versione nell'ultima versione che Ho consultato ipotizzano quasi sempre che gli altri su cui si lavora siano non vuoti e che gli elementi che vengono aggiunti eventualmente siano elementi davvero nuovi e gli elementi che vengono cancellate siano elementi che ci sono nell'albero mentre io queste ipotesi semplificative non le farò gestivo un codice che affronta tutti i casi possibili e questo leggermente complica le cose quindi le differenze che trovate sono dovute in buona parte a questi aspetti in piccola parte anche piccole questioni di gusto vero Non credo che sia un enorme problema lo ha detto ciò che proviamo a entrare nel discorso la nostra funzione inserta br che cosa deve fare dovrebbe andare a capire Qual è la posizione corretta nell'albero Dopodiché dovrebbe costruire un nuovo albero binario di ricerca con un solo nodo che contiene l'informazione nuova giunta e appendere questo nuovo albero come figlio di sinistra o di destra alla posizione Al nodo che è stato raggiunto nella ricerca stessa per prima cosa consideriamo il fatto che piova l'albero potrebbe essere vuoto cioè la funzione sarà la funzione ricorsiva per semplificarci la vita perché la struttura dell'albero di ricerca è ricorsiva e ci sarà un caso base e questo caso base Sarà che l'albero è vuoto domanda Che cosa bisogna fare quando un albero è vuoto e questo caso è piuttosto semplice perché se l'albero è vuoto vuol dire In sostanza che dobbiamo costruire un nuovo albero costituito dalla informazioni aggiunta fornita e questo nuovo albero devi andare a prendere il posto dell'albero vuoto quindi procediamo di conseguenza se per caso l'albero ti è un albero vuoto Ecco che la nostra operazione consiste nel sostituire il corrente valore dell'albero ti con quello ottenuto da una funzione di costruzione di una costruzione di un albero generico la quale costruzione opera non superando operatore operando Ma semplicemente su un numero intero i ricevendo come sotto alberi di sinistra a destra due sotto alberi vuoti Basta queste quanto Ovviamente questo richiede di costruire la funzione costruisce avviare che andremo a costruire sulla falsariga di costruisce albero altrimenti che cosa succede eh altrimenti succedono varie cose bisogna capire se andare a sinistra o se andare a destra Ma come vi dicevo ammettiamo anche la possibilità di entrare nell' albero col numero 6 e scoprire che in realtà in radice questo numero già c'è in radice o nella radice del sotto altro su cui stiamo lavorando questa cosa 1:21:40 Numero 6 e scoprire che in realtà in radice questo numero già c'è in radice o nella radice del sotto altro su cui stiamo lavorando per cosa vale ricorsivamente su qualunque sotto albero binario di ricerca stiamo sul quale stiamo lavorando quindi in effetti abbiamo un altro caso base che però è un caso base abbastanza particolari nel quale non succede nulla io scrivo poi cancello se per caso i è identico a ti freccia Attenzione qui un'altra piccola modifica è un po' fastidioso vedere tii per cui ho qua scriveva Val e questo per adesso non l'abbiamo usato quindi non si ripete da nessuna parte Se per caso i è uguale attiva al In sostanza non dobbiamo fare nulla Questo è un secondo caso base quello in cui l'oggetto da aggiungere è identico alla radice ovviamente non dovendo fare nulla neanche lo scriviamo Quindi possiamo tranquillamente cancellare questa parte e sostituirla con i due casi ricorsivi col caso i minore etival e col caso i ma giovedì ti va che andiamo ad aggiungere notate che devo scriverli entrambi nel senso che questo IV finale va lasciato proprio perché altrimenti in questo Elsa andrai a ricomprendere anche il caso in cui è uguale attiva al per cui Questi sono i due casi di corsi in questi due casi ricorsivi che cosa dobbiamo fare e dobbiamo sostanzialmente andare ad aggiungere i a sottoalbero sinistro oppure sottoalbero destro per cui sembrerebbe una cosa molto semplice sembrerebbe una banale chiamata e però adesso mi rendo conto di aver fatto una stupidaggine cioè di aver scritto tutto dentro member mentre invece dobbiamo scrivere tutto dentro in Sert stavo guardando altre cose Perfetto Allora dobbiamo andare a chiamare questa cosa in insert ABR ma il numero ID o andiamo inserire non intima lo Inseriamo nel sottoalbero sinistro Analogamente qua di sotto andremo inserire

tutto nel sottoalbero invece destro domanda siamo a posto così Apparentemente sì nel senso che effettivamente in questo modo Noi abbiamo dobbiamo abbiamo fatto che cosa abbiamo inserito questo elemento i all'interno del sottoalbero di sinistra ma se ci va giornate un attimo Vi renderete conto che questo non sa assolutamente che il nuovo albero diciamo che in che maniera avviene questa aggiunta semplice scendendo sotto albero sinistro Adesso finisco adesso eccetera eccetera fino a che si arriva un albero vuoto Cioè fino a che si arriva questo caso base in questo caso base avviene una allocazione del nuovo albero che viene appeso al vecchio in che maniera sostanzialmente assegnando all'albero ti sto facendo un altro errore Infatti avviene sostanzialmente in che maniera prendendo questo nuovo albero e assegnando l'ho come figlio di sinistra all'albero ti quindi ne stiamo dicendo vai sul sotto al giovedì sinistra inseriscilo e questo sotto albero di sinistra diventa il nuovo sotto albero di sinistra Analogamente questo diventa il nuovo sotto albero di destra quando ti DTS era vuoto in sostanza quello che succede che io passo un sotto altro vuoto entro in questa chiamata Costruisco il nuovo albero questo viene assegnato a te DTS e quindi ti punta come figlio sinistro al nuovo nodo ma manca un piccolo dettaglio il piccolo dettaglio è che ogni nodo ha anche un puntatore al padre oltre ad avere dei puntatori due figli e questo nuovo nodo che è stato costruito qua dentro non ha assegnato il puntatore al nodo padre e quindi va fatta a mano va fatta a mano come andando dire che questo nuovo nodo che prima non c'era e che adesso c'è a come nodo padre che cosa ha il nodo qui in sostanza questa operazione spesso non è necessaria Cioè se ti litighi in realtà non è stato modificato stiamo semplicemente eri confermando che il padre del figlio di destra è ti ho che il padre del figlio di sinistra e ti ma in quel caso in cui ti DTS utility di era inizialmente vuoto allora 1:26:35 Vuoto allora questo valore vuoto che entra in insert Quando esce è invece un puntatore al nuovo nodo appena locato dalla costruzione e questo nuovo appena locato non ha il puntatore padre correttamente ha segnato quindi dobbiamo assegnarle l'ho qua dentro Non è banalissimo Immagino ma spero di essere riuscito a descrivere nel dettaglio che cosa succede forse qualche disegno può aiutare posso solo consigliare questo non funziona ancora il tutto perché abbiamo la funzione costruisce ABR da realizzare Cosa vuol dire semplice che la andiamo a definire o ti chiamare allora contrariamente a quello che abbiamo fatto sempre in tutto il corso qui non abbiamo un main prima del quale scrivere le dichiarazioni e dopo scrivere le definizione Qui abbiamo una selva di funzioni che più o meno dovrebbero essere tutte indipendenti Ma questa funzione in in realtà dipende da costruisce a Pieve cioè il compilatore quando arriva a questo punto deve già conoscere Che cosa vuol dire costruisce ABR Allora per conoscerlo o noi andiamo a inserire una dichiarazione dentro a breve. h ci andiamo a scriverla qua in maniera che essendo inclusa in questo punto la il compilatore conosca Che cosa fa costruisce ABR Ma questo è andrebbe la funzione costruisce AVR accessibile anche dall'esterno da chiunque includa bmw.ch e questa cosa non la vogliamo fare perché Costruisci avviare una funzione qui importante in albero binario qua dentro invece una funzione che lavora sulla struttura topologica dell' albero binario di ricerca e noi non vogliamo che nessuno lavori sulla struttura topologica tranne chi è strettamente autorizzato cioè le funzioni che sono qua dentro allora per permettere a inserta brd chiamare costruisce ABR c'è un secondo modo che non abbiamo mai usato finora che quello di scrivere la definizione di costruisce AVR prima della definizione di insert in maniera che il compilatore nella sua passata sappia che cosa fa costruisce ABR prima di sapere che cosa fa inserta br quindi questo è un altro modo di ordinare le funzioni è più delicato perché richiede un ordine fare funzioni mettendo a volta che voi avete le vostre dichiarazioni in alto sotto potete fare quello che vi pare e mi piace però è un modo che garantisce di fermare impedire l'accesso a funzioni che devono rimanere abbastanza segrete Allora Cosa vuol dire vuol dire che noi abbiamo il nostro la nostra funzione costruisce ABR che restituisce un albero binario di ricerca prende in ingresso un numero intero prende

in ingresso due ABR che si chiamano figlio di sinistra TS e figlio di destra ti Di chi è questa funzione cominciamo a metterla vuota Dopodiché per convincerli che la cosa dal punto di vista formale è conveniente possiamo vedere che effettivamente tutto funziona come avvertimento secondo il quale la funzione costruisce ABR è arrivata in fondo di una funzione non-void senza vestiti e i valori avviare deve restituire qualche cosa di cui per il momento scrivo un Return not lì ma solo per poter avere la correttezza sintattica e a questo punto possiamo invece entrare nell'implementazione di costruisce ABR che riflette quella di costruisce albero perfetto Come si fa a costruire una br e praticamente la stessa cosa dobbiamo allocare un elemento radice dobbiamo riempire la sua componente informativa dobbiamo segnare Sistemare tutti i puntatori i puntatori sono identici a quelli di un albero binario 1:30:30 E quindi davvero praticamente una copia quello che può cambiare che qui abbiamo puntatori a nodo questo qui dovrebbe essere ABR Ma poi sappiamo che ne abbia ne sono contatori quindi non tutte questioni puramente formali Allora diciamo che la BRT variabile interna lo troviamo lì allo chiamo un nodo con la nostra bella funzione malloc che comprende una cosa che era di amicizia di misura di un nodo verifico il fatto che il nodo sia un nodo Esatto si chiama No do comunque forse viene a chiamarlo nodo ABR ma è lo stesso Non credo che nessun programma Uno mescoli alberi e alberi binari sennò ci sarebbe un problema di doppia definizione di due tipi diversi con la stessa con lo stesso nome E abbiamo la solita verifica del fatto che la locazione sia andata a buon fine dobbiamo vedere se per caso ti è uguale annuncio è stato L'evocazione C'è stato un errore nella locazione di una VR Se ciò è andato a buon fine noi possiamo andare a prendere una per una queste operazioni Qua ovviamente modificandole nel modo opportuno cioè possiamo dire che il campo Val è il numero intero i che il padre di questo modo per adesso è nessun nodo ma non oggi non esiste quindi non mettiamo a nulla non ho qui in questa libreria non esiste il padre. No dai un puntatore ci scriviamo null tanto sappiamo che fuori bisognava sistemare questo campo padre Qualcuno potrebbe dire Ma perché non gli passo il nome del Padre Perché allora non sarebbe una funzione di costruzione sarebbe la funzione di appendi e potrebbe anche essere ragionevole come implementazione ho voluto tenermi il più possibile vicino alle dispense per questo fatto questa operazione quindi ti padre Per adesso nulla Poi subito dopo verrà sistemato assegnandogli l'effettivo padre a cui appendiamo il nuovo albero Dopodiché andiamo a dire che il figlio di sinistra ETS se per caso questo ti non era Nullo il figlio di sinistra deve puntare ti come padre Questa volta poi non succede nel modo e nel modo in cui noi stiamo usando questa funzione il figlio di destra e ti di eventualmente se non è nullo puntati come padre Dopodiché possiamo finalmente restituire il nuovo albero ti all'esterno in maniera da poter valutare se non abbiamo fatto pasticci questo conclude la procedura di inserimento e siccome abbiamo ampiamente usato questa procedura nel caricamento del file Eccola qua abbiamo usata in ciascun su ciascun numero dovremmo essere in grado di avere il nostro albero fatto e finito in memoria il problema è che non abbiamo niente per vederlo e quindi Salvo Una compilata per verificare la correttezza sintattica qui non ci succede niente di utile perché devo che adesso abbiamo inserito tutto quanto ma anche se stampiamo non si vede nulla ci fermiamo tranquillamente è urgente avere una procedura di stampa Come funziona una eventuale procedura di stampa e torniamo qua allora è chiaro che noi vorremmo in questo caso stampare 1 2 3 4 6 8 e sembra un andare su e giù per l'albero Ma siccome la stampa e ordinata e siccome va alla proprietà fondamentale degli ABR che il ciascun nodo è strettamente successivo a quelli del sottoalbero sinistro è strettamente precedente quelli del sottoalbero destro è chiaro che se io voglio stampare un albero Prima devo stampare il sottoalbero sinistro poi devo stampare il nodo radice poi devo stampare il sottoalbero destro e questa cosa vale ripetutamente ricorsivamente su ogni sottoalbero quindi la stampa è evidentemente una stampa in in ordine un po' Com'era la stampa delle espressioni aritmetiche in notazione infissa

qui ha completamente diverso l'argomento ma la stampa coordinata degli elementi di un dizionario rappresentato come albero binario di ricerca si implementa come una visita in in ordine questo device siete perfettamente in grado di realizzarla tuo nipote Te li suggerisco di mettere in pausa e realizzare questa stampa la soluzione della do fra qualche secondo la soluzione In che cosa consiste in un algoritmo ricorsivo di visite in ordine Se vi ricordate come l'abbiamo fatta o comunque c'è anche ci pensate un attimo e rendete conto che ci sarà un caso base che quello di albero vuoto in cui non sappiamo niente e poi ci sarà invece il caso generico domanda davvero se l'albero è vuoto 1:35:20 In cui non sappiamo niente e poi ci sarà invece il caso generico domanda davvero se l'albero è vuoto Non dobbiamo stampare nulla Non val la pena di stampare un puntino uno spazio qualcosa risposta in questo caso No perché sostanzialmente se l'albero è perché in sostanza in una visita di un albero tutte tutti i figli delle foglie sono alberi vuoti c'è una quantità di alberi vuoti e di conseguenza non ha molto senso stampare qualcosa Perché sennò questo qualcosa Ce lo ritroveremo un sacco di volte per cui se l'albero non è vuoto il caso base qui non si vede perché consiste nel non fare nulla se l'albero non è vuoto dovremmo stampare il sottoalbero sinistro e questo con una chiamata ricorsiva stampa ABR sull'albero ti freccia TS Analogamente alla fine ci sarà una stampa del sottoalbero destro e in mezzo che cosa abbiamo in mezzo abbiamo la stampa del elemento informazione Questo elemento è un numero intero e Katy treccia val val la pena di scriverlo così no perché io visto che non essendoci per niente spazi venga tutto appiccicato spazi dove forse mettendoli sia prima sia dopo avremo dei doppi io proverò a mettere uno spazio solo dopo e poi vediamo se questa cosa funziona Se è gradevole alla vista allora tutto ciò che ci tocca fare adesso è ricompilare lanciare e provare a vedere se la stampa questa volta funziona Allora c'è chiaramente qualche cosa di strano ma in realtà è abbastanza una stupidaggine Nel senso che noto che abbastanza spesso io introduco questi valori fittizi di restituzione e poi mi dimentico di aggiornarli nel momento in cui riempio la funzione stessa nel caricamento della br abbiamo fatto una simpatica carica Dopodiché al esterna mio vestito un albero vuoto e questo Evidentemente non va bene dobbiamo restituire l'albero ti che abbiamo appena costruito il che dovrebbe consentirci finalmente di andare a vedere tutto quello che era stato caricato notate che questi numeri sono perfettamente ordinati per valori crescenti e a questo punto siamo in grado non soltanto di stamparli ma siamo anche in grado di Inserire nuovi valori perché la funzione insert l'abbiamo realizzata possiamo inserire per esempio il valore che so stampare nuovamente e vedere che fra il 43-50 si è creato il numero 45 possiamo inserire il numero 1020 e vedere che in fondo è comparso il numero 1020 a questo punto siamo in grado di fare qualcosina in più Quindi usciamo e prepariamoci alla fase successiva fase successiva nella quale dobbiamo scegliere qualche altra funzione da implementare ben piantato creazione distruzione vuoto inserimento ci rimangono la cancellazione che però è l'operazione più complicata la lasceremo per ultima Abbiamo fatto anche la stampa ci mancano member min e Max sono tre funzioni piuttosto simili fra loro Quindi vi l'indicazione di farle sostanzialmente tutte e tre insieme perché dico che sono simili Perché sono tutte funzioni che in sostanza esplorano l'albero per trovare un qualche cosa che cosa la Esplora per capire se un dato numero andiamo il 5 al solito a partire o no questo albero e come lo fa andando alla ricerca nel sottoalbero giusto confrontando Scusate il numero che abbiamo in in mano con la corrente radice e vedendo se per caso questo numero identico Allora l'abbiamo trovato se il numero che abbiamo in mano il più piccolo Sara eventualmente nel sottoalbero sinistro Non può essere nel testo Mentre se il numero è maggiore della radice se c'è e nel sottoalbero detto Non può essere Nel sinistro e questo ci permette di scendere in uno dei due sottoalberi applicare ricorsivamente lo stesso test la stessa funzione quindi la struttura È abbastanza evidente ricorsiva basata su un caso base che sarà quello di albero vuoto con un secondo va a caso base che quello di dire questo elemento che ci hanno dato è uguale

alla radice oppure no E poi ci saranno i il caso di corsivo vai a sinistra oppure vai a destra seconda di Qual è il valore il confronto tra il valore che è stato fornito e Ivan non è conservato nella radice questa avremo poi da dire qualcosa su questa funzione vedremo che siccome la ricorsione te 1:40:10 È stato fornito che i valori e conservato nella radice questa avremo poi da dire qualcosa su questa funzione vedremo che siccome la ricorsione terminale si può trasformare meccanicamente in un algoritmo iterativo Ma questo lo faremo a posteriori prima di farlo Vi suggerisco di provare a implementare questo algoritmo che abbiamo descritto a parole in sostanza ripeto se l'albero è vuoto non si fa nulla perché evidentemente non si fa nulla si restituisce falso perché evidentemente l'elemento i non fa parte dell'albero Se invece l'albero non è vuoto si verifica se per caso sia il nodo radice e altrimenti si va a vedere nel sottoalbero sinistro nel sottoalbero destro con opportune chiamate ricorsive Quindi abbiamo detto se ti è un albero vuoto allora approfittiamo le abbiamo Write on Force altrimenti sicuramente esiste una radice possiamo confrontare i con il valore in radice Può darsi che sia identico Beh se è identico E allora chiaramente dobbiamo restituire tu e altrimenti che cosa succede altrimenti poteva succedere che l'elemento stia nel sottoalbero di sinistra oppure che l'elemento stia nel sottoalbero di destra è tutto Questo dipende In sostanza dal confronto fra i etival quindi se per caso i è minore liti freccia Val Dovremmo ripetere il tutto sul sotto albero di sinistra deve restituire il valore che ne ricaviamo quindi restituiremo member ABR di nel sottoalbero ti freccia TS Eccolo qua con le maiuscole Minuscole corrette in fine infine e a questo punto non ha più senso testare il caso di ma giovedì ti va perché è chiaro che se non è uguale non è minore e maggiore Però io lo lascio sotto forma di commento diciamo per ricordarci che cosa significa in questo caso istituiremo member AB rdpd molto semplice possiamo andare a compilare provare a lanciare il tutto ci facciamo stampare qualcosa e ci chiediamo se l'elemento che so 64 fa parte dei spettivamente fa parte mentre l'elemento 63 non ne fa parte e quindi il tutto sembra perlomeno con un paio di esempi sembra funzionare Prova a questo punto possiamo chiederci Siccome la ricorsione eccola qua o qua è finale perché finale perché è addirittura distruzione che sta dentro al ritorno quindi è l'ultima operazione che si esegue vuol dire che si può realizzare una implementazione iterativa di questa funzione quindi questa qui Sostanzialmente è l' implementazione ricorsiva della nostra funzione e potremmo anche decidere che invece vogliamo fare un implementazione iterativa che è tendenzialmente preferibile perché evita tutto il nome Red tutto lavoro in più di area locale sullo specchio vengo di attivazione di copiare Che cosa dico che arriva lo vedi che è sempre lo stesso perché continuava a cercare sempre lo stesso numero e io volevo dirti che sono che cosa Solo le radici dei sotto alberi che dirvi Andiamo a visitare ma sostanzialmente quello che mi interessa è semplicemente il fatto che questa radice sia Zero sia nulla oppure che il contenuto della radice sia il numero i non mi interessa nient'altro Quindi tutti i precedenti passaggi sono del tutto inutili Cosa vuol dire questo così anche in sostanza quello che possiamo fare saltare tutte queste chiamate eseguire la ricerca sotto l'albero sinistro sottoalbero destro fintanto che non siamo in quale situazione nel caso di base cioè fintanto che siamo o in un albero vuoto o in un albero che ha come radice dell'elemento i e questo non è difficile in sostanza che cos'è che dobbiamo sostanzialmente dire che Procederemo avanti avanti avanti fintanto che non vale una delle due condizioni di fine c'è fintanto che non siamo in un caso base fintanto che non siamo non siamo notate la negazione nel caso di albero vuoto e non siamo nella anche nell'altro caso base che è il caso in cui di è uguale a il valore contenuto dentro l'albero 1:45:00 In cui is è uguale il valore contenuto dentro l'albero fintanto che non siamo in nessuno di questi due casi Che cos'è Che dobbiamo fare Dobbiamo chiederci se proseguire andando a sinistra Proseguendo a destra quindi dobbiamo chiederci se io è più piccolo di ti va al E allora che cosa si farà si andrà a considerare come albero corrente il sottoalbero sinistro faccio una cosa che potrebbe sembrare azzardata cioè sostituisco tutti col

sottoalbero sinistro perché sembra azzardata perché se gli vado modificare ti Ma si cominci il passaggio dei parametri per copia questo ti che sto ricevendo non è lo stesso Ti della funzione che mi ha chiamato è un altro ti è un interno alla funzione member che io posso continuare a cambiare tranquillamente e lo sostituisco con sotto anche un sinistro in questo caso mentre se invece o l'altro caso e mi faccio notare che se sono dentro vuol dire che io non è uguale e d'altra parte non è neanche minore ne deriva che è certamente Maggiore quindi non è necessario scrivere Leaf è quello che si farà Sara semplicemente passare al sotto albero di destra a questo punto quando si esce da questo ciclo che è il che corrisponde alla continua locazione di ambientis di record di attivazione sullo Stack quando finalmente si esce i casi sono due o siamo usciti perché ti ha i nostri o siamo usciti perché ti va È uguale i Allora siccome ti potrebbe essere notte io prima tetto quello cioè se ti è uguale a nostri evidentemente restituisco Restituisco Falls nell'altro caso restituisco tu non ho neanche bisogno di interrogarmi sul fatto che mi sia uguale posso restituire tu direttamente Ma questo significa questa cosa che adesso scrivo e poi mo cancello questa cosa qui else Return True In sostanza si può anche tranquillamente compare riassumere dicendo che restituisco il fatto che ti non sia un albero vuoto se sono uscito con un albero non vuoto significa che ho trovato la radice Questo è quanto non voglio fare entrambe le implementazioni potrei commentare via questa un modo possibile di farlo approfitto per farmi vedere una finezza è di andare a dire che questo pezzo di codice viene considerato dal precompilatore soltanto se la costante simbolica ricorsiva è definita altrimenti andiamo a considerare questo altro pezzo di codice non so se si capisce il precompilatore neanche il compilatore quando arriva qui va ad analizzare se qualcuno ha scritto the final ricorsiva da qualche parte e qui non c'è scritto define ricorsiva non c'è scritto qui e non c'è scritto neanche nel mail però se qualcuno lo scrive invece entreremo Siccome ricorsiva non è definita tutto questo pezzo di codice da fare è cancellato dal precompilatore e rimane invece questa parte qui Vi consiglio di fare la prova cioè di fare la compilazione con meno e maiuscola di vedere che effettivamente in avviare zero questa cosa è scomparsa di rifarla scrivendo da qualche parte prima una define ricorsiva e di vedere che in questo caso il precompilatore prende questa parte eh Cancella questa alta in questo modo potete avere codici diversi semplicemente con l'aggiunta e la cancellazione di una riga e questo può a volte essere comodo Allora vediamo se questa cosa continua a funzionare Mi aspetto di sì sperando di non aver fatto pasticci ci andiamo a stampare ovviamente non fa nulla perché minuscolo il comando tutto quanto andiamo a chiederci se il numero 20 è membro effettivamente lo è e poi ci chiediamo se invece è mendo il numero uno e mi dice che lo è in questo è un cattivo messaggio evidentemente Ma la motivazione è molto bene ed è che pur avendo detto correttamente che questo condizioni di permanenza è il contrario di questa condizione qui non ho fatto la stessa cosa c'è un ovetto che gli deve essere diverso cattiva al per rimanere nel ciclo per cui avevo scritto che si rimane finché è uguale allora si usciva immediatamente fornendo il valore vero Di conseguenza cui possiamo uscire ricompilare e testare Print se per caso il numero 1 giustamente non ne fa parte altrimenti qualche altro 1:49:50 Link se per caso il numero uno giustamente non ne fa parte altrimenti qualche altro numero invece ne fa parte e siamo a posto possiamo passare alla implementazione delle funzioni minime Massimo che hanno una storia molto simile cioè sono funzioni anche qui in cui c'è una discesa in uno dei due sottoalberi quindi un implementazione ricorsiva con un caso base di albero vuoto Non c'è il caso l'altro caso base e però ne implementazione ricorsiva con ricorsione terminale che si può facilmente trasformare in iterativa Allora solito Provate a fare la vostra implementazione non è comunque per niente complicata come come come compito è la soluzione è la seguente se per caso L'albero è un albero vuoto Dobbiamo inventarci un valore di default che è poi questo di cui abbiamo parlato in precedenza altrimenti che cosa succede altrimenti è chiaro che l'elemento minimo sta nell'albero

di sinistra non sta nell'albero di destra però attenzione ho detto non esiste il caso base secondo non è corretto è chiaro che se un elemento c'è ma non ha un figlio di sinistra Beh allora quello è un caso base Anzi è proprio il caso base che stiamo cercando Cioè se noi stiamo cercando il minimo di un albero quello che facciamo e andare sempre giù a sinistra Finché si trova un numero che non ha albero di sinistra e quindi non ha altri numeri che sono strettamente minori e questa è la definizione di minimo di un insieme è un numero che non sa e che è minore o uguale a tutti gli altri cioè che non ha nessun altro numero che sia strettamente minore di lui di conseguenza Questa è la seconda è il secondo caso base nel quale Noi andiamo a testare se per caso il figlio di sinistra dell'albero corrente è vuoto e in questo caso sappiamo che comunque Esiste un valore corrente e questo valore corrente guarda caso è esattamente il minimo Quindi restituiamo t-ball ed è soltanto se anche questo non funziona che andiamo a riapplicare ricorsivamente quindi possiamo essere sicuri che il risultato sia il min AVR di che cosa del sottoalbero di sinistra il quale sotto a sinistra c'è siamo sicuri che ci sia e quindi questo è in effetti quanto e ci fornisce la implementazione ricorsiva che stavamo cercando lamentazione ricorsiva e che possiamo anche procedere a verificare velocemente dicendo perfetto diamo una vostra solita stampa cerchiamo il minimo che è 20 e a questo punto si è cancellassi mo ma non possiamo cancellare Allora diciamo che inseriamo 19 e senza neanche andare a stampare il minimo è diventato 19 e se inseriamo meno 5 il minimo è diventato meno 5 e quindi tutto sommato Direi che possiamo darci per soddisfatti prima di fare l'implementazione integrativa Io direi che è decisamente banale andare a costruire la funzione massimo perché non si fa altro che sostituire tutto ciò che è Max con min e tutto ciò che è sotto albero sinistro col sottoalbero destro e ovviamente anche qui c'è una chiamata a Max Cerchiamo di non essere troppo sbrigativi però mi sembra che tutto sia stato corretto in corrispondenza facciamo una nostra prova compriamo ovviamente rilanciamo facciamo la stampa e vediamo che il massimo e 997 che effettivamente l'ultimo numero D'altra parte se a questo punto inseriamo mille il massimo diventa mille non possiamo cancellare niente quindi non possiamo far altro che aggiungere altri numeri in Sert 2090 e vedere se sostituiscono via i precedenti e direi che possiamo essere relativamente soddisfatti dovrebbe essere semplice vi invito a mettere in pausa e provare per i fatti vostri dovrebbe essere semplice realizzare incrementazione iterativa in cosa consiste consiste nella al solito scorrere in orto albero andando di passo in passo dove verso sinistra verso sinistra verso sinistra quindi ancora più facile di prima o verso destra verso destra verso destra fermandosi quando nei due casi di base cioè fermandosi quando si fa un albero vuoto oppure un albero il cui figlio è vuoto Evidentemente è chiaro che questi due casi sono in un certo senso vedi non sono entrambi necessari perché se se l'aveva è vuoto è chiaro che è un figlio vuoto 1:54:40 Perché se se l'aveva è vuoto è chiaro che è un figlio vuoto e via dicendo cioè la prima delle 2 che viene incontrata Come condizione quella di albero col figlio vuoto A meno che L'albero in partenza sia vuoto quindi in realtà non vi porteremo Neanche due condizioni me li porteremo una soltanto Questa è l'unica Se volete finezza difficoltà che presenta l'implementazione provate e vi do a momenti la mia soluzione Allora come nell'altro caso andiamo a inglobare questo pezzo di codice nella nostra del nostra direttiva if-else andiamo a costruire l'altro caso come in diff la stessa cosa faremo poi sotto nel caso del massimo e andiamo a prepararci a realizzare una implementazione questa volta iterativa in cosa consiste l'implementazione iterativa molto semplice consiste nel dire In realtà che se il caso se il caso se l'albero corrente è vuoto allora restituiremo in Max nel caso del minimo dopodiché siamo sicuri che esiste un nodo figlio e possiamo andare giù Figlio figlio figlio sul lungo il sottoalbero sinistro fintanto che il figlio di sinistra del nodo corrente rimane non vuoto è questo quello a cui alludevo dicendo che non useremo entrambe le condizioni perché questa condizione ti albero vuoto si trova una sola volta all'inizio solo se l'intero albero vuoto altrimenti si usa

solo la seconda vedi in questo caso ciò che si farà Sara banalmente di continuare a seguire la strada del sottoalbero di sinistra e quando finalmente si arriva a violare la condizione di Parma quindi ad avere un nodo il cuscino di sinistri vuoto Ecco che allora banalmente andremo a restituire il valore che si è trovato in questo caso Detto ciò la stessa cosa si può fare per l'implementazione del massimo con ciò che va a banalmente mutato cioè andando a restituire in Mine se per caso L'albero è vuoto andando a scendere da sotto albero di destra anziché di sinistra e Ma per il resto non è cambiato assolutamente nulla e quindi possiamo corrispondentemente andare a compilare e a vedere che il minimo e 20 il massimo 997 se inseriamo 5 e inseriamo mille vedremo che il minimo è 5 e il massimo e 1000 e questo è quanto ci occorreva per cui a questo punto L'unica cosa che si manca per concludere l'implementazione e realizzare la procedura di cancellazione che come vi avevo accennato è la procedura un pochino più complicata e sofisticata perché presenta di diversi cavi particolari e diverse operazioni però adesso andiamo a considerarla un passo alla volta Allora a questo punto possiamo affrontare la funzione più complicata di gestione dell' albero binario di ricerca che è la cancellazione perché è complicata in realtà già nel gestire gli alberi binari avevamo avuto un problema relativo alla cancellazione Il problema è il fatto che se io cancello un elemento per esempio se cancello l'elemento due non è assolutamente ovvio che cosa debba succedere in un albero binario per mantenere la proprietà di essere albero binario Tanto è vero che gli alberi binari e avevamo adottato la strategia osa ma sufficiente per gli scopi che ci eravamo prefissi di cancellare un intero sotto albero tutto ciò che è appeso al nodo due cose che qui non si può fare perché la cancellazione e qui dobbiamo gestire un insieme di elementi ordinati e vogliamo cancellare un elemento non un sottoinsieme di elementi che tra l'altro non è sotto il nostro controllo perché non siamo noi a decidere chi è figlio a sinistra destra del nodo 2 l'ordine che lo stabilisce Allora che cos'è che si può invece fare qualche cosa di più sofisticato intanto osserviamo che ci sono dei nodi che sono cancellabili senza grandi problemi e sono le foglie l'uno il 3 e Lotto si cancellano de allocando e ovviamente sistemando i puntatori nel nodo padre in maniera corretta poi ci sono dei modi che sono un pochino sistemabili con un po' di scotch ma non molto e sono i nodi che hanno un solo figlio sono interni Man un solo figlio non due Se io volessi cancellare in modo qua 1:59:30 Insomma non molto e sono i nodi che hanno un solo figlio sono interni Man un solo figlio non due Se io volessi cancellare in modo 4 In effetti mi troverai di fronte a una situazione simile a quella di una lista bidirezionale Ognuno di questi nodi a un puntatore ai ai due nodi figli e poi è un puntatore al padre quindi il L'albero è percorribile bidirezionalmente E se invece di avere due figli ce n'è uno solo 243 formano effettivamente una lista per cui se io voglio cancellare il 4 è abbastanza automatico che il 32 si scambiano reciprocamente i loro puntatori che si de All'occhio il quato è tutto vada a posto tranquillamente Quindi l'unico problema vero È quello che corri che riguarda i nodi con due figli lo cerchiamo di impostare ovviamente una procedura ricorsiva la procedura ricorsiva avrà un caso base che quello di albero vuoto nel quale non c'è nulla da fare ed è il caso in cui io voglio cancellare qualcosa che non è nel dizionario poi ci sarà invece il caso in cui io vado a cercare questo elemento che voglio cancellare lo trovo il corpo che non ha figli o che ha un figlio solo e in questo caso la cosa non è difficile perché si tratta di eventualmente vi adattabile gamifant padre è unico figlio L'unico figlio magari non c'è e poi del locale il nodo stesso e questo Sarà un caso base relativamente semplice poi ci sarà un terzo caso base che è qui Appunto per questo che più complicata tre casi base invece di averne uno o due il terzo caso base quello del nodo che ha due figli cugina inventarsi qualcosa fa un balzo di ingegno e in questo caso il balzo di ingegno è simile a quello che si faceva nel caso delle tabelle Se vi ricordate delle tabelle venivano permettevano di cancellare un elemento in tempo costante semplicemente sovrascrivendo l'elemento da cancellare con l'ultimo e riducendo la dimensione della tabella come dire che io

posso cancellare il 2 Basta che ci scriva sopra un altro elemento terminale terminale vuol dire foglia in un albero e questo elemento foglia può essere a sua volta cancellato facilmente sembra un'idea geniale Se non fosse che in realtà però non si può banalmente fare questa cosa prendendo una foglia qualsiasi perché c'è la struttura ordinata dei nodi dell'albero che incastra molto vicina mentre la struttura dei nodi stessi quindi non è possibile sovrascrivere un nodo interno con una foglia qualsiasi Fortunatamente però è possibile sempre trovare almeno una foglia che permette che può sovrascrivere il nodo stesso anzi l'altra se ne possono trovare due Ricordatevi stiamo parlando di nodi che hanno due figli e quindi hanno un sottoalbero sinistro è un sottoalbero destro entrambi non vuoti ora la cosa interessante è che c'è una foglia nel sottoalbero sinistro e c'è una foglia nel sottoalbero destro che sono entrambe cancellabili e sostituibili al nodo di partenza quali sono queste foglie semplice sono l'elemento massimo del sottoalbero di sinistra elemento minimo del sottoalbero di destra perché sono foglie perché sono il massimo e il minimo il massimo non può avere figli di destra perché è un Scusate diciamo una piccola in proprietà Nel senso che sto parlando di foglie ma in realtà non volevo dire foglio Quello deve nodi che hanno al massimo un sottoalbero figlio perché hanno il massimo sotto l'albero figlio perché l'evento massimo di un sottoalbero non può avere il sottoalbero destro elemento minimo di un sotto albero non può avere sotto albero sinistro quindi per farla breve se io voglio cancellare il nodo 2 quello che potrei fare andare a cercare il massimo nel sottoalbero sinistro che in questo caso siamo fortunati è una foglia ed è l'uno e questo elemento può essere cancellato abbastanza semplicemente e sostituito al 2 oppure posso prendere l'evento minimo del sottoalbero destro che anche in questo caso è una foglia ad esempio mi ha un po' tratta in inganno e in realtà quello che potrebbe succedere che ci fosse un 3:30 compreso fra il 3 e il 4 quindi un figlio destro del tre il tre è comunque l'elemento Mini come una foglia questo per darvi un esempio di di che cosa potrebbe succedere però è comunque possibile andare a trovare il minimo elemento del sottoalbero destro quindi questo modo tre cancellando stoppando la sottoalbero destro stesso e andare a sovrascrivere l'elemento due e questo è quello che effettivamente succede quello che effettivamente fa la procedura Delete many capite che gli elementi base sono al solito una divisione in caso di base e casi ricorsivi cattivi consigli sono banali si risolvono richiamando la funzione stessa alcuni casi base sono banali quello di albero vuoto 2:04:20 Quindi capite che gli elementi base sono al solito la divisione in casi base e casi ricorsivi che arrivi così vi sono banali si risolvono richiamando la funzione stessa alcuni casi base sono banali quello di albero vuoto ad esempio Altri meno e poi c'è e invece questa necessità di avere una funzione ausiliaria che la funzione che dato un nodo che ha Al massimo un figlio però eventualmente anche 0 ricollega il figlio che eventuale con il padre in maniera da garantire la struttura dell'albero Allora proviamo a buttare giù lo schema dopodiché vi inviterò a lavorarci da soli e vi fornirò la soluzione cose che dobbiamo fare in sostanza cominciamo elencare i casi base se l'albero ti è vuoto non si fa nulla e questo è molto semplice in realtà quello che scriveremo Sara se l'albero è foto si farà qualcosa Sei un caso in cui non si fa nulla quindi neanche Compare lo scrivo per renderlo esplicito poi ci sarà invece il caso in cui la radice dell'albero di a un solo figlio e questo è il secondo caso base Scusatemi un attimo Sto percorrendo i tempi quello che noi stiamo cercando è l'elemento i Quindi quello che il caso base quando gli è la radice dell'albero ti è a un solo figlio eccolo qua poi c'è un altro caso base che dice che io e la radice dell'albero ti è questa volta due figli e questo è il caso complicato e poi ci sono invece i casi ricorsivi cioè il caso in cui se i precede la radice dell'albero ti e infine il caso in cui i segue la radice Questi sono i due casi ricorsivi Quindi quello che dovreste essere facilmente in grado è di sistemare questo caso base i due casi ricorsivi probabilmente anche il caso di singolo figlio e se vi sentite in grado provare anche realizzare questa altra funzione che è quella che determina che risolve il problema dei due figli Questa funzione userà sia la funzione ausiliaria

che collega direttamente l'unico figlio al padre per il nodo che va a sostituire quello che stiamo cancellando ma anche ci sarà una funzione che determina questo modo che va a sostituire il quello che stiamo cancellando se io voglio cancellare il 2 prima chiamerò una funzione che va in cerca oqui del massimo o qua del minimo quello che sostanzialmente faremo sarà andare alla ricerca del massimo nel sottoalbero di sì classe non ricordo male Ho un piccolo vuoto Esatto Sì dopodiché determinato il massimo nel sottoalbero di sinistra con una funzione andremo ad applicare questa l'altra funzione che quella che Cancella tale nodo è semplicemente andremo a sovrascrivere l'elemento che vogliamo cancellare Ok Allora diciamo che adesso avete fatto il vostro tentativo e vediamo in che cosa può consistere una soluzione mi ha detto che se l'albero è vuota non si fa nulla questo neanche lo scriviamo se l'albero non è vuoto se ti è diverso da nostri invece solo allora faremo una serie di cose possibili e queste cose possibili saranno e due chiamate ricorsive e i due casi base per semplicità questione di gusto mio personale siccome le due chiamate ricorsive sono molto semplici Le anticipo così me la sbrigo abbastanza velocemente Che cosa succede se per caso l'elemento i è minore di ti va al molto semplice succede che bisogna applicare Delete ABR bisogna applicarlo e sotto albero di sinistra bisogna cancellare i dall'albero Ti frecciati lì è risultato Che cos'è il risultato è il nuovo sotto albero di ho detto sinistra di nuovo sotto albero di sinistra TS e questo risolve il primo caso ricorsivo dopodiché abbiamo l'altro caso di corsivo Elsa quello in cui invece I am a giovedì ti va al E questo vuol dire che ha Peter 2:09:10 Vi terremo la funzione dell'iter sottoalbero di destra o tenendo il nuovo sotto albero di destra perfetto applichiamo affrontiamo ora il caso base più semplice quello di figlio singolo se non siamo capitati in nessuno di questi due casi è evidente che è uguale Attiva quindi effettivamente gli è la radice non va bene neanche di scriverlo però Dobbiamo verificare che ci sia un solo figlio come si fa a verificare che ci sia un solo figlio e bisogna che uno dei due sia un albero vuoto quindi bisogna vedere o che TTS sia un no togli questa è la prima possibilità oppure ora possiamo andare a vedere Katy freccia ti dissi a un nutri in questo caso dobbiamo fare una serie di cose di operazioni che adesso andiamo un po' meglio a discutere altrimenti è evidente che gli altri ci sono entrambi e faremo tutta una serie di altre operazioni che cosa succede nel caso in cui ci sia un solo albero quello che bisogna fare Sostanzialmente è cancellare il nodo il nodo puntato cioè la radice dell'albero stesso e appendere direttamente il sottoalbero al padre quindi dovremmo appende direttamente l'unico sottoalbero al nodo padre della radice faccio notare che potrebbe anche non esistere un modo padre quindi Qui ci sono una serie di casi e sottocasi eccetera eccetera e questa cosa e dopodiché dovremmo che distrugge dealloca il la radice stessa tutte queste cose non le facciamo Con un'unica funzione qui diamo un nome così cominciamo a non sovraccarica Arci e io questa funzione l'ho chiamata toglie nodo ABR è una funzione che lavora su che cosa lavora su un albero e su un nodo dell'albero sulla radice può sembrare un po' barocco scrivere due volte ti lo è assolutamente io sto in qualche maniera cercando di salvare la possibilità di una implementazione fatta con vettori e indici implementazione che non ho fatto in proprio quindi non sono sicuro che tutto questo basti ma è evidente che se avessimo l'implementazione a Vittoria indici i puntatori alla radice non sarebbero più dei puntatori sarebbero dei numeri interi cioè la radice di Un tal sotto albero sarebbe l'elemento 5 elemento 20 del vettore che rappresenta questo albero complessivo e quindi avrei bisogno non solo dell'indice ti del Avrei bisogno anche di rappresentare l'albero nel suo complesso non è detto che funzioni consentitemi di scrivere in questo modo poi comunque tutto il resto funziona abbastanza corrispondentemente Cosa fa questa funzione che abbiamo scritto per il momento ce ne disinteressa siamo affinché il tutto abbia senso abbiamo bisogno che la funzione sia definita sopra esattamente come abbiamo fatto per la funzione costruisce ABR e di conseguenza avrei una funzione Void toglie il nodo ABR la quale in funzione Void avrà come argomenti Vabbè come argomenti

il nodo che si vuole togliere lo chiamerò n e il dell'albero stesso in cui vogliamo ragionare per il momento questa funzione è la lasciamo vuota e abbiamo risolto il più semplice l'intermedio fra i casi base più semplice ma quello di cui non si fa nulla perché la deve vuoto Cosa rimane da fare rimane da fare il caso più complicato Cosa succede se abbiamo da togliere un nodo che ha due figli anziché uno soltanto è bene in questo caso quello che dovevo fare Sara andare a capire per prima cosa qual è il nodo che andrà a sovrascrivere quello corrente Quindi anche qui abbiamo bisogno di determinare un nodo E questa volta esplicitamente quindi ho bisogno di una variabile nodo.in e in questa variabile No da questa variabile nodo andiamo ad assegnare Che cosa è l'elemento 2:14:00 Determinare un nodo e questa volta esplicitamente quindi ho bisogno di una variabile nodo.in e in questa variabile No da questa variabile noto andiamo ad assegnare Che cosa è l'elemento massimo del sottoalbero sinistro Bisogna dargli un nome Io l'ho chiamato a Max per dire che è l'elemento che corrisponde al valore massimo di una funzione all'interno di una b r e la funzione in realtà è data dal dizionario del sottoalbero sinistro sotto l'albero sinistro questo mi dai il nodo Massimo nell'albero di sinistra una volta che l'ho trovato quello che possiamo fare dire benissimo Adesso ce l'abbiamo prima di cancellarlo sopra scriviamo il valore della radice quindi ti va al diventa = n Val a questo punto abbiamo due volte il valore di n ce l'abbiamo come elemento massimo del sottoalbero di sinistra Questo vuol dire che possiamo tranquillamente togliere il nodo stesso quindi andremo a togliere il nodo n dall'albero Ti noto che in realtà nella mia soluzione la funzione toglie nodo un po' come tutte le funzioni che abbiamo considerato restituisce a sua volta un altro Allora gli faccio restituire in entrambi i casi l'albero ti è a questo punto tutto ciò che rimane è di andare a dichiarare vuota questa funzione e andare a riempire questi due termini quindi dovremmo andare qua sopra a determinare determina il nodo che corrisponde al valore massimo nell'albero di e questa è una funzione che restituisce un puntatore a nodo e ricevi in ingresso un abrt piuttosto semplice come potete facilmente immaginare e quest'altra invece la funzione che appende il sottoalbero l'unico dei due sotto Alberi Del Noto n al nodo padre del mondo n restituendo alla fine restituendo alla fine l'albero che se ne ottiene ok più o meno abbiamo scritto tutto quello che dobbiamo fare comincerei dalla più facile delle due funzioni che evidentemente la smacks perché era più facile perché somiglia in maniera fortissima alla implementazione del Max ABR andiamo a vedere questa implementazione del massimo che qua sotto Prendiamo pure quella integrativa si tratta Se no do è se l'albero è vuoto di restituire che cosa non il valore più infinito Massimo meno infinito ma nel caso di restituzione di un nodo un puntatore fittizio null Dopodiché invece andremo giù per il sottoalbero destro destro destro fino a che non Troveremo un modo che non ha sottoalbero destro e che quindi necessariamente è il valore massimo è il nodo Massimo e invece di restituire il valore restituire il nodo quindi non è che cambi molto si può pesantemente ispirarsi a questa funzione e andare a cercare di riempire la funzione Eccola qua Max e dire che cosa che se per caso l'albero su cui stiamo lavorando è vuoto restituiremo nulla È un puntatore fittizio Se invece non è così correremo lunghi sotto alberi desti desti destri fino a ottenere il puntatore alla radice che è il nodo che ci interessa effettivamente Quindi questa parte del nostro compito è stata abbastanza semplice Adesso viene quella più complicata qui dobbiamo fisicamente togliere un modo da un albero Questa cosa è molto simile alla cancellazione di un elemento da una lista bidirezionale con qualche complicazione legata al fatto che in realtà l'elemento non si sa bene se usa per scendere per andare in una direzione il sottoalbero di sinistra o quello di destra dobbiamo scoprirlo e il fatto che è un'altra il nodo potrebbe anche non avere un predecessore Perché non è una lista con sentinella perché se per caso stiamo cancellando la radice il nodo n non ha un nodo padre a cui appendere il sottoalbero figlio quindi c'è da fare un pochino di casi è di sotto casi però cominciamo a scrivertelo per punti Allora prima cosa che dobbiamo fare determinare il

sottoalbero figlio L'unico 2:18:50 Devo fare determinare il sottoalbero figlio L'unico sotto albero figlio L'unico sotto albero non vuoto del nodo n e questa è la prima cosa può darsi che ci sia Può darsi che non ci sia potrebbe anche essere nulla Quindi eventualmente potrebbe anche essere un nutri in effetti è questo il caso in cui il nodo è una foglia cosa che potrebbe anche succedere una volta che abbiamo trovato questo che cos'è che dovremmo fare e dovremmo sostanzialmente appendere questo sottoalbero al nodo padre tenendo conto del fatto che sotto albero può non esserci e il nodo padre può non esserci quindi appende se esiste un sottoalbero lo appende al nodo padre che può se questo esiste e quindi dobbiamo tenere il conto di tutti questi casi Una volta fatto questo siamo a posto Mi Nel senso che dobbiamo cancellare il nodo stesso ma potrebbe anche succedere che le cominciamo a cancellare in modo stesso Sì allora meglio spezzare questo operazione Allora se esiste il sottoalbero appende il sottoalbero al modo padre però anche il nodo padre deve entrare in comunicazione col sottoalbero Quindi praticamente deve tidu Ire il sottoalbero nelle informazioni del nodo padre all'albero corrente nel nodo padre fatto questo potrebbe anche succedere una situazione un po' limite la situazione limite è quella in cui stiamo cancellando la radice il che significa in sostanza che anche l'albero ti cambia Quindi se n è la radice cambia l'albero chi Punta il sottoalbero e finalmente dovremmo essere in grado di deallocare il nodo n in riassumo un attimo qui a parte abbastanza ovvia Stiamo cercando Qual è il sottoalbero che effettivamente esiste potrebbe anche non esserci Dopodiché questo sotto albero che abbiamo trovato ammesso che esista lo appendiamo al modo padre ammesso che questo esista il nodo padre se esiste deve puntare il sottoalbero quindi nelle informazioni del nodo padre bisogna far puntare il sottoalbero Può darsi che tutte queste operazioni vadano addirittura toccare la struttura la radice dell'albero ti perché magari avevo cancellato la radice infine possiamo del locale il nodo Dopodiché fate tutte queste belle cose possiamo finalmente restituire l'albero complessivo Allora procediamo una per una eseguire queste operazioni prima operazione Come si fa a trovare Qual è l'albero vuoto Non è difficile se si abbiamo già sappiamo già l'utente il programmatore che usa questa funzione deve già sapere di aver già testato che ci sia un solo albero figlio altrimenti il programma è sbagliato ma noi l'abbiamo fatto abbiamo fatto qua sotto con questo test quindi non lo rifacciamo questo test Allora se abbiamo questa situazione che cosa possiamo dire possiamo dire che ho tffs in modo sinistro è vuoto o è vuoto in modo destro cominciamo a crearci una variabile TF che punterà il sottoalbero esistente questa variabile per cominciare io potrei pensare che sia sotto albero di sinistra se per caso nel fare questa ipotesi sono andato a portare un altro vuoto evidentemente vuol dire che l'unico figlio esistente invece è il sottoalbero destro in questo è un modo per andare a puntare quello dei due che non è vuoto ipotizzando che uno solo dei due non sia vuoto fatto questo se siamo arrivati a trovare effettivamente un albero non vuoto perché se per caso l'abbia nel vuoto Qui abbiamo finito questa operazione non non non deve essere 2:23:40 Effettivamente un albero non vuoto perché se per caso l'abbia vuoto Qui abbiamo finito questa operazione non non non deve essere neanche fatta per in questo caso Attendiamo questo TF al nodo padre Cosa vuol dire vuol dire che il suo campo padre GF padre non deve essere ti non deve essere scusa te non deve essere il nodo da cui siamo partiti e qui mi accorgo che ho fatto un pasticcio Allora Scusate Qui stavo guardando il sottoalbero di sinistra è sotto l'albero di destra del nodo n Ok quindi a seconda dei casi TF punta sotto l'albero sinistro o sottoalbero destro del nodo n se sto puntando a qualcosa Vado a scrivere nel modo padre che non è più n ma è il padre di N E questo potrebbe essere vuoto Non è vuoto Non importa Può darsi che ne sia la radice come padre abbia nulla vuol dire che TFR come padre nulla c'ho fatto Bisogna salire nel Padre ammesso che il padre esista e dobbiamo andare a recuperare dobbiamo andare a recuperare quale dei due figli del padre era n quindi dobbiamo capire se n era figlio di sinistra o figlio di destra

di suo padre come si fa a distinguerlo semplice e andare a vedere se n è uguale al figlio di sinistra di suo padre oppure invece sei figlio di destra Evidentemente o è un caso o è l'altro non ci sono terzi un dato Allora se era figlio di sinistra Cosa significa significa che adesso il figlio di sinistra del padre di n non sarà più n Ma sarà banalmente TF nell'altro caso invece non è più il figlio di destra di n non è più n il figlio di destra di suo padre ma è TF quindi stiamo aggiornando le informazioni contenute all'interno del logo padre Dopodiché potrebbe anche essere che non ci sia una tuo padre in questo è un problema perché vuol dire che n padre sarebbe nulla e quindi qui stiamo scrivendo una stupidaggine allora teniamo conto di questa cosa che succede se n padre è nulla succede sostanzialmente che siamo in radice e quindi siamo in questo caso qui lo anticipiamo e diciamo che se per caso n padre non cioè il nodo da cancellare è la radice Ci limitiamo a dire che l'albero ti di l'albero ti diventa l'albero PF Altrimenti altrimenti sappiamo che esiste un nodo padre e quindi possiamo proseguire con questa bella catena di If ed else quindi altrimenti andiamo a vedere se il nodo che stiamo cancellando la figlia di sinistra o di destra finalmente abbiamo sistemato tutti i puntatori e ha sistemato i puntatori del padre Abbiamo sistemato i puntatori del figlio Possiamo tranquillamente andare al locale il nodo n e restituire l'albero così ottenuto E questo dovrebbe essere la conclusione Speriamo andiamo a compilare e avevo vieni compilazione non ce ne sono proviamo a lanciare il nostro codice solito vediamo questa volta provo a cancellare il nodo 59 ed effettivamente il 50 c'è ancora niente errore trovato anche se non sono sicura di che cosa si tratti sostanzialmente Siccome la funzione toglie il nodo viene chiamata all'interno di diverse chiamate ricorsive n è la radice in questo caso qui in cui stiamo cambiando l'albero ti non è solo quando è la radice dell'intero albero potrebbe anche semplicemente essere la radice del sottoalbero su cui stiamo lavorando per cui in effetti questo test qui è scorretto e va sostituito dal testo che dice n = te in questo caso stiamo cancellando la radice dell'albero e quindi andiamo a risistemare quanto sopra è il risultato vi compilando 2:28:30 Mo a risistemare quanto sopra è il risultato vi compilando nuovamente dovrebbe questa volta tornare e infatti riusciamo a cancellare il nodo 50 ecc eccetera eccetera Ok piuttosto complicato In realtà il concetto fondamentale è in che maniera si svolge questa operazione a volte come nella cancellazione dalle liste altre volte invece come nella cancellazione da tabelle Però fruttando anche questo meccanismo di cancellazione come nelle liste di togliere un nodo facendo risalire l'unico sotto albero esistente A questo punto io direi che in realtà siamo abbiamo concluso l'esercizio e quindi possiamo terminare la lezione

Figura 11.3: Dizionari: operazioni

Sia $\mathcal{T}$ l'insieme di tutti i possibili dizionari su U

I dizionari ammettono tipicamente le seguenti operazioni

- calcolo del minimo: dato un dizionario, ne restituisce l'elemento minimo

  $$\min : \mathcal{T} \to U$$

  Se il dizionario è vuoto, restituisce un valore fittizio $+\infty$

- calcolo del massimo: dato un dizionario, ne restituisce l'elemento massimo

  $$\max : \mathcal{T} \to U$$

  Se il dizionario è vuoto, restituisce un valore fittizio $-\infty$

Figura 11.4: Dizionari: operazioni

In matematica basta definire un oggetto per crearlo

Nelle implementazioni concrete, questo in genere non vale
Quindi è opportuno definire

- creazione: crea un dizionario vuoto

$$\text{crea} : () \to \mathcal{T}$$

- distruzione: distrugge un dizionario

$$\text{distrugge} : \mathcal{T} \to ()$$

Si noti la scomparsa del concetto di posizione proprio di altre strutture che rappresentano insiemi

- non si accede agli elementi del dizionario tramite una posizione, ma solo tramite il loro ordine

Figura 11.5: Albero binario di ricerca

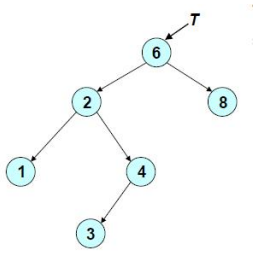
Un albero binario di ricerca (ABR) T su un insieme ordinato U è un

- albero binario
- in cui tutti i nodi sono diversi tra loro
- in cui ogni nodo segue tutti quelli del proprio sottoalbero sinistro

$$a_i \succ a_j \quad \text{per ogni } j \in T_s(i) \text{ e per ogni } i \in T$$

- in cui ogni nodo precede tutti quelli del proprio sottoalbero destro

$$a_i \prec a_j \quad \text{per ogni } j \in T_d(i) \text{ e per ogni } i \in T$$



Gli ABR gestiscono insiemi non generici, ma totalmente ordinati

Figura 11.6: ABR: implementazione con puntatori
Gli *ABR* hanno le stesse implementazioni degli alberi binari

Nell'implementazione a puntatori:

- l'albero corrisponde a un puntatore al nodo radice
- ogni elemento dell'albero corrisponde a una struttura con
  - il dato $a \in U$
  - un puntatore alla radice del sottoalbero sinistro (`NULL` se non esiste)
  - un puntatore alla radice del sottoalbero destro (`NULL` se non esiste)
  - un puntatore al nodo padre (`NULL` se non esiste)

```
#define NO_TREE NULL                                    (albero vuoto)

typedef nodo *ABR;                          (l'albero è l'indirizzo della radice)

typedef struct _nodo nodo;
struct _nodo {
   U a;                                          (U è il tipo del nodo generico)
   nodo *Ts;
   nodo *Td;
   nodo *padre;
};
```

Lezione 13

La lezione è centrata su un esercizio per impratichirsi sui seguenti argomenti:

- realizzazione di un ambiente interattivo testuale
- alberi binari di ricerca rappresentati con puntatori

Chi notasse eventuali incoerenze o errori, oppure avesse dubbi sul contenuto di queste pagine e dei codici, è pregato di segnalarmeli per contribuire a migliorare la qualità dei materiali del corso.

Problema

Si vuole scrivere un programma `ricerca.c` che carica da un file di testo un insieme di numeri interi tutti diversi tra loro e permette all'utente di eseguire una serie di operazioni sull'insieme stesso:

- `member` i: chiede al programma di indicare se il numero intero i fa parte dell'insieme oppure no;
- `insert` i: chiede al programma di aggiungere il numero intero i all'insieme;
- `delete` i: chiede al programma di cancellare il numero intero i dall'insieme;
- `print`: chiede al programma di stampare l'insieme per valori crescenti;
- `empty`: chiede al programma di indicare se l'insieme è vuoto oppure no;
- `min`: chiede al programma di indicare l'elemento minimo dell'insieme;
- `max`: chiede al programma di indicare l'elemento massimo dell'insieme;
- `exit`: chiede al programma di terminare.

Formati di ingresso e di uscita Il file di ingresso che contiene l'insieme iniziale dei numeri interi li riporta semplicemente uno dopo l'altro separati da spazi o a capi. Per esempio:

```
95 77 748 40 242 303 793 573 247 727 87 491 832 594 302 229 967
```

All'istruzione `member` i il programma risponde stampando a video i `in T` oppure i `non in T`, a seconda del risultato (i è il numero indicato dall'utente).

All'istruzione `print` il programma risponde stampando a video i numeri in ordine crescente, su una sola riga, separati da spazi bianchi.

All'istruzione `empty` il programma risponde stampando a video `T e' vuoto` o `T non e' vuoto` secondo il risultato.

Alle istruzioni `min` e `max` il programma risponde stampando `min(T) =` i oppure `max(T) =` i, dove i è l'elemento minimo o massimo richiesto.

Traccia della risoluzione

Prima di eseguire l'esercizio, si possono scorrere interamente i lucidi per acquisire il concetto di dizionario, di albero binario di ricerca e le basi della implementazione degli alberi binari di ricerca con puntatori.

L'esercizio parte dal file `ricerca0.c`, che contiene solo la consueta funzione `InterpretaLineaComando` per recuperare il nome del file dalla linea di comando e copiarlo nella corrispondente variabile.

Prima fase (`ricerca.c`) Avviamo la realizzazione *top-down* dell'algoritmo scomponendo il problema in due fasi, risolte da corrispondenti funzioni chiamate nel `main`, dichiarate fra i prototipi prima del `main` e definite dopo il `main` (all'inizio vuote, poi da riempire):

1. `CaricaABR`: carica i numeri interi dal file indicato nella linea di comando in un albero binario di ricerca;
2. `EsegueComandi`: riconosce ed esegue i comandi forniti dall'utente da tastiera.

La prima funzione si limita ad

- aprire il file;
- creare un albero vuoto;
- leggere i numeri dal file;
- inserirli via via nell'albero;
- chiudere il file ;
- restituire l'albero.

L'implementazione è estremamente semplice, ipotizzando di aver disponibili le funzioni per creare un albero vuoto e per inserirvi numeri interi. Queste funzioni (e le altre introdotte nei lucidi) sono dichiarate nel file `abr.h`, mentre il file `abr0.c` ne contiene le definizioni vuote. La dichiarazione della struttura dati è praticamente identica a quella dell'albero binario discussa nella lezione precedente. La differenza principale è che il dato contenuto in ogni nodo è un semplice numero intero `val`, anziché un operando e/o un operatore, ma questo dipende banalmente dallo specifico problema che stiamo affrontando. Rispetto alla definizione astratta, si è aggiunta una funzione di stampa ordinata.

La seconda funzione è abbastanza interessante, perché realizza (diversamente da tutti gli esercizi svolti sinora) un programma che interagisce con l'utente per un tempo indefinito, finché questi non indica di terminare. Il programma consiste in un semplice ciclo che:

- legge da tastiera l'operazione indicata dell'utente;
- riconosce l'operazione e il suo eventuale operando;
- esegue l'operazione.

La lettura dell'operazione può avvenire con la funzione `gets`, ma è più sicuro usare la funzione `fgets(Comando,ROW_LENGTH,stdin);`, che nasce per la lettura da file di testo, ma si può adattare alla lettura da tastiera specificando `stdin` come il file da leggere, e ha il vantaggio di specificare il numero massimo di caratteri letti, in modo da evitare che si superi la dimensione della stringa di lettura `Comando`. Per riconoscere l'istruzione, osserviamo che vi sono due categorie di operazioni: quelle con un operando intero (da `member` a `delete`) e quelle senza operandi. La funzione `sscanf`, applicata sulla stringa `Comando` con la stringa di formato `"%s %d"` distingue i due casi in base al fatto che essa restituisce 2 nel primo caso e 1 nel secondo. Definita la classe cui appartiene l'operazione, si può identificarla esattamente confrontando l'operando con quelli sopra elencati. Un modo alternativo di procedere potrebbe essere di indicare esplicitamente gli operandi nella stringa di formato (per esempio, `sscanf(Comando," member %d",&val);`)[2] e usare una cascata di costrutti di selezione (`if ... else if ...`). Una volta identificata l'operazione, basta chiamare la corrispondente funzione di gestione dell'albero e stampare il risultato a video nel formato richiesto.

Seconda fase (`abr1.c`) La fase successiva dell'esercizio richiede di cominciare a realizzare la libreria per la gestione dell'albero binario di ricerca. Cominciamo con le funzioni più semplici:

- `creaABR`: crea un albero vuoto;
- `distruggeABR`: distrugge un albero binario di ricerca;
- `ABRvuoto`: restituisce `TRUE` se l'albero è vuoto, `FALSE` altrimenti;
- `insertABR`: restituisce l'albero ottenuto aggiungendo un elemento all'albero dato.

Le prime tre funzioni non riguardano il contenuto informativo, ma solo la topologia dell'albero, che è identica a quella dell'albero binario. Non stupisce quindi che siano identiche alle corrispondenti funzioni dell'albero binario. Anche la loro complessità sarà la stessa: costante per la creazione e il test di vuotezza, lineare per la distruzione.

La funzione `insertABR` e quelle che vedremo nelle fasi successive dell'esercizio, invece, tengono conto del contenuto informativo, e quindi sono abbastanza diverse da quelle dell'albero binario. In particolare, in un albero binario di ricerca la posizione degli elementi è assegnata automaticamente, anziché essere indicata in modo esplicito, come negli alberi binari. L'implementazione qui descritta tende ad avvicinarsi a quella delle dispense del corso, ma vi sono le seguenti differenze:

1. le dispense fanno in genere l'ipotesi semplificativa che l'albero su cui si opera non sia vuoto, mentre qui consideriamo il caso generale;
2. le dispense assumono che i valori da inserire nell'albero non ne facciano già parte e quelli da cancellare ne facciano parte, mentre qui consideriamo anche i casi rimanenti;
3. le dispense assumono che le funzioni passino i parametri per indirizzo, mentre in C i parametri vengono passati per valore;

[2] Lo spazio iniziale nella stringa di formato serve per riconoscere il comando anche nel caso in cui l'utente digiti qualche spazio prima della parola chiave `member`.

4. per semplificare alcuni passaggi faremo uso della funzione `costruisceABR`, che coincide con la funzione `costruiscealbero` descritta nella lezione precedente; nelle dispense vengono invece usate le funzioni `CREA_NODO_SIN` e `CREA_NODO_DES`.

La funzione ausiliaria `costruisceABR` viene usata solo all'interno della libreria, ma non è accessibile all'utente esterno, per mantenere la libreria vicina alla struttura astratta descritta nei lucidi. Affinché possa essere chiamata dalla funzione `insertABR`, deve essere definita prima di quella, in modo che il compilatore l'abbia già letta quando arriva a occuparsi della funzione chiamante.

Procediamo con la funzione `insertABR`, al solito con un'implementazione ricorsiva:

- primo *caso base*: se l'albero è vuoto, la funzione `costruisceABR` restituisce la soluzione, cioè un nuovo albero con due sottoalberi vuoti e il nuovo dato come informazione associata alla radice;
- secondo *caso base*: se il nuovo dato coincide con la radice dell'albero corrente, si esce senza far nulla, restituendo l'albero corrente;
- *caso ricorsivo*: a seconda che il nuovo dato preceda o segua quello associato alla radice, si considera il sottoalbero sinistro o destro e si procede ricorsivamente nel sottoalbero individuato; al termine, si restituisce l'albero corrente (questo è importante, perché il nuovo dato inserito modifica la radice di uno degli alberi incontrati durante l'esecuzione e quindi bisogna accertarsi che tale radice sia modificata).

La complessità di un inserimento è proporzionale al numero di chiamate ricorsive, dato che ad ogni chiamata si esegue un numero costante di operazioni. Quindi, è proporzionale alla profondità h dell'ABR: $\Theta(h)$. L'intera costruzione di un ABR di n elementi ha quindi complessità in $\Theta(hn)$. La profondità h è compresa fra $\log_2 n$ e n, secondo che l'albero sia più o meno bilanciato. Vale il caso pessimo quando gli elementi vengono inseriti in ordine crescente o decrescente, e quindi scendono via via sempre nel sottoalbero destro o sinistro. Vale il caso ottimo quando gli elementi vengono inseriti in modo del tutto casuale, e quindi i vari sottoalberi tendono ad avere tutti la stessa profondità.

Terza fase (`abr2.c`) Nella fase seguente, realizziamo le funzioni di consultazione dell'albero, cioè:

- `stampaABR`: stampa a video gli elementi dell'albero in ordine crescente;
- `memberABR`: determina se un dato elemento appartiene o no all'albero;
- `minABR`: restituisce l'elemento minimo dell'albero;
- `maxABR`: restituisce l'elemento massimo dell'albero.

Con la funzione `stampaABR` ci mettiamo in grado di vedere l'albero caricato da file, verificando che il codice sia corretto semanticamente, e non solo sintatticamente. La funzione si riduce a una semplice visita in in-ordine, dato che, per definizione, il valore conservato nella radice di un albero è segue quelli contenuti nel sottoalbero sinistro e precede quelli contenuti nel sottoalbero destro. Ci si può chiedere se stampare qualcosa nel caso di albero vuoto: conviene non farlo perché durante la visita ogni foglia dell'albero ha due alberi vuoti, e quindi otterremmo molte stampe

non desiderate. La funzione è ovviamente lineare nel numero degli elementi ($\Theta(n)$), come nel caso degli alberi binari.

Procediamo con le funzioni di interrogazione dell'albero, partendo da `memberABR`. L'implementazione è ricorsiva:

- primo *caso base*: se l'albero è vuoto, l'elemento cercato non sta nell'albero;
- secondo *caso base*: se l'elemento cercato coincide con quello conservato in radice, ovviamente sta nell'albero;
- *caso ricorsivo*: a seconda che il nuovo dato preceda o segua quello associato alla radice, si riapplica la funzione al sottoalbero sinistro o a quello destro.

Qui la complessità è proporzionale al numero di chiamate ricorsive, cioè alla profondità h dell'albero: $\Theta(h)$. Siccome la chiamata ricorsiva è terminale, è possibile ricavare facilmente e meccanicamente una semplice implementazione iterativa. Le varie chiamate ricorsive, in effetti, consistono semplicemente nel salvare sulla pila di sistema i vari sottoalberi visitati (sinistri o destri secondo l'esito del confronto fra il valore di i e quello conservato nel nodo corrente). Giunti al caso base:

- se si arriva a un sottoalbero vuoto, si restituisce `falso`;
- se la radice coincide con i, si restituisce `vero`.

La ricerca dell'elemento minimo con `minABR` ammette anch'essa un'implementazione ricorsiva di complessità $\Theta(h)$:

1. primo *caso base*: se l'albero è vuoto, restituisce un valore convenzionale corrispondente a $+\infty$;
2. secondo *caso base*: se la radice dell'albero non ha un sottoalbero sinistro, il suo valore associato è l'elemento minimo, e viene direttamente restituito;
3. *caso ricorsivo*: se l'albero ha un sottoalbero sinistro, si riapplica la funzione a tale sottoalbero.

Ancora una volta, la chiamata ricorsiva è terminale, ed è possibile ricavare un'implementazione iterativa in modo meccanico. Anziché salvare sulla pila di sistema i vari sottoalberi visitati (sempre sottoalberi sinistri). Al termine della ricorsione, si arriva al caso base, in cui la foglia non ha un sottoalbero nella direzione richiesta, e l'algoritmo restituisce il valore associato alla foglia individuata. Di conseguenza, l'implementazione iterativa di `minABR` consiste nel:

- restituire $+\infty$ se l'albero è vuoto;
- scendere lungo il sottoalbero sinistro finché si arriva a un nodo che non ha più tale sottoalbero, e restituire il valore associato al nodo.

Per la funzione `maxABR`, valgono considerazioni assolutamente analoghe a `minABR`, con la sola differenza che sia l'implementazione ricorsiva sia quella iterativa scendono sempre lungo il sottoalbero destro e che la funzione restituisce $-\infty$ se l'albero è vuoto.

Quarta fase (`abr3.c`) Per concludere, affrontiamo la funzione più complessa dell'esercizio, cioè la funzione `deleteABR`, che cancella un elemento da un albero dato. Ancora una volta la funzione è ricorsiva:

- primo *caso base*: se l'albero è vuoto, non fa nulla;
- *caso ricorsivo*: a seconda che il dato da cancellare preceda o segua quello associato alla radice, riapplica la funzione al sottoalbero sinistro o a quello destro;
- secondo *caso base*: se l'elemento da cancellare è conservato in radice, si possono distinguere due casi:
  - se la radice ha un solo nodo figlio (o nessuno), si può direttamente appendere tale figlio (se esiste) al nodo padre, "cortocircuitando" l'albero;
  - se la radice ha due figli, eliminarla compromette la struttura gerarchica dell'albero; la soluzione è (come nella cancellazione da tabelle) sovrascriverla con un altro elemento, e cancellare quest'ultimo, ma per non compromettere l'ordine totale degli elementi bisogna scegliere in modo oculato quale cancellare.

Discutiamo in maggiore dettaglio le due operazioni più complicate delineate più sopra. L'operazione che "cortocircuita" l'albero è abbastanza semplice a parole, ma presenta molte piccole differenze a seconda che il figlio esista o no, che sia a sinistra o a destra e che vi sia un nodo padre a cui appendere il figlio oppure no (se si cancella la radice, non esiste un nodo padre). Per semplicità di lettura, e per essere fedeli alla modalità *top-down*, accorpiamo il tutto in un'apposita funzione `ABR toglienodoABR (ABRnodo *n, ABR T)`, che elimina un nodo `n` da un albero `T` restituendo il nuovo albero `T` modificato. La funzione procede come segue:

1. determina l'unico sottoalbero esistente (eventualmente nessuno);
2. se c'è un sottoalbero lo appende al padre (eventualmente assente);
3. se c'è un padre, determina in quale sottoalbero del padre sta il nodo da cancellare e sostituisce ad esso il sottoalbero del nodo cancellato;
4. finalmente, dealloca il nodo.

La funzione può essere usata solo se il nodo ha al massimo un figlio, e per efficienza non controlla che la condizione sia rispettata: lo deve fare l'utente. Non riportiamo tale funzione nel file di intestazione `abr.h` perché non è verosimile che un utente esterno lo usi per operare sull'albero e perché la funzione presuppone esplicitamente un'implementazione a puntatori nella definizione del suo secondo argomento, che è un `nodo *`. Affinché possa essere chiamata dalla funzione `deleteABR`, deve essere definita prima di quella, in modo che il compilatore l'abbia già letta quando arriva a occuparsi della funzione chiamante.

Nel caso in cui invece il nodo da cancellare abbia due figli, bisogna cercarne un altro che sia cancellabile semplicemente e che possa essere messo al posto di quello da cancellare senza compromettere l'ordinamento totale. Un possibile candidato è l'elemento massimo che precede quello da cancellare[3], dato che è certamente una foglia e che si può spostare senza cambiare l'ordine complessivo. Per trovare tale nodo, definiamo una funzione `argmaxABR` che esegue le stesse operazioni di `maxABR`, ma restituisce un nodo anziché un dato, e la applichiamo al sottoalbero sinistro del

[3] Un altro sarebbe l'elemento minimo che lo segue.

nodo da cancellare. In questo modo, troviamo il massimo del sottoalbero sinistro, che è esattamente il nodo cercato. Anche questa funzione, che è di servizio e usa esplicitamente (nel tipo del risultato) l'implementazione a puntatori, non comparirà nel file di intestazione, ma verrà semplicemente definita prima della funzione chiamante `deleteABR`. Ottenuto il nodo, ne copiamo il dato sulla radice e lo cancelliamo, dato che ora è un duplicato. Possiamo usare `toglienodoABR` senza compromettere la struttura perché il nodo certamente non ha un sottoalbero destro. Nel complesso, anche nel caso pessimo, la complessità è proporzionale al numero di chiamate ricorsive, cioè alla profondità h dell'albero: $\Theta(h)$.

11.1 Esercizi

11.1.1 Esercizio

Si modifichi la libreria realizzata nell'esercizio di laboratorio in modo che possa trattare insiemi di parole (sequenze di caratteri senza separatori), anziché di numeri.

Capitolo 12

Algoritmi di ordinamento "efficienti"

Questo capitolo riprende e approfondisce argomenti trattati anche nelle dispense di teoria, cioè algoritmi di ordinamento la cui complessità non è quadratica, ma inferiore. Più precisamente, questi algoritmi raggiungono il limite asintotico inferiore di complessità $\Theta(nlogn)$ che le dispense di teoria mostrano essere invalicabile per gli algoritmi di ordinamento basati su confronti. A stretto rigor di logica, non tutti gli algoritmi che discuteremo hanno questo caso pessimo: *HeapSort* e *MergeSort* raggiungono questo caso limite, mentre *QuickSort* ha ancora un caso pessimo $\Theta(n^2)$. Si tratta però di un caso pessimo estremamente raro e sfortunato, e il caso medio è $\Theta(nlogn)$, è molto frequente e ha costanti di proporzionalità tipicamente favorevoli. Questo è il motivo del suo nome: *QuickSort* sta per "ordinamento veloce".

Dopo aver scorso alcuni concetti teorici, affronteremo l'implementazione di questi algoritmi in linguaggio C per il semplice caso di ordinare un insieme di numero interi, e concluderemo con qualche considerazione sull'uso pratico di questi algoritmi in situazioni meno addomesticate.

Sia U un insieme dotato di un ordine debole $\preceq$ *(sono ammessi i doppioni)*

Il problema dell'ordinamento ha come

- istanza: qualsiasi vettore V su U
- soluzione: il vettore V' permutazione di V tale che

$$V[i] \preceq V[j] \text{ per ogni } i \leq j$$

Esempio:

$V =$

5	2	8	4	7	1	3	6

$V' =$

1	2	3	4	5	6	7	8

Figura 12.1: Il problema dell'ordinamento

Il problema dell'ordinamento è definito come il problema di trasformare un vetto-

re V di elementi di un dato insieme U in un altro vettore V' che è una permutazione del primo. L'insieme U, detto *insieme universo* o *insieme base* deve essere dotato di una relazione di ordine debole, cioè deve esistere una relazione binaria per cui, presi due elementi dell'insieme, siamo sempre in grado di dire che il primo precede il secondo o che lo segue (quindi la relazione è completa), e questa relazione deve essere riflessiva e transitiva. Nei testi si impone a volte che l'ordine sia totale, cioè che tutti gli elementi di U siano ordinati. Nelle applicazioni questo non è strettamente necessario. Per esempio, in un insieme di persone da ordinare per età possono esistere persone della stessa età che non sono la stessa persona. La sostanza dei fatti non cambia molto.

Dato un insieme universo e un suo sottoinsieme rappresentato da un vettore V indicizzato da 1 a n, si vuole ottenere un altro vettore indicizzato da 1 a n che sia una permutazione del primo, cioè in cui gli indici si scambiano fra loro. La condizione essenziale è che quando un elemento ha indice minore o uguale di un altro ($i \leq j$), il primo elemento $V[i]$ preceda (non strettamente) il secondo elemento $V[j]$.

12.1 Selection Sort

L'algoritmo *SelectionSort* divide il vettore V in due tabelle:

- il sottovettore iniziale contiene gli elementi più piccoli disordinati
- il sottovettore finale contiene gli elementi più grandi ordinati

La seconda tabella è vuota all'inizio, poi cresce un elemento alla volta:

- l'elemento massimo della prima tabella si sposta nella seconda e diventa il suo primo elemento

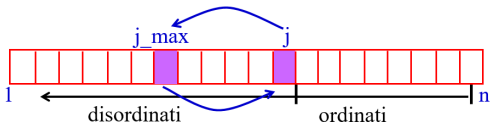


Figura 12.2: SelectionSort

Abbiamo già descritto nel Capitolo **??** due algoritmi di ordinamento quadratici basati sul graduale allargamento di un insieme ordinato fino a comprendere tutti gli elementi del vettore stesso. Abbiamo visto in particolare l'algoritmo di *SelectionSort*, che nelle dispense di teoria non è trattato. Questo algoritmo gestisce sul vettore V due tabelle: la parte iniziale del vettore è una tabella che contiene gli elementi più piccoli del vettore, in generale in modo disordinato; la seconda parte del vettore è una tabella che contiene gli elementi più grandi in modo strettamente e rigorosamente ordinato. Inizialmente, la partizione è banale: la prima tabella contiene tutto il vettore, mentre la seconda tabella è vuota. Siccome la seconda tabella è vuota, possiamo tranquillamente ipotizzare che contenga gli elementi più grandi del vettore, e che siano perfettamente ordinati (su un insieme vuoto si può ipotizzare che valga qualsiasi proprietà). L'algoritmo procede allargando via via la seconda tabella a spese della prima, e mantenendo valide queste ipotesi:

1. che la seconda tabella sia ordinata;
2. che contenga gli elementi più grandi.

Se si riesce a fare questo, al termine dell'algoritmo la seconda tabella ha interamente divorato la prima e costituisce l'insieme di tutti gli elementi del vettore (non solo i più grandi, perché non ce ne sono altri) perfettamente ordinati. Come si fa a mantenere le due proprietà? Molto semplicemente, l'elemento spostato di passo in passo dalla prima tabella alla seconda non è scelto a caso, ma è l'elemento massimo della prima tabella. Questo perché spostandolo dalla prima alla seconda tabella sappiamo dove metterlo: va in cima, perché è minore o uguale a tutti gli elementi originali della seconda tabella. Inoltre, sappiamo che non si sposterà nel seguito dell'algoritmo, perché gli elementi rimasti nella prima tabella gli sono precedenti, e quindi non gli usurperanno mai la posizione che ha conquistato nel passo in cui è stato spostato. L'idea è semplice, e abbiamo visto che anche l'implementazione è molto semplice.

```
SelectionSort(V,n)                                  
{                                                   
  for (j = n; j > 1; j--)                           Σ_{j=1}^{n} (...)
  {                                                 
    i = TrovaIndiceMassimo(V,j);                    g(j)
    Scambia(&V[i],&V[j]);                           Θ(1)
  }                                                 
}                                                   

TrovaIndiceMassimo(V,n)                             g(j) = ...
{                                                   
  iMax = 1;                                         Θ(1)
  for (i = 2; i <= n; i++)                          Σ_{i=1}^{j} (...)
    if (V[i] > V[iMax]) iMax = i;                   
  return iMax;                                      Θ(1)
}                                                   
```

SelectionSort è un algoritmo quadratico: $T(n) \in \Theta\left(n^2\right)$

Figura 12.3: SelectionSort: complessità

La Figura 12.1 riporta lo pseudocodice di *SelectionSort*: un ciclo va allargando la seconda tabella con un indice j che cala da n fino a 2, e il suo corpo consiste nella ricerca dell'elemento di valore massimo nella prima tabella e nello scambio e aggiornamento dell'indice j che spostano tale elemento dalla prima alla seconda tabella. La ricerca, implementata in modo sequenziale, costa $\Theta(n)$ e ripeterla n volte costa $\Theta\left(n^2\right)$, da cui la complessità quadratica di *Selection Sort*. Questo è inefficiente.

Se riuscissimo a trovare l'elemento massimo in tempo costante, potremmo ottenere una complessità lineare, perché eseguiremmo n iterazioni di complessità costante. Fortunatamente, esiste una struttura che consente di determinare l'elemento massimo di un insieme in tempo costante.

Questa struttura si chiama *heap*, e ha lo scopo di "gestire" un insieme di elementi, consentendo di eseguire inserimenti, cancellazioni e modifiche, ma soprattutto di

La ricerca dell'elemento massimo rende *SelectionSort* inefficiente

Se richiedesse tempo costante, la complessità scenderebbe a $O(n)$

La struttura dati astratta *max-heap* consente di

- gestire un insieme (compresi inserimenti e cancellazioni)
- determinarne l'elemento massimo in tempo costante

Si può usarla per rappresentare la prima tabella, ma occorre

1. costruirla al principio quando coincide con l'intero vettore
2. mantenerla aggiornata quando perde l'elemento massimo

Queste operazioni sono però abbastanza efficienti

- aggiungere un elemento nuovo
- modificare un elemento
- cancellare un elemento (*ci serve solo questa operazione, in realtà*)

richiedono tempo logaritmico

Figura 12.4: Inefficienza di *SelectionSort*

determinarne l'elemento massimo (in base a una data relazione d'ordine). Questo consente di sostituire la prima tabella del vettore (quella disordinata) con un *heap*. Fatto questo, la ricerca dell'elemento massimo richiederà solo tempo costante. Ciò non è privo di costi: i costi consistono, anzi tutto, nel fatto che l'*heap* inizialmente deve contenere tutti gli elementi del vettore dato, e costruirlo richiede tempo: bisogna impostare l'intero vettore in maniera che sia un *max-heap*. Dopo di che, bisogna che l'*heap* rimanga aggiornato nonostante che l'elemento massimo venga estratto e spostato nella seconda tabella. Infatti, l'*heap* va decrescendo. Non è automatico che cancellare un elemento di un *heap* sia un'operazione efficiente. Entrambe le operazioni (costruzione ed estrazione) hanno un costo.

La cosa interessante è che si riesce a gestire in tempo lineare la costruzione di un *heap* e in tempo logaritmico la cancellazione dell'elemento massimo. Più in generale, qualsiasi aggiunta, cancellazione o modifica di un elemento è possibile in tempo logaritmico.

Che cos'è un *heap*? Un *heap* è semplicemente un albero binario radicato e ordinato, come quelli trattati nel Capitolo **??**, cioè un albero i cui nodi hanno un sottoalbero sinistro e un sottoalbero destro, eventualmente vuoti. Si tratta però di un albero binario *quasi completo*.

Un albero binario completo è costituito da nodi interni tutti di grado 2 e da un unico livello di foglie, tutte alla stessa profondità h. Un albero binario quasi completo ha solo foglie a profondità h e foglie a profondità $h-1$; inoltre, le foglie a profondità $h-1$ stanno “in fondo a destra”, mentre quelle a profondità h stanno “in fondo a sinistra” e c'è al massimo un nodo di grado 1, che ha soltanto il figlio di sinistra (vedi Figura 12.1). Sostanzialmente, è un albero che, se non è completo, manca solo di foglie che stanno nell'ultimo livello e sulla destra. Queste proprietà topologiche definiscono gli *heap* in genere.

Un *max-heap* è caratterizzato da due proprietà:

1. è un albero binario (*Lezione* 12) quasi completo, cioè con
   - foglie di profondità h o $h-1$
   - al più un nodo di grado 1
     - a profondità h
     - col solo figlio sinistro
     - con tutti i nodi alla sua destra nello stesso livello di grado nullo

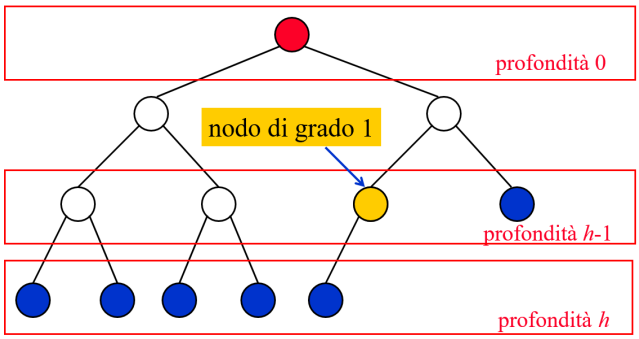


Figura 12.5: La struttura *max-heap*

Un *max-heap* è caratterizzato da due proprietà:

2. i nodi sono etichettati con valori tratti da un insieme U ordinato e l'etichetta di ogni nodo non precede quelle degli eventuali nodi figli

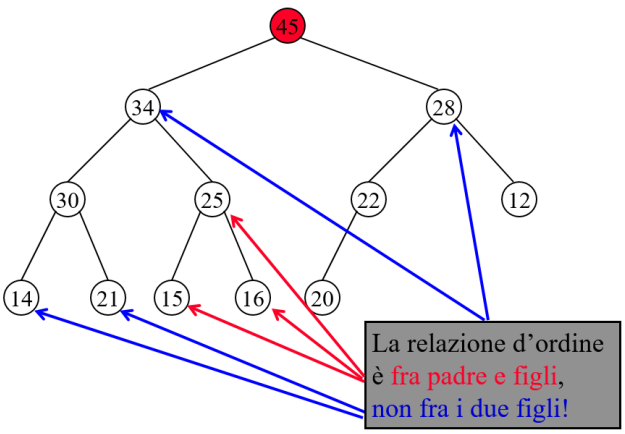


Figura 12.6: La struttura *max-heap*

Il *max-heap* è caratterizzato dal fatto che i nodi sono dotati di etichette che appartengono a un insieme universo U dotato di una relazione di ordine debole, e le etichette obbediscono a dei vincoli: ogni nodo è associato a un'informazione chi precede in senso non stretto le informazioni associate ai nodi figli (eventualmente presenti). Quindi le etichette delle foglie sono libere, ma un nodo come la radice dell'albero nella Figura 12.1, che ha etichetta 45 deve avere valori maggiori o uguali dei due figli (34 e 28). Il nodo di etichetta 28 è maggiore di 22 e 12, il nodo 22 è maggiore dell'unico figlio (20), mentre il nodo 12 è una foglia e quindi non deve soddisfare nessuna particolare proprietà. L'ordinamento è imposto fra ascendenti e discendenti, fra padri e figli. Nessuno ordine è imposto sui livelli, tra fratelli. Per esempio, la radice ha figli (34 e 28) tali che il primo è maggiore del secondo, mentre il nodo 30, in basso a sinistra, ha figli (14 e 21) con il primo minore del secondo. Entrambe le possibilità sono lecite: la relazione non è completa.

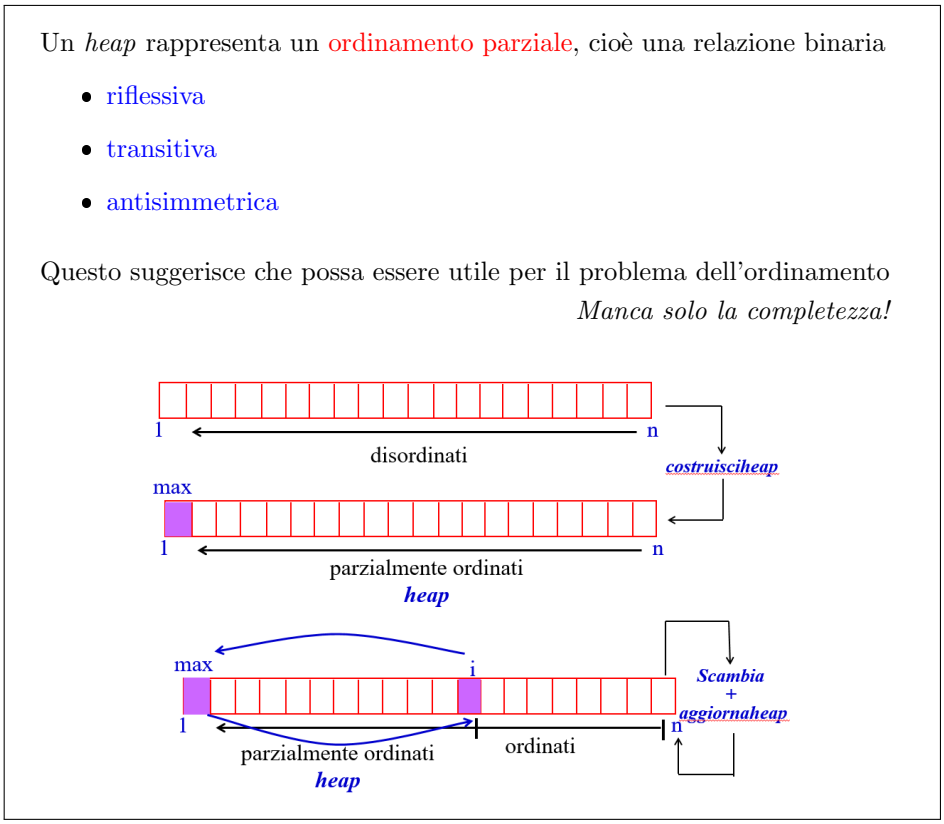


Figura 12.7: *Heap* e ordinamenti parziali

Una relazione d'ordine esiste: un *heap* rappresenta un ordinamento parziale, dotato delle proprietà riflessiva, transitiva e antisimmetrica. Riflessiva vuol dire che ogni nodo è maggiore o uguale a se stesso. Transitiva vuol dire che, se un nodo padre è maggiore uguale al figlio, e il figlio maggiore o uguale a un nipote, ovviamente il padre sarà maggiore o uguale al nipote. L'antisimmetria, a stretto rigore, è la proprietà per cui due nodi che si precedono vicendevolmente sono lo stesso nodo. In realtà le etichette possono essere identiche (infatti, la relazione d'ordine sull'universo U è debole, ma è anche vero che i due nodi non sono lo stesso nodo.

Tutto questo suggerisce che, in effetti, un *heap* non sia del tutto ordinato, ma almeno parzialmente lo sia. Ora, nell'algoritmo *SelectionSort* abbiamo una prima

tabella disordinata che cede elementi a una seconda tabella ordinata. Se implementiamo il primo sottoinsieme con un *heap*, avremo una prima tabella che nasce disordinata, viene resa parzialmente ordinata imponendo la struttura di *heap*, e poi si possono far cedere al primo sottoinsieme elementi alla tabella totalmente ordinata cercando ogni volta l'elemento massimo. Il punto fondamentale è che l'elemento massimo è sicuramente il primo. L'operazione di eliminazione del primo elemento può essere realizzata come nelle tabelle non ordinate sovrascrivendo il primo elemento con l'ultimo e riducendo la dimensione della tabella. Questa operazione però distrugge la proprietà di *heap*, che va reintrodotta con un'opportuna funzione.

Osserviamo di passaggio (ma ne faremo uso in altri capitoli) che esistono anche *min-heap*, che hanno la stessa struttura topologica dei *max-heap* (albero binario quasi completo), ma godono di una relazione parziale di ordine fra nodi padri e nodi figli esattamente opposta: il nodo padre deve avere etichetta minore o uguale a quella dei due (eventuali) nodi figli.

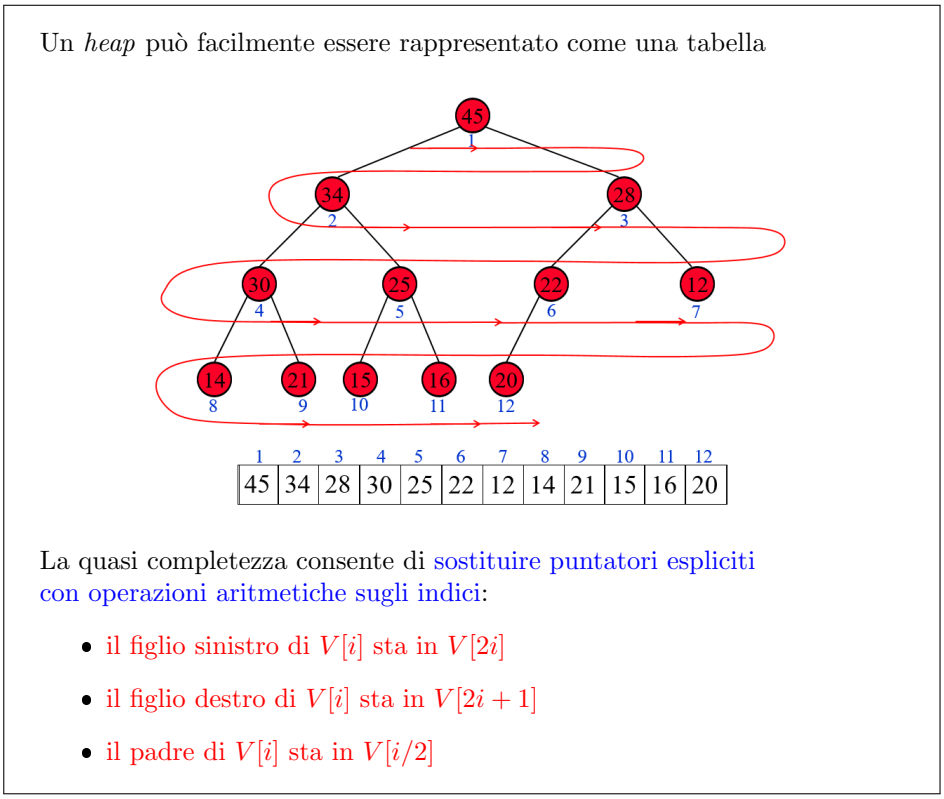


Figura 12.8: *Heap* e tabelle

Una proprietà fondamentale degli *heap* è che essi sono rappresentabili non solo come alberi binari (con i puntatori), ma soprattutto come vettori o tabelle, un po' come le pile e le code potevano essere rappresentate come liste a puntatori, ma anche semplicemente come vettori o tabelle.

Si tratta semplicemente di riportare le etichette dei nodi, livello per livello, in una tabella (o un vettore, se il numero di elementi rimane costante). La radice sarà l'elemento di indice 1, i suoi figli sinistro e destro saranno gli elementi 2 e 3, e così via. Come si vede nella Figura 12.1, questo è consentito soltanto dal fatto che l'albero binario è quasi completo. Se l'albero non fosse quasi completo, e avesse lacune in mezzo, anziché solo nella parte in basso a destra, la tabella avrebbe celle

non utilizzate, in numero non prevedibile a priori. Invece, in questo modo abbiamo il esatto di celle strettamente necessario, e soprattutto la possibilità di muoversi come si fa di solito in un albero binario scendendo dal padre al figlio di sinistra o di destra, e risalendo da un nodo al nodo padre, non con puntatori espliciti, ma con operazioni aritmetiche sugli indici degli elementi nella tabella. Se si considera la radice, che ha indice 1, il figlio di sinistra ha indice esattamente doppio (2), e il suo figlio di sinistra ha indice 4, cioè tutte le volte si raddoppia l'indice. Invece, per passare al figlio di destra, basta raddoppiare e incrementare di 1 l'indice: il figlio di destra della radice (indice 1) ha indice $2 \cdot 1 + 1 = 3$, il figlio di destra di questo nodo ha indice $2 \cdot 3 + 1 = 7$. Se si vuole risalire da un nodo al padre, basta dividere per due l'indice del nodo con la divisione intera, cioè trascurando il resto: $i/2$ è il padre del nodo i. Tutto questo rende estremamente semplice navigare in un albero binario quasi completo. Infatti, un albero binario quasi completo, che sia o non sia un *heap*, si rappresenta in generale in questo modo, che rispetto all'implementazione a puntatori fa risparmiare molto spazio (tutti i puntatori).

Le operazioni fondamentali sugli *heap* sono

- creazione a partire da un semplice vettore non ordinato

$$\text{creaheap} : \mathcal{V}_{n,U} \to \mathcal{H}_{n,U}$$

- aggiornamento, cioè ripristino dell'etichettatura corretta in un nodo, assumendo che essa valga in ogni altro nodo

$$\text{aggiornaheap} : \mathcal{H}_{n,U} \times \mathbb{N}^+ \to \mathcal{H}_{n,U}$$

(*qui c'è un lieve abuso, dato che non si parte da un* heap *corretto*)

Figura 12.9: Operazioni fondamentali sugli *heap*

Le operazioni fondamentali che ci serve fare sugli *heap* sono:

la possibilità di crearli e la creazione avviene prendendo un vettore generico non ordinato di n elementi dell'insieme Universo trasformandolo in un heap di elementi sull'universo la seconda operazione importante che di cui avremmo bisogno è quella di aggiornamento questa operazione di aggiornamento In cosa consiste e consiste nel fatto che il nostro la nostra struttura a rigore non è esattamente unità infatti la notazione che sto usando è parzialmente sbagliata non l'ha trovata una migliore e comunque lavoriamo solo qui quindi non è così importante comunque l'idea è che abbiamo un vettore In sostanza che dovrebbe rappresentare unip che in realtà non lo rappresenta perché in una posizione i è sbagliato cioè quella posizione a un etichetta non è maggiore uguale a quella di entrambi i figli ma è minore a quella di uno dei due figli o addirittura di entrambi i figli e quello che bisogna fare in sostanza è riparare questo guasto ipotizzando che però tutto il resto del vettore sia corretto goda della proprietà hip quindi che sia un guasto assolutamente limitato per cui ho scritto che la funzione Aggiorna i parte da unip e da una posizione intera compreso fare un ed Ennio e ti tuition hip in realtà non parte da ogni parte da qualcosa che potrebbe anche non esserlo ma che se non lo è non lo è a causa di un'unica posizione Dopodiché Invece quello che dobbiamo ottenere effettivamente unip del tutto corretto Allora adesso cerchiamo di capire come realizzare queste operazioni partiamo da l'aggiornamento perché la creazione fa uso dell'aggiornamento e quindi Così sappiamo già In che cosa consiste allora l'aggiornamento sostanzialmente consiste nel

dire io ho una certa posizione un certo modo nel quale l'etichetta è sbagliata perché invece di essere maggiore uguale a quella dei due figli è minore di quella di uno o di entrambi i figli alla quello che bisognava fare per prima cosa andare a capire qual è

0:18:20 Che bisogna fare in sostanza è riparare questo guasto ipotizzando che però tutto il resto della vettore sia corretto goda della proprietà hip quindi che sia un guasto assolutamente limitato per cui ho scritto che la funzione Aggiorna i parte da unip e da una posizione intera compreso fra 1 e dello e istituisce unip in realtà non parte da ogni parte da qualcosa che potrebbe anche non esserlo ma che se non lo è non lo è a causa di un'unica posizione Dopodiché Invece quello che dobbiamo ottenere effettivamente unip del tutto corretto Allora adesso cerchiamo di capire come realizzare queste operazioni partiamo da l'aggiornamento perché la creazione fa uso dell'aggiornamento e quindi Così sappiamo già In che cosa consiste allora l'aggiornamento sostanzialmente consiste nel dire io ho una certa posizione un certo modo nel quale l'etichetta è sbagliata perché invece di essere maggiore uguale a quella dei due figli è minore di quella di uno o di entrambi i figli alla quello che bisognava fare per prima cosa andare a capire quale dei due figli ha l'etichetta più grande perché perché qui ci troviamo di fronte a tre nodi che non sono messi bene ma in realtà di quei tre nodi il massimo deve salire a occupare la posizione del padre è quella prima operazione che si fa per aggiornare le unità è determinare tra i due figli quello che ha l'etichetta massima lì sotto nello pseudocodice vedete la determinazione del figlio di sinistra moltiplicando i per due quella del figlio di destra moltiplicando i per due motivi fumando uno e dopodiché c'è proprio la determinazione di quale di queste tre posizioni corrisponde al valore massimo c'è un test ulteriore il test ulteriore è che le posizioni SD siano posizioni minori e uguali al n perché perché può darsi che in realtà il nodo considerato non abbia figli ora se il nodo non ha figli o se ne ha uno soltanto se non ha figli È chiaro che va tutto bene perché non ci sono problemi con i proprietari persone a uno soltanto il confronto va fatto solamente con quella lo ha determinato il figlio di etichetta massima Bisogna capire se questa etichetta è corretta Cioè è minore o uguale a quella del nodo padre Oppure non lo è e se non lo essere più grande dobbiamo scambiare queste due etichette e questo punto automaticamente dei tre nodi che potenzialmente stiamo considerando la nuova etichetta del padre sarà quella giusta e cosa sarà successo sarà successo che uno dei due figli eh e l'altro dei figli invece è stato modificato Quindi è possibile che a Valle di questa operazione sostanzialmente uno dei due figli si sia si trovi ad essere in una posizione sbagliata nuovamente come ha cambiato la sua etichetta Può darsi che non rispetti più la proprietà hip bene Cosa succede a questo punto molto semplice che abbiamo già una procedura che serve a riparare unico in realtà posizione è la procedura stessa di aggiornare la richiamiamo ricorsivamente Infatti l'ultima operazione E questa giornata hip che viene eseguita nella posizione di Max come potete facilmente immaginare questa è una ricorsione terminale è abbastanza facilmente sostituibile con una versione iterativa non lo faremo può essere però un buon esercizio ovviamente da fare detto ciò passiamo in realtà in grado anche di costruire un heap Che cos'è che si tratta di fare sostanzialmente si tratta di partire da un vettore che è un semplice vettori quindi è completamente disordinato e Questo significa che tutti Possiamo sempre vederlo come un albero binario quasi completo imponendo la forza quella struttura ad albero che abbiamo visto prima visitando i nodi per livelli Ma la proprietà ed in generale non va da nessuna parte è però possibile imporre la forza come sfruttando la proprietà Aggiorna hip la cosa importante è che bisogna sostanzialmente Ricordarsi di aggiornare tutti i per quelli che hanno la dice in ciascuno dei nodi dell'albero procedendo in ordine inverso dalle foglie alla radice non dalla radice alle foglie per il semplice motivo che la procedura Aggiorna hip sostanzialmente ipotizza che la proprietà sia proprietà Iper sia rispettata in tutti i livelli inferiori e di conseguenza procediamo dalle foglie alla radice

Stiamo procedendo da nodi i cui livelli inferiori sono già stati trattati quindi sono già corretti all' indietro e sono modi in cui invece è possibile che la procedura debba ancora essere applicata per cui la funzione che hai non fa altro che correre tutti gli elementi dall'ultimo al primo 0:23:10 Ancora essere applicata per cui la funzione che non fa altro che scorrere tutti gli elementi dall'ultimo al primo all'indietro in linea di principio da na1 chiamando la funzione Aggiorna hip in realtà ci si può anche rendere conto del fatto che è inutile applicare la procedura Aggiorna hip sulle foglie per il banale motivo che non avendo figli non è possibile che la proprietà Ipsia violata in essi Allora dove cominciano le foglie in un heap di n elementi esattamente nella posizione n mezzi al solito trovata con la divisione intera per il banale motivo che tutte le posizioni perché gli eventuali figli sarebbero due volte io due volte di più uno e quindi due volte i vuol dire due per rende mezzi cioè n che è l'ultimo elemento possibile quindi n mezzi è l'ultimo nodo possibile che può ancora avere dei figli se si parte nei mezzi più uno il figlio di sinistra e due volte nei mezzi più uno quindi n + 2 n + 2 sicuramente di figli non ne ha Allora questo Cosa significa significa che stiamo eseguendo che vende mezzi volte la funzione Aggiorna eapci si potrebbe porre il problema di valutare la complessità questa valutazione complessità è già stata fatta nella teoria e si è arrivati alla non ovvia conclusione che la complessità di questo algoritmo è lineare teta DN Nonostante che si chiamino n mezzi volte delle funzioni che potenzialmente sono logaritmiche ma il motivo è che in realtà questi logaritmi intendono essere lì a via molto velocemente piccoli e quindi alla fine la somma viene tetta DNA Ma i dettagli li avete già visti nella lezione di teoria allora a questo punto possiamo concludere la inizializzazione introduzione diciamo questa lezione dicendo che la procedura di spot consiste nel creare inizialmente unip sul vettore di dato nel scorrere gli elementi andando a prendere tutte le volte il primo elemento io uguale a uno e scambiandolo con l'ultimo elemento della prima tabella di quella dello stesso quindi mi sostanzialmente andando a scambiare l'elemento vd1 con l'elemento vdj di iota dove iota va decrescendo da n giù fino a fino a uno e siccome questo distrugge la proprietà di Hip sul parte iniziale sul primo sottovettore che è quello compreso fra le posizioni 1 e le posizioni iota e questo comporta che l'IP stesso debba essere riaggiornato e qui come vedete un magicamente riaggiornato corretto un errore al volo il link che viene aggiornato è costituito da quella parte del vettore B che sta dalla posizione 1 la posizione 7:59 e la parte che va corretta è la posizione 1 la come vedete accostando Selection Sort and heap Sort sono quasi lo stesso algoritmo qui si può discutere sul fatto che siano lo stesso algoritmo No io spesso tendo a dire che sono lo stesso algoritmo come struttura dati diversa in cui invece di lavorare su una tabella disordinata una tabella ordinata all'interno di un vettore si lavora su un hip e su una tabella ordinata implementati entrambi all'interno dello stesso vettore lightshot a un aggravio in rosso costituito dalla costruzione del Lip in tempo lineare ed all'aggiornamento in tempo logaritmico logaritmo di yota ad ogni iterazione prendi perenni interazioni ma ha anche un grosso vantaggio il corso ba sta nel fatto che la funzione che determina l'elemento Massimo all'interno del lettore che costava tetta DNA è ridotta a una semplice determinazione dell'elemento uno che poi non Scriveremo neanche così nel codice Scriveremo direttamente vd1 e questo abbassa la complessità temporale da teta di ennequadro accetta di n log n Allora questo punto direi che siamo perfettamente in grado di andare a mettere le mani nel codice e si tratta non tanto di un problema pratico quanto proprio di andare a implementare questo algoritmo nella sua forma più semplice è dato un file che contiene un elenco di 100 numeri interi qualsiasi con un po' di ripetizioni e dopodiché abbiamo Fai iniziare questo fa iniziare lo stesso file che abbiamo usato per cominciare l'implementazione di insertion Sort Selection Sort cioè un file che contiene l'interpretazione della linea di comando che semplicemente va a prendere dal dall'utente il nome del file che conterrà i numeri da ordinare Dopodiché carica questo ho detto vedi in 0:28:00 Dall'utente il nome del file che conterrà i numeri da ordinare Dopodiché ca-

rica questo vettore di interi andando anche dal locale il vettore vivi è definito come vinta È vinta è il classico puntatore intero che maschera in qualche modo il fatto che si tratta di un vettore dinamico e dopodiché noi dovremmo ordinare il vettore stesso e abbiamo già la funzione che lo stamperà in maniera da poter verificare che sia tutto andato a buon fine non andiamo a determinare a vedere come funziona il caricamento procediamo invece direttamente nell'implementazione cosa dobbiamo fare dobbiamo ordinare il vettore e chiamiamolo anche direttamente col suo nome l'algoritmo è un algoritmo di Sort che lavora sul vettore V di Lunghezza n Questo è quanto questo algoritmo possiamo andare più sopra a fornire un dichiarazione avrà come parametro un vettore di interi è un intero e possiamo anche dire Che ordina il vettore V di Lunghezza n con l'algoritmo heapsort a questo punto costo viviamo la nostra dichiarazione vuota in basso e possiamo se vogliamo verificare al solito la correttezza del codice possiamo anche provare a compilarlo con tutte le opzioni di sicurezza questo file si chiama Sottozero vogliamo costruire un file heapsort. è questo file lanciato sulla fai i soldi che cosa fa lo stampa così come è perché lo carica ha una funzione di ordinamento che non fa nulla e lo stampa così come quindi perlomeno se assicuri di aver letto correttamente i dati possiamo anche confrontare questi numeri con quelli che ci se ci vengono fornite inizialmente verificare che sono effettivamente gli stessi nello stesso ordine Allora che cos'è che dobbiamo fare molto semplice possiamo anche ispirare codice Dobbiamo creare unip dobbiamo andare gradualmente a ridurre la prima tabella allargando invece la seconda ea ogni passo sostanzialmente scambiare il primo elemento dell'hip che anche il massimo con l'ultimo elemento che è quello in posizione iota e ri aggiornare diminuire la dimensione dell'heap quindi portarlo dai 8 ai 7:59 eri aggiornare la posizione uno che è scorretta Quindi in sostanza quello che ci tocca fare è una funzione crea hip che questa volta lavora sul intero vettore V di Lunghezza n e ovviamente doveva in qualche maniera dichiararle definirla dopodiché faremo scorrere questo intero iota che quindi va in qualche modo dichiarato questo iota lo facciamo scendere da n giù fino a uno qui è idiota maggiore di uno ne avevamo parlato ragionando sulla selections l'idea è che quando ci rimane un unico elemento non c'è bisogno di cercare il massimo e metterlo dall'ip nel vettore ordinato è già siamo già ordinati ed è già nella posizione giusta Ecco perché non ci fermiamo in uno ma ci fermiamo in due un pochino prima e ognuno di questi passi abbiamo detto che andiamo a determinare la posizione massima che è la posizione io uguale a uno ma è inutile calcolare le esplicitamente la possiamo riportare direttamente all'interno della funzione scambia e dovremmo ovviamente andare a scambiare queste due posizioni che sono due numeri interi andremo a scambiare vd1 lo cambieremo con vdj la posizione finale dell'hip le passiamo per argomento perché vanno fisicamente modificate questi due numeri Dopodiché tranquillamente Ci limitiamo ad aggiornare Lip quale hip Dip V che va dalla posizione 1 alla posizione yacht a meno 1 quella che va modificata quella posizione critica che potrebbe violare la sicuramente quasi sicuramente mi voleva la proprietà è la posizione 1 questo punto siamo a posto e possiamo procedere a definire le tre funzioni crea hip scambia e aggiorna i pkwy si apre una piccola discussione Cella di natura assolutamente estetica e stilistica abbiamo visto che quasi sempre abbiamo adottato la strategia di prendere la chiamata riportarla all'inizio nell'elenco dei prototipi e costruire una dichiarazione e poi sotto una definizione nel lavorare su alcuni 0:32:50 E costruire una dichiarazione e poi sotto una definizione nel lavorare su alcune librerie abbiamo fatto una cosa leggermente diversa Cioè abbiamo detto ma in realtà la funzione Chiama crea hip è solo una È solo questa qui è solo heapsort E allora tanto vale scriverla subito sopra Cioè possibile perfettamente lecito andare a definire la funzione crea hipp comincio a scriverla che è una volta che riceve un vettore di interi e riceve un intero è perfettamente possibile prendere questa funzione e andare addirittura a definirla qui perché tanto il compilatore quando scorre questo file trova la dichiarazione è la definizione di Crea hip prima

dell’uso questo è corretto e si fa quando si vuole nascondere questa funzione secondaria dicendo che viene usata soltanto qui dentro nel nostro caso va bene un modo va bene non c’è una vera differenza Visto che stiamo facendo un esercizio molto semplice Diciamo che in linea di principio se questa funzione di heapsort Prevedo che possa essere da utilizzare Beh allora bisogna capire probabilmente finirà in una libreria a parte non starà dentro questo fa il CID prima capire se questa libreria nella sua interfaccia nel suo fa il punto h esporrà al mondo esterno heapsort e basta E allora tanto vale tenere crea hip nascosta oppure invece forniva al mondo esterno un hip funzionale cioè un vettore Con eventualmente altre funzionalità per esempio l’aggiunta la cancellazione la creazione e la modifica di numeri piuttosto che altro per azioni del genere a quel punto fosse una operazioni di creazione e potrebbe essere utile per comunicare con il mondo esterno e allora io seguirò questa strategia che però non è assolutamente ovvia e andrò ad aggiungere la funzione crea Hip elenco delle intestazioni Perché Perché nel momento in cui decidessi di fare questa libreria mi troverò ad avere qua sotto tutti i corpi delle funzioni cioè il file. c e mi troverò ad avere qua sopra tutte le dichiarazioni cioè tutti i prototipi e quindi mi dovevo ad avere il file. h e sarà molto più comodo che non andare in giro a cercare le cose allora Questo che stiamo facendo è in sostanza la creazione crea un hip nel lettore sulla Vettore ti ho detto vedi di Lunghezza n Adesso possiamo scrivere la funzione sopra o sotto non importa L’importante è che l’intestazione preceda l’uso quindi crea i play in alto e precede questa chiamata questa funzione Qui per il momento la lasciamo vuota per la solita modalità top-down e poi ci vediamo con le altre funzioni cioè con scambia e aggiorna ipg la funzione scambia Che cosa farà in sostanza scambia i numeri interi Diamogli dei nomi a e b esempio è una funzione Void che riceve un puntatore a un numero intero a è un puntatore a un numero intero B e deve scambiarli e poi abbiamo già che ci siamo facciamo anche questa una funzione che aggiorna Lip V di Lunghezza n si chiama yacht a meno della chiamata ma la lunghezza degli in generale la n e la posizione che va corretta qui è la posizione 1 nella chiamata ma in generale sarà la posizione i cinegiornali di Lunghezza n a partire dalla posizione i cosi l’abbiamo resa più generale questa funzione è valida In ogni caso è una funzione Void che riceve un vettore di interi che riceve una Lunghezza n e una posizione it queste due funzioni si possono portare sotto facciamo i due corpi vuoti dopo aggiungo un paio di considerazioni di natura anche qui molto stilistica e poco più avrete probabilmente notato che ho messo scambia maiuscolo le altre Minuscole semplicemente perché tendenzialmente ho tenuto minuscoli in tutto il corso i nomi delle funzioni di base che lavorano su strutture dati mentre scambia potrebbe esserlo non esserlo ma non è una funzione che lavora sul libro e una funzione che lavora su due numeri interi Dopodiché possiamo procedere direi ah ah sì l’altra cosa che volevo dire era che se per caso che noi in realtà abbia 0:37:40 Possiamo procedere direi ah ah sì l’altra cosa che volevo dire era che se per caso che noi in realtà abbiamo realizzato questa struttura da tip gestendo esplicitamente un vettore è un intero una tabella non abbiamo costruito come abbiamo fatto in tutte le lezioni precedenti una libreria che nascondesse l’IP all’interno di una struttura con delle funzioni ad hoc eccetera eccetera sarebbe stato la strategia corretta diciamo volendo perseguire fino all’ultimo la l’approccio alle strutture dati astratte in realtà qui L’obiettivo della lezione e soprattutto algoritmi di ordinamento e questi algoritmi sono talmente semplici ed elementari che ho voluto evitare di sovraccaricare la lezione anche con questo aspetto Quindi in realtà è perfettamente possibile fare una incapsulamento che mi ha bloccato potrebbe essere un buon esercizio esercizio consigliabile ma non lo facciamo in questa lezione Allora la funzione di creazione del Lip come abbiamo visto nei lucidi sostanzialmente consiste nel chiamare la funzione di aggiornamento sulla prima metà del vettore partendo dalla metà e andando all’indietro potrei quasi fare la tempesta Ma insomma vado lentamente in maniera da e scriverlo da descrivere quello che sto facendo Partiamo dalla posizione n mezzi

e andiamo via via regredendo all'indietro nella nella lungo il vettore fino ad arrivare alla posizione 1 alla prima e ad ognuno di questi passi andiamo ad aggiornare Lip costituito dal vettore B di Lunghezza n nella posizione i lo scambio è la solita di solito meccanismo che utilizza una variabile ausiliaria che fa da buffer diciamo questa variabile si carica di uno dei due numeri interi quindi per esempio * Pia questo uso del P mi ricorda il fatto che voglio copiarmi il numero intero a non il puntatore Dopodiché nel numero intero a vado a scrivere lo so lo scrivo con il numero B e poi il numero B lo so lo scrivo con il valore che ho salvato da parte banale tutte le funzioni operazioni di scambio che sono estremamente frequenti in algoritmi di ordinamento poi finiscono per usare delle procedure di questo genere come funziona L'aggiornamento del Lippo e te la procedura Più significativa e complessa abbiamo detto che l'idea e intanto di capire quale dei due figli ha l'etichetta Maggiore quindi per prima cosa dobbiamo conoscere i due figlie l'ora chiamiamoli SD sinistro e destro e ovviamente il figlio di sinistra corrisponderà alla indice iperdue mentre il figlio di destra l'indice Iper due più uno non c'è nessun bisogno di scrivere esplicitamente S è soltanto per ricordarselo cioè potremmo usare due per i in in giro nel nell'algoritmo Ma questo rende tutto più chiaro A questo punto possiamo cercare di capire quale per fare due figli ha l'indice Massimo o meglio quale fai due figli e il padre ha l'indice Massimo partiamo con una ipotesi Allora ci serve l'indice Massimo Quindi abbiamo una variabile ad hoc partiamo con l'ipotesi che sia il padre che è ancora l'ipotesi più tranquilla perché vuol dire che allora non bisogna fare assolutamente nulla e poi cerchiamo di valutarla che succede se per caso invece è il figlio di sinistra adesso è più grosso cioè che succede se vds è maggiore DVD Imax succede che allora Shh candidarsi come a candidarsi come figlio come nodo di etichetta massima Questa scrittura è potenzialmente sbagliata perché stiamo dando per scontato che è se esista come figlio ma in realtà non è detto quindi la prima cosa che dobbiamo valutare se per caso essere veramente un figlio cioè se è femmina = n solo allora andremo a vedere se ha etichetta maggiore del padre altrimenti qui rischiamo Allora mi scuso per la transizione ma c'è stato un blocco stavamo dicendo che stiamo cercando di capire quale fra il padre e figlio di finito il figlio di destra è l'indice dell'elemento Massimo e abbiamo controllato che il figlio di sinistra fosse effettivamente un nodo vero è che la sua etichetta superasse quella del Padre in questo caso possiamo andare a dire che il Max non è più il padre e il figlio di sinistra e poi 0:42:30 Quando vedeva che la sua etichetta superasse quella del Padre in questo caso possiamo andare a dire che il Max non è più il padre e il figlio di sinistra e poi possiamo andare a ripetere questo test sul figlio di destra valutando Intanto se veramente un modo ragionevole esistente poi se la sua etichetta supera quella del nodo Massimo che potrebbe a questo punto essere il padre o il figlio di sinistra e nel caso la superi andare a salvare il valore di di in Imax ti ho detto quello che potrebbe succedere e che comunque il padre sia correttamente iMac so cioè che è uguale in tal caso non si fa nulla ma se per caso hai Max non è il padre e solo allora andremo a apportare le modifiche del caso la prima modifica sta nel fatto che il l'etichetta deve passare devono scambiarsi le etichette il padre e il maggiore dei due figli Queen andare a scambiare i valori che sono valori interi del vettore di imposizione di Max e il valore del vettore di imposizione li passo per indirizzo perché devono fisicamente modificarsi Dopo di che cosa succede succede che la posizione in questa volta diventata corretta perché contiene il massimo fra i due figli e però uno dei due figli quello che non è Imax è corretto E però quello che invece hai Max potrebbe non essere più corretto quindi il vettore V di Lunghezza n Lip vidi di Lunghezza n nella posizione Imax ma quantomeno verificato per non dire eventualmente aggiornato a questo punto dovremmo essere in grado di fare una prova cioè di andare a compilare e valutare se l'ordinamento funziona e quanto meno su questo semplice esempio l'algoritmo funziona ci sono buone speranze che si Effettivamente da questo punto di vista siamo in grado di abbiamo realizzato la heapsort Non ricordo se ho

fatto notare che questa procedura ricorsiva a una ricorsione terminale Quindi in effetti è trasformabile in una procedura iterativa è un buon esercizio farlo Non non lo facciamo in questa in questa lezione terminato l'argomento li ripasso Insomma dell'argomento deepsort possiamo passare a un secondo algoritmo di ordinamento efficiente che è l'algoritmo di ordinamento per fusione cioè merge Sort In che cosa consiste sostanzialmente sostanzialmente l'idea molto semplice è quella del Divide et impera Intanto sono algoritmi ricorsivi si amersfoort Sort sia il successivo quicksort e di conseguenza useremo pezzi del vettore V che indicizza eremo fra un indice sinistro testo Io li indicherò con sd anziché supporre che il vettore Evviva da1n allora lui come al solito e quindi applica il principio della divisione del problema da un intero vettore a due sottovettori due per semplicità questi due sottovettori sono nel caso di me al Sort esattamente identici sono metà e metà e si ottengono determinando l'elemento mediano questo perché il modo più semplice di farlo in assoluto dopodiché se è la principio c'è la la fase della imperatore della risoluzione dei due sotto problemi che viene fatta ricorsivamente richiamando l'algoritmo e Supponendo che l'algoritmo di per sé di alla soluzione corretta cioè che in ciascuno dei due sottovettori riescono i permuti gli elementi in maniera che siano allora interno ordinati Dopodiché la questione è che dobbiamo vi combinare le soluzioni cioè adesso abbiamo un vettore costituito di due sottovettori di lunghezza più o meno identica ciascuno dei due ordinato Ma nel complesso i due sottovettori non sono ordinati quello che bisogna fare fonderli e questo fa la procedura famosa procedura me quindi riassumendo Questa è la struttura è un algoritmo ricorsivo a un caso base il caso base Qual è il caso base quello in cui il lettore è talmente piccolo da essere già ordinato quindi o è di Un solo elemento o è nullo Quand'è che una vittoria Un solo elemento quando gli indici sinistro resto coincidono S = Di quand'è che è vuoto per convenzione possiamo dire che quando l'indice sinistro è addirittura a destra del destro vuol dire che il vettore vuoto quindi quando esce ma giovedì di prova per essere maggiore o uguale a Dio abbiamo il caso base che consiste nel non fare nulla per questo non si vede nulla nello del codice non seguo codice invece scemino e di di ci troviamo di fronte effettivamente alle chiamate di così le Allora queste chiamate ricorsive Che cosa fanno applicano Prima di tutto la divisione dei problemi e sotto problemi che è una 0:47:20 Martedì così allora queste chiamate dico Siri Che cosa fanno applicano Prima di tutto la divisione del problemi e sotto problemi che è una semplicissima operazione di calcolo dell'indice mediano S più di mezzi poi ci sono le chiamate ricorsive sui due sotto problemi e quindi Infine c'è la combinazione delle due soluzioni una unica soluzione con questa funzione merge Sembra tutto molto banale c'è da fare una piccola osservazione utile la sedazione è che l'indice mediano viene calcolato ascendesse più di mezzi Ma questo SPD mezzi arrotondato per difetto o per eccesso e i due vettori includono quale dei due vettori include l'indice mediano questi Sembrano dei problemi stupidi ma non lo sono affatto perché se per caso noi calcolassi Mo l'indice mediano arrotondando lo per difetto Ma poi ha più passimo questo indice mediano al secondo sottovettore quindi modello SM 11 e da m.a.d. Ecco Che cadrebbe la correttezza dell'algoritmo per il banale motivo che i due sottovettori non sarebbe strettamente più piccoli del vettore di partenza diamo l'esempio S = 5 = 6 se abbiamo s = 5,65 + 6 fa 11 / 25 e mezzo arrotondato per difetto fa 5 ma se l'elemento 5 viene a più pazzo al secondo sottovettore vuol dire che il primo va da 5 sam1 che quattro Quindi è vuoto mentre il secondo vada m che è 5 a di' che sei e quindi è identico al vettore di partenza e siccome dentro questa chiamata c'è un'altra chiamata ricorsiva perfettamente identiche capito che abbiamo una infinita di chiamate ricorsive con uno Stack che esplode semplicemente perché abbiamo sbagliato la decomposizione del problema Quindi se si arrotonda per difetto l'indice medianova del primo sottovettore ci potrebbe anche a rotonda recesso mettendo l'indice mediano del secondo Questa è semplicemente la scelta più comune Allora c'ho detto possiamo dedicarci invece a vedere che cosa Perché

l'algoritmo funziona più o meno molto velocemente per una questione di induzione matematica perché si dimostra facilmente che funziona per vettori molto piccoli di zero o un elemento basta non fare nulla e perché vale un principio di induzione matematica per cui se si suppone induzione forte se si suppone che l'algoritmo sia valido su tutti i vettori fino a dimensione ennemenouno Beh allora vuol dire che queste due chiamate amersfoort che lavorano su vettore di dimensione n mezzi arrotondata più o meno per difetto o per eccesso sono corretti Quindi avevo che otteniamo due sottolettore ordinati D'altra parte è vero come abbiamo visto prima che si arriva al caso base quello che manca è semplicemente di dimostrare che la funzione merge è veramente partendo da due sottovettori riproduce un vettore ordinato complessivo se questo vale è facile fare l'ultimo passaggio dalla dimostrazione per induzione la quale dimostra che se noi abbiamo una algoritmo valido fino a dimensione nemmeno uno Allora l'algoritmo vale anche per la dimensione n e quindi a questo punto vale per tutti le dimensioni Cioè per tutte le possibili istanze detto ciò si può analizzare la complessità No Ho dimenticato la procedura merde Giusto Allora ma Che cosa fa la procedura merge la procedura medi sostanzialmente prende i due sottovettori quello che vada s a m e quello che vada mp1a di e li ricompone in un unico vettore ordinato Come è possibile farlo Ci vuole un vettore d'appoggio esiste una versione molto più sofisticata che non ha bisogno di questa cosa ma è al di là degli scopi dei Confini di questo corso in realtà si tratta di avere un vettore di appoggio B nel quale copiare Via via gli elementi dei due sottovettori cioè prendiamo il frutto vettori iniziale e finale e confrontiamo gli elementi minimi del primo e del secondo il minimo dei due lo mettiamo in fondo al vettore B poi andiamo di nuovo a prendere il minimo dei due elementi e la mettiamo in fondo AB è chiaro che in questo modo stiamo ricostruendo in maniera ordinata il vettore B contiene elementi che sono tutti più piccoli o uguali a quelli dei due sottovettori e quindi siamo va tutto bene finché i due sottovettori non sono entrambi non vuoti se uno dei due sottolettori è vuoto questa cosa non può più procedere perché non si può più prendere il minimo ma quello che si può fare a prendere il sottovettore residuo è in linea di principio metterlo in fondo al vettore B poi vedremo che non facciamo questa cosa ma facciamo una cosa leggermente più furba quando abbiamo fatto tutto questo Noi abbiamo il vettore complessivo ordinato in B dobbiamo ricopiarlo INA o invii 0:52:10 Quando abbiamo fatto tutto questo Noi abbiamo il vettore complessivo ordinato in B dobbiamo ricopiarlo INA o invii nel vettore corrente ed è chiaro che tutto questo è lineare perché ciascuno degli elementi viene copiato una prima volta dal vettore via il vettore B è una seconda volta dal vettore B il vettore a ha detto ok Quindi la complessità di merge è lineare questo ci permette di fare l'analisi di complessità che non faremo attentamente Perché l'avete già vista la richiameremo soltanto abbiamo una funzione ricorsiva una algoritmo ricorsivo in cui il caso base è in tempo costante perché consiste nel valutare se SM = di oppure minorili poi c'è Invece il caso di corsivo che è costituito da una operazione in tempo costante che il calcolo dell'indice mediano da due le chiamate ricorsive su vettore di dimensione n mezzi per difetto o per eccesso e poi da una me perché invece il Mehari è i vari teoremi Teorema fondamentale delle equazioni ricorrenti o altre metodi con cui si svolgono queste questioni ci mostrano ci permettono facilmente di dimostrare che la complessità sta nello spazio theta di n log n e quindi è ottimale a meno di costanti moltiplicative tutto questo è particolarmente ci permette di procedere alla implementazione Che cos'è che sostanzialmente dobbiamo fare torniamo al nostro punto di partenza andiamo a recuperare una copia di questo file Sort 0 che avevamo Però non vorrei un attimo ok Eccoci qua andiamo a recuperare una copia in questo file so Zero da cui eravamo partiti quello che ha soltanto il caricamento dei dati e la loro stampa in maniera da essere totalmente puliti e questa volta invece di ordinare il vettore con la procedura il Sort la ordineremo con la procedura me sorta è chiaro che da un punto di vista astratto si tratta di fare sostanzialmente la stessa cosa Cioè

li prende il vettore V di Lunghezza n dobbiamo andare come sempre a costruirci la nostra dichiarazione non sto prototipo dicendo che abbiamo questo Vettore che è un Lindt con Lunghezza n dobbiamo andare a ordinare il vettore di di Lunghezza n con l'algoritmo merge-sort questa volta e possiamo andare a recuperare a costruire un corpo vuoto per la nostra funzione che andremo Poi tranquillamente a riempire possiamo ovviamente ispirarci alla pseudocodice che è praticamente un codice la funzione è rigorosamente ricorsiva c'è un caso base che sarebbe S maggiore o uguale a di nel quale non si fa assolutamente nulla ma noi è inutile che lo scriviamo quindi direttamente passiamo a considerare il invece il caso in corsivo in cui Esse è minore di 18 è il vettore a qualche cosa da fare che cosa ha da fare e bisogna andare a recuperare la posizione mediana osserviamo una cosa intanto sono andato giù bello piatto in realtà amersfoort non lavora su un vettore V di Lunghezza n dovendo essere chiamata ricorsivamente nella definizione di messo ad ci vogliono indice sinistro e l'indice destro quindi torniamo un attimino a Monte spesso succede si tratta di partire da un vettore che ha indice Sinistro 1 indice destro n la nostra definizione lavora su un sottovettore del vettore di compreso fra gli indici del SDN che di questo ordina il sottovettore del vettore di compreso fra s e d con l'algoritmo di merce short ed è questo che ci apriamo ci apprestiamo effettivamente a realizzare come algoritmo Allora a questo punto tutto torna abbiamo un indice mediano n che è quello che possiamo Per prima cosa andare a valutare che cosa dobbiamo valutare dobbiamo valutare la semisomma di s e d di questo è di per sé una divisione intera perché se li sono 0:57:00 Se è di di questo è di per sé una divisione intera perché SD sono interi e due un intero Quindi automaticamente questo questa operazione tronca Il risultato non dobbiamo fare niente di particolare in ci possiamo limitarci a chiamare ricorsivamente la funzione emerso volta sul vettore di fra gli indici slm la funzione mersolat sull'indice sul vettore lì fra gli indici mp1 e di e infine possiamo andare a fondere queste due parti del vettore di quella compresa fra S ed m e quella compresa fra mp1 che evidentemente inutile scrivere come pagamento è l'indice di questo l'enorme vantaggio tipico delle funzioni ricorsive che il costo delle chiamate sono in realtà già state dichiarate definite L'unica cosa che manca è preparare la funzione merge al solito qui possiamo metterla qua sopra potevo metterla in alto c'è da chiedersi se la funzione mese ipotizzando uno spostamento in una libreria potrebbe mai capitare di avere una funzione di dover chiamare da fuori una funzione emerge è abbastanza probabile che la funzione merge sia puramente ausiliaria per una chiamata in mezz'ora mentre la creazione e l'aggiornamento di unip invece suonano un pochino più generalmente utili per cui o questa volta tanto per far vedere anche un'altra stile di programmazione vado a mettere la funzione merge non fra i prototipi in alto ma semplicemente all'interno delle varie dichiarazioni definizioni e precisamente andiamo a costruirci una funzione che fonde il sotto vettore ordinato vi dalla posizione esser posizione m con il sotto vettore ordinato vi dalla posizione mp1 alla posizione di a questo punto abbiamo una funzione Void che riceve un vettore di interi e riceve tre semplici indici numerici interi Questa è la funzione che adesso ci apprestiamo a realizzare in che modo si può realizzare la funzione merge Allora abbiamo detto che la Funzione merge consiste nell'andare a scorrere la prima metà e la seconda metà del vettore di quella compresa fra sdm quella compresa fra mp1 e di siccome le dobbiamo scorrere ci vorranno due cursori quindi ci vorranno un indice per la prima è unica per la seconda metà Io ti chiamerei i 1 e i Ohi s e i di come preferite volevo fare DSi di Allora a questo punto dovremo però puoi anche andare a scrivere dei valori all'interno di un vettore ausiliario P Quindi intanto ci serve un vettore ausiliario B che sarà un vettore dinamico perché non sappiamo a priori quanto sarà lungo dipende dal dai valori di sdd la dimensione di bit e poi ci servirà un indice che con AB che potremmo chiamare i Questo potrebbe essere l'integrale di tutto ciò che ci serve allora Che cos'è che dobbiamo fare prima cosa dobbiamo andare ad allocare il vettore ausiliario Poi dovremmo andare in sostanza

a scorrere i sottovettori vsm EV da m più 1 grado Finché uno dei due non si svuota non è vuoto spostando ogni volta il minimo dei due nel vettore ausiliario P Questo è quello che dobbiamo fare quando tutto questo succederà e allora dobbiamo gestire il residuo cosa intendo con residuo intendo quel pezzetto finale o del primo o del secondo sottovettore che non è stato completamente copiato in B e questo lo faremo in maniera abbastanza astuta quindi non dettaglio già subito in che maniera lo faremo Diciamo che gestisce il residuo del primo o del secondo sotto lettore e quando finisci ne abbiamo sistemato il tutto possiamo andare a copiare il vettore ausiliario 1:01:50 Abbiamo sistemato tutto possiamo andare a copiare il vettore ausiliario di in V fatto tutto questo non ci serve più il lettore ausiliario possiamo liberarcene punti possiamo deallocare il vettore ausiliario B Allora alcune cose sono molto facili da fare al solito può valere la pena che mettiate in pausa e proviate a realizzare da soli queste queste funzioni la implementazione e la locazione di B è molto semplice si tratta banalmente di chiamare la solita funzione culloch callo cell Azzera anche è importante azzerarlo no Se volete essere molto efficienti potreste fare una malloch e Andrebbe bene lo stesso io faccio tanto per quanto deve essere lungo questo vettore B deve permettersi di ospitare al limite Tutti gli elementi dal indicesse all'indice di quindi dimmelo sp1 ok di meno S più uno io farò più due addirittura giusto per ragionare col vettore B che va da 1 a limite massimo è un po' uno spreco sono tutti Piccoli dettagli sui quali si può effettivamente ragionare Ma che non sono molto interessanti dal punto di vista della algoritmica quindi li evidenzio e poi però li ignoro testo Questo va fatto se la locazione è andata a buon fine quindi andiamo a vedere se per caso il vettore B il puntatore B è nullo che nel caso andiamo a segnalare che c'è stato un errore nella locazione del vettore ausiliario e a questo punto possiamo effettivamente in questo caso abbandonare il programma perché c'è poco da fare in fondo andremo a deallocare b quando non ci serve più veniamo la parte interessante Come si fa a scorrere questi due vettori Finché uno dei due non è vuoto allora Che cosa sono i s i d sono gli indici che stiamo andando a guardare questo vuol dire che correndo da sinistra a destra i due sottolettori da sinistra a destra perché prendiamo sempre l'elemento minimo PS inizialmente coinciderà con l'indice s e p inizio del primo sottovettore mentre ridi coincide con M + 1 cioè all'inizio del secondo sottovettore Ehi Che cos'è è l'indice nel quale si va a scrivere all'interno del vettore B Qui si potrebbero fare due cose sostanzialmente Si potrebbe mettere io uguale a uno e poi scrivere in quel vettore quindi ipotizzando che i sia la prima posizione libera in B Oppure si potrebbe dire che io e la ultima posizione occupata in B e allora si potrebbe dire io uguale a zero seguo questa strada non tanto perché sia ovviamente migliore ma perché alla fine quando implementeremo questa parte finale un pochino più complicata nella gestione delle residuo verrà più spontaneo procedere in questo modo ci sarà più avere in i l'ultima posizione occupata del vettore B quindi comincia con valore Nullo allora come facciamo a sapere se due vettori sono non Nulli entrambi e molto semplice non vuoti entrambi quando abbiamo terminato di vedere entrambi i lettori e finché i s rimane all'interno del vettore sinistro che finisce in.me finché i di rimane all'interno del vettore destro che finisce in di finché valgono entrambe queste condizioni Ecco che entrambi sottovettori sono non sono stati esauriti a questo punto vuol dire che abbiamo due elementi minimi uno di qua e uno di là e quindi possiamo sicuramente andare a prenderne uno e scriverlo dentro B ed ecco perché incrementiamo l'indice i per dire adesso scrivimi in questa posizione quale dei due elementi scriviamo possiamo andare a scrivere o l'elemento che sta nel vettore in posizione i.s.o. l'elemento che sta nel vettore in posizione Id il più piccolo dei due domande se sono uguali e se sono uguali è questione di preferenza Allora diciamo per motivi che saranno un pochino più chiari anche dopo che se sono uguali forse preferiamo il primo e di conseguenza noi diremo che se vds è più piccolo strettamente e questo è sicuro ma pure anche uguale devi dirgli di Allora di questi due conviene che dentro nel io vado a finire i s e quindi ha 1:06:40 Allora di questi

due conviene che dentro nel vettore ausiliario Vado a finire i s e quindi andremo a scrivere nella posizione Jesi ma del vettore ausiliario il valore di V bts finito No non è finito perché abbiamo incrementato Abbiamo letto il primo elemento del vettore sinistro quindi l'indice sinistro deve incrementare nel caso contrario invece che cosa facciamo la cosa complementare andremo a scrivere in B l'indice ID del vettore di partenza e andremo a incrementare l'indice di lettura ID l'indice di scrittura è già automaticamente incrementato qui e quindi quando siamo in fondo al ciclo stiamo correttamente puntando con i l'ultima posizione del vettore di ausiliario a questo punto quando usciamo abbiamo finito uno dei due sottovettori e l'altro no ci sono due casi Può darsi che ci rimanga un pezzo del primo Può darsi che ci rimanga un pezzo del secondo Ecco se ti rimane un pezzo del secondo in realtà Cosa vuol dire vuol dire che la parte finale del secondo sottovettore è costituita da elementi che sono più grossi di tutti quelli del vettore ausiliario c'è più grossi di tutti quelli sia del primo che del secondo sottolettore sono gli elementi massimi del vettore Ma se elementi massimi del vettore sono in fondo alla seconda metà vuol dire che sono in fondo al vettore cioè sono esattamente dove dovrebbero essere per cui se il secondo sottovettore a un residuo non c'è niente da fare per cui soltanto nel caso in cui ci sia un residuo del primo sottovettore dobbiamo fare qualcosa ci stiamo investito del primo sottovettore perché se c'è un residuo nel secondo Gli elementi sono già nella posizione corretta il pezzo finale del secondo sottovettore è già dove deve essere come facciamo invece a gestire vestito del primo sottolettore e vuol dire che gli elementi massimi dell'intero vettore sono in fondo non all'intero vettore Ma il primo sottovettore e allora dobbiamo semplicemente spostando bisogna semplicemente si può stare taglia elementi in fondo al vettore stesso se se abbiamo degli elementi fa la posizione GS è la posizione m Questi elementi da Yes am devono spostarsi in avanti fino ad arrivare alla posizione di E come si fa Non è difficile si tratta di scorrere finché i s non diventa uguale a m di passo in passo incrementando i s noi dobbiamo andare a prendere l'elemento che sta in posizione ISS e dove dobbiamo scriverlo nello stesso vettore in una posizione che sta più avanti Dove sta questo è il punto un pochino critico evidentemente sta in posizione ISS più qualcosa questo qualcosa è sostanzialmente la lunghezza del sottovettore di destra potremmo anche calcolarla ma c'è un modo molto più tranquillo di procede vede quello di scalare questo indice PS in che maniera togliendo m e sommando di perché questa espressione ha senso Perché caso particolare 6S è uguale m cioè se siamo arrivati in fondo Allora questa espressione è vidi e io sto contando l'elemento mediano Lo sto mettendo in fondo a destra che è esattamente quello che voglio fare Vedi tutto sommato al netto del fatto che i due elementi devono scalarsi con una differenza fissa uno se il fisso è abbastanza ovvio che questo o se sia di meno m cioè la differenza tra l'indice destro del tutto il vettore è l'indice detto del sottovettore di sinistra penso che siano abbastanza confusa ma con un minimo di calma in realtà è tutto abbastanza logico è ragionevole fatto questo quindi abbiamo sistemato l'eventuale residuo di Des delle sottovettore di sinistra e se non c'hai un residuo semplice Se non c'hai un residuo dire che i s Edge addirittura maggiore di m quindi in questo ciclo while non entreremo il residuo di destra non c'è da fare niente perché è già dove deve essere quello che dobbiamo fare copiare quel pezzo di elementi che sono stati scritti in B e dobbiamo copiare di nuovo in v e quanti sono questi elementi a rigore non lo sappiamo no lo sappia 1:11:30 Dobbiamo copiare di nuovo invii e quanti sono questi elementi a rigore non lo sappiamo no lo sappiamo sono io sono esattamente i Questi elementi perché abbiamo inizializzato ia0 Ecco perché l'abbiamo messo a zero anziché uno e tutte le volte che c'è un elemento nuovo abbiamo incrementato i Quindi in effetti un modo possibile per fare tutto questo è di andare a decrementare via via i fino a 15 e maggiori di 0 e ad ogni passo andiamo a copiare ad ogni passo ovviamente de clementia mo i e ad ogni passo dobbiamo andare a copiare che cosa in un opportuno elemento di vi dobbiamo andare a copiare l'elemento iesimo

del vettore ausiliario B e qui c'è il solito discorso cosa scriviamo qua dentro sarà un qualche cosa di identico a quello che abbiamo fatto sopra Cioè IVA decrescendo questo indi anche lui decrescendo ma bisogna inventarsi l'offset come è fatto lo stent allora questa volta lo se sarà fatto così in sostanza è che quando B è uguale a 1 vi deve prendere il primo elemento dell'intero vettore Come si chiama l'indice del primo elemento dell'intero vettore si chiama essere indice di sinistra quindi quando io uguale a uno qua dentro dobbiamo avere essere per cui hai meno 1 + s e questo mi dà la procedura di copia a sinistra Come vedete non è esattamente un copiare i due sottovettori in B e poi copiare B nel vettore di partenza e qualche cosa di un pochino più astuto ma sono veramente dettagli dal punto di vista algoritmico non cambia granché Allora a questo punto in teoria siamo in grado di compilare questo nuovo file che si chiama sempre sotto zero ma che da un eseguibile che è Sort e con un po' di fortuna dovremmo poter verificare che effettivamente il risultato è ordinato e probabilmente il codice anche ragionevolmente corretto Ok Allora a questo punto possiamo scivolare verso il terzo algoritmo della giornata che significa Cioè che ci siamo Possiamo preparare un terzo file iniziale questo è sotto devo merge possiamo andare a recuperare questo file iniziare qui che poi utilizzeremo per realizzare la funzione Quick Sort algoritmo quicksort ma prima diamo un'occhiata ai alle basi dell' algoritmo quicksort veloce è un nome È paradossale perché in realtà non garantisce nel caso pessimo di essere veloce a una conquista quadratica nel caso pessimo Però succede che se la scelta del cosiddetto elemento pivot di cui parleremo fra un minuto è una scelta non particolarmente sfortunata l'algoritmo si rivela invece essere Enel hoogendijk con delle costanti particolarmente vantaggiose solo quando la scelta del pivot è pessima regolarmente e in tutte le chiamate o nella gran maggioranza delle chiamate algoritmo degenera in una complessità quadratica algoritmo applica sempre strategia Divide et impera c'è sempre una questione di dividere il vettore in due sottovettori però non è più una divisione equa in due sottovettori esattamente identici perché è una divisione che viene fatta che viene il cui risulta da una operazione più raffinata e non una divisione fatta meccanicamente in maniera da ottenere due risultati della stessa lunghezza Questo è il vantaggio lo svantaggio del metodo la divisione raffinata è basata sul fatto di ottenere già in partenza da una parte degli elementi piccoli dall'altra degli elementi Grossi perché perché se andiamo a ordinare in due sottovettori in un primo sottovettori gli elementi piccoli in un secondo sottotetto gli elementi grossi evidente che poi per V combinare i due sottovettori non veniva a fare praticamente nulla perché sono già vicendevolmente ordinati i piccoli a sinistra e i grossi grandi a destra con l'elemento che fa da discrimine che questo famoso elemento pivot messo esattamente in mezzo fra i due sottovettori quindi l'idea è una divisione non banale in due parti uguali col calcolo di un elemento mediano in tempo tetta di uno ma un po' raffinata con una procedura che finiremo Partition che andiamo un pochino meglio 1:16:20 Ma non tempo tetta di uno ma un po' raffinata con una procedura che definiremo Partition è che andiamo un pochino meglio analizzare nel seguito e una procedura di chiamata ricorsiva che risolvere così 22 sotto problemi anche qui dobbiamo dimostrare che i due sotto problemi siano entrambi vigorosamente più piccoli di quello di partenza ma quella cosa è facile perché perché abbiamo tolto le mento più Quindi è chiaro che mal che vada anche se uno dei due sottovettori è vuoto l'altro ennemenouno Quindi è più piccolo di quello di partenza e infine un'operazione di ricombinazione che ancora una volta banale perché si tratta di mettere insieme due sottovettori ciascuno dei quali ordinato e con il primo costituito di elementi minori del secondo quindi la ricombinazione consiste nel non fare niente Anche qui ci sarà un caso base che ha lo stesso perché de la Mer soft e di tutti gli algoritmi di ordinamento In pratica cioè quella di dire che è un pittore con zero un elemento nuovi chiede nulla per essere coordinato e il tutto si giocherà sul fatto di realizzare in maniera abbastanza efficiente questa procedura di partizionamento Allora la procedura di pseudocodice

Quindi come abbiamo detto si compone al solito di un caso base invisibile che esse Maggiore quella di di un caso di corsivo invece che si vede con una procedura di costruzione dei due sotto polemiche questa Partition 1 quale procedura restituisce un indice Q che è il modo con cui si restituisce l'elemento pivot q e l'indice all'interno del vettore in cui sta il pivot una procedura Partition In sostanza rimescola permuta gli elementi del vettore in maniera tale che da esse acumen uno ci siano gli elementi piccoli quelli minori e uguali al PV in posizione in cui ci sia il cibo e dalla posizione qq1 lunedì ci siano invece di elementi grandi questi due sottovettori non sono già di per sé ordinati vengono ordinati dalle due chiamate ricorsive quicksort da esse acumen uno è da cui più uno di è qui sotto non si vede ma c'è la procedura di ricombinazione che consiste nel prendere il vettore da essa come uno già ordinato l'elemento q e il rettore del Q1 di già ordinato e costruire con questi due vettori è un elemento il vettore ordinato vi Evidentemente non c'è nulla da fare perché tutto questo è avvenuto in loco all'interno del vettore lì ok Non solo grazie a questa al fatto che i due sottolettori vanno fino a come uno è partendo da cui più uno come dicevo prima è garantito che entrambi sotto poi mi sono più piccoli quello che potrebbe succedere di dannoso è che l'indice chiusi a s o che l'indice cui si addice que la Partition restituisca minimo o l'elemento Massimo e in questo caso uno delle due chiamate è inutile l'altra è di dimensione è nemmeno uno e quindi l'intero Franco ritmo è facile dimostrare che ha una complessità che n quadro perché tdn è data da alcune operazioni più un tetta DN che questa Partition più una chiamata tdn meno uno che evidentemente è molto grosso è su un un problema più piccolo ma non sufficientemente piccolo che l'algoritmo sia corretto è al solito dovuto alla induzione forte nel senso che nel caso base al merito non fa nulla ed è giusto non fa nulla si suppone che l'algoritmo funzioni correttamente su problemi più piccoli dn15 nemmeno uno Quindi queste due parti siano corrette L'unica cosa che è importante che la funzione Partition effettivamente metta nelle prime posizioni gli elementi più piccoli del Pivo e nelle ultime quelli più grandi e a questo garantito che la procedura funziona mentre Come dicevamo per quanto riguarda la complessità e ci sono questi c'hai questo caso pessimo Critico del genere e invece il caso medio nel quale con opportune ipotesi di probabilità sulla distribuzione dei numeri all'interno del vettore si riesce a dimostrare che mediamente la complessità n log n va anche detto che ci sono metodi per determinare un pivot che non sia totalmente Assurdo sono metodi stocastici che consistono Nel scegliere un pva caso ci sono metodi deterministici in realtà che consentono spendendo un po' di fatica di scegliere prima un pilo sensato poi di usarlo per partizionare in due vettori che non siano nessuno dei due vuoto e di conseguenza arrivare a garantire la complessità n log n quindi Qui salta una versione 1:21:10 Seguenza arrivare a garantire la complicità e me lo gang quindi quicksort a una versione che anche nel caso pessimo Enel Gauguin non affronteremo quella versione in o Affronterò neanche l'estrazione casuale Per non chiamare in causa generatori di numeri pseudocasuali e altri argomenti che non sono nei limiti del corso quello che in effetti io farò sarà scegliere un pivot assolutamente arbitrario e sarà il primo elemento del sottovettore l'elemento in posizione S del sottovettore può essere un'idea intelligente può essere un'idea stupida è un'idea stupida quando quando S in realtà è l'elemento minimo o quando essere l'elemento massimo perché perché se l'elemento minimo la partizione metteva creerà un vettore un primo sotto vettore vuoto perché non c'è niente di più piccolo di esse mentre D'altra parte se è l'elemento Massimo che aveva un primo sottovettore che contiene tutti gli elementi il secondo sarà vuoto per cui se siamo di fronte a un vettore che è già ordinato o che già ordinato al contrario questa scelta del pivot è deleteria in ogni altra situazione probabilmente tende Adesso invece abbastanza valida come qualsiasi altra Allora torniamo al nostro codice come sempre dobbiamo andare a affermare chiamare la nostra funzione lì Siccome la funzione funziona ricorsivamente dobbiamo dare i due estremi del vettore sul quale La picchiamo quindi nel Maine saranno 1 ed n e a

questo punto possiamo andare a costruirci la nostra dichiarazione e definizione con una funzione Void che lavora su un vettore di interi e gli indici s e d è il nostro commento dirà che andiamo a ordinare il sotto vettore del vettore di compreso tra SSD con l'algoritmo Quick Sort con questa versione vi dicevo che sceglie il pivot nella maniera più semplice come primo elemento del sottovettore stesso Allora questa algoritmo ha la solita struttura ricorsiva per cui se esce maggiore uguale di cioè nel caso Base non fa nulla se è Se è minore di di invece abbiamo detto che abbiamo una prima fase di costruzione del nostro sotto problema con la funzione Partition che lavora sul lettore VHS e di Air restituisci in uscita l'indice dell'elemento pivot quindi ci serve un parametro una variabile Q che riceva questo indice fatta la divisione del problema in sotto problemi possiamo risolvere ricorsivamente sotto problemi da esse acumen Uno è il sotto problema da q + 1 fino a di e poi avremo la ricombinazione che però non c'è e quindi non la scriviamo a questo punto tutto ciò che ci occorre è a a definire la funzione ausiliaria Partition ancora una volta faccio la scelta di non costruire un prototipo in alto perché non penso che ci possa essere qualche altra funzione che abbia bisogno esattamente della funzione Partition così come la scriviamo di conseguenza mi limito semplicemente a partizionare il vettore di Fra s e d in un sottovettore vi dsq di elementi minori uguali al PDS e un sotto vettore tdq più uno di di elementi che invece sono strettamente maggiori dvds è quel rigore vds però è il valore iniziale quindi forse è meglio se qui scriviamo vivo è più per il semplice motivo che alla fine di tutta questa operazione l'elemento che stava in posizione S all'inizio viene poi spostato in posizione q e infatti qui dobbiamo ancora dirlo parti partiziona non partizionare il vettore in questa maniera e One l'elemento vivo in BBQ a questo punto abbiamo una funzione 1:26:00 Boh in BBQ a questo punto abbiamo una funzione che restituisce un numero intero riceve un vettore di interi e due indici interi SD è che possiamo andare ad analizzare un pochino più nel dettaglio Allora qui ci serve un po' di pazienza perché dobbiamo capire come procede questa funzione Partition la funzione che cosa fa la funzione Intanto abbiamo detto assume come vivo convenzionalmente l'elemento vds per cui vds fino all'ultimo non lo toccheremo all'ultimo lo prenderemo e lo porteremo in posizione Q Questo è il resto del vettore dalla posizione sp1 la posizione di Andrea rimescolato in maniera da avere in un primo sotto vettore e gli elementi minori uguali al più in un secondo sottovettore quelli maggiori E questa cosa può venire fatta in Val in letteratura molte diverse versioni di Partition ma l'idea comune a tutte queste versioni a di avere due sottovettori due tabelle che inizialmente sono del tutto vuote e lasciano fuori come residuo gli elementi del vettore V non ancora considerati e pian pianino queste due tabelle crescono crescono come andando a guardare un elemento non ancora considerato e decidendo se ha più parlo alla prima sottotabella alla prima tabella o la seconda tabella Quindi in sostanza abbiamo una parte disordinata del vettore che all'inizio è tutto tranne il pivot e poi abbiamo due tabelle inizialmente vuote che vanno crescendo c'è chi mettere queste due tabelle all'inizio e alla fine del vettore c'è chi le mette tutte e due alla fine che le metti tutte e due all'inizio sono questioni di scelta che tipicamente a seconda di Qual è il dato possono essere più o meno efficienti ma non c'è una versione a meno la conoscenza che sia nettamente più efficiente delle altre io scelgo quella descritta sia nelle dispense sia nella teoria che è la seguente Allora l'idea è che a parte il pivot in posizione s ne abbiamo due tabelle che stanno nella prima parte del vettore quindi partono a sinistra e vanno pian pianino allargandosi verso destra inglobando gli elementi che non sono ancora stati considerati e che stanno fra sp1a di ora Cos'è che succede succede che siccome queste due tabelle Sono entrambe nella metà sinistra tutte le volte che arriva un nuovo elemento che va nella seconda tabella basta allargarla e siamo a posto ma tutte le volte che l'elemento che leggiamo deve andare nella prima tabella e ci si crea un problema perché in mezzo fra il nuovo elemento è la prima tabella c'è la seconda tabella Come si fa a spostare l'intera tabella chiaramente nulla si scala tutta quello

che si fa è uno scambio si tratta In sostanza di prendere il primo elemento della seconda tabella è l'elemento in questione quello che deve spostarsi scambiarli fra loro e ridefinire Il primo elemento della seconda tabella che adesso è cambiato come ultimo elemento della prima tabella D'altra parte la seconda tabella verrà allargata sulla destra Quindi in pratica la prima tabella continua ad allargarsi verso destra quella degli elementi piccoli mentre la seconda tabella quella degli elementi grossi a volte si allarga e a volte invece si scala spostando sia il primo sia l'ultimo elemento avanti di un passo più difficile a dirlo che non sa farlo vedere sostanzialmente dobbiamo dare un pochino di definizioni scegliere delle convenzioni Allora abbiamo bisogno di dire dove comincia dove finisce ogni tabella prima tabella chiaramente comincia In sp1 quindi non c'è bisogno di un indice Ma dove finisce domanda Diciamo che finisce in D1 elemento di destra della prima tabella adesso qui bisognerebbe anche capire se questo elemento di destra è l'ultimo elemento che appartiene la tabella o è invece il primo che non vi appartiene la scelta che viene più comoda è di dire che è l'ultimo elemento della prima tabella possiamo anche segnare ce l'ho questo è l'ultimo elemento della prima tabella cioè quella degli elementi minori e uguali al PV D'altra parte ci serviva poi anche un t2c se aveva un elemento di sinistra un indice di sinistra o di destra della seconda tabella indice di sinistra non ci serve perché se sappiamo che la prima tabella finisce in 1:30:50 Ci serviva un elemento di sinistra un indice di sinistra o di destra della seconda tabella indice di sinistra non ci serve perché se sappiamo che la prima tabella finisce in D1 la seconda comincia In di uno più uno dei due invece ci serve allora qui verrebbe spontaneo definire di due come l'ultimo elemento della seconda tabella ma invece per motivi di diciamo di efficienza viene più comodo dire che è il primo elemento a destra della seconda tabella Cioè la seconda tabella finisce in di 1:59 non in D2 E questa comprende gli elementi che sono maggiori del Pivo Basta scrivere una convenzione seguirla correttamente alcune sono leggermente più efficienti di altre lo Che cos'è che bisogna sostanzialmente fare è molto semplice procedere fino a che le due tabelle e in particolare la seconda non è andata a toccare la fine del vettore Allora siccome la seconda tabella finisce in di 1:59 questo Cosa significa significa che andiamo a prendere tutti gli elementi che stanno fuori delle due tabelle cioè gli elementi dalle sinistra più 1 fino allenamento di destra compreso Tutti questi elementi vengono corsi uno per uno e per ciascuno si cerca di capire se va nella prima puntata seconda sottotabella stiamo semplicemente scrivendo gli elementi del vettore saltiamo S perché è chiaro che essendo il pivot sta nella prima tabella Quindi da sp1 in poi li guardiamo tutti e per ciascuno cerchiamo di capire se va nella tabella prima o nella tabella seconda e come si fa si fa un banale confronto con il pivot che abbiamo detto essere l'elemento vds Allora se per caso l'elemento vtd due è più grosso dell'elemento pivot o questo il caso facile perché è il caso in cui di due deve stare nella seconda tabella e quindi basta incrementare di due ma l'incremento di due già lo facciamo qua per cui in questo caso non si fa nulla caso in cui non si fa nulla È inutile scriverlo guardiamo solo toccato il caso in cui si fa qualcosa è quello in cui l'elemento che stiamo guardando in realtà è più piccolo del pivot e quindi deve andare a finire nella prima tabella Come si fa a farlo finire nella prima tabella semplice lo andiamo a mettere all'inizio della seconda tabella quindi andiamo sostanzialmente a allargare la prima tabella Questo è un errore perché la prima tabella questo punto include un elemento più grosso del Pivo Ma se quell'elemento più grosso del Pivo che sta in D1 lo scambiamo con di due siamo perfettamente a nostro agio quindi si tratta banalmente di andare a scambiare l'elemento V che sta in D1 con l'elemento V che sta in D2 al solito passando lì per indirizzo in questo modo noi in realtà stiamo andando a shiftare in avanti la prima tabella perché di uno che è il suo estremo di destra si allarga e quindi l'estremo di sinistra della seconda tabella si sposta anch'esso è di 2 che all' estremo di destra la seconda tabella si sposta a destra per cui a seconda dei casi ho la seconda tabella si allarga o la seconda tabella si sposta di un passo

verso destra abbiamo finito No Non esattamente nel senso che adesso la procedura è veramente partizionata in maniera corretta ma l'elemento più volta in posizione S mentre l'elemento pivot In realtà dovrebbe stare in posizione Q non soltanto che anche capire che cos'è questo posizione Q di cui stavamo parlando è semplicemente è la posizione nella quale va il pivot domanda Dove va il PVC Allora siccome tutti gli elementi della seconda tabella sono strettamente maggiori non vanno toccati gli elementi della prima tabella invece sono minori uguali al pivot quindi l'ultimo che sta in posizione di uno si può tranquillamente scambiare con il pivot e mi è possibile andare a scambiare l'elemento che sta in posizione D1 con l'elemento pivot che sta in posizione S E A questo punto il pivot sta in di uno è di uno sta in ass soprattutto quello che succede è che adesso sappiamo che il pivot sta in posizione D1 e possiamo restituirlo al esterno alla funzione Quick Sort che adesso sa dove comincia e dove finisce 1:35:40 Restituirlo al esterno alla funzione quicksort che adesso sa dove comincia e dove finisce ciascuna delle due tabelle che cosa dobbiamo fare non molto sostanzialmente L'unica cosa che ci manca e scambia scambia l'abbiamo già implementata Ma abbiamo cambiato via via il codice quindi ci limitiamo a implementare abbastanza velocemente è la solita procedura che scambia due numeri interi con l'ausilio di un terzo numero Temp temporaneo tempo va a prendere il primo numero il primo numero viene sovrascritto dal secondo e finalmente il secondo viene sovrascritto dal primo che era stato conservato Allora a questo punto dovremmo poter andare a il codice Sì davvero il caso che abbiamo fatto un piccolo ti avevo voglia che in teoria un compilatore dovrebbe segnalare Ma in questo caso non lo fa vi faccio vedere l'errore sta nel fatto che qui noi abbiamo una fantastica variabile di uno che viene utilizzata ma non viene mai inizializzata e è sempre buona norma verificare che tutte le variabili abbiano un valore iniziale sensato subito prima di dove vengono usate di due viene usata viene inizializzata qui è uguale a se uno di uno No Qual è il valore iniziale di D1 d1e l'ultimo elemento della prima tabella pianta bene l'altra è vuota Che significa che sicuramente o destro sta addirittura a sinistra del sinistro ma potremmo anche dire che l'ultimo elemento che è minore uguale al primo è il pivot stesso cioè che di uno sta in posizione S e tutte le volte che si trova un elemento nuovo che è minore uguale a vdsd uno viene spostato con il primo che diventa anche lui da mettere nella prima tabella andrò in posizione s + 1 e poi in posizione spo2 eccetera eccetera e questo dovrebbe In effetti sistemare le cose e consentirci di effettivamente di ordinare il nostro settore Allora tutto questo chiude la lezione sull'implementazione di tutte queste procedure c'è una cosa da dire che noi abbiamo ordinato un banale vettore di interi nella maggior parte dei casi Questo potrebbe succedere nella maggior parte dei casi può capitare di avere delle strutture più sofisticate da ordinare provate a pensare di avere un elenco di persone che vanno ordinate per cognome per numero di matricola per età per media per qualsiasi altro cosa succede succede che a questo punto ci si può porre il problema di dire ma questi oggetti che noi stiamo ordinando li trattiamo come se fossero dei numeri interi Cioè li mettiamo in un vettore e poi li scambiamo o invece la cosa non è molto sensata è chiaro che è sempre possibile farlo però è anche chiaro che se gli oggetti di due vds eccetera eccetera tutti questi oggetti che qui allegramente scambiamo con questo meccanismo di delle tre carte se tu non ho già ti grandi Ognuno di Queste copie può essere una cosa pesante per esempio se si tratta di parole e queste assegnamenti non si possono fare si devono fare delle stelle in Copy String Copy se la parola una lunghezza massima costa di uno ma è un poeta di uno che può essere 256 non è uno come in questo caso allora C'è da pensarci un attimo se gli oggetti da ordinare sono grandi può essere fastidioso doverli fisicamente ordinare per non solo perché costa costa non costa complessità nel senso di ordini di grandezza è sempre nello gannet Ma questa sulle costanti e poi anche perché se per caso dobbiamo ordinare Questi elementi in vario modo ogni volta distruggiamo l'ordinamento precedente magari l'ordinamento inizia ad avere un senso magari io voglio avere sia l'ordinamento per

età sia l'ordinamento per cognome sono due ordinamenti diversi Cosa faccio Faccio due copie di una struttura molto grande spreco spazio spreco tempo per copiare Insomma C'è da pensarci un attimo Allora le cose che si possono fare sono diverse ma la principale consiste in sostanza nel non ordinare direttamente questi oggetti ma nel costruirsi un vettore di posizioni e ordinare le posizioni cioè nel farlo 1:39:30 C'è da pensarci un attimo se gli oggetti da ordinare sono grandi può essere fastidioso doverli fisicamente ordinare per non solo perché costa costa non costa complessità nel senso di ordini di grandezza è sempre n log n ma costa sulle costanti e poi anche perché se per caso dobbiamo ordinare Questi elementi in vario modo ogni volta distruggiamo l'ordinamento precedente magari l'ordinamento inizia ad avere un senso magari io voglio avere sia l'ordinamento per età sia l'ordinamento per il cognome sono due ordinamenti diversi Cosa faccio Faccio due copie di una struttura molto grande spreco spazio spreco tempo per copiare Insomma C'è da pensarci un attimo Allora le cose che si possono fare sono diverse ma la principale consiste in sostanza nel non ordinare direttamente questi Ma nella costruirsi un vettore di posizioni e ordinare le posizioni cioè nel fare l'ordinamento indiretto vedremo un esempio di questa cosa in una delle prossime elezioni le prossime due delle ultime due ma il principio è decisamente generale perché Cosa vuol dire fare un vettore di posizioni posizione un concetto astratto che abbiamo già visto declinato in due modi cioè puntatori indirizzi puntatori agli oggetti che stiamo trattando oppure indici numerici indici numerici hanno senso se sono riferiti a un vettore cioè l'idea è che se voi avete un vettore di che so strutture contenenti informazioni complicate non andate a toccare quelle strutture costruite un Bettolino di interi o un Bettolino di indirizzi Dopodiché quello che viene effettivamente assoggettato a questi scambi è passato a tutte queste funzioni partitioning so a te via dicendo è il vettore di puntatori è vero una cosa è vero che quando si fanno i confronti come questo video 2 minore uguale avs e ci sono sempre dei confronti alla base delle funzioni degli ordinamenti di basati su confronto per definizione vedete che in merge noi abbiamo dei confronti fondamentali che sono questi vds minore-uguale avidi di per esempio e analogamente nel funzione heapsort che abbiamo realizzato i confronti c'erano ed erano nella giornata hip è uno questi due i confronti questi confronti lavorano direttamente sul vettore V Ma se il vettore V contiene dei numeri che sono gli indici in realtà stiamo dobbiamo lavorare non su Viber dobbiamo lavorare sul vettore che contiene le vere informazioni nell'indice vidimax Oviesse o vedi di questo è semplicemente un anticipo la ciliegina per chiudere la lezione che andremo a realizzare più compiutamente in seguito ma in seguito non avremmo il tempo per osservare Quali sono i vantaggi di questa scrittura che viene in mente è più complicata è più fastidiosa anche perché cambia anche cambiano i parametri di queste funzioni i vantaggi sono uno di poter evitare scambi operazioni di scambio su strutture di grandi dimensioni due nel poter rispettare l'ordine corrente degli elementi determinando altri ordini evidentemente con delle procedure di appoggio ce lo svantaggio e che c'è un vettore di appoggio fatto di puntatori ho fatto di interi c'è anche un altro aspetto l'altro aspetto e che può darsi che i dati non siano in un vettore e se i dati non sono in un vettore ma sono in una lista o in un albero o in qualunque altra oggetto le gli algoritmi come li abbiamo visti non sono proprio applicabili sono applicabili se uno costruisce un vettore di puntatori Invece mercoledì indicizzate di no perché hai detto vedi gli indici interi in quanto posizioni hanno senso soltanto se da qualche parte Esiste un vettore di partenza in so che sto facendo solo delle considerazioni hai attivamente confuse ma in mancanza del tempo per un esercizio finito e completo mi sembrava utile davi questi spunti di riflessione prima di definitivamente chiudere anche questa lezione

Figura 12.10: Aggiornamento

L'aggiornamento di un *heap* in un nodo dato consiste nel

- determinare il figlio con l'etichetta massima
- confrontare tale etichetta con quella del nodo dato:
  se superiore,
  - scambiare le due etichette
  - applicare ricorsivamente la procedura al nodo figlio

```
aggiornaheap(V,n,i)                                  T(h)
{
  s := 2 * i; { figlio sinistro }                    Θ(1)
  d := 2 * i + 1; { figlio destro }                  Θ(1)
  iMax := i;                                         Θ(1)
  if (s <= n && A[s] > A[iMax]) iMax = s;            Θ(1)
  if (d <= n && A[d] > A[iMax]) iMax = d;            Θ(1)
  if (iMax != i)                                     Θ(1)
  {
    Scambia(&A[i],&A[iMax]);                         Θ(1)
    aggiornaheap(V,n,iMax);                          T(h-1)
  }
}                                                    Quindi complessità Θ(log n)
```

Figura 12.11: Costruzione

La costruzione di un *heap* su un vettore dato consiste nel

- aggiornare gli *heap* costituiti dai suoi sottoalberi
- in ordine inverso, perché l'aggiornamento presuppone la correttezza di tutti gli *heap* ai livelli inferiori (quindi con indici successivi)
- trascurando le foglie, perché sono certamente *heap* corretti

```
creaheap(V,n)
{
  for (i = n/2; i >= 1; i--)
    aggiornaheap(V,n,i);
}
```

L'analisi di complessità (non elementare) mostra che $T(n) \in \Theta(n)$

Figura 12.12: *HeapSort* e *SelectionSort*
Si può quasi dire che sia lo stesso algoritmo con strutture dati diverse

```
SelectionSort(V,n)
{

  for (j = n; j > 1; j--)
  {
    i = TrovaIndiceMassimo(V,j);
    Scambia(&V[i],&V[j]);

  }
}
```

```
HeapSort(V,n)
{
  creaheap(V,n);
  for (j = n; j > 1; j--)
  {
    i = 1;
    Scambia(&V[i],&V[j]);
    aggiornaheap(V,j-1,1);
  }
}
```

così che la complessità temporale scende da $\Theta\left(n^2\right)$ a $\Theta\left(n \log n\right)$

Figura 12.13: Ordinamento per fusione (*MergeSort*)
Dato un vettore V, sia $V[s,d]$ il sottovettore contenente gli elementi di V con indici compresi fra s e d

L'algoritmo *MergeSort* applica la strategia detta *divide et impera*

- divide: suddivide il vettore in due sottovettori di $n/2$ elementi calcolando l'indice mediano (*è il modo più semplice, non l'unico!*)
- impera: ordina i sottovettori ricorsivamente, lasciando invariati quelli di lunghezza ≤ 1 (caso base)
- combina: fonde i due sottovettori producendone uno solo ordinato

Figura 12.14: *MergeSort*: pseudocodice

```
MergeSort(V,s,d)
{
  if (s < d)
  {
    m = ⌊(s+d)/2⌋;
    MergeSort(V,s,m);
    MergeSort(V,m+1,d);
    Merge(V,s,m,d);
  }
}
```

I due sottovettori sono sempre più piccoli del vettore V grazie a

- l'arrotondamento per difetto
- la divisione in $V[s, m]$ e $V[m + 1, d]$

Senza questa combinazione la ricorsione proseguirebbe all'infinito
(esaurendo lo *stack*)
(*per esempio, si consideri* $s = 5$ *e* $d = 6$)

Figura 12.15: *MergeSort*: correttezza

L'algoritmo funziona per induzione matematica (forte)

- nel caso base ($s \geq d$), il vettore V è ordinato
  Dunque l'algoritmo funziona su vettori di $n \leq 1$ elementi
- nel caso ricorsivo
  1. i due sottovettori sono strettamente più corti
     Dunque $n_1 < n$ e $n_2 < n$
  2. l'ipotesi induttiva garantisce che siano ordinati
     Se l'algoritmo ordina tutti i vettori di qualsiasi lunghezza $\ell < n \ldots$
  3. la funzione *Merge* garantisce che il risultato sia ordinato
     $\ldots$ l'algoritmo ordina anche qualsiasi vettore di lunghezza n

L'algoritmo ordina qualsiasi vettore V di qualsiasi lunghezza n

Figura 12.16: La procedura *Merge*
La procedura ricombina le soluzioni dei due sottoproblemi

Parte con le due metà del vettore V ordinate al proprio interno
e un vettore ausiliario B vuoto:

1. finché i due sottovettori sono non vuoti
   - confronta i loro elementi minimi
   - sposta il minore dei due in fondo al vettore ausiliario B
2. quando uno dei due sottovettori è vuoto
   - copia l'altro in fondo al vettore ausiliario B
3. quando sono vuoti entrambi
   - ricopia il vettore ausiliario B sul vettore iniziale V

La complessità temporale è ovviamente lineare $\Theta(n)$

Figura 12.17: *MergeSort*: complessità

```
MergeSort(V,s,d)
{
  if (s < d)                                   Θ(1)
  {
    m = ⌊(s+d)/2⌋;                             Θ(1)
    MergeSort(V,s,m);                          T_MS(n_1)
    MergeSort(V,m+1,d);                        T_MS(n_2)
    Merge(V,s,m,d);                            Θ(n)
  }
}
```

È facile mostrare che $n = d - s + 1 = n_1 + n_2$ con

$$n_1 = m - s + 1 = \left\lceil \frac{n}{2} \right\rceil \qquad n_2 = d - m = \left\lfloor \frac{n}{2} \right\rfloor$$

Figura 12.18: *MergeSort*: complessità

Per semplicità consideriamo sequenze in cui n è una potenza di 2

- i due sottovettori hanno entrambi lunghezza $n/2$

Di conseguenza

- quando $n = 1$, il tempo di calcolo è $T(1) \in \Theta(1)$
- quando $n > 1$, il tempo di calcolo $T(n)$ è la somma di
  1. $\Theta(1)$ per il calcolo dell'indice mediano (divide)
  2. $2T(n/2)$ per la soluzione dei due sottoproblemi (impera)
  3. $\Theta(n)$ per la ricomposizione della soluzione (combina)

$$T_{MS}(n) = \begin{cases} \Theta(1) & \text{per } n = 1 \\ 2T_{MS}(n/2) + \Theta(n) & \text{per } n > 1 \end{cases}$$

Si dimostra che $T_{MS}(n) \in \Theta(n \log n)$

Figura 12.19: Ordinamento "veloce" (*QuickSort*)

Dato un vettore V, sia $V[s, d]$ il sottovettore contenente gli elementi di V con indici compresi fra s e d

L'algoritmo *QuickSort* applica la strategia detta *divide et impera*

- divide: suddivide il vettore in due sottovettori
  (*in modo diverso da MergeSort!*)
- impera: ordina i sottovettori ricorsivamente,
  lasciando invariati quelli di lunghezza ≤ 1 (caso base)
- combina: fonde i due sottovettori producendone uno solo ordinato

La strategia è la stessa del *MergeSort*, ma i due sottovettori contengono

1. gli elementi non superiori a un valore di soglia
2. gli elementi superiori a un valore di soglia

Il valore soglia è il valore di un elemento specifico (elemento *pivot*),
che viene estratto e poi reinserito in mezzo fra i due sottovettori ordinati

Figura 12.20: *QuickSort*: pseudocodice

La suddivisione diversa implica che:

1. la fase “divide” è più sofisticata:
   sposta gli elementi raccogliendone alcuni al principio e altri alla fine
2. la fase “combina” è banale:
   basta concatenare i due sottovettori ordinati

```
QuickSort(V,s,d)
{
  if (s < d)
  {
    q = Partition(V,s,d);          { q è la posizione finale del pivot }
    QuickSort(V,s,q-1);
    QuickSort(V,q+1,d);
  }
}
```

I due sottovettori sono più piccoli di V perché non contengono il *pivot*
(*questo evita la ricorsione infinita*)

Figura 12.21: *QuickSort*: correttezza

L’algoritmo funziona per induzione matematica (forte)

- nel caso base ($s \geq d$), il vettore V è ordinato
  Dunque l’algoritmo funziona su vettori di $n \leq 1$ elementi
- nel caso ricorsivo
  1. escludendo il *pivot*, i sottovettori sono strettamente più corti di V
     Dunque $n_1 < n$ e $n_2 < n$
  2. la funzione *Partition* garantisce che i valori del primo sottovettore siano non superiori al *pivot* e quelli del secondo superiori
  3. l’ipotesi induttiva garantisce che siano ordinati
     Se l’algoritmo ordina tutti i vettori di qualsiasi lunghezza $\ell < n$...
  4. il concatenamento crea banalmente un vettore ordinato
     ...l’algoritmo ordina anche qualsiasi vettore di lunghezza n

L’algoritmo ordina qualsiasi vettore

Figura 12.22: *QuickSort*: Partition
Vedi file `QuickSort-Partition.pdf`

Figura 12.23: Uno schema mnemonico
I diversi algoritmi visti sinora sono legati da alcune idee di fondo

1. il modo di procedere; ci sono algoritmi che
   a) costruiscono un sottovettore ordinato
      e lo ricombinano con gli elementi esterni, uno alla volta
   b) costruiscono due sottovettori, li ordinano e li ricombinano
2. la distribuzione dello sforzo; ci sono algoritmi che impiegano
   a) poco sforzo nella costruzione dei sottovettori ordinati e
      molto sforzo nella ricombinazione
   b) molto sforzo nella costruzione dei sottovettori ordinati e
      poco sforzo nella ricombinazione

	Costruzione banale e ricombinazione sofisticata	Costruzione sofisticata e ricombinazione banale
Allargamento di un sottovettore	InsertionSort	SelectionSort (HeapSort)
Fusione di due sottovettori	MergeSort	QuickSort

Figura 12.24: Implementazioni ricorsive di *InsertionSort* e *SelectionSort*
Vedi file `InsSort-SelSort-Ricorsive.pdf`

12.2 Esercizi

12.2.1 Esercizio

Ispirandosi alla funzione *Merge*, che fonde due vettori ordinati in uno solo, si realizzi una funzione *Union*, che fonde due insiemi ordinati (rappresentati da vettori/tabelle) in uno solo senza elementi ripetuti.

12.2.2 Esercizio

Ispirandosi alla funzione *Merge*, si realizzi una funzione *Intersection*, che determina l'intersezione di due insiemi ordinati (rappresentati da vettori/tabelle).

12.2.3 Esercizio

Ispirandosi alla funzione *Merge*, si realizzi una funzione *SetDifference*, che determina l'insieme differenza di due insiemi ordinati (rappresentati da vettori/tabelle).

Capitolo 13

Programmazione dinamica

Questo capitolo è dedicato alla *Programmazione Dinamica*, in particolare applicata a problemi di Ottimizzazione Combinatoria[1]

La Programmazione Dinamica è un modo di progettare algoritmi che applica la strategia già discussa di dividere un problema in sottoproblemi, risolvere i sottoproblemi stessi e combinare le loro soluzioni in modo da ottenere la soluzione del problema complessivo, ma lo fa in maniera più efficiente, sfruttando le operazioni intrinsecamente ripetute. Si applica a problemi di molti generi diversi, ma gli esempi discussi in questo capitolo sono tutti esempi di Ottimizzazione Combinatoria[2]. È quindi opportuno prima di tutto definire l'Ottimizzazione Combinatoria.

Un *problema di ottimizzazione*, al contrario di quelli cui si è abituati fin dalle scuole elementari, non ha una soluzione singola (l'equazione $x+3 = 4$ è un problema la cui soluzione è $x = 1$), ma tante. Ognuna di queste soluzioni ha un valore, o un costo, assegnatole da una *funzione obiettivo*. Il problema consiste nel trovare, all'interno di un *insieme di soluzioni ammissibili*, una che abbia valore massimo o costo minimo. Ne derivano problemi di massimizzazione e problemi di minimizzazione.

Perché "combinatoria"? Perché l'insieme delle soluzioni ammissibili ha una definizione piuttosto particolare. Si suppone che sia dato un *insieme base* finito B[3] e che l'insieme delle soluzioni ammissibili X sia formato da sottoinsiemi dell'insieme base. Chi conosce il calcolo combinatorio ricorderà senz'altro che le combinazioni di n oggetti a k a k sono sottoinsiemi non ordinati di k oggetti estratti senza ripetizioni da un insieme di n oggetti. Quindi, sono sottoinsiemi di quest'ultimo. L'Ottimizzazione Combinatoria riguarda tutti i problemi le cui soluzioni sono combinazioni, cioè sottoinsiemi di un insieme base. Di conseguenza, X è sottoinsieme di 2^B, cioè dell'insieme delle parti di B, ovvero la collezione di tutti i sottoinsiemi di B, a partire dal sottoinsieme vuoto, passando per i singoletti (sottoinsiemi formati da un solo elemento), le coppie, le terne, ecc... fino ad arrivare all'insieme B complessivo.

[1] Nota operativa: le ultime lezioni del modulo di laboratorio sono dedicate ad argomenti che vengono trattati in seguito nelle lezioni di teoria. Questo è dovuto al fatto che l'esame richiede lo svolgimento di un progetto, che richiede alcune settimane di tempo, a cui vanno aggiunti diversi giorni per la valutazione (trattandosi dell'appello di giugno, il numero di iscritti è spesso alto, e comunque sconosciuto a priori. Tutto ciò impone di terminare il corso intorno alla metà di maggio, e quindi trattare in laboratorio gli ultimi argomenti prima che vengano introdotti nelle lezioni di teoria. Ogni studente troverà il modo più consono alle proprie caratteristiche per seguire queste lezioni: riprenderle dopo aver seguito quelle di teoria, o approfittare di una prima esposizione informale alla quale far seguire i formalismi esposti nel modo più appropriato, o portare avanti in parallelo i due approcci usando le dispense e registrazioni disponibili.

[2] Questo dipende anche dal fatto che la mia attività di ricerca è concentrata sull'ottimizzazione combinatoria.

[3] Nelle dispense di teoria viene indicato con E.

I problemi di Ottimizzazione Combinatoria (OC) sono definiti da

- un insieme base finito B
- una regione ammissibile $X \subseteq 2^B$ i cui elementi, detti soluzioni, sono opportuni sottoinsiemi $x \subseteq B$ dell'insieme base
- una funzione obiettivo $f : X \to \mathbb{N}$ che dà un valore a ogni soluzione

Si tratta di trovare una soluzione di valore minimo o massimo

$$\min_{x \in X} f(x) \quad \text{oppure} \quad \max_{x \in X} f(x)$$

I problemi di Ottimizzazione Combinatoria hanno moltissime applicazioni

Figura 13.1: Ottimizzazione Combinatoria

È facile mostrare che esistono esattamente 2^n sottoinsiemi di un insieme di n elementi, per il banale motivo che ogni sottoinsieme può avere o non avere (quindi, ci sono due casi) ciascuno degli n elementi dell'insieme base. Una funzione obiettivo f associa a ciascuna sottoinsieme ammissibile, cioè a ciascuna soluzione un valore o un costo, e il problema consiste nel trovare una soluzione che abbia valore minimo o massimo della funzione obiettivo all'interno della regione ammissibile. Ipotizzeremo che le funzione obiettivo abbiano valori naturali, cioè numeri interi non negativi. Nelle dispense di teoria si parla di numeri reali positivi, ma la distinzione è da un lato sofisticata, e dall'altro non necessaria: lavorando con un computer i numeri reali sono in effetti tutti razionali, e questi sono riducibili a numeri interi a patto di fissare opportunamente il denominatore, ovvero l'unità di misura. Quindi, l'ipotesi qui adottata non è assolutamente limitativa.

I problemi di Ottimizzazione Combinatoria hanno una quantità enorme di applicazioni. A un occhio matematico un po' superficiale possono sembrare strani. In effetti, se l'insieme base B è finito, anche la collezione 2^B è finita, e quindi qualsiasi problema di Ottimizzazione Combinatoria ha un numero finito di soluzioni[4], dato che X contiene solo una parte di 2^B. Ma se il numero di soluzioni è finito, apparentemente, il problema non sussiste: basta scorrere tutte le soluzioni, valutarle, conservare la migliore e restituirla al termine. Questo è vero, cioè ogni problema di Ottimizzazione Combinatoria ammette un algoritmo risolvente che si chiama *algoritmo esaustivo* e che fa esattamente quanto detto. Tuttavia, questo algoritmo valuta $O(2^n)$ sottoinsiemi (magari ha qualche trucco per non valutarli tutti, ma nel caso pessimo il numero di sottoinsiemi valutati ha quell'ordine i grandezza) e per ciascuno valuta se il sottoinsieme corrisponde una soluzione ammissibile o no, e in caso positivo valuta il valore della funzione obiettivo. Anche se si suppone che il test di ammissibilità e la valutazione della funzione obiettivo comportino un tempo polinomiale, comunque il numero esponenziale di sottoinsiemi da valutare, e quindi l'algoritmo esaustivo è in complesso esponenziale. Questo algoritmo, quindi, non è pratico: è applicabile solo a istanze piccole.

La ricerca matematica, quindi, indaga se esistano algoritmi polinomiali per i problemi di Ottimizzazione Combinatoria. Per alcuni problemi, ci sono algoritmi

[4]Si può dimostrare facilmente che vale anche l'opposto, cioè che avere un numero finito di soluzioni è una definizione alternativa equivalente, più immediata, ma anche meno interessante.

Algoritmo esaustivo: trova una soluzione ottima scorrendole tutte

E allora dove sta il problema?

L'algoritmo esaustivo

- valuta $O\left(2^{|B|}\right)$ sottoinsiemi $x \subseteq B$
- per ognuno valuta se è una soluzione ammissibile ($x \in X$)
- per ogni soluzione ammissibile valuta il valore ($f(x)$)

Anche se le due valutazioni sono polinomiali, il risultato è esponenziale

La ricerca matematica mostra che

- alcuni problemi di OC ammettono algoritmi polinomiali esatti
- tutti i problemi di OC ammettono algoritmi polinomiali euristici, cioè che non garantiscono di trovare l'ottimo su ogni istanza

Algoritmi "non corretti", a rigore, però utili in pratica

Figura 13.2: Algoritmi risolutivi

esatti di complessità polinomiale, che sono molto utili in pratica. Per altri problemi, d'altra parte, non è noto alcun algoritmo polinomiale. Esiste anzi una nota congettura, la famosa congettura $\mathcal{P} \subset \mathcal{NP}$, che è uno dei sette "problemi del millennio", risolvendo i quali si vince un milione di dollari messo in palio dal Clay Mathematics Institute. Siccome è noto (praticamente per definizione) che $\mathcal{P} \subseteq \mathcal{NP}$, la congettura sostiene che alcuni problemi appartenenti all'insieme $\mathcal{NP}$, che comprende problemi di Ottimizzazione Combinatoria dotati di opportune proprietà (sorvoliamo sui dettagli), non appartengono all'insieme $\mathcal{P}$, che raccoglie i problemi che ammettono algoritmi polinomiali. Nessuno sa se la congettura sia vera o falsa, ci si lavora da circa 50 anni, e il consenso pratico è che molto probabilmente sia vera.

Che cosa si fa quando si trova di fronte si trova di fronte a un problema per cui non si conoscono algoritmi polinomiali esatti, e si sospetta che non ce ne siano? Si va alla ricerca di algoritmi polinomiali *euristici*. La parola "euristico" viene ovviamente dall'espressione "Eureka" (cioè "Ho trovato"), del famoso aneddoto su Archimede. Si tratta di algoritmi che trovano una soluzione appartenente a X, quindi ammissibile, realizzabile in pratica, ma non certamente ottima. Lo sforzo è di progettare algoritmi che producano quanto meno la miglior soluzione possibile. In pratica, un algoritmo euristico trova la soluzione ottima per alcune istanze del problema, ma non per tutte (o almeno non è dimostrato che la trovi per tutte). Nei casi in cui l'algoritmo non trova la soluzione ottima, potrà trovarne una più o meno buona.

Un algoritmo di questo genere è chiaramente scorretto, ma non è inutile, perché in un tempo polinomiale (veloce) genera una soluzione che può essere applicabile in pratica, e che forse spesso costa non molto più dell'ottimo. Da un punto di vista pratico, questo ha un notevole valore economico, ma anche conoscitivo. Agli algoritmi euristici è dedicato in parte il prossimo capitolo, perché l'algoritmo *greedy*, che ne costituisce il tema principale, è una modalità generica di risolvere problemi

di Ottimizzazione Combinatoria, a volte in modo esatto, altre in modo euristico[5].

Si vuole scegliere da un insieme di oggetti voluminosi un sottoinsieme di valore massimo che si possa racchiudere in uno zaino di capacità limitata

- un insieme B di oggetti elementari
- una funzione $v : B \to \mathbb{N}$ che descrive il volume di ogni oggetto
- un numero $V \in \mathbb{N}$ che descrive la capacità di uno zaino
- una funzione $\phi : B \to \mathbb{N}$ che descrive il valore di ogni oggetto

La regione ammissibile contiene i sottoinsiemi di oggetti di volume totale non superiore alla capacità dello zaino

$$X = \left\{ x \subseteq B : \sum_{j \in x} v_j \leq V \right\}$$

L'obiettivo è massimizzare il valore complessivo degli oggetti scelti

$$\max_{x \in X} f(x) = \sum_{j \in x} \phi_j$$

Figura 13.3: Il problema dello zaino (KP)

Nel seguito, considereremo un problema di Ottimizzazione Combinatoria e cercheremo di risolverlo applicando la Programmazione Dinamica. Questo problema si chiama *problema dello zaino*, o *Knapsack Problem*. Il problema riguarda un insieme di oggetti dotati di un volume e di un valore. Si vuole determinare un sottoinsieme di questi oggetti che abbia un valore complessivo massimo, ma un volume complessivo non superiore a un limite, che viene interpretato come la capacità di uno zaino nel quale si vogliono mettere questi oggetti. L'idea fondamentale è che soltanto il volume sia limitante, non la forma o altri aspetti: ad esempio, una canna da pesca è abbastanza piccola ma può non entrare in uno zaino, ma di questo ci disinteressiamo, come se gli oggetti fossero liquidi o malleabili. Il problema è formalmente identico a quello di investire somme di denaro avendo un *budget* che limita l'investimento totale, oppure di dedicare periodi di tempo ad attività avendo un orizzonte temporale complessivo. L'idea del volume e del valore sono quindi semplicemente delle etichette.

quindi andiamo un pochino più formalmente a dire che abbiamo un insieme B che è l'insieme base poi per cui concludere mo che il problema di ottimizzazione combinatorio hai fatto di oggetti esiste una funzione volume di piccolo che associa ogni oggetto un numero naturale numero intero maggiore uguale a zero avremo un numero di grande che è la capacità dello zaino 0:14:30 Naturale numero intero maggiore uguale a zero avremo un numero di grande che è la capacità dello zaino è una funzione che associa ad ogni oggetto nell'insieme il valore di quell'oggetto le soluzioni ammissibili la regione ammissibile costituita da tutti i sottoinsiemi e Que-

[5] Agli algoritmi euristici in genere è dedicato anche un corso che tengo per la laurea in informatica, ma aperto a studenti di matematica e che richiede solamente le competenze di questo corso di algoritmi.

sto dimostra che il pronome di ottimizzazione combinatoria non tutti i sottoinsiemi ma solo quelli il cui volume totale cioè la somma dei volumi dei singoli oggetti non supera il valore di grande della capacità D'altra parte l'obiettivo che ci poniamo è una funzione additiva cioè la fdx è la somma di una funzione ausiliaria che è il valore dei singoli oggetti definita sugli elementi dell'insieme del sottoinsieme che stiamo studiando quindi F su un sottoinsieme la somma di FIDI iota per tutti gli oggetti che appartengono a quei sottoinsieme ICS e quello che non vogliamo fare massimizzare la È abbastanza evidente che questo è un problema di ottimizzazione combinatoria ci si può chiedere come risolverlo e oggi lo risolveremo in maniera esatta con vari algoritmi che hanno molto in comune fra loro ma sono diversi e domani domani nella prossima lezione lo risolveremo invece in maniera e ho visti Ca perché gli algoritmi che svilupperemo oggi saranno algoritmi in realtà non polinomiali vediamo un esempio detto per familiarizzare col problema Abbiamo 6 oggetti che indichiamo con le lettere da a ad F ciascuno di questi oggetti ha un valore corrispondente è indicato come 724 541 e ha un volume e i volumi sono corrispondentemente 53231 e uno c'è una frac di volume = 8 in basso a sinistra vedete una soluzione ammissibile che il sottoinsieme costituito dagli oggetti cd ed è che È ammissibile perché la somma e volumi è pari a c d e quindi 3 a 2 + 3 + 1 di totale fa 6 che è minore di 8 mentre invece il valore di questa soluzione è la somma dei valori Quindi 4 + 5 + 4 che fa 13 è la soluzione ottima non lo sappiamo la troveremo dimostrerò che la soluzione ottima diventava alle 15 quindi non è ottima e il la soluzione invece che avete in basso a destra è un sottoinsieme non ammissibile quindi non è una soluzione adatta ed è costituita dagli elementi ac e d perché non è ammissibile pur essendo un sottoinsieme perché il volume di a è 5 quello dici e due quello di dietro è la somma di 5 + 2 + 3 fa 10 che è strettamente maggiore di 8 È un peccato perché il valore invece è 7 + 4 + 5 che fa 16 e sarebbe invece migliore che non 13 questo credo che chiarisca definitivamente il problema quella domanda è come risolviamo questo problema è la risposta è che esiste un algoritmo ricorsivo che è abbastanza semplice nella sua struttura è che funziona in questo modo Allora l'idea è che noi abbiamo il nostro zaino la nostra istanza di in quella di zaino che è una quaterna la quaterna costituita da un insieme B grande tra le funzioni e vi piccolo e dalla capacità di grande qualunque stanza del Pollino dello zaino è definita da 47 di questo genere o la cosa interessante è che questa istanza si può In effetti ricondurre a due altre istanze più piccole che sono dei sotto problemi risolti i quali Noi siamo in grado di risolvere l'istanza di partenza come funziona questa riduzione questa discussione semplicemente scegliamo un elemento qualunque per esempio l'ultimo elemento l'elemento ennesimo dell'insieme B grande è chiaro che quell'elemento ho parte della soluzione ottima o non ne fa parte non ci sono altre possibilità ma cosa vuol dire questo Supponiamo di considerare il caso in cui l'elemento non fa parte della soluzione Questo vuol dire che se lo buttiamo via da l'insieme B e quindi consideriamo l'insieme Fatemelo chiamare BN meno uno che poi sarebbe meno n In pratica cioè l'insieme degli oggetti dall'indice uno all'indice n meno 1 e fatemi considerare le funzioni in piedi Sugli Sugli insieme piccolo quindi le restrizioni di tali funzioni che io continuo a chiamare feehily anche se in realtà si potrebbe essere pignoli si potrebbe cambiare il nome sono le restrizioni dopodiché abbiamo uno zaino dello stesso volume uguale identico Ecco se siamo in grado di risolvere all'ottimo questo problema in realtà la soluzione che troviamo è anche la migliore delle soluzioni del problema di partenza fra quelle che non contengono l'oggetto ennesimo sotto l'ipotesi di no 0:19:20 La migliore delle soluzioni del problema di partenza fra quelle che non contengono l'oggetto ennesimo cioè sotto l'ipotesi di non prendere l'oggetto ennesimo In poche parole se io considero ipotesi di rifiutare l'oggetto non ho più bisogno di risolvere il mio problema sullo zaino con N elementi Ma posso risolvere un problema con lo stesso zaino e nemmeno un elementi che evidentemente un po' nella più piccolo questo è banale ma è chiaro che non pos-

siamo limitarci a risolvere questo problema perché ogni tanto capiterà di prendere l'oggetto ennesimo per il semplice motivo che sennò si va avanti ricorsivamente a buttare via l'ennesimo getto e si arriva ad avere una soluzione completamente vuota non ha senso Allora può darsi che invece l'oggetto ennesimo vada preso che succede se prendiamo l'oggetto ennesimo succede che lo prendiamo lo mettiamo nello zaino e ci troviamo di fronte a un problema di zaino con N meno 1 elementi insieme bm1 esattamente come prima le stesse restrizioni di piedi di è un volume di zaino che in realtà è più piccolo Cioè come se noi invece di avere lo zaino con dentro un oggetto avessimo uno zaino un po' più piccolo quanto più piccolo più piccolo di Bienne perché viene il volume occupato dall'ennesimo getto nello zaino originale quindi è come se avessimo lo zaino più piccolo è un insieme di oggetti più piccolo Trovata la soluzione su questo zaino più piccolo Abbiamo la soluzione originale quasi Nel senso che quella soluzione Liolà soltanto incrementata via aggiungendo le l'oggetto ennesimo con il suo valore Quindi in effetti non possiamo trovare preso il problema di zaino due sotto problemi anche loro problemi di zaino le cui soluzioni ci suggeriscono la soluzione ottima del problema di partenza basta prendere la migliore fra la soluzione in cui si rifiuta l'ennesimo getto è la soluzione in cui l'ennesimo getto viene accantonato si riduce la dimensione dello zaino e poi però si somma il valore dell' ennesimo oggetto questa cosa dà luogo a una equazione ricorrente che vedete qua in alto la quale equazione ricorrente riguarda il valore ottimo di un problema non riguarda il tempo di calcolo di un algoritmo ma per il resto è una questione di corrente si può trattare come tutte le equazioni ricorrenti del caso che avete visto in precedenza cioè questa equazione ha un caso base e ha due casi ricorsivi Andiamo un pochino più nel dettaglio il caso bene quelle in cui video oggetti di grande vuoto quindi a n = 0 oggetti Qual è la miglior soluzione se non ci sono oggetti e di non prendere nessun oggetto è l'unica soluzione possibile è il suo valore 0 si vedono è molto facilmente tanta parte in generale perché ci sono due casi ricorsivi Perché in effetti io ho ipotizzato due possibilità per quella di buttar via l'ennesimo getto di infilarlo nello zaino ma non mi è venuto in mente che magari l'oggetto nello zaino non ci sta è possibile che l'oggetto non ci stia nello zaino Allora se l'oggetto non ci sta non ho due casi ne ho uno solo sono costretto a buttarlo via Se invece l'oggetto ci sta o due oggetti Ecco perché abbiamo due casi ricorsivi è un primo caso di corsivo semplice che corrisponde la situazione in cui ci sono più oggetti cioè almeno un oggetto n maggiore di 0 e il volume di quell'oggetto però supera il volume dello zaino la capacità dello zaino in tal caso ottima su n oggetti con capacità di coincide con la soluzione ottima con i primi ennemenouno oggetti e con capacità di Perché Perché l'unica cosa da fare buttar via l'ennesimo getto Se invece vedi n minore uguale al grande Ecco che abbiamo due possibilità e Sceglieremo la migliore la massima o ci limitiamo a tenere la soluzione ottima senza l'ennesimo oggetto ma con tutto il volume di zaino disponibile o prendiamo la soluzione ottima con N meno 1 getti uno zaino più piccolo di un fattore bdnf e poi sfruttiamo questo fattore vdn per inserire l'ennesimo getto nello zaino e guadagnare un premio un valore fil.in la migliore di queste due soluzioni è chiaramente la migliore soluzione complessiva Quindi siamo in grado di risolvere il nostro problema la cosa interessante è che questo è questione ricorrente si riferisce soltanto due valori interi mentre Le istanze sono descritte da 4 oggetti dei quali il primo insieme il secondo il terzo sono funzioni e il quarto è un numero intero come è spiegabile questa cosa in realtà è molto semplice l'insieme B è un insieme di oggetti elementari 0:24:10 Come è spiegabile questa cosa in realtà è molto semplice l'insieme B è un insieme di oggetti elementari e si descrive con la sua cardinalità non ci interessa sapere altro Esattamente come nel caso dei grafi l'insieme dei vertici o dei nodi era descritto con la cardinalità n quindi questo B diventa n sì e vi sono delle funzioni definite subito cioè definite sui numeri da 1 N se invece di avere n abbiamo ennemenouno oggetti come vi dicevo in precedenza p e v andrebbero sostituite dalle loro per restrizioni ma non è un grosso problema

da un punto di vista matematico informatico continuare a chiamare le funzioni alla stessa maniera da pulita matematico Direi che è solo una questione di notazione dal punto di vista informatico siccome una funzione sarà un vettore indicizzato da 1 N se il vettore lo voglio indicizzare da 1 NM in uno mi basta accorciarlo cioè so' sti una tabella ma la tabella È ancora un vettore con la sola differenza che c'è un indice è nemmeno uno che mi dice Quanti elementi della tabella io devo guardare Ecco perché un'istanza è descritta non da 4 ma in realtà soltanto da n soltanto da due parametri e questi due parametri sono dei numeri interi tranquillamente tutto questo ci suggerisce un semplice algoritmo risolutivo che è poi questo descritto qua sotto l'idea Qual è l'idea è che se per caso n = 0 il valore ottime 0 Altrimenti se il ennesimo oggetto ha un volume che supera la capacità di grande non facciamo altro che applicare l'algoritmo a un'istanza ridotta in cui gli oggetti ce ne sono e nemmeno uno le funzioni Fi e vi sono ristrette ai primi e nemmeno un oggetti quindi sono comunque gli stessi vettori ma gestiti fino n meno 1 e la capa lo zaino è la stessa di grande sei invece volume dell'ennesimo getto e minore uguale a b grande allora risolveremo separatamente il problema suo è nemmeno un oggetti con le stesse funzioni ristrette e con il totale la capacità totale di grande oppure il problema suo è nemmeno urgenti con le due funzioni con un capacità ridotta di un fattore v n e poi somme Remo al risultato il premio il valore Fi dell'ennesimo oggetto la migliore di queste due soluzioni fa freezer If you non sarà la soluzione complessiva questo algoritmo Vi faccio notare ci restituisce il valore della soluzione non ci restituisce la soluzione però per cominciare lo risolveremo così è possibile in realtà farsi restituire anche la soluzione non è banalissimo E siccome scopriremo che questo algoritmo è del tutto inefficiente direi che non approfondiamo questa cosa anche se potrebbe essere una esercizio puramente a livello di esercizio di programmazione ripeto l'algoritmo che così abbiamo costruito non è un algoritmo da applicare in pratica però che cosa succede succede che vale la pena di mettere le mani sul codice Quindi a questo punto andiamo a prendere i materiali e quello che troverete è che esiste un file 90.cm è predisposto a risolvere i problemi di questo genere contrariamente alla maggior parte delle lezioni ho arricchito un po' il codice cioè questa volta non abbiamo solo la classica funzione che interpreta la linea di comando andando a caricare il nome del file dei dati Eccola qua ma abbiamo qualcosa di più non abbiamo soltanto l'inclusione di standard standard input output String non abbiamo solo la definizione della lunghezza delle stringhe il tipo logico boolean e i vettori che ci servono questa volta abbiamo anche la definizione dei dati e delle soluzioni della trackball cioè delle strutture che sono costruite per rappresentare istanze e soluzioni del problema dello zaino in che cosa distanza di un problema di zaino abbiamo detto che è una quaterna una quaterna costituita da un insieme che però essendo fatto di oggetti elementari è semplicemente la cardinalità n da un numero intero che è la capacità di è da due funzioni p e v valore volume che corrispondono a dei vettori Infatti sono dei vintage cioè sono dei vettori dinamici di interi Per quanto riguarda la soluzione dovrà presenteremo con il suo valore f e quando risolveremo anche il problema di trovare la soluzione non solo il valore avremo un vettore di incidenza di te vedi Uni Se vi ricordate un sottoinsieme di un insieme si può rappresentare in vari modi ma uno dei modi più efficienti è quello del vettore di incidenza 0:29:00 I vari modi ma uno dei modi più efficienti è quello del vettore di incidenza occupa spazio per cui abbiamo n oggetti l'occupazione lineare non è terribile ma è molto veloce nel aggiunta cancellazione è nella test di appartenenza e quindi abbiamo fatto questa scelta di usare un vettore dinamico di valori logici dopodiché abbiamo anche delle funzioni e una funzione che carica i dati una funzione che distrugge i dati una funzione che crea la soluzione è una funzione che distrugge la soluzione e infine una soluzione che una funzione che stampa la soluzione sono tutte fatte con la classica struttura che ben conosciamo e somigliano molto a day alle funzioni che abbiamo visto per gestire strutture dati al punto che verrebbe quasi da dire ma

perché non abbiamo fatto una libreria per gestire problemi di zaino la risposta è puramente per semplicità potrebbe essere un'ottima idea di fare una libreria isolando staccando questa questa parte per costruire una tutta questa parte iniziare per costruire una un file di intestazione tutta questa parte iniziale per costruire un file di intestazione e isolando ciò che segue cioè le funzioni di caricamento dei dati da file di distruzione dei dati di creazione e distruzione di una soluzione di stampa per costruire invece il file della libreria non andava nei dettagli parliamo invece di formato abbiamo quattro file a disposizione quattro esempi di stanze che prendono il nome dal numero di elementi molto piccolo di 6 elementi che poi è esattamente l'esempio che abbiamo visto disegnato in questo lucido Dopodiché c'è un fai un po' più grosso da 10 elementi uno da 24 e una delle tue a 50 elementi quello da 24 alla caratteristica che i valori numerici sono piuttosto grossi mentre invece gli altri file hanno tutti sia quelli da 6 da 10 ma anche quello da 50 è fatto tutto di più Merini piuttosto piccoli da due o tre cifre allora qual è il formato che Abbiamo ipotizzato che i dati sono costituiti da quattro righe una riga che dà il insieme B quindi il numero di elementi preceduto da n = 1 seconda riga che dà i valori degli elementi uno in fila all'altro separati da spazi preceduto Duffy uguale Poi abbiamo i volumi separati da spazi e preceduti da V = E infine abbiamo la capacità è un numero intero Istituto Dami grande uguale queste ultime due righe in realtà non rappresentano gli atti rappresentano la soluzione ottima che io mi sono precalcolato a parte che ho riportato qui in maniera che la avessimo come riferimento sto sfruttando il fatto che i dati Siccome il la funzione di caricamento dei dati legge solo le prime quattro righe poi nelle righe successive uno può scrivere quello che vuole potevate metterlo si poteva mettere queste informazioni a parte in altri file è solo per comodità che sono finite lì allora Che cos'è che dobbiamo fare adesso quello che dobbiamo fare che vi invito a fare perché veramente un operazione molto semplice e di andare qui nel centro del file della funzione mail a scrivere una procedura ricorsiva basata su questa equazione ricorrente che abbiamo commentato e su questo schema su questo pseudocodice di algoritmo determini la soluzione ottima dei nostri dei nostri problemi e delle nostre quattro distanze È chiaramente una cosa assolutamente semplicissima prima di avviarmi a farlo faccio notare che per la prima volta in questo corso Io ho introdotto alcune operazioni che servono a registrare tempi di calcolo qui in alto o incluso la libreria Time. h che mi permette di utilizzare due oggetti mi permette di utilizzare Intanto il tipo clock ti e poi mi permette di utilizzare la funzione clock queste due funzioni questa funzione questo tipo che cosa fanno molto semplice la funzione clock va a misurare il numero di unità elementari di tempo su questo devo qualcosa dopo che è passato dall'inizio dell'esecuzione del programma restituisce un numero sostanzialmente che è sostanzialmente è un numero intero positivo ma viene mascherato come tipo clock è questo numero Mi dice quanto tempo è passato dal inizio 0:33:50 Intero positivo ma viene mascherato come tipo clock è questo numero Mi dice quanto tempo è passato dall'inizio dell'esecuzione prima di eseguire il nostro algoritmo visto che qui Scriveremo la chiamata l'algoritmo e subito dopo la differenza di questi due numeri fine meno inizio mi dice quante unità elementari di tempo sono trascorse dall'inizio alla fine dell' algoritmo questo numero viene convertito in numero reale X evitare divisioni troncate è diviso per il clock persec questo quello che pensi che è una costante simbolica che è dichiarata dentro in Time. h e che venisse quante unità temporali ci sono in un secondo a seconda delle macchine tipicamente i valori più tipici sono un milione o miliardo Cioè sostanzialmente la unità temporale di tempo singola unità di tempo registrata in Di solito è un milionesimo o è un miliardesimo di secondo ho detto una stupidaggine vuoi un millesimo o è un milionesimo di secondo quindi si tratta o di 1000 clock per Sec o di un milione di klostersepp la seconda delle macchine a questo punto abbiamo un numero reale che ti dice quanti secondi sono trascorsi dall'inizio alla fine possiamo tranquillamente stamparlo a video detto ciò possiamo avviarci quindi direi potete mettere in pausa

e risolvere il problema Oppure potete seguire quello che stiamo facendo cos'è che vogliamo fare vogliamo chiamare un algoritmo ricorsivo per il problema di zaino no problem questo algoritmo riceve i dati e restituisce i risultati alla cui verrebbe abbastanza spontaneo dire che siccome i dati li ho chiamati i che sarebbe stanza è la soluzione L'ho chiamata S potrebbe valere la pena di scrivere una cosa di questo genere scrivo e poi correggo S Ale a algoritmo ricorsivo suino e questo non è del tutto sbagliato È un po' fastidioso perché in uscita restituire una struttura è una cosa che non abbiamo mai fatto che tendenzialmente non si fa perché semplicemente perché è un oggetto complicato che al suo interno contiene un puntatore perché la soluzione è un contiene un vi booleane il WLAN è un puntatore Quando si gioca con questi puntatori non si sa mai esattamente bisogna sapere esattamente cosa si sta facendo per cui più che scrivere così sarebbe ragionevole scrivere in quest'altro modo però diciamo che in realtà noi in questa fase iniziale stiamo ipotizzando che l'algoritmo ci restituisca solo il valore della funzione E allora diciamo correggendo che questo algoritmo istituisce spf cioè il campo F Eccolo qua il campo valore della funzione obiettivo in questa determinata soluzione mentre il campo essepuntoinox per il momento lo lasceremo del tutto vuoto e qui abbiamo accantonato in parte il problema perché non passiamo i per un motivo molto semplice che questo deve essere un algoritmo ricorsivo Che significa che quando all'interno dell'algoritmo noi Scriveremo tutto quello che ci serviva poi dovremmo passare all'algoritmo l'istanza ridotta ora la distanza ridotta sarebbe una scatola di questo genere con dentro i nuovi dati del problema ridotto ora questa cosa è perfettamente lecita Ma comporterebbe all'interno della chiamata ricorsiva di costruirsi una seconda istanza trasformando l'istanza padre in ciascuna delle due istanze figlie perché vi ricordo che abbiamo due chiamate ricorsive diverse e questo vorrebbe dire al locale due volte strutture che poi una volta fatto tutto quello che si deve andrebbero anche deallocate distrutto e quindi c'è tutto noi Red di gestione di memoria di copia di trasformazione di dati che se si può evitare è meglio a volte non si può però se si può è meglio Allora cos'è che si può fare una cosa che si può fare che spesso si fa e dire siccome i due problemi figli sono molto simili al problema padre prendiamo i dati del problema padre cioè prendiamo questa istanza i che stiamo effettivamente utilizzando la prendiamo e la modifichiamo in maniera da poterla usare come istanza del problema figlio cosa vorrebbe dire vuol dire che all'interno della istanza i n vada è che ha inventato eventualmente vi va ridotto Fi 0:38:40 Cosa vorrebbe dire vuol dire che all'interno della istanza i n va decrementato eventualmente viva Vidotto Filippo e fortuna si lasciano come sono perché abbiamo già detto che è sufficiente dire che vanno considerate fino al valore n qual è il problema Il problema è che nel momento in cui modifichiamo distanza ed entriamo qua dentro in uscita dobbiamo recuperare i dati del problema padre questo a volte si fa effettivamente c'è a volte per gestire passando ricorsivamente dei dati da Bonaparte Albona figlio è possibile che si vada non modificare i dati stessi e poi si recuperino a posteriori i dati del problema padre però qui si può fare ancora di meglio si può sfruttare il fatto che in realtà di questi quattro dati due non cambiano ha fatto perché si è vi vengono semplicemente ristretti attraverso il trucco di avere una cardinalità che Cala e gli altri due sono dei numeri naturali sono i numeri interi e il meccanismo di chiamata ricorsiva produce una copia che quando io chiamo algoritmo ricorsivo Enel sui dati e nemmeno uno è in realtà è nel record di attivazione sto costruendo una copia di è nemmeno una copia di lì quindi non ho bisogno di recuperare a posteriori dati del problema originale perché i dati del problema originale rimangono nel record di attivazione del problema originale intoccati quindi l'idea Sostanzialmente è che se invece di passare i che viene passato per indirizzo e quindi viene modificato eccetera eccetera io passo i quattro campi separatamente posso Quando cambi separatamente vuol dire il valore n del della struttura ed il valore il valore vi è infine il valore di grande Ecco che all'interno della mia funziona algoritmo di corsi 0:40:40 Il valore vi

e infine il valore di grande Ecco che all'interno della mia funzione algoritmo ricorsivo posso in realtà semplicemente sostituire nelle chiamate ricorsive i valori chiamati con quelli che mi servono Effettivamente non so se sono stato chiaro ma direi che la cosa più sensata è procedere e ovviamente diventerà addirittura banale a quel punto quello che noi vogliamo In effetti ottenere a questo punto solito meccanismo di scrittura della dichiarazione un prototipo e della definizione quello che vogliamo fare è sostanzialmente di risolvere il l'istanza di marcia problem con N oggetti di valore VII volume e capacità e con capacità non sono gli oggetti che hanno capacità vi come lo risolviamo con un algoritmo o con l'algoritmo ricorsivo naturale quello che viene in mente a chiunque ragioni sul problema Questo funziona e mi deve restituire un intero riceve un intero n riceve un vettore di interi Fi è un vettore di interi V E riceve Infine un altro intero che è la capacità di grande A questo punto ci siamo svincolati dalla tortura Chi è qui sotto Possiamo tranquillamente andare a riempire la nostra funzione questo algoritmo ricorsivo alla struttura che abbiamo visto qua sopra Praticamente possiamo quasi copiarla se non effettivamente copiarla però lo facciamo un passo alla volta per ragionarci sopra è in sostanza che cos'è che dobbiamo fare in sostanza e dobbiamo dire c'è un caso base caso base il caso in cui n è uguale a zero in questo caso restituiamo come valore della nostra soluzione Zero altrimenti abbiamo due casi ricorsivi c'è il primo caso di corsivo nel qual è il volume dell'elemento ennesimo non eccede quindi non sta all'interno del dello zaino in DVD n in questo caso è maggiore di grande e bene allora la nostra soluzione è semplicemente identica alla chiamata dell'algoritmo di corsivo su un problema e qui viene buono quello che dicevamo prima che non è i che avremo ricevuto prima copiato e modificato e trasformato ma è dato dal valore n meno 1 che a questo punto verrà copiato all'interno delle strutture del record di attivazione di questa chiamata ricorsiva e quindi sarà naturalmente distinto dal n di partenza e poi ci saranno il vettore f**** Tanto è sempre lo stesso e non viene mai toccato non viene mai modificato da nessuna chiamata il volume stesso discorso è la capacità dello zaino che in questo caso specifico è assolutamente identica altrimenti ce l'ha il tuo caso l'altro caso segnamocelo ma non è necessario imporlo è il caso in cui VN è minore uguale alla capacità dello zaino questo è semplicemente un promemoria in questo caso noi abbiamo due possibili soluzioni e dobbiamo vestire la migliore delle 2 potremmo anche fare Max di Però cerchiamo di essere un pochino più ariosi e chiari andiamo a prenderci i valori 0 e fai uno che sono i valori di Fi valori della Chiusa 10 ep 1 per rimanere coerente coi lucidi forse sarebbe più chiaro di f0 F1 perché la funzione obiettivo F Comunque sia 0 intendendo il valore ottimo che abbiamo calcolato qua sopra quello del problema in cui si rifiuta l'ennesimo getto Quindi abbiamo solo i primi anni meno 1 e la capacità dello zaino e rimane la stessa mentre più uno è dato da che cosa è dato dalla soluzione nella quale si prendono i n meno 1 oggetti su uno zaino più piccolo è ridotto del valore di dn e a questa soluzione in realtà si va a sommare il valore di N perché l'oggetto non viene effettivamente buttato ma anzi viene inserito all'interno dello zaino A questo punto dei due valori 0:45:20 Viene effettivamente buttato ma anzi viene inserito all'interno dello zaino A questo punto dei due valori andiamo a prendere il massimo che significa qui potete farlo in tanti modi approfitto per ripassare il operatore ternario Sefi uno è maggiore di zero Allora restituiamo fi1 altrimenti restituiamo p071 dei vari modi in cui si può scrivere questa soluzione tutto sommato Questo è quanto non c'è altro di particolare da fare e mo vedere se la cosa convince il compilatore cioè sì compilando con le solite opzioni il programma na70 il file Nef zero e costruendo il nostro programma vista sintattico non ci sono problemi possiamo andare a vedere se le i file i risultati sono corretti allora Se lanciamo il programma su input vediamo che la soluzione è 15 ed effettivamente qui ci siamo Mi ero segnato le soluzioni ottime le soluzioni ottime Sono effettivamente pari soluzione ottima di questo caso effettivamente pari a 15 non c'è un yx = per il semplice motivo che non abbiamo calcolato la soluzione soltanto il

valore Dopodiché possiamo valutarlo su 10 e vediamo che è 309 e qui si può andare a controllare ed effettivamente risultato dovrebbe essere 309 il risultato successivo è più elaborato ed è 13 milioni Euro 549000,94 mentre questo punto ci rimane solo da provare il file l'istanza da 50 e qui casca l'asino perché casca l'asino perché come vedete le prime due istanze sono state risolte in meno di un milionesimo di secondo la terza è stata risolta in 7 centesimi che è pochissimo ma è già molto di più questo è un problema che è grosso il doppio 50 cm però non ci sta mettendo il doppio perché Perché non è un algoritmo lineare Che razza di algoritmo è questo algoritmo e se ci pensate un attimo allora non è banalissimo la equazione ricorrente non è ovvia però se uno ci pensa vede che tutto sommato abbiamo per ogni problema Abbiamo al massimo due problemi figli la cui dimensione è nemmeno uno ciascuno di questi avrà due problemi figli di dimensione è nemmeno due eccetera Non è difficile vedere che si rischia non sempre perché c'è un caso base qui e c'è Ci sono casi in cui ci si ferma e si ha un figlio solo però non è difficile vedere che si rischia di avere un albero binario con n + 1 livelli a livello 0 a livello n Cosa vuol dire e vuol dire che abbiamo 2^n cetera eccetera il conto Esatto Non lo faccio nodi da affrontare in Ognuno dei quali le operazioni sono costanti perché vedete che sono confronti sottrazioni meno 1 meno vdn e confronti di questo genere ora la domanda è quanto posso aspettarmi che duri questo algoritmo posso andare a prendere un caffè come faccio c'è la definizione di complessità asintotica nel caso pessimo che ci aiuta 2 alla n è un numero che ci dà il tempo di calcolo certo andrebbe X una costante C che non conosciamo ma in fondo possiamo fare un attimo faccia approssimata cosa ci dice questa stima molto semplice che qui abbiamo due alla sesta Cioè se non sbaglio 64 operazioni moltiplicate per una costante bisogna fare una stima di quanto operazioni il nostro computer farà in un secondo ragionevolmente con un clock di gigahertz possiamo pensare a miliardi di operazioni sto facendo le Stime molto a volte ma vedete che la cosa ha un senso quindi 2 alla sesta operazioni e te ne fa miliardi in un secondo che ovvio che non ci arriva neanche un milionesimo tue radici ma sono 1000 1000 su miliardi a un milionesimo siamo lì Qui invece abbiamo due alle 24 Allora cominciamo a fare i conti in qualche modo che cos'è 2^{24} su 24 è questo bel numero qua se Supponiamo che ci sia un miliardo operazione ogni miliardesimo di secondo possiamo dividere per un miliardo e vedere che il risultato è 0,0 16 Qui abbiamo 0,07 abbiamo sbagliato ma l'ordine di grandezza è abbastanza simile

0:50:10

Felici Qui abbiamo 0,07 abbiamo sbagliato ma l'ordine di grandezza è abbastanza simile E allora che cosa rischiamo che succeda per 2 alla 50 sostituiamo ci viene fuori un numero che espresso in secondi È inquietante perché se lo dividiamo per i secondi che ci sono in un'ora e poi lo dividiamo per le ore che ci sono in un giorno teniamo a 13 giorni quello che è successo in realtà è che mentre parlavamo il calcolo è terminato per cui la nostra stima di 13 giorni è stata abbastanza lontana dalla realtà che rimasta sui 3 minuti intorno perché ma sostanzialmente perché questo due la 50 ipotizza che la albero binario vado giù giù fino alle foglie è credibile questa cosa Se guardiamo i valori dei volumi che sono di una o due cifre su un totale di capacità che ha tre cifre evidente che il numero di oggetti che stavano in questo zaino tendenzialmente non sarà mai di 50 sarà molto più basse e quindi in effetti questo due da 50 e una stima per eccesso però da un punto di vista pratico l'idea che la doppia vela dimensione non porta a un raddoppio del tempo ma una crescita molto più grande è in effetti una stima estremamente utile che mi permette di capire che questo tipo di algoritmo probabilmente non è così valido se al posto di 420 scrivessimo dei valori più alti probabilmente avremo dei grossi problemi a risolvere anche istanze da 50 Nodi all'ora che cos'è che bisogna fare bisogna cercare di capire se ti ha sbagliato qualcosa ed è quello che cercheremo di fare nella seconda metà della lezione che per adesso si interrompe Ben ritrovati Allora eravamo rimasti nella

prima metà della lezione con il dispiacere di vedere il nostro bel algoritmo ricorsivo impiegare 3 minuti e potenzialmente addirittura anche diversi giorni per risolvere ogni stanza tutto sommato piccola fatta di 50 oggetti che non è certamente un numero alto si può pensare istanze anche in Ghiaia di oggetti io a capire dov'è il problema allora dov'è il problema per capirlo possiamo vedere questo piccolo esempio che non è l'esempio che abbiamo visto in precedenza un esempio ulteriormente Vidotto fatto solo di 4 oggetti nel quale però appare abbastanza chiara la questione La questione è che l'algoritmo di ricorsivo continuo a risolvere ripetutamente sempre le stesse problemi vediamo l'esempio Allora noi abbiamo un insieme una stanza con 4 oggetti si suppone che non l'ho scritto Mi correggo l'ho scritto in alto a destra abbiamo una capacità a Pavia sei Questo significa che possiamo concentrarci sul sul l'ultimo elemento e quindi possiamo andare a decidere se l'elemento numero 4 può essere fissato nello zaino Ho cancellato e avremo quindi due sottoproblemi vedete le due frecce che escono dal sotto problema iv-vii V6 se tu hai sotto quelle mi sono tre Phoebe sei che corrisponde a buttar via e quarto elemento e tre figli v5 che corrisponde a fissare l'elemento nello zaino e quindi sottrarre alla capacità sei il valore invidi quattro cioè uno ciascuno di questi due per uno ne genera altri due quindi sopra avremo due figli di 6 e 2 fili 5 perché il terzo elemento anche so volume = 50 = 1 mentre invece tre fili 5 decidendo se fissare fuori o dentro dalla soluzione delle mento 3 logo a due fili 5 e due fili quato Ed ecco la sorpresa il problema due fili 5 compare due volte e siccome questi problemi vengono tutti risolti vedete che non solo quel problema lì ma tutti i problemi figli sono esattamente identici ai problemi figli della coppia che sta subito sopra bene Abbiamo un sottoalbero costituito da 1 2 3 4 5 e 6 sotto problemi che si ripete esattamente identico Ma la cosa non si esaurisce lì perché come vedete ancora più sotto il sotto problema 2 fili 4 che di per sé è nuovo però genera un sotto problema 1 film di 4 che si può trovare anche in un'altra zona dell'albero e che a sua volta ha dei sotto problemi perfettamente identici quindi siccome sottopormi tendono a esplodere combinatorica mente ogni

0:55:00

Che a sua volta Adesso problemi perfettamente identici quindi siccome sottopormi tendono a esplodere combinatorica mente ogni sotto problema copia da luogo a un intero sotto albero di sotto problemi copia e questo Evidentemente a Cava il tempo di calco in questo esempio l'aggravio non è forte possiamo contare i sottoproblemi vedere che quelli che sono doppioni sono 1 2 3 4 5 6 7 8 9 10 10 su un totale che se non ricordo male è di 28 Quindi non in realtà pochissimi dovrebbero essere 12 di sotto 12 sopra fa 24 poi sono 27 in tutto che comunque è una percentuale è degna di nota che cosa succede però assunto lei ma più grande ma più grande realtà succede che la frazione e poi è inutile che vengono risolti nuovamente nuovamente nuovamente tante volte Basta vedere quante volte viene risolta ad esempio il problema 0,5 Ce ne sono due due volte viene risolto 0 V 1 somma ci sono parecchie copie potete pensare che il numero delle copie esploda dopo un po' Ebbene questa cosa Evidentemente è inaccettabile produce dei risultati che sono estremamente inefficienti Come si può evitare tutto questo discorso l'idea è abbastanza semplicemente quella di andare a tutte le volte che si risolve un problema e andare a salvarsi la soluzione perché potrebbe essere utile per un per uno scopo futuro perché può darsi che poi in un momento successivo della computazione ci sia da risolvere lo stesso problema questo è l'idea di fondo della programmazione dinamica conservare le soluzioni dei sotto problemi in maniera da non doverle ricalcolare Nella speranza che i problemi si ripresentino e che quindi non sia necessario risolvere di nuovo Ma basti consultare una struttura opportuna e in cui sono conservate le soluzioni la programmazione dinamica da luogo a due grandi categorie di algoritmi che sono in realtà sostanzialmente se non identici sulla base di del modo in cui viene realizzata forse ve li tratta in maniera top Town o in maniera bottom-up la modalità top-down

a corrisponde sostanzialmente agli algoritmi ricorsivi che abbiamo appena visto nei quali invece di andare a risolvere tutti sotto problemi ogni volta che ho sotto problema Viene ritrovato viene costruito per prima cosa si va a cercare in un'opportuna struttura se per caso sotto prima è già stato risolto e In caso positivo invece di risolverlo si prende semplicemente la soluzione da quella struttura terza parte In caso negativo invece sotto problema andava risolto andare io la prima volta e alla fine della soluzione quello che si fa e andare a salvare la soluzione nella struttura stessa quindi il concetto in pratica è che invece di avere dei casi baseq elementari stimi e dei casi ricorsivi Vai ad eventuali come nel caso del nafta che mi aveva due introduciamo un ulteriore possiamo caso base nel qual è il problema non è in realtà base ma è già stato risolto e quindi la soluzione del problema è banale si può fare in tempo costante accedendo a questa struttura ausiliaria ovviamente tutto questo costa spazio lo spazio necessario a gestire questa struttura Che genere di struttura potrebbe essere Non è difficile dipende dal problema quindi va costruita Sulla base del problema ma qui abbastanza evidente che i problemi sono caratterizzati non da fidavi ma da due numeri interi di cui il primo numero intero è compreso fra 1 e denner secondo numero intero è un po' meglio zero e denner perché l'ultima l'ultimo livello di foglie è costituito da zero insiemi vuoti Dopodiché Invece l'altro numero che corrisponde alla capacità di grande è compresa fra la capacità dello zaino è un numero intero compreso tra 0 e la capacità dello zaino iniziale Ecco perché ho detto che sta correndo quella dello zaino iniziale È tipica del problema alla radice Quindi in realtà tutto questo è facilmente rappresentabile in una matrice che abbia da 0 a 50 4 su una dimensione è da 0 a 6 sull'altra e non è difficile poi vedere che i livelli in questo caso corrispondono ai valori da 0 fino a quattro 1234 potrebbero essere le colonie o potrebbero essere le righe se Trasformiamo la matrice mentre Invece nell'altra direzione nel possiamo accoppare tutti questi stati in un massimo di 7 righe o viceversa colonne setras poniamo questo cosa corrisponde in pratica non avevo un albero ma a far correre

0:59.50

Colonne si trasformano questo cosa corrisponde in pratica non avevo un albero ma a far corri a far uscire la freccia dal nodo per esempio tre figli di 5 e non a farla andare in un altro No domani a far arrivare al nodo 25 che è già stato generato da in precedenza quindi questo albero che potenzialmente esplodeva nella binaria moltiplicando per due il numero dei nodi ad ogni livello invece viene compresso e la sua larghezza la sua ampiezza non può superare il valore di V grande più 1 quindi al massimo oppure 7:00 righe e infatti qui vedete che ci sono 1-2-3-4-5-6-7 che corrispondono esattamente ai valori idv compresi fra 0 1 2 3 4 5 e 6 sono tutti i valori possibili questo questa matrice è la struttura nella quale Noi andremo a salvare i risultati dei problemi questo è ciò che ci provo di fare effettivamente adesso cioè di realizzare in modalità da un algoritmo di programmazione dinamica Poi vedremo la realizzazione e bottom-up che è legata semplicemente al fatto che tutti gli algoritmi ricorsivi hanno una espressione iterativa e in particolare gli algoritmi ricorsivi di programmazione dinamica consentono di lavorare iterativamente su queste matrici su queste strutture che conservano i sotto problemi che gli dia Qual è l'idea che abbiamo una matrice in una struttura in cui poniamo tutti sotto problemi possibili quindi una matrice Zero Da 0 a 4 e da 0 a 6 ruote 5 X 7 dopodiché alcuni questi sotto poi mi sono banali Sotto polemicone n = 0 per esempio tutti questi problemi qua corrispondono a massimizzare la il risultato ottenibile prendendo oggetti da un insieme vuoto c'è poco da massimizzare il valore 0 però questo è un caso base Dopo di che cosa fa a partire dai problemi più semplici possiamo combinare le soluzioni dei problemi più semplici per risolvere i problemi più complicati vieni a partire da questa da questo livello qui costruiremo le soluzioni del livello successivo con un elemento e poi del livello con due elementi poi del livello con 3 fino ad arrivare alle so-

luzioni del livello con quattro elementi capire che questo è un algoritmo tipicamente interattivo basato sul fatto che si ricombinano le soluzioni quindi viene a mancare la decomposizione del problema in sotto problemi perché è fatta già all'inizio costruendo la matrice la struttura di tutti sotto problemi mentre invece rimane la parte di ricombinazione dei delle soluzioni dei sotto problemi nelle soluzioni dei problemi ma questo lo vediamo poi con calma allora per andare a realizzare questo algoritmo ricorsivo in modalità top Town partiamo dal algoritmo ricorsivo delle proprio Eccolo qua che abbiamo realizzato in precedenza questo algoritmo ricorsivo dobbiamo modificarlo per tener conto del fatto che sfruttava una informazione ausiliaria Allora questo comporta un pochino di modifiche non algoritmi che non concettuali ma di natura sostanzialmente di programmazione e di struttura di architettura del codice Che cos'è che vogliamo effettivamente fare avremo un algoritmo ricorsivo che userà oltre alla definizione del problema stesso la struttura ausiliaria matrice alla quale dobbiamo dare un nome questa struttura avrà evidentemente allocata all'inizio di allocata Alla fine allora fare tutto questo nel mail non è propriamente bellissimo per evitare di farlo io andrò a costruirmi una funzione che incapsula tutto quello che occorre quindi la locazione iniziale l'algoritmo viene proprio e la deallocazione finale Dopodiché ci dedicheremo algoritmo come facciamo molto semplice io ho dava un qualche nome questa funzione potrei chiamarla programmazione dinamica ricorsiva per il naso problem non è un nome bellissimo ma può funzionare e questa operazione questa funzione è la voleva sui dati e vestiti erano risultato tutti quei ragionamenti che ho fatto riguardo al dividere l'istanza nelle sue componenti qui non valgono perché perché questa funzione non è ricorsiva è solo una interfaccia esterna una scatola che contiene la funzione stessa quindi qui mi permetto In effetti di andare a incorporare i dati Dopodiché posso anche stampare in uscita e risultati Allora già che ci siamo lo restituiamo li passando lì per l'indirizzo e tutto sommato siamo a posto così Quindi l'unico motivo per cui qui avevo diviso tutto questo era il fatto di poter sfruttare la modifica uno per uno degli elementi per poter 1:04:40 Avevo diviso tutto questo era il fatto di poter sfruttare la modifica uno per uno degli elementi per poter costruire i sotto problemi questa funzione qui andiamo a Locarno a Locarno ce l'ha a dichiarar se la qua in alto è quello che faremo sarà a risolvere di stanze in affitto problem distanza hit del Natale Coven Ottenendo la soluzione S con l'algoritmo di programmazione dinamica ricorsiva e questo mi dice esattamente che cosa stiamo cercando di fare è una funzione che restituisce niente perché la soluzione in realtà è il secondo dei dati Questo è un soluzione soluzione No problem passato per indirizzo perché va modificato ma ama anche perché in questo modo è più sufficiente passare l'indirizzo passare il risultato e dopodiché anche i dati del nastro al problem li passiamo per indirizzo semplicemente per questioni di efficienza controllo che si chiamino Sita tkp soluzione KT a questo punto siamo in grado di costruirci il nostro solito corpo vuoto e all'interno di questo corpo è alta Che cos'è che effettivamente faremo andremo ad allocare la matrice ausiliaria che ci occorre ci occorre una matrice ausiliaria Allora andiamo per gradi cominciamo a scrivere la traccia dell'algoritmo stesso andiamo allocare la matrice ausiliaria dei delle soluzioni dei sotto problemi cui dovremmo dare un nome Dopodiché andremo a ovviamente andremo anche a andremo anche a inizializzarla Perché noi dobbiamo essere in grado di riconoscere per ogni sotto problema se conosciamo o non conosciamo la soluzione quindi Inizializza la matrice ausiliare delle istruzioni sotto problemi in maniera stabile se conosciamo o no la soluzione Dopodiché potremmo finalmente andare a risolvere il problema è questo il vero e proprio algoritmo di programmazione dinamica ricorsiva per il naspro poi finalmente andremo a liberare questa lettura dati questa matrice la matrice ausiliaria abbiamo da fare ancora un pochino di cose sostanzialmente abbiamo bisogno di gestire una matrice Lo sapete che una matrice dinamica non è altro che un vettore dinamico i cui elementi sono a loro volta vettori dinamici potremmo fare la matrice come doppio puntatore a

intero io mi toglierò lo sfizio di andare a nascondere un po' tutto e quindi di dire che una matrice di interi non è altro che un vettore dinamico di vettori di interi vint questo non è particolarmente necessario mi serve semplicemente a distinguere quella che è una matrice da quello che potrebbe essere un doppio puntatore intero Quello che mi interessa in particolare però è Andrea dal locale questa matrice diamole un nome la matrice potremmo chiamarla più grande tanto per darle un nome perché contiene in realtà sostanzialmente i valori di zero edifici uno tutti i valori di fi che abbiamo ottenuto in questo Durante quegli esecuzione di questo algoritmo è questa matrice andrà allocata quindi in capsule amo il tutto Diciamo che stiamo allocando una matrice di interi questa matrice avrà un certo numero di righe un certo numero di colonne Quante ali e Quante colonne Allora abbiamo detto che il numero di righe e di colonne corrisponde al numero di elementi da più uno perché si va da 0 al numero totale di elementi e corrisponde al numero alla capacità dello zaino più uno perché gli indici andranno da zero al valore di grande quindi andiamo ad avere per esempio P freccia n + 1 righe e ti freccia di più 1 colonne il piffero è ceduto il fatto che i dati stanno nella stanza i però è un puntatore quindi si prende più il Freccia i campi sono al campo n e Campogrande qua dentro andremo a fare tutti i controlli del caso quindi non vado a verificare il risultato perché lo faremo qua dentro per fare tutto questo abbiamo bisogno di una funzione che faccia 1:09:30 Risultato perché lo faremo qua dentro per fare tutto questo abbiamo bisogno di una funzione che faccia questa locazione vado ad dichiarar mela e poi a definirla la dichiaro qua sopra Perché tutto sommato potrebbe un certo punto vale la pena di avere una libreria per matrici giusto per questo però è una funzione che semplicemente all'Oca una matrice di Int con NR righe e NC colonne e si può tiza che si parta da zero Quindi questa funzione restituisce una matrice interi riceve in ingresso un certo numero di righe riceve in ingresso un certo numero di colonne e dopodiché questa sarà una funzione che dovremmo andare a realizzare in qualche modo per il momento ci scrivo un bel 21 nulla Così siamo a posto qui poi cosa dobbiamo fare inizializzare la matrice Allora potremmo scrivere una funzione anche qui però è molto semplice idea che valore ha un problema in cui la soluzione non è nota che a dargli un numero che sia impossibile per qualunque sotto problema siccome questi sotto problemi hanno una funzione valore positiva e non è possibile che non ci sia un valore negativo Allora tanto per scrivere in qualche modo dei valori che mi faccia capire che il problema non è stato risolto kg meno 1 in tutti quanti che vuol dire sostanzialmente che devo scorrere righe e colonne e righe colonne corrisponderanno le righe all'indice i dei nodi degli elementi e le colonne a valori di capacità più o meno alta le chiamerò ogni tanto per richiamare il fatto che il valore massimo è V maiuscolo e quindi avrò uniche va da 0 i30n e avrò un vip che invece va da 0 fino a p freccia di grande E per ognuno di questi valori Noi andiamo a scrivere dentro la matrice nella cella Ivi andiamo a scrivere meno uno segui preferite mettere una costante simbolica con un valore negativo cambiabile a piacere non c'è niente di male anzi potrebbe essere anche più sensato a questo punto avremo l'algoritmo l'algoritmo somiglierà molto a questo algoritmo qua Questo algoritmo ricorsivo di per il lo zaino Non copierò esattamente il nome stesso oppure Sì diciamo che lo usiamo e poi lo modifichiamo Quindi vado a chiamare questo algoritmo è il risultato andrà a essere scritto nella cella F della soluzione p.s. mi chiamerò questo algoritmo questo algoritmo funzionerà sui dati di freccia n funzionerà sulla funzione valore P freccia Fi sulla funzione volume di freccia vi è sul valore di capacità P freccia vi è questo funzionava più o meno come già abbiamo fatto ma con quel caso base aggiuntivo che abbiamo detto per avere il caso base aggiuntivo gli dobbiamo passare anche la matrice Fi grande Questo significa che risalendo la dichiarazione la definizione dell'algoritmo di corsivo devono cambiare perché qua dentro avremmo bisogno anche della matrice di grande quindi con l'algoritmo di corsivo non algoritmo ricorsivo che usa la matrice ausiliaria grande Ecco che abbiamo cambiato qualcosina quindi

possiamo andare di sotto a cambiare l'intestazione e ovviamente ti Avvia il punto e, qui cambieremo ancora qualcosa terminiamo top down la realizzazione della funzione di corsiva cosa dobbiamo fare del locale la matrice ausiliaria questo si potrebbe fare con una funzione ad hoc Diciamo che per fare un po' in fretta Io mi limito a scorrere da 0 a p i freccia n più più e vado a deallocare la riga i-esima dell'amatrice Fi e concludo de allocando anche la costola di questa bella matrice Sarebbe concettualmente corretto fare una funzione qui ma per fare in fretta non lo faccio 1:14:20 Sarebbe concettualmente corretto fare una funzione qui ma per fare in fretta non lo faccio Ok Allora a questo punto che cos'è che ci manca in teoria dovremmo avere tutto abbastanza corretto sintatticamente può essere un buon momento fermarsi e verificarlo ovviamente dal punto di vista semantico niente va bene ma è per non accumulare i pasticci Allora siccome ci sono un po' di modifiche vediamo di che cosa si tratta vediamo che ci sono nella chiamata questo algoritmo Ah beh certo tutte le chiamate interne avrebbero bisogno della matrice Fi In ognuna di queste va modificata passando gli Amatrice Fi Dopodiché ancora una volta alla riga 91 questa chiamata non ci serve più la chiamata nel main non l'abbiamo più e a questo punto c'è un punto e, che ci siamo dimenticati alla riga 269 bisogna andare fino a p n e questo quantomeno sintatticamente corretto non provo neanche a Lanciano perché evidentemente Qui c'è una di allocazione di una matrice che non è stata bloccata Quindi è verosimile che succede un pasticcio sistemiamo questa locazione è la classica costruzione di una matrice ci serve un cursore e dopodiché dobbiamo Per prima cosa andare al locale la matrice stessa per cui ci serve anche una matrice M questa matrice m non è altro che il risultato di una callo o malloch Vito che abbiamo visto che la allo chiamo al momento la inizializziamo anche potrebbe andar bene scrivere malloch io vado avanti in questa maniera è una matrice dnr righe ognuna della quali è un vettore di interi Ovviamente a questo punto invece Dobbiamo verificare se la locazione è andata a buon fine segnando Lore nell' allo azione della matrice m eventualmente e uscendo e a questo punto qui abbiamo la (da chiudere a questo punto dobbiamo fare esattamente la stessa cosa riga per riga Quindi per chi che va da 0 a PN su ogni singola riga noi possiamo procedere a fare che cosa ad allocare la Jesi ma vita però il risultato sarà un vettore di interi e i singoli elementi saranno interi e saranno n c e andiamo a verificare che il tutto abbia funzionato perché è meglio non si sa nulla perché se per caso è nulla Abbiamo un errore nella locazione della riga MPX se proprio vogliamo farci dire tutto quello che è successo questo dovrebbe essere quanto a questo punto Ricordiamoci di nuovo restituire nulla divertiti ed Amatrice che abbiamo faticosamente costruito Perfetto allora adesso abbiamo finalmente tutto quanto quello che succede che il nostro algoritmo è ancora quello vecchio non uso alla matrice filtro Cioè ci si può aspettare che quantomeno funzioni Qui abbiamo un p freccia n che giustamente sbagliato Questo NR e a questo punto non si aspetta che l'algoritmo funziona esattamente come prima visto che lo stesso Salvo una piccola interfaccia che non fa che all'Oca e dell'oca una matrice misuriamo le stanze piccole vediamo che non funziona lo stesso Allora questa è la classica situazione in cui si sono allocate da Locate male le cose con il stare attenti a vedere che cosa sia combinando E qui effettivamente nella matrice ne abbiamo deciso di bloccare una matrice con Enrique NCIS colonne specificando che cosa che le righe vanno da 0 1:19:10 Enrique NCIS colonne specificando che cosa che le righe vanno da 0 a n r mean 10 api freccia n e n r l i n + 1 Questo significa che ne abbiamo a Locate esattamente Nev e per ciascuna di esse da 23:59 dobbiamo andare ad allocare la riga Quindi abbiamo avvocato Naviga in più ha più pandola a una cella di memoria che non fa parte del vettore stesso e a questo punto dovremmo essere invece in grado di gestire il tutto ed effettivamente Diciamo che funziona possiamo fare le prove sulle altre istanze Magari anche un po' più grosse non quella da 50 e risultati sono relativamente confortanti vedete che il tempo di calcolo era nettamente aumentato perché perché abbiamo allocato riempito avvo-

cato che non sembra poi un'idea così geniale tutto sommato l'aspetto geniale sta nel fatto che adesso però possiamo sfruttare all'interno della funzione ricorsiva la conoscenza delle funzioni ricorsive questa qua la conoscenza delle soluzioni ai passi precedenti Come si fa ascoltare questa conoscenza ti ho detto con un ulteriore caso base questo caso base nel caso in cui la soluzione in nota quand'è che la soluzione nota quando Fi non è meno 1 ma è un valore maggiore uguale a zero Quindi in effetti esiste Mi scuso per la brusca interruzione in realtà Nel digitale ho messo in pausa la registrazione e poi sono andato avanti a parlare per svariati minuti e quel punto era quasi impossibile recuperare il punto di origine quindi mi stavamo completando l'implementazione di questo algoritmo ricorsivo per il nap Sack basato sul uso di una matrice ausiliaria e abbiamo detto che avendo la matrice ausiliarie non sappiamo che quando il contenuto della matrice stessa per il problema al corrente Quindi n.fi grande è maggiore uguale a zero significa che questo problema in qualche momento è stato già risolto ne conosciamo La soluzione è la soluzione è proprio figli e neve quindi possiamo limitarci a vestitu irefi dnb Altrimenti possiamo proseguire con tutti gli altri casi Allora questo dovrebbe sostanzialmente sistemare il tutto salvo il fatto che però negli noi dobbiamo salvare il risultato altrimenti non capiterà mai di entrare in questo utile e gradevole caso baset Quindi in effetti noi dobbiamo aggiungere alla realizzazione degli altri casi il salvataggio infidi nvt che cosa del risultato che nel caso base di insieme vuoto e zero mentre In altri casi base Sarà che cosa sarà il valore della chiamata ricorsiva e qui Evidentemente ce lo Salviamo però poi dobbiamo restituirlo all'esterno quindi dobbiamo andare al terminato il tutto a restituire Fi dnb mentre in questo ultimo caso consisteva il tutto nel restituire fidi NV avendo assegnato afidi NV il migliore dei due risultati ottenuti in precedenza allora tutto questo si può ulteriormente sgrossare osservando che in pratica in tutti i casi non restituiamo fili NV perché è il valore calcolato l'unico in cui non succede qui Dovresti tu chiamo zero ma qui restituiamo In effetti assegniamo Fi dnv a zero e quindi possiamo effettivamente restituirlo anche qua a questo punto se ci sono quattro casi che terminano tutti alla stessa maniera non ha senso tenerli separati possiamo fare un'unica restituzione finale del contenuto della matrice E questo ci permette di cancellare queste operazioni dai singoli casi il che ci semplifica moltissimo e oltretutto in un caso che è questo in cui si era già Noto e consiste nel non fare assolutamente nulla ma questo è una situazione che abbiamo già trovato in molti algoritmi ricorsivi tutti quelli di ordinamento per cui si tratta sostanzialmente solo nel caso in cui fidi fidi NV è negativo solo nel caso in cui non è stato ancora risolto il problema di andare a lanciare tutti i casi precedenti Quindi questo è un altro modo equivalente più semplice 1:24:00 Tolto il problema di andare a lanciare tutti i casi precedenti Quindi questo è un altro modo equivalente più semplice di descrivere la stessa cosa ora se il tutto è stato realizzato senza particolari a voi possiamo andare a vedere qual è l'effetto sui risultati e sui tempi di calcolo Per quanto riguarda CLS possiamo andare in ricompilare il nostro codice e provare lanciarlo sistematicamente una per una sulle istanze di stanza piccola otteniamo lo stesso risultato in tempo velocità massimo distanza da 10 stesso discorso su distanza da 24 vedete che c'è un rallentamento Abbiamo impiegato un secondo e 15 lo stesso risultato anziché 0,07 centesimi Cioè Euro 0,07 secondo che è un forte aggravio che cosa succede nel caso dell'istanza da 50 istantanea la soluzione quindi un calo brusco da 3 secondi a 3 minuti fa niente neanche un milionesimo è un aumento di un fattore Boh 10-15 è ragionevole tutto questo per vedere se è ragionevole mia fare un po' di teoria c'è studiare la complessità asintotica nel caso pessimo non è elementari Simo qui perché bisogna capire qual è la complessità di questa funzione ricorsiva che ha un caso base un po' strano per cui non è un caso base facilmente trattabile come quelli a cui siamo abituati Però possiamo fare un po' di ragionamenti che genere di ragionamenti Allora il seguente intanto dentro questo IV entreremo soltanto una volta sola perché non appena entriamo qualunque strada si scelga il valore di Fidene vi sarà assegnato

e sarà maggiore uguale a zero Quindi in effetti in ogni problema si entra una volta sola Le volte successive si non si fa altro che avessi tuire in uscita il valore della Cella se trascuriamo questa cosa Che cosa che succede sostanzialmente succede che la il nostro albero Sì visitato Ma in realtà è tutte in ciascun sottoproblema si entra e si eseguono le operazioni una volta sola Quanti sono i sotto problemi non sono 2^n o dell'ordine di 2 anni Come viene calcolato prima perché questo qui non è un albero è un albero che si richiude su se stesso da 3 cv 5 c'è la freccia che va su questo nodo già visitato ed è già fatto una stima per eccesso di questi nodi perché gli abbiamo assegnati a Celle di una matrice una matrice che ha qui a colonne corrispondenti agli elementi e sono n + 1 colonne e righe corrispondenti ai livelli di capacità da 0 a b grande quindi sono più grande più uno nella nostra implementazione abbiamo messo sulle righe le II sulle colonne Levi però non cambia nulla Il lavatrice sempre Grossa n + 1 per di più uno In ognuna di queste Celle c'è un calcolo che è in tempo costante quindi tutto questo spinge in direzione di ipotizza che il nostro algoritmo sia di complessità o grande dnp1 di più non c'è o grande DN per ora e né per vivi nel problemi piccoli e 6 x 8 48 fa ridere anche qui abbiamo un 10 x 1765 che fa ridere anche nel 50 abbiamo 50 x 420 che fa 20 mila non è molto tutto sommato Ma nella istanza intermedia Qui abbiamo 6 milioni e rotti per 24 Quanto fa 6 milioni e rotti per 24 proviamo a calcolarlo Allora abbiamo che su 6 milioni 400-480 notificato 24 e otteniamo 154 milioni In sostanza ma 2 alla 24 invece cioè il vecchio algoritmo ricorsivo il vecchio i tuoi corsivo Agusta 16 milioni non 170 quindi il vecchio lui Così io probabilmente una decina di volte più veloce il che torna perché era invece che un secondo Euro 0,07 Cioè come dire 10 centesimi di secondo E questo tutto sommato CV con forza sul fatto che tutto ciò abbia senso allora domanda è meglio l'algoritmo di programmazione dinamica o è meglio algoritmo ricorsivo sembrerebbe che dipenda dal istanza La risposta è sì dipende dagli stanza per stanza e con valori idv molto grandi probabilmente può convenire la algoritmo ricorsivo ma Tenete conto che 2^n è decisamente veloce cioè confrontare 2^n Da una parte con n x v dall'altra e significa che vi deve essere per perdere vi deve essere nettamente maggiore di $2^n/n$ e questo tende ad essere un numero veramente grande per cui sono poche le istanze cercarle con lanternino sono un po' 1:28:50 E questo tende Adesso è un numero veramente grande per cui sono poche le istanze cercare col lanternino Sono poche le stanze in cui la programmazione dinamica viene dominata detto ciò allora Uno potrebbe dire ma questo algoritmo di complessità o DNA è un algoritmo polinomiale in fondo n è lineare via lineare viene da dire che sia polinomiale Lo è o non lo è Ecco in realtà detto strettamente parlando questo algoritmo non è polinomiale perché non è polinomiale per capirlo bisogna andare un pochino a rivangare la definizione di dimensione di un'istanza io gli ho detto le misure distanza n se ci sono n oggetti Ma questa è una definizione informale la vera definizione di dimensione di ogni stanza è quanti caratteri sono necessari per rappresentarla Quanti simboli sono necessari per rappresentarle questi simboli è chiaro che dipende perché più è alto n e più caratteri ci vogliono e la cosa è tipicamente proporzionale ma in realtà l'aspetto di vi cambia un pochino le cose perché perché vi può essere un numero molto piccolo o può essere un numero molto grande può essere 8 o può essere questo 6 milioni e rotti e la dimensione dell'istanza dipende poco davi Cioè se vi cresce se vi passa da 8 a 6 milioni la lunghezza del file non è cresciuta di molto da un carattere è passato semplicemente a 7 caratteri l'aumento E logaritmico perché Perché per rappresentare in cifre 6 milioni bastano 7 cifre decimali Per rappresentare un numero come 8 ne basta una cioè la dipendenza della dimensione del file da questo parametro vie logaritmica la dipendenza dell'algoritmo per parametro B invece nave e lineare rispetto logaritmico e esponenziale per vi è l'esponente è 2 alla logaritmo di di conseguenza si può dire In effetti che l'algoritmo che abbiamo costruito non sia polinomiale ma in un certo senso sì esponenziale è vero che provi non è un parametro sensibile importante è come il numero di elementi è un parametro che

potrebbe essere piccolo anche per distanze molto grandi Io potrei avere i numeri non eccessivi con milioni e milioni di oggetti e l'ora che cosa succede niente Succede che esiste una classe di algoritmi che sono non polinomiali Ma che hanno una non polinomiali TAC e non dipende da n dipende da altri parametri e si può sperare che per alcune stanze molto comuni questi parametri non eccedano in dimensione e questi sono algoritmi che vengono detti pseudopolinomiale tutto questo extra rispetto al programma del Corso Secondo me è interessante da un punto di vista anche concreto Quindi abbiamo un algoritmo pseudopolinomiale di programmazione dinamica vi avevo anticipato il fatto andiamo avviciniamoci a l'ultimo argomento che la programmazione dinamica si può realizzare anche bottom-up cioè in effetti è perfettamente possibile andare a gestire questa famosa matrice non dall'alto in basso creando i problemi risolvendo gli scrivendo nella matrice E quando ci si ritorna usando il dato scritto ma si può metodicamente costruire la matrice dal livello più basso al livello più alto costruendo la Punto c'è costruendo la soluzione di ciascun problema a partire dalle soluzioni di sotto problemi io risolvo il livello 0 sapendo che con niente zero elementi l'ottima è sempre zero se non c'è niente Non si può guadagnare nulla e i problemi del livello 1 li posso risolvere semplicemente Nando Le due soluzioni corrispondenti al livello zero Quindi il problema 1 fighi sei Lo risolvo a partire dei problemi Zero fidi 6:06 V2 l'uno FT4 lovisolo partire da devo finì 400 eccetera eccetera e questa cosa la posso fare su tutto questo livello e poi su tutto questo e poi su tutto questo e poi sono tutto questo l'idea è quindi aveva un algoritmo iterativo il quale elenca tutti sotto problemi in una bella attrice e nel caso dello zaino risolvere i problemi più semplici quelli del livello 0 e poi ricombina le soluzioni dei problemi semplici per ottenere quelle di problemi complessi e risale fino ad arrivare dove arrivare al problema più complesso in assoluto che è quello dato Qual è la complessità di questo algoritmo che adesso vediamo sarà abbastanza semplice scorrere tutti i problemi se per ognuno di questi problemi Il tempo richiesto è costante la complessità sarà ancora una volta o DN x v cioè proporzionale 1:33:40 Ognuno di questi problemi Il tempo richiesto è costante la complessità sarà ancora una volta o di n x v cioè proporzionale al numero di sotto problemi e allora Uno potrebbe dire ma perché faccio questo anziché l'alto metodo quello ricorsivo per il semplice motivo che la ricorsione come sapete a un overhead di allocazione sullo Stack dei record di attivazione di coppia di deallocazione che tutto sommato può sempre valere la pena di cancellare e quindi in effetti si può avere una preferenza per una pensione integrativa adesso prima di ragionarci so Provo a far vedere un attimo in che maniera potrebbe funzionare questa versione ragionando sul esempio il solito quello che abbiamo visto nei luci di alcuni all'inizio questo con 6 oggetti e ragionando Appunto su questi 6 oggetti e sulla dimensione di zaino che va fino al valore Quindi abbiamo la nostra bella matrice con le righe associate agli elementi da 0 a 6 e con le colonne associate ai valori di capacità da 08 Cosa possiamo fare noi sappiamo che conseguimenti la soluzione è Chiaramente sempre nulla quindi possiamo risolvere questi casi base semplicemente mettendo degli zeri a questo punto siamo in grado di risolvere i problemi con un elemento sostanzialmente soltanto con il primo semplicemente combinando i due sotto problemi di livello 0 che gli corrispondono o il singolo sotto problema perché sapete che a volte Potrebbe non esserci due sotto problemi in particolare come l'elemento uno ha valore 7 e Volume 5 È chiaro che se il l'elemento a Volume 5 e lo zaino ha capacità 0 1 2 3 o 4 non è possibile infilarlo questo elemento Quindi in realtà questi sotto problemi quelli che corrispondono a Un solo elemento è a zaini di capacità compresa tra 0 e 4 hanno come unica soluzione possibile ancora una volta Zero qui cominciano ad esserci invece due casi possibili se lo zaino capacità 5 Allora io posso o non prendere l'elemento 115 ereditare la soluzione del livello precedente che vale zero oppure posso prenderlo che vuol dire evitare dal livello precedente La soluzione che usa 5 unità in meno di zaino cioè anziché 50 ma vado 5 Passi a sinistra prendo questa soluzione e al valore

di questa soluzione sommo il valore del nuovo elemento che è 7 quindi o prendo 0 o prendo il valore che tra 5 Passi a sinistra sommato 70 + 7 e chiaramente è meglio dello più 7 faccio lo stesso qua con uno zaino di capacità 6 o prendo 0 o prendo il valore che sta 5 Passi a sinistra più il valore del nuovo elemento che evidentemente Domina è la stessa cosa succede anche nei passi successivi po' più interessante l'elemento due elemento valore 2 e volume 3 avendo volume 3 gli zaini con 012 non possono accettarlo e quindi Qui non c'è niente da fare L'unica possibilità e recuperare la soluzione del livello precedente e rifiutare l'elemento due c'ho fatto Se invece ho uno zaino di dimensione 3 questo elemento ci sta quindi posso confrontare La soluzione che vale zero con la soluzione che ha due come valore di base più quella ottenuta al livello precedente riducendo la dimensione dello zaino di tre perché tre è il volume dell'oggetto in sé prendo 0 gli sommo due che è il valore del nuovo getto e a questo risultato sommo confronto con questo risultato la soluzione precedente e quindi ottengo due Qui abbiamo la stessa identica lo stesso identico valore e qui finalmente Sì invece si deve confrontare Che cosa si deve confrontare la soluzione da 7 con la si ottiene sommando abbiamo detto due al valore che si trovava a tre passi prima 1 2 3 cioè 00 + 2 fa 2 confrontato con 7 chiaramente è meglio 7 Analogamente Sarà meglio 7 qui Sarà meglio 7 qui quando arriviamo allo zaino vero e proprio di dimensione 8 dobbiamo confrontare La soluzione che non prende l'oggetto 6:58 è la soluzione che prende invece l'oggetto due di valore 2 e occupa le sue tre unità quindi devi andare a cercare tre passi indietro come va la soluzione migliore valeva 7 il sommiamo il 2 è qui otteniamo il valore 9 1:38:30 Invece l'oggetto due di valore due e occupa le sue tre unità Quindi bisogna andare a cercare tre passi indietro come era la soluzione migliore valeva 7 il sommiamo il 2 è qui otteniamo il valore 9 e a questo punto possiamo in realtà proseguire Analogamente accelero un po' per non metterci tanto tempo l'oggetto 3A volume 2 quindi le prime due Celle sono impossibili da riempire Qui invece si può confrontare lo zero con 4 più il valore che abbiamo qui cioè due passi prima e quindi scriviamo quattro anche qui chiudiamo qua tu Anche qui Scriveremo 4 Qui invece sono confronteremo 7 con 4 più due cioè con 6 e vincerà il 7 e analogamente direi che continuerà a vincere il 7 anche qui perché confronto 7 con 2 più 4 ma quando arrivo Qui confronto 7 con 7 + 4 e quindi andiamo ad avere invece un valore 11 e qui confronteremo nove con 7 + 4 cioè con 11 che vince quanta possibilità è un oggetto di volume 3 quindi tre passi indietro e valore 5 quindi dobbiamo sommare 5ck volume 3 i primi tu i primi tre zaini sono ereditano la soluzione del livello precedente 004 Qui invece ti può confortare il 4 con 5 più la soluzione a tre passi indietro e quindi vince vince 5 Dopodiché avremo ancora 5 poi avremo il confronto fra 7 e 4 + 5 9 che vince poi avremo ancora nove qui avremo un confronto fra 11 e 12 3 che fa 4 + 5 9 per cui vince l'undici e finalmente qui abbiamo un 12 perché 12 perché confrontiamo l'11 con 12377 più 5/12 ed ecco il 12 penultima riga ancora più velocemente stavolta il volume 1 quindi soltanto il caso zero e a verità la precedente i casi successivi confrontano la precedente con la Celletta più a sinistra quando quattro quindi qui confronto 00 + 4 e fra quattro ancora confronto 40 + 4 fa 4 qui confronto 5 con 4 più 4 che fa 8 e vince Otto qui confronto 5 con 5 più 4 che fa 9 poi Confronto nove con 5 + 4 e vuol dire che ho due possibilità perfettamente equivalenti Cioè posso non so mai nulla ed evitare il 9 o posso sommare quato è a partire da questo 5 Dopodiché possiamo avere 9 + 4 13 che vince sul 99 + 4 13 che vince sul 11:11 + 4/15 che vince sul 12:00 ultimo elemento stavolta si somma 1 e si guarda la Celletta sinistra questo rimane Zero questo è 4 che è meglio dicevo più 15 cioè 4 più uno è meglio di 48 è meglio di 4 + 159 è identico 8 + 19 Qui c'è un 10 meglio 9 più uno è meglio di 91390 è meglio di 9 + 1 10 Qui c'è 14 che questo 13 + 1 ed è meglio di 13 e qui confrontiamo 13 più uno cioè 14 con 15 e questo è quanto andiamo a fare questa cosa a livello di codice Come si fa a fare questa cosa a livello di codice Allora riparto dal codice precedente e vado questa volta a sostituire la funzione PD ricorsiva con la funzione PD inte-

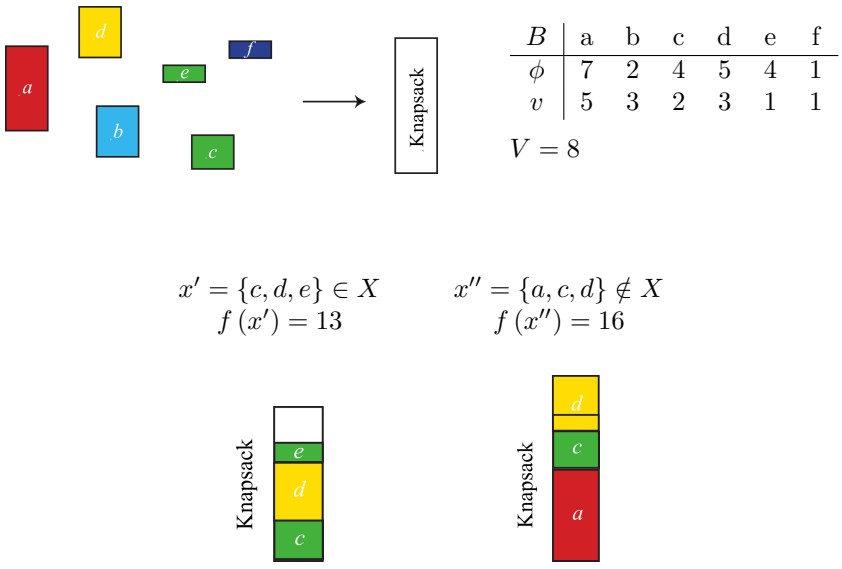
rattiva andiamo a modificare interattiva con l'algoritmo di PD di programmazione dinamica iterativa Analogamente qua sotto andremo a sostituire la ricorsione con la versione iterativa e ovviamente in questa versione non avremmo questa procedura ricorsiva però la tengo perché vedremo che avremo qualcosa di molto simile in realtà è questo è ovviamente interessante di per te allora ci servirà Comunque la matrice questo assolutamente non cambia quello che invece andrà pesantemente a cambiare e il fatto che non chiameremo l'algoritmo ricorsivo con il togliamo pure questa parte qui avremo un algoritmo di programmazione dinamica iterativa per il naso problem qui non vedo la necessità di fare una funzione perché non abbiamo chiamate ricorsive che 1:43:20 Me stessa potremmo farlo non farlo non è così importante e come abbiamo fatto nelle esempio quello che faremo e procedere di riga in riga per i che cresce da 0 a n e di colonna in colonna andando a riempire le varie Celle quindi non è strettamente necessario inizializzare la matrice Anzi direi che non è necessario inizializzare la matrice visto che la Riempiamo completamente quello che faremo recupero il ciclo savas cover e riga per riga da 0n colonna per la colonna da 0v e riempire le singole Celle Allora come si fa molto semplice Andiamo su ciascuna singola cella a risolvere il corrispondente problema il sotto problema di dimensioni Eevee Come si fa a risolvere questo sotto problema e allora ci sarà ovviamente il caso in cui = 0 Qualcuno potrebbe anche più semplicemente a risolverlo a parte io ho voluto tenere tutto insieme quindi potremmo chiederci se per caso inizi a zero bene se io zero è abbastanza chiaro che la soluzione del problema consisteva nell' andare a scrivere nella cella il valore 0 è questa è l'inizializzazione che abbiamo fatto qua nella prima riga e questa cosa Se ci pensate un attimo corrisponde pari pari a questo caso base se n = 0 fili n v = 0 questo è esattamente la stessa operazione dopodiché abbiamo questo caso quello in cui il volume del CNA esimo qui Jesi mo elemento Kevin è superiore a che cosa è superiore alla capacità quindi che cosa succede se il volume degli esimo elemento supera la capacità corrente semplicemente succede che si calcola si va a prendere il valore nella riga precedente con lo stesso volume Cioè In sostanza si va ad assegnare a i lì che cosa il valore che la matrice Fi aveva in corrispondenza alla riga Imen uno è in corrispondenza esattamente la stessa colonna vi vedete la forte somiglianza formale fa questa assegnamento e quest'altro algoritmo ricorsivo non è altro che il contenuto qui diamo per scontato che entrano in questa chiamata ci sia il valore di fichi e quindi sì basta farti tranquillamente leggerlo e lo vedi e nemmeno uno e invii grande che corrispondono agli mn1 che avevi piccolo qua dentro a questo punto c'è infine il terzo caso il terzo caso è quello in cui invece ci sono le due possibilità cioè l'oggetto può essere effettivamente sia preso sia non preso perché non è necessario scriverlo Perché Ovvio è la condizione complementare perché il volume dell'oggetto è minore o uguale a che cosa alla capacità dello zaino corrente si vanno a calcolare le due soluzioni e si va a prendere la migliore delle due quindi proprio per mostrare anche la corrispondenza formale di queste cose il valore di zero e ovviamente a questo punto qua mi serve un zero e mi serve un figuno Allora il valore di zero Non sarà altro che che cosa sarà il contenuto di dell'amatrice fi in corrispondenza al rigga Imen uno è in corrispondenza alla colonna vip che non cambia la capacità del nostro problema questo corrisponde paro paro alla chiamata ricorsiva nel caso Zero Nel caso uno invece andremo a prendere fidi n che in questo caso è parte della istanza Eccola qua Eh ti fidi elemento iesimo e poi andremo a andare a consultare nella matrice quale cella la cella i meno 1 e la cella non vi mangi meno il volume dell'elemento Jesi moscetti i v dell'elemento di dell'elemento chiesimo e questo sistema anche quest'altro caso in sostanza è come se avessimo chiamato la funzione ricorsiva e la funzione ricorsiva ci avesse dato la sua soluzione perché tanto è stata calcolata al passo precedente a questo punto non è ovviamente bisogna raggruppare 1:48:10 Calcolata al passo precedente a questo punto non è finito e qui Ovviamente bisogna raggruppare in un blocco dobbiamo andare a settare che cosa andare a settare il valore di film E come lo sentiamo

possiamo andare a prendere qua sopra perché esattamente la stessa cosa andiamo a settare prendendo il massimo dei due valori tutto questo è molto simpatico non è solo simpatico ma in realtà funziona anche correttamente e se non abbiamo fatto particolari errori dovrebbe essere tranquillamente possibile compilarlo Qui ci sono degli n Davy maiuscolo che sono evidentemente risultato di tempesta Infatti questo al solito ieri ma c'è la prima colonna e in teoria Tutto questo ci dovrebbe permettere di risolvere una alla volta le nostre istanze Naturalmente no Ah certo perché ci siamo dimenticati di restituire il risultato finale Allora fatto tutto questo La domanda è Dov'è il risultato finale è veduta tuo finale evidentemente sta in questa cella qua perché questa la cella n di grande è la cella che corrisponde all'intero problema quindi in sostanza nella soluzione nel campo F andiamo a scrivere pdnd grande A questo punto abbiamo buone speranze questo n evidentemente Lenny dell'istanza e questo vi è il della istanza a questo punto abbiamo buone speranze di riuscire a risolvere i singoli problemi Uno per Uno Infatti otteniamo il solito 15 il solito 309 il solito 24 con un pochino di rallentamento forse Persino peggio di prima è il solito 50 istantaneo perché è rallentato ulteriormente perché non vi sarà forse sfuggito il fatto che nel passare in rassegna sistematicamente tutte le righe tutte le colonne No in realtà stiamo risolvendo anche dei sotto problemi che non esistono Quindi questa versione iterativa è apparentemente più pesante ancora di quella ricorsiva domanda l'ora perché la versione iterativa ti solite quella che effettivamente si esegue non si esegue quella ricorsiva l'ho fatta didatticamente per mostrare Com'è la dinamica top Town Come si passa da una all'altra Perché Perché intanto Vi amiamo gli overhead di allocazione E deallocazione perché evitiamo di avere una sequenza di sullo Stack ma sequenza di n chiamate Dov'è ne potrebbe essere migliaia Potrebbe non esserci spazio sullo Stack E poi perché in realtà in realtà quello che succede è che comunque sia appunto eseguire molto più velocemente i singoli passi può compensare anche il fatto di eseguire delle Celle in più ti risolvere dei sottoporre In più va detto che esistono pensioni di questa implementazione iterativa che noi non facciamo oggi nei quali In effetti si riesce a saltare le c'è degli utili per non è necessario effettivamente come qui scorrere sicuramente necessario scorrere tutte le righe Ma tutte le colonne tutti i volumi di non è strettamente necessario però questo sono complicazioni sofisticazioni molto utili che cedono i limiti del corso e quindi non li presento Ecco perché ha comunque senso fare questa questa cosa allora rimane l'ultimo argomento l'ultimo argomento che fino adesso noi abbiamo recuperato il valore della F ma abbiamo ignorato il valore della ICS Come si fa a recuperare la soluzione lo faccio vedere qui Se noi partiamo da questo 15 ne sappiamo che il 15 è stato ottenuto o risalendo con la soluzione Dal Passo precedente il che significa che l'oggetto sei Non è stato preso oppure andiamo a vedere l'oggetto sei l'oggetto sei a volume 1 vuol dire che questo 15 ottenuto dalla soluzione del livello precedente con volume con capacità ridotta di 1 Cioè questo 13 Allora se 15 è uguale a 13 più uno vuol dire che ci siamo arrivati in diagonale prendendo l'oggetto Mentre se 15 è uguale a 15 cioè Los precedente ma sulla stessa colonna vuol dire In sostanza che l'oggetto non l'abbiamo preso né alcuni casi potrebbe essere che vado a entrambe abbiamo visto dei casi in cui c'era mi pare un 9 non so più se era questo questo in cui si poteva forse questo in cui si Sali 1:53:00 Che valgono entrambe abbiamo visto dei casi in cui c'era mi pare un 9 non so più se era questo questo in cui si poteva forse questo in cui si saliva Allora qui all'oggetto 5 ho detto 5 a premio quattro se sono in questo punto io posso o non prendere l'oggetto 5 e avere una soluzione che valeva 9 e non prende un oggetto 5 oppure posso partire da questa soluzione che invece valeva 5 prendere l'oggetto di indice 5 che apre mio 4 e 5 + 4 mi dà 9 quindi ho due modi equivalenti di prendere questa soluzione Posso scegliere io faremo una scelta molto semplice che quella di dire che se rimango sulla stessa colonna è meglio anche se è uguale di conseguenza tendenzialmente in caso di parità Diciamo che l'oggetto non lo prendiamo però che tanto Vuol dire semplicemente che

ci sono soluzioni equivalenti Allora tutto questo significa in sostanza che è possibile esplorando la matrice anche ricostruire la soluzione Come è possibile ricostruire la soluzione è tutto sommato abbastanza semplice Allora l'idea è la seguente partiamo dal valore finale Cioè ipotizziamo che il volume impegnato sia tutta la capacità e qui mi ha fare pipì freccia quindi piccolo perché non viene maiuscolo Eccolo A questo punto possiamo retrocedere da n su fino a zero quindi possiamo Avi salire per i che va da n giù fino a 0 ovviamente il caso Zero non ci interessa andando lì indietro e ad ogni passo quello che possiamo fare chiederci sono arrivato a questa situazione dalle strato precedente o invece in diagonale in alcuni casi la diagonale non esiste Se vi ricordate tutti questi casi avevano una caratteristica che siccome l'oggetto ha un volume molto grosso andare in diagonale vorrebbe dire sfogare in negativo quindi se per caso il mio l'oggetto del volume corrente che si chiama p v p maiuscolo in stanza volume dell'oggetto Jesi mo se questo è maggiore della della Allora necessariamente sappiamo che la nostra soluzione a inyx digli il valore falso gli ha detto che sono dei valori booleani tra parentesi se la soluzione è stata creata con una culloch è già falsa dentro però per pulizia sto segnando tutti i valori altrimenti che cosa succede altrimenti succede che in generale avrò due possibilità e che dovevo capire quale delle due è quella che effettivamente è stata ascoltata quindi dovevo capire In sostanza se io sono ottenuto il mio valore partendo dalla cella al livello precedente Imen uno con la stessa capacità di o se invece l'ho ottenuto questa valore partendo dalla cella del livello precedente con volume di meno vi meno il volume occupato nell'istanza dal i esimo oggetto più che cosa più il premio figli il valore dell'oggetto iesimo a seconda che io sia arrivato qui da zero o da fy1 deciderò se non prendere oppure prendere il tutto In sostanza se per caso fidi i v è identico quest'altro Che cosa significa significa che magari è anche identico fy1 Però io fra i due scelgo questo e quindi sono contento effetto psdx Sii Loseto a falso Altrimenti significa che necessariamente sono arrivato salendo dalla diagonale e di conseguenza avrò che il valore quello vero Qui se volete potete confrontare potresti dire che in questo caso filivivi è uguale a quest'altra combinazione E allora in questo caso l'oggetto dovrete per forza prenderlo non vi dovrebbe sfuggire il fatto che però se l'oggetto lo prendete dovete spostarvi non in verticale ma in diagonale E quindi 1:57:50 Il fatto che però se l'oggetto lo prendete dovete spostarvi non in verticale ma in diagonale E quindi il valore corrente di vi deve essere a sua volta decrementato di che cosa di questo volume in maniera che poi siate in grado di procedere con la vostra ricerca all'indietro Ok allora questo punto non dovrebbe sfuggire il fatto che mi attacchi uno non l'abbiamo usato l'ho scritto solo per per poter poi scrivere qua di fianco di che cosa si trattasse con questa operazione si può tranquillamente togliere è rigorosamente parlando anche feed-0 non vi serve assolutamente a nulla anche questo l'ho scritto per far capire di che cosa si stesse parlando e tutto sommato anche qui l'espressione risulta parecchio più semplice Allora a questo punto se non abbiamo commesso particolari e voi Incrociamo le dita Qui abbiamo un p minuscolo alla riga 284 quindi che diventa P maiuscolo poi c'è un n non dichiarato perché qui dobbiamo fare pipì freccia n che scende e con un po' di fortuna possiamo provare a valutare le solite nostre quattro istanze e vediamo che la soluzione sarà 135 che effettivamente se andiamo a vedere i risultati Se prendiamo gli oggetti 135 Cioè In questo caso AC ed è il volume occupato e 5 più 2 più 18 che è minore uguale 8 e il premio 7 + 4 + 4 che fa 15 affettivamente e questa è la soluzione ottima Dopodiché possiamo andare a indagare l'istanza 10 Dove istantaneamente si trova la soluzione 234 sei Sui 10 elementi distanza da 24 Eccola qua ci mette un goccino ma vi dà la vostra soluzione e infine distanza da 50 istantaneamente vi restituisce il risultato finale a questo punto direi che abbiamo abbastanza indagato a fondo su questa materia quello che possiamo fare è riassumere qualche considerazione sulla complessità abbiamo visto che da algoritmo ricorsivo ci ha dato due alla nzd due la n si potrebbe essere anche molti meno se fosse veramente 2N ci mettevamo 13 giorni a risolvere l'istanza 50

non 3 minuti però comunque sia questo il numero dei problemi ammissibili nel caso pessimo ciascuno che chiede tempo costante quindi in totale l'algoritmo esponenziale le programmazioni dinamiche sia top-down e bottom-up limitano il numero dei problemi ammissibili a un tetta dnv dove se via piccolo perfetto Lago di Como e praticamente lineare Levi però può anche essere molto grande potrebbe essere al limite esponenziale in n e quindi andare peggio che non il 2 n anche se abbiamo visto che hai bisogno cercare un istante un pochino costruita ad hoc questo algoritmo di complicità attenta dnv Non è polinomiale lo è per limitato si dice in gergo che è pseudopolinomiale E questo deve chiudere l'argomento di oggi e la prossima lezione Oi affronteremo il problema dello zaino da un punto di vista diverso cioè dal punto di vista della ricerca con un algoritmo il più possibile efficiente veloce di soluzioni che però non hanno la garanzia di essere ottime quindi sono a euristiche e questo algoritmo sarà un algoritmo greedy cioè un algoritmo che è ingordo e va a procurarsi gli elementi della soluzione visto che la soluzione è un sottoinsieme uno alla volta sulla base di un criterio che al momento sembra ottimo ma che in realtà può provocare dei danni a lungo termine ed è per questo che la soluzione non è garantita essere ottima E questo chiude per oggi la lezione

Figura 13.4: Esempio



Lezione 15

La lezione è centrata su un esercizio per impratichirsi sui seguenti algoritmi per il problema dello zaino:

- algoritmo ricorsivo banale;
- algoritmo ricorsivo di programmazione dinamica *top-down*;
- algoritmo iterativo di programmazione dinamica *bottom-up*.

Figura 13.5: Un algoritmo ricorsivo per il problema dello zaino
Qualsiasi problema di zaino (B, ϕ, v, V) si può risolvere riconducendolo a due problemi di zaino più piccoli

Considerato un oggetto qualsiasi (per es., l'oggetto $i = n$), si può solo:

1. rifiutare l'oggetto
2. accettare l'oggetto (se non è troppo grande: $v_n \leq V$)

In entrambi i casi, ciò che rimane è ancora un problema di zaino:

1. se n viene rifiutato, si tratta di:
   - riempire uno zaino di capacità V con oggetti tratti da $B \setminus \{n\}$
2. se n viene accettato, si tratta di:
   - riempire uno zaino di capacità $V - v_n$ con oggetti tratti da $B \setminus \{n\}$
   - aggiungere l'oggetto n alla soluzione

La soluzione ottima del problema è la migliore fra quelle dei sottoproblemi

Figura 13.6: Un algoritmo ricorsivo per il problema dello zaino
Si può quindi scrivere l'equazione ricorrente

$$f^*(n, V) = \begin{cases} 0 & \text{se } n = 0 \\ f^*(n-1, V) & \text{se } n > 0 \text{ e } v_n > V \\ \max\left[f^*(n-1, V), \phi_n + f^*(n-1, V - v_n)\right] & \text{se } n > 0 \text{ e } v_n \leq V \end{cases}$$

dove si riconoscono:

- un caso base: insieme di oggetti vuoto
- due casi ricorsivi:
  1. zaino troppo piccolo per contenere l'ultimo oggetto, per cui si può solo rifiutare l'oggetto stesso
  2. zaino abbastanza capiente da contenere l'ultimo oggetto, per cui la soluzione è la migliore fra scegliere e rifiutare l'oggetto stesso

I valori $f^*(n, V)$ sono ottimi di vari problemi di zaino, dove n identifica

- la restrizione dell'insieme B al sottoinsieme $B_n = \{1, \ldots, n\} \subseteq B$
- la restrizione delle funzioni ϕ e v da B a B_n

Figura 13.7: Un algoritmo ricorsivo per il problema dello zaino

$$f^*(n,V) = \begin{cases} 0 & \text{se } n = 0 \\ f^*(n-1,V) & \text{se } n > 0 \text{ e } v_n > V \\ \max\left[f^*(n-1,V), \phi_n + f^*(n-1,V-v_n)\right] & \text{se } n > 0 \text{ e } v_n \leq V \end{cases}$$

I vari problemi si costruiscono facilmente riducendo n e V:

- per restringere l'insieme base B e i vettori `phi` e `v` agli elementi leciti
- per ottenere la capacità residua

```
if (n == 0)
  return 0;
else if (v[n] > V)
  return AlgoritmoRicorsivoKP(n-1,phi,v,V);
else
{
  phi0 = AlgoritmoRicorsivoKP(n-1,phi,v,V);
  phi1 = phi[n] + AlgoritmoRicorsivoKP(n-1,phi,v,V-v[n]);
  return max(phi0,phi1);
}
```

Figura 13.8: Esempio

È un algoritmo molto inefficiente: risolve ripetutamente problemi uguali

Per esempio, $\phi = [\ 6\ \ 8\ \ 3\ \ 2\ ]$, $v = [\ 4\ \ 2\ \ 1\ \ 1\ ]$ e $V = 6$

QUESTA FIGURA NON COMPILA: VA RECUPERATA

10 *problemi uguali su* 28

Figura 13.9: Programmazione dinamica

La programmazione dinamica si basa sull'idea di

- conservare le soluzioni dei sottoproblemi per non ricalcolarle

Si può realizzare in due modalità

- top-down: come algoritmo ricorsivo che
  - divide un problema in sottoproblemi e li affronta uno per uno
  - verifica se il sottoproblema corrente è già stato risolto:
    - in caso negativo, lo risolve e salva la soluzione in una struttura
    - in caso positivo, recupera la soluzione dalla struttura

  I problemi risolti diventano dei nuovi casi base, risolti in $\Theta(1)$
- bottom-up: come algoritmo iterativo che
  - elenca tutti i sottoproblemi possibili
  - risolve i problemi più semplici
  - combina le soluzioni dei problemi più semplici per ottenere quelle dei problemi complessi

Figura 13.10: Valutazione di complessità

Per l'algoritmo ricorsivo, abbiamo:

- $\Theta(2^n)$ sottoproblemi ammissibili
- tempo $\Theta(1)$ per risolvere ciascuno

e quindi complessità temporale $\Theta(2^n)$

Per la programmazione dinamica (*top-down* e *bottom-up*), abbiamo

- $\Theta(nV)$ sottoproblemi ammissibili
- tempo $\Theta(1)$ per risolvere ciascuno

e quindi complessità temporale $\Theta(nV)$, che è

- non polinomiale in genere (V *occupa spazio* $\log V$)
- polinomiale per V limitata

In gergo si definisce pseudopolinomiale

Figura 13.11: Il problema del cammino minimo

Dati

- un grafo orientato $G = (N, A)$ con $n = |N|$ nodi e $m = |A|$ archi
- un nodo origine $s \in N$ e un nodo destinazione $t \in N$
- una funzione di costo definita sugli archi $c : A \to \mathbb{N}$

si trovi un sottoinsieme di archi $X^* \subseteq A$ tale che

1. X^* sia un cammino orientato
2. X^* vada da s a t
3. X^* abbia costo totale minimo:

$$c_{X^*} \leq c_X \text{ per ogni } X \text{ cammino orientato da } s \text{ a } t$$

con $c_X = \sum_{(i,j) \in X} c_{ij}$

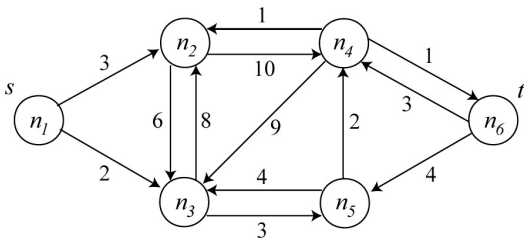


Figura 13.12: Varianti

Il problema del cammino minimo ha una definizione non banale

- se G e c ammettono circuiti di costo negativo, è mal posto
  Si può percorrere il circuito indefinitamente
- se X non deve contenere circuiti, non sono noti algoritmi polinomiali
  È uno dei "Problemi del Millennio"

Ci sono algoritmi polinomiali per grafi e/o funzioni di costo specifiche

- grafi G privi di circuiti
- funzioni di costo c non negative
- grafi G privi di circuiti di costo negativo rispetto a c

Ci sono algoritmi specifici in base all'origine e destinazione, per cercare

- i cammini minimi da ogni nodo a ogni nodo
- i cammini minimi da un nodo a ogni nodo
- i cammini minimi da un nodo a un altro (*troncano i precedenti*)

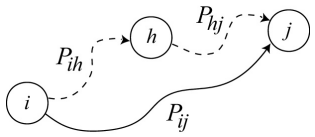
Consideriamo la ricerca dei cammini minimi da ogni nodo a ogni nodo

Figura 13.13: Operazione triangolare

Descriveremo un algoritmo che

- determina se G e c ammettono circuiti di costo negativo
- se non ne ammettono, fornisce un cammino di costo minimo per ogni coppia di nodi

Operazione triangolare: dato un cammino P_{ij} da i a j e un nodo $h \in N \setminus \{i, j\}$, sostituire P_{ij} con un cammino P_{ih} da i a h e un cammino P_{hj} da h a j



Quando $c_{P_{ij}} > c_{P_{ih}} + c_{P_{hj}}$, il cammino risultante è migliore dell'originale

È un'operazione promettente, ma

- come dovremmo applicarla, e quando smettere?
- la soluzione finale ottenuta è certamente ottima?

Figura 13.14: Un teorema fondamentale

Se $N_h = \{n_i \in N : i \leq h\} = \{n_1, \ldots, n_h\}$ il sottoinsieme dei primi h nodi e $P_{ij}^{*\,(h)}$ è il cammino di costo minimo da i a j nel sottografo indotto da $\{n_i, n_j\} \cup N_h$ e $\ell_{ij}^{(h)}$ il corrispondente costo, per $h = 0, \ldots, n-1$, allora

$$\ell_{ij}^{(h+1)} = \min\left(\ell_{ij}^{(h)}, \ell_{i,h+1}^{(h)} + \ell_{h+1,j}^{(h)}\right)$$

Dimostrazione: *Il cammino* $P_{ij}^{*\,(h+1)}$ *o include* n_{h+1} *o non lo include*

Se $n_{h+1} \notin P_{ij}^{*\,(h+1)}$, *allora* $P_{ij}^{*\,(h+1)}$ *è anche il cammino di costo minimo da* i *a* j *nel sottografo indotto da* $\{i, j\} \cup N_h$, *e il suo costo è* $\ell_{i,j}^{(h)}$

Se $n_{h+1} \in P_{ij}^{*\,(h+1)}$, *allora* $P_{ij}^{*\,(h+1)}$ *va da* n_i *a* n_{h+1} *e da* n_{h+1} *a* n_j.
Per il principio di ottimalità

- *il primo sottocammino è ottimo nel sottografo indotto da* $\{n_i, n_j\} \cup N_{h+1}$; *poiché non usa* j, *è anche ottimo nel sottografo indotto da* $\{n_i\} \cup N_{h+1} = \{n_i, n_{h+1}\} \cup N_h$ *e il suo costo è* $\ell_{i,h+1}^{(h)}$
- *il secondo sottocammino è ottimo nel sottografo indotto da* $\{n_i, n_j\} \cup N_{h+1}$; *poiché non usa* i, *è ottimo anche nel sottografo indotto da* $\{n_j\} \cup N_{h+1} = \{n_{h+1}, n_j\} \cup N_h$ *e il suo costo è* $\ell_{h+1,j}^{(h)}$

LA FIGURA NON COMPILA: VA RECUPERATA 1

LA FIGURA NON COMPILA: VA RECUPERATA 2

Figura 13.15: Programmazione dinamica

L'algoritmo di Floyd-Warshall (come quello di Dijkstra) è un esempio di programmazione dinamica

- si decompone il problema in sottoproblemi associati a indici ordinati
- si dimostra che la soluzione ottima di un sottoproblema di indice $h+1$ dipende solo dalle soluzioni ottime di quelli di indice $1, \ldots, h$ (principio di ottimalità)
- si costruisce la soluzione ottima partendo da quelle dei sottoproblemi

Questo approccio generale fu proposto da Richard Bellman nel 1940

Naturalmente, funziona solo se il problema gode del principio di ottimalità

Chi notasse eventuali incoerenze o errori, oppure avesse dubbi sul contenuto di queste pagine e dei codici, è pregato di segnalarmeli per contribuire a migliorare la qualità dei materiali del corso.

Problema

Si vuole scrivere un programma `knapsack.c` che aiuti un ladro entrato in una gioielleria ad uscirne col massimo bottino possibile. La gioielleria contiene un insieme $B = \{1, \ldots, n\}$ di n oggetti preziosi, per ognuno dei quali è noto il valore ϕ_i e il volume v_i. Il ladro ha uno zaino di capacità V, e per semplicità si suppone che il volume totale dei preziosi rubati sia sufficiente a indicare se possono essere contenuti nello zaino oppure no (trascurando il problema della loro forma, che evidentemente potrebbe creare zone inutilizzate nello zaino e ridurne la capacità). Alternativamente si può pensare a un escursionista che debba decidere quali oggetti portare con sé in una gita, avendo fissato un peso massimo per il proprio bagaglio (ignorando questa volta forme e volumi degli oggetti). Il programma deve indicare quali oggetti prendere per rispettare il vincolo massimizzando il valore totale del sottoinsieme di oggetti scelti.

Formati di ingresso e uscita Il file di ingresso che contiene l'istanza del problema consiste di quattro righe. La prima fornisce il numero di oggetti n, preceduto dalle parole chiave `n =`. La seconda fornisce i valori ϕ_i degli oggetti, indicati da numeri interi separati da spazi, preceduti dalle parole chiave `phi =`. La terza riga riporta i volumi v_i degli oggetti, indicati da numeri interi separati da spazi, preceduti dalle parole chiave `v =`. Infine, l'ultima riga contiene la capacità V dello zaino, preceduta dalla parola chiave `V =` .

Sono disponibili quattro esempi, rispettivamente con $n = 5$, 10, 24 e 50 oggetti. Il primo corrisponde all'esempio dei lucidi, ed è:

```
n = 6
phi = 7 2 4 5 4 1
v = 5 3 2 3 1 1
V = 8
```

Il programma deve stampare a video il valore dell'ottimo del problema, preceduto dalle parole chiave `f* =`. Dovrebbe anche (ma per semplicità lo faremo solo nell'ultima fase della risoluzione) stampare in una seconda riga la soluzione ottima, rappresentata come elenco degli indici degli oggetti che ne fanno parte, separati da spazi, racchiusi fra parentesi tonde e preceduti dalle parole chiave `x* =`. Il risultato nel caso dell'esempio sopra riportato è:

```
f* = 15
x* = ( 1 3 5 )
```

I quattro file di esempio forniti riportano le due righe con la soluzione dopo le quattro con i dati, sfruttando il fatto che questo non disturba la lettura dei dati stessi.

Traccia della risoluzione

Si può cominciare scorrendo i lucidi che introducono l'Ottimizzazione Combinatoria, l'algoritmo esaustivo, la distinzione tra algoritmi esatti ed euristici e il problema dello zaino, che è ovviamente un buon modello per il problema da risolvere.

L'esercizio parte dal file `knapsack0.c`. Questo file contiene molte cose, per potersi dedicare immediatamente all'aspetto algoritmico del problema. Ovviamente, contiene la consueta funzione `InterpretaLineaComando` per recuperare il nome del file dalla linea di comando e copiarlo nella corrispondente variabile. Contiene direttive di inclusione, costanti simboliche e dichiarazioni di tipo già usate più volte nelle lezioni precedenti. Contiene però anche le strutture e le funzioni necessarie a gestire istanze e soluzioni per un problema di zaino:

- `datiKP` descrive un'istanza attraverso due numeri interi (il numero degli oggetti `n` e la capacità `V`) e due vettori dinamici di numeri interi (il valore `phi` e il volume `v`); sono già disponibili funzioni per caricare un'istanza da file di testo (nel formato sopra descritto) e per distruggerla;
- `soluzioneKP` descrive una soluzione attraverso un numero intero (il valore totale della soluzione `f`) e un vettore di incidenza, cioè un vettore dinamico di valori logici `x`; sono anche disponibili funzioni per creare, distruggere e stampare (nel formato sopra descritto) una soluzione.

Il codice contiene anche variabili e istruzioni per misurare il tempo di esecuzione degli algoritmi che andremo realizzando, dato che faremo alcune osservazioni interessanti su questo tempo. In particolare, la funzione `clock()` assegna ad una variabile di tipo `clock_t` il numero di unità elementari di tempo (tipicamente, millisecondi o microsecondi) trascorse dall'inizio dell'esecuzione. La differenza dei valori registrati al principio e al termine dell'esecuzione fornisce la durata. Per esprimerla in secondi, occorre dividere il numero di unità elementari trascorse per quello delle unità contenute in un secondo, che è fornito dalla costante simbolica `CLOCKS_PER_SEC`. Di conseguenza, si include anche la libreria `time.h`.

Prima fase (`knapsack1.c`) I lucidi propongono un algoritmo ricorsivo che risolve il problema dello zaino in maniera abbastanza simile a una versione ricorsiva della *InsertionSort*[6], vale a dire

1. riducendo il problema con l'eliminazione di un elemento;
2. risolvendo il problema ridotto;
3. riaggiungendo l'elemento eliminato alla soluzione del problema originale.

L'analogia non funziona completamente, perché in generale non è detto che l'elemento eliminato sia reinseribile nella soluzione del problema originale (lo zaino potrebbe essere ormai troppo pieno), e d'altra parte non è neanche detto che l'elemento eliminato si possa ignorare (altrimenti li butteremmo tutti e otterremmo una soluzione vuota, che è la peggiore in assoluto). Dobbiamo quindi complicare la ricorsione ammettendo due possibili casi ricorsivi: quello in cui l'ultimo elemento non trova posto nella soluzione ottimale del sottoproblema e quello in cui vi trova posto. Il caso in cui l'ultimo elemento trova posto nello zaino richiede che la capacità sia manipolata (ridotta) in modo da lasciare il posto necessario. A volte, il primo

[6] Che non è in programma, ma viene descritta nei materiali di approfondimento della lezione 14.

caso è banalmente l'unico possibile, perché lo zaino è di per sé troppo piccolo per contenere l'ultimo elemento, e quindi non è necessario calcolare la soluzione ottima del sottoproblema per sapere che l'elemento va buttato. Riassumendo, abbiamo:

- un caso base, nel quale l'insieme degli oggetti potenziali è vuoto;
- due casi ricorsivi, nei quali l'insieme degli oggetti potenziali non è vuoto e:
  1. lo zaino è troppo piccolo per contenere l'ultimo oggetto dell'insieme, così che questo va scartato, producendo un insieme di oggetti potenziali più piccolo;
  2. lo zaino è abbastanza grande da contenere l'ultimo oggetto dell'insieme, così che questo si può scartare, producendo un insieme di oggetti potenziali più piccolo da inserire nello stesso zaino, oppure scegliere, producendo un insieme di oggetti potenziali più piccolo da inserire in uno zaino parzialmente occupato (che equivale a uno zaino più piccolo).

Nel primo caso ricorsivo, c'è una sola chiamata ricorsiva (come nell'*InsertionSort*); nel secondo, invece, ce ne sono due, e la soluzione ottima dell'intero problema è la migliore fra le soluzioni ottime dei due sottoproblemi.

Fortunatamente, i due sottoproblemi sono entrambi rappresentabili come problemi di zaino:

1. il sottoproblema in cui si rifiuta l'ultimo elemento, è semplicemente uno zaino sui primi $n-1$ elementi con la stessa capacità V: l'insieme B viene ristretto al sottoinsieme $\{1,\ldots,i\}$ ponendo $i = n-1$, e le funzioni ϕ e v vengono banalmente ristrette al nuovo sottoinsieme;
2. il sottoproblema in cui si accetta l'ultimo elemento, è ancora uno zaino sui primi $n-1$ elementi, con la stessa restrizione di B, ϕ e v, ma la capacità dello zaino diventa $V - v[n]$, dato che l'oggetto fissato va ad occupare spazio nello zaino originale (cioè il nuovo zaino è la parte rimasta libera dello zaino originale).

Quando $v_n > V$, il secondo sottoproblema è impossibile, e quindi c'è solo il primo.

Nella prima fase dell'esercizio, realizziamo l'algoritmo ricorsivo descritto più sopra, che è molto semplice ed elegante. Va osservato che, siccome l'algoritmo nelle chiamate ricorsive opera su sottoproblemi del problema originale, bisogna trovare il modo per passare tali sottoproblemi alla procedura stessa. Ci sono diversi modi per farlo:

1. ricostruire i dati di ogni sottoproblema copiando, ed eventualmente adattando, quelli del problema originale (questo è molto inefficiente sia rispetto allo spazio sia rispetto al tempo);
2. modificare i dati del sottoproblema corrente e passarli alla procedura ricorsiva; nel caso specifico, si tratta di decrementare `n` ed eventualmente ridurre `V` (i vettori possono rimanere invariati, dato che il decremento di `n` nasconde automaticamente l'elemento scartato); se i dati sono passati per indirizzo, però, bisogna ricordarsi di ripristinarli subito dopo la chiamata, dato che le chiamate annidate ai livelli inferiori introdurranno ulteriori modifiche (per esempio, decrementando `n` fino a zero);
3. passare alla procedura ricorsiva copie modificate dei dati, in modo da non alterare gli originali; per far questo, basta sfruttare il passaggio dei parametri

per copia, cioè non fornire alla procedura l'intera istanza, ma solo i campi `n`, `phi`, `v` e `V`, modificando `n` e `V` così che ogni singola chiamata abbia i propri valori indipendenti[7].

Scegliamo l'ultima strada, che produce il codice più semplice e vicino all'equazione ricorrente, anche se forse la seconda usa qualche operazione in meno. Per semplicità, chiediamo che l'algoritmo restituisca solo il valore della soluzione, e non l'intero risultato[8].

Seconda fase (`knapsack2.c`) L'algoritmo ricorsivo risolve molto in fretta i primi tre problemi di esempio, mentre il quarto non accenna a terminare nemmeno dopo svariati minuti. Il motivo è che si tratta di un algoritmo molto inefficiente: nell'albero delle chiamate ricorsive, infatti, ogni nodo corrisponde a un sottoproblema in cui gli oggetti sono accettati, scartati oppure ancora non presi in considerazione. Le foglie, in particolare, corrispondono ad accettare o scartare tutti gli oggetti, e quindi a sottoinsiemi completamente determinati. Nel caso pessimo (quando il test sulla capacità non interviene mai), le foglie sono 2^n e i nodi complessivi circa il doppio. I primi tre problemi di esempio hanno, rispettivamente, $n = 6$, $n = 10$ e $n = 24$ elementi, e quindi potenzialmente $2^6 = 64$, $2^{10} = 1024$ e $2^{24} = 16\,777\,216$ foglie, che si possono tranquillamente scorrere in frazioni di secondo. Il quarto esempio, invece, ha $n = 50$ oggetti, e quindi $2^{50} \approx 1.126 \cdot 10^{15}$ foglie: anche ipotizzando di esplorare un miliardo di foglie al secondo, occorrerebbero circa 13 giorni per visitarle tutte.

Ora osserviamo che i sottoproblemi sono in effetti individuati semplicemente dai due numeri naturali `n` e `V`, per cui ci sono al massimo $(n+1)(V+1)$ sottoproblemi, e non 2^n. In pratica, molti dei sottoproblemi generati e risolti dall'algoritmo ricorsivo si ripetono, e quindi vengono risolti ripetutamente sprecando tempo. A seconda dei valori di `n` e `V`, può prevalere una o l'altra stima, ma la differenza può essere enorme. La soluzione è, ogni volta che si risolve un sottoproblema, salvarne il risultato in un'apposita struttura. Questa struttura consenta, di fronte a un altro sottoproblema, di verificare se sia nuovo oppure già affrontato, e nel secondo caso di recuperare la soluzione senza doverla ricalcolare. Dove conservare le soluzioni dei sottoproblemi? Essendo questi indicizzati su due numeri naturali, viene spontaneo conservarli in una matrice bidimensionale, con indici `i` e `v`, rispettivamente estesi da `0` a `n` e da `0` a `V`. Ogni sottoproblema, con la relativa soluzione, sarà associato ad una cella. Dobbiamo quindi definire una matrice, allocarla e inizializzarla con un valore che indichi il fatto che il sottoproblema non è ancora stato risolto (un valore intero negativo può andare bene, dato che ogni valore non negativo è potenzialmente una soluzione sensata). Quindi incapsuleremo la procedura ricorsiva realizzata nella prima fase in una procedura `PDricorsivaKP` che alloca e inizializza la matrice, chiama la procedura ricorsiva e infine dealloca la matrice. Inoltre, modificheremo la procedura ricorsiva affinché sfrutti la matrice e la aggiorni, cioè:

- aggiungeremo un altro caso base, corrispondente alle situazioni in cui il sottoproblema corrente risulta già risolto, cioè la matrice contiene un valore non negativo: in tale caso, restituiremo il valore contenuto nella matrice (in tempo costante);
- modificheremo i casi ricorsivi in modo che, una volta calcolato il risultato, prima di restituirlo lo salvino anche nella cella appropriata della matrice.

[7] I due vettori non subiscono modifiche, e quindi si possono passare impunemente senza farne copia.

[8] Ottenere il risultato comporta di passare alla procedura ricorsiva le strutture per contenerlo, e aggiornarle nel modo corretto. È un buon esercizio, ma inutilmente complicato per un algoritmo che poi abbandoneremo in quanto inefficiente.

Applicando questo algoritmo agli esempi, osserviamo che l'istanza con $n = 50$ oggetti viene ora risolta molto in fretta. Il motivo è semplice: la nuova procedura risolve al massimo `(n+1)(V+1)` sottoproblemi, cioè nel caso specifico $51 \cot 421 = 21\,471$, che è nettamente meno di $2^n \approx 10^{15}$. D'altra parte, il terzo esempio risulta invece rallentato. Infatti, in esso $n = 24$ e $V = 6\,404\,180$, per cui $2^n = 2^{24} = 16$ milioni, mentre $(n + 1)(V + 1) \approx 160$ milioni.

Terza fase (`knapsack3.c`) Il modo più comune di realizzare l'algoritmo di programmazione dinamica per il problema dello zaino non è tuttavia quello appena descritto (che viene definito *top-down*), ma un algoritmo iterativo, definito di programmazione dinamica *bottom-up*. L'idea di fondo è che generare i sottoproblemi in modo ricorsivo sia una spesa inutile, dato che i sottoproblemi, in fin dei conti, consistono semplicemente in coppie di numeri non negativi (i, v), e mantenere un'architettura di allocazioni e deallocazioni sullo *stack* di sistema solo per scorrere queste coppie appare sproporzionato. L'idea è di procedere invece sistematicamente, risolvendo tutti i sottoproblemi possibili per $i \leq n$ e $v \leq V$. Ogni sottoproblema (i, v) si può risolvere in tempo costante a patto di disporre delle soluzioni ottime dei sottoproblemi con $i - 1$ oggetti: basta applicare l'equazione ricorrente. Dunque, basta un tempo costante per estendere la soluzione dai problemi più piccoli a quelli immediatamente più grandi. L'idea è "risolvere" tutti i problemi di dimensione $i = 0$, salvare i risultati (tutti nulli) nella matrice che abbiamo introdotto più sopra, e procedere a riempirla salendo progressivamente da $i = 0$ a $i = n$, sfruttando in ogni riga i il contenuto delle celle della riga precedente $i - 1$. L'estensione avviene con un banale costrutto di selezione che ripete (*mutatis mutandis*) i casi dell'equazione ricorrente e dell'algoritmo ricorsivo. Rispetto alla programmazione dinamica *top-down*, questo algoritmo risolve un numero maggiore di sottoproblemi (li risolve tutti, anziché solo quelli davvero ottenibili sviluppando l'albero delle chiamate ricorsive), ma evita tutta la fase di costruzione dei sottoproblemi. Potrebbe quindi essere più o meno efficiente, secondo che il numero dei sottoproblemi effettivi sia più o meno vicino a $(n + 1)(V + 1)$[9]

In questa variante, è abbastanza semplice ricostruire a posteriori la soluzione ottima, oltre a determinarne il valore. Anzi tutto, l'ottimo del problema corrisponde al contenuto della cella (n, V) della matrice, dato che il problema complessivo ha n oggetti e capacità V. Partendo da quella cella, è possibile ricostruire a ritroso il percorso seguito per raggiungerla fino ad arrivare alla cella origine $(0, 0)$. Infatti, ogni cella (i, v) può essere raggiunta solo da due possibili celle, entrambe al livello immediatamente precedente: $(i - 1, v)$ e $(i - 1, v - v_i)$. Se si è arrivati dalla prima, l'elemento i non è stato aggiunto alla soluzione parziale, e quindi il valore della soluzione deve essere lo stesso; se si è arrivati dalla seconda, invece, l'elemento i è stato aggiunto, e quindi il valore della soluzione deve essere cresciuto. L'idea è confrontare i valori conservati nelle due potenziali celle di partenza con quello della cella di arrivo per dedurre da dove si sia effettivamente partiti. Questo determina il valore della variabile x_i. Può capitare che le due potenziali celle di partenza siano entrambe compatibili con quella di arrivo. Questo succede quando il problema ammette diverse soluzioni ottime equivalenti. In tal caso, sceglieremo la soluzione con $x_i =$ `false`, ma questo è del tutto arbitrario (corrisponde a preferire le soluzioni che non usano gli elementi di indice maggiore).

[9] In realtà è anche possibile costruire solo i sottoproblemi utili, ma questo comporta una struttura dati leggermente più complicata, che consenta di saltare le celle inutili della matrice.

Il cammino minimo da tutti i nodi a tutti i nodi

La parte conclusiva della lezione è dedicata all'algoritmo di Floyd-Warshall per il calcolo di tutti i cammini minimi in un dato grafo orientato, cioè del cammino minimo da ogni nodo origine a ogni nodo destinazione del grafo. L'esercizio è puramente algoritmico, e l'algoritmo è molto semplice da realizzare, per cui non aggiungiamo commenti qui.

Capitolo 14

Algoritmo greedy

Questo capitolo è dedicato a una metodologia generale per progettare algoritmi per risolvere problemi di Ottimizzazione Combinatoria. È quindi uno sviluppo naturale del precedente. Si può parlare di *algoritmo greedy* al singolare, ovvero di algoritmi *greedy* al plurale secondo quanto si restringe l'intepretazione: come un ben preciso modo di ottenere soluzioni per questa famiglia di problemi, un algoritmo generale che funziona per qualsiasi problema di questa categoria, oppure come tutta la famiglia di varianti nelle quali lo schema di fondo viene di volta in volta riadattato e reso più flessibile sulla base delle osservazioni empiriche delle prestazioni che l'algoritmo fondamentale ha.

PARAGRAFO DA SPOSTARE: La programmazione dinamica è un metodo per progettare algoritmi, ma non necessariamente solo per problemi di Ottimizzazione Combinatoria. Noi abbiamo visto solo un esempio di questo genere. Le lezioni di teoria presentano esempi ancora di algoritmi di ottimizzazione (per il problema del cammino minimo), ma anche di algoritmi per problemi di altra natura. Siete caldamente invitati a implementare questi algoritmi, perché si suppone che a questo punto siate in grado di camminare sulle vostre gambe e realizzare algoritmi diversi da quelli visti in laboratorio, in preparazione sia del progetto d'esame sia di eventuali future applicazioni sul lavoro.

Nei problemi di Ottimizzazione Combinatoria

- una soluzione x è un sottoinsieme di un dato insieme base B finito
- le soluzioni appartengono a una famiglia $X \subseteq 2^B$ di sottoinsiemi che soddisfano opportune condizioni
- la funzione obiettivo $f : X \to \mathbb{N}$ dà un valore a ogni soluzione

Si tratta di trovare una soluzione di valore minimo o massimo in X

Algoritmo esaustivo: trova una soluzione ottima ma è esponenziale

- alcuni problemi di OC ammettono algoritmi polinomiali esatti
- tutti i problemi di OC ammettono algoritmi polinomiali euristici, cioè che non garantiscono di trovare l'ottimo su ogni istanza

Figura 14.1: Ottimizzazione Combinatoria

Ricordiamo che i problemi di Ottimizzazione Combinatoria sono quelli nei quali esiste un insieme base B finito e una famiglia X di sottoinsiemi di B, che è una parte dell'insieme 2^B delle parti di B. I sottoinsiemi di X si chiamano soluzioni, o soluzioni ammissibili, e rappresentano tutte le soluzioni al problema pratico che viene descritto attraverso il modello. A ciascuna di queste soluzioni una funzione obiettivo f assegna un valore intero non negativo. Lo scopo del problema di Ottimizzazione Combinatoria è determinare una soluzione che abbia valore minimo (oppure valore massimo) fra quelle dell'insieme X. Esiste sempre un algoritmo esaustivo, che scorre uno per uno tutti i sottoinsiemi di B, valuta se sono soluzioni ammissibili e, in caso positivo, valuta la funzione obiettivo, conservando a parte il valore e la soluzione migliori trovati via via. Quindi è possibile risolvere banalmente tutti i problemi di Ottimizzazione Combinatoria per enumerazione, ma è assurdo farlo appena la dimensione dell'insieme base supera un valore abbastanza ridotto. Questo non dipende molto dalla tecnologia, come si è visto in una delle prime lezioni. Gli algoritmi esponenziali esauriscono le risorse della tecnologia molto presto e qualunque miglioramento da quel punto di vista serve a poco in pratica.

Alcuni problemi di Ottimizzazione Combinatoria ammettono algoritmi esatti. L'esempio che abbiamo visto in realtà non era rigorosamente polinomiale, a pseudopolinomiale. Però esistono anche algoritmi polinomiali esatti e sono trattati nelle dispense di teoria, in particolare per problemi su grafo, come quelli del cammino minimo e dell'albero minimo. Ci sono però anche problemi per i quali non sono noti algoritmi esatti polinomiali, e in base a una teoria a cui ho alluso nella scorsa lezione si congettura che algoritmi polinomiali esatti non possano esistere. Esistono allora algoritmi polinomiali euristici, cioè che cercano una soluzione che sia ammissibile, anche se non necessariamente di qualità ottima.

La programmazione dinamica è un tentativo di andare a passare in rassegna tutti i sottoinsiemi dell'insieme base, e in particolare tutte le soluzioni, non in modo esplicito, ma a gruppi[1]. Affrontando il problema dello zaino, si è costruita una matrice in cui ogni cella corrisponde a un sottoproblema (i, v), in cui si considerano solo i primi i oggetti di partenza e solo una parte di capacità v dello zaino dato. In ogni cella (i, v) l'algoritmo scrive un valore che è la soluzione ottima del sottoproblema. È possibile interpretare la cosa anche in un altro modo, cioè associando ad ogni cella quelle soluzioni del problema originario (con tutti gli n elementi e con lo zaino originale di capacità V) che godono di questa proprietà specifica: che fra i primi i elementi quelli contenuti nella soluzione abbiano volume non superiore a v. È come dire che con quegli elementi si può risolvere un problema di zaino con capacità v avendo a disposizione solo i primi i elementi. Il valore scritto nella cella è sia il valore della migliore soluzione del sottoproblema sia il valore parziale di quel pezzo della soluzione ottima del problema completo che comprende solo i primi elementi. Il vantaggio del metodo è quindi di considerare le soluzioni non una per una, ma a blocchi, e in ogni blocco considerare solo la migliore.

L'idea ora è cercare di ridurre l'insieme dei sottoproblemi, fino a ottenerne un numero molto ridotto. Non è sempre possibile, anzi di solito non lo è. Però è possibile indagare se esistano problemi interessanti nei quali si riesce a ridurre i sottoproblemi al punto che, invece di esplorare un'intera matrice, si esplora un semplice vettore monodimensionale di problemi. Tale vettore corrisponderebbe via via ad allargare il problema di un elemento alla volta. Se avessimo avuto solo i, anziché i e v, avremmo potuto "allargare" la soluzione ottima di passo in passo, fino a trovare la soluzione ottima per tutti gli n elementi. Per il problema dello zaino questo non è possibile, per altri problemi lo è. C'è quindi un filone di studi dedicato alla ricerca delle condizioni che consentono di risolvere un problema in

[1] Controllare che la cosa fosse esplicita nel capitolo precedente

questo modo. Si tratta ovviamente di un modo molto efficiente, perché il numero di stati da esplorare è decisamente ridotto. Questo è uno dei modi di vedere l'algoritmo *greedy*: funziona quando si riesce a costruire una programmazione dinamica che ha questa proprietà.

Un altro modo di vedere tutto questo è abbandonare l'idea di esplorare tutte le soluzioni, cioè puntare in partenza a costruire un sottoinsieme nel modo più semplice possibile: partire con un sottoinsieme vuoto e allargarlo un passo alla volta, scegliendo un elemento che paia particolarmente promettente e aggiungendolo al sottoinsieme corrente finché si decide di fermarsi. Questa è una modalità di ragionamento "euristica", che però a volte si rivela esatta. Dà luogo all'algoritmo *greedy*, cioè *ingordo*, proprio perché il sottoinsieme corrente "mangia" ogni volta l'elemento più promettente. Sotto opportune condizioni, questo algoritmo fornisce una soluzione ottima, negli altri casi no. Applicandolo al problema dello zaino, scopriremo che non funziona in generale, ma funziona in circostanze particolari, e questo ci indirizzerà allo studio delle condizioni di ottimalità.

Procederemo in modo informale, rimandando alle dispense di teoria tutte le condizioni e proprietà teoriche. In particolare, verrà trattato un teorema che fornisce le condizioni sulla base delle quali l'algoritmo *greedy* risolve all'ottimo un problema di Ottimizzazione Combinatoria. Un algoritmo *greedy* per noi sarà un oggetto che segue lo schema che andremo a dettagliare. Tutta la notazione servirà solo per chiarezza, ed evitare ambiguità, non per dimostrare teoremi.

Un algoritmo greedy A aggiorna ad ogni passo t un sottoinsieme $x^{(t)}$:

1. in $t = 0$ parte da un sottoinsieme vuoto: $x^{(0)} = \emptyset$
   (*perché ovviamente è parte di una soluzione ottima*)
2. termina se nessun sottoinsieme più grande può essere ottimo:

   $$x \cup \{i\} \notin \mathcal{F}_A \text{ per ogni } i \in B \setminus x$$

   $\mathcal{F}_A$ raccoglie i potenziali sottoinsiemi di soluzioni ottime
   (*potenziali, non sempre sicuri*)
3. fra gli elementi $i \in B \setminus x$ tali che $x \cup \{i\} \in \mathcal{F}_A$
   sceglie l'elemento $i^{(t)}$ che ottimizza un criterio $\phi_A(i, x)$
   (*tiene x "ammissibile" e cerca di tenerlo "ottimo"*)
4. aggiunge $i^{(t)}$ al sottoinsieme corrente $x^{(t)}$: $x^{(t+1)} := x^{(t)} \cup \left\{i^{(t)}\right\}$
   (*non si torna più indietro nella scelta!*)
5. torna al punto 2

Per alcuni problemi, trova soluzioni ottime; per altri no

Figura 14.2: Gli algoritmi greedy

Si parte con un sottoinsieme $x^{(0)}$ vuoto. Perché? Molto semplice: volendo raggiungere una soluzione ottima, che è un sottoinsieme dell'insieme base, il sottoinsieme vuoto ne è sicuramente una parte, qualunque sia la soluzione ottima. Quindi, partire da un insieme vuoto non è un errore. Se partissimo da un altro insieme, potremmo avere già sbagliato dall'inizio.

Nel generico passo intermedio, verifichiamo se sia possibile allargare il sottoin-

sieme corrente x. Questo significa che il nostro sottoinsieme x, unito a un elemento i esterno (cioè in $B \setminus x$), deve ancora far parte di uno spazio indicato con $\mathcal{F}_A$, e detto *spazio di ricerca* ($\mathcal{F}$ sta per l'inglese "find", cioè "trovare", o per "feasible", cioè "ammissibile"). Questo spazio di ricerca è proprio dell'algoritmo che stiamo esplorando (da cui il pedice A) e serve a guidarlo in maniera tale da non abbandonare del tutto le speranze di arrivare una soluzione ottima.

Per concretizzare, consideriamo il problema dello dello zaino. Ha senso partire da uno zaino vuoto? Assolutamente sì. Quando smette di aver senso aggiungere un elemento i a una soluzione parziale corrente x? La risposta più naturale è che quando l'elemento da aggiungere fa sforare la capacità dello zaino non ha senso aggiungerlo, perché si esce dall'insieme delle soluzioni ammissibili. Siccome l'algoritmo aggiunge elementi, ma non ne toglie mai, l'uscita dallo spazio di ricerca è definitiva: aggiungere quell'elemento proibisce di arrivare a una soluzione ottima perché condanna x a restare inammissibile per tutto il resto dell'esecuzione. In generale, alcune aggiunte hanno senso, altre non ne hanno. Lo spazio di ricerca, cioè la collezione $\mathcal{F}_A$, indica esattamente questo: raccoglie tutti i potenziali sottoinsiemi di soluzioni ottime, tutti quelli che hanno qualche probabilità di stare all'interno di una soluzione ottima, e quindi di non farci sbagliare.

Perché diciamo "potenziali"? Perché se avessimo la garanzia che x rimanga sottoinsieme di una soluzione ottima, a furia di allargare x, prima o poi, otterremmo una soluzione ottima, e questo non è sempre il caso. Di solito, si riesce a garantire l'ammissibilità del risultato finale, ma al solito questo dipende dal problema.

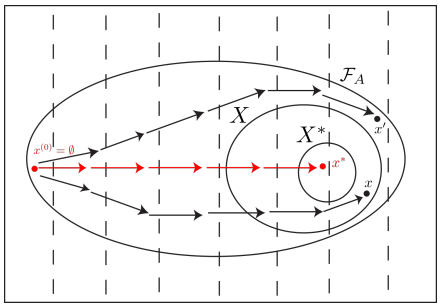
In generale, ci sarà più di un elemento appartenente a $B \setminus X$ che aggiunto a x consenta di rimanere nello spazio di ricerca. Fra questi, l'algoritmo deve scegliere in base a qualche criterio. Esisterà quindi una funzione $\phi_A(i, x)$, che è propria dell'algoritmo A considerato e dipende dall'elemento i che si valuta se aggiungere, ma anche dalla soluzione x da cui si parte. Questo criterio deve stimare la qualità dell'aggiunta di i a x. Come si misura la qualit'à di un'aggiunta? L'idea è che deve tenere la soluzione x il più possibile vicina all' ottimo, e possibilmente anche lontana dal diventare inammissibile.

Scelto uno degli elementi, quello che ottimizza il criterio dato, aggiungeremo l'elemento al sottoinsieme corrente e non torneremo mai più indietro su questa scelta. L'algoritmo *greedy* è un algoritmo costruttivo, che non distrugge mai quello che ha fatto. Questo lo rende poco flessibile, molto rigido.

L'algoritmo termina quando non è più possibile allargare il sottoinsieme corrente, perché qualsiasi elemento aggiuntivo fa uscire dallo spazio di ricerca (o, eventualmente, perché non ci sono più elementi esterni). A questo punto, si è ottenuta, auspicabilmente, una soluzione ammissibile, che va valutata.

Possono succedere principalmente tre cose, rappresentate nelle Figure 14.3, 14.4 e 14.5. In queste figure, il rettangolo esterno rappresenta l'insieme delle parti 2^B, diviso da tratteggi verticali in livelli che raccolgono i sottoinsiemi di pari cardinalità. Il primo livello a sinistra contiene solo il sottoinsieme vuoto, che ha zero elementi. Poi c'è l'insieme dei singoletti, che hanno un elemento, poi le coppie, le terne, via via fino ad arrivare in fondo a destra all'unico sottoinsieme che contiene tutti gli $n = |B|$) elementi. L'insieme delle parti 2^B contiene lo spazio di ricerca $\mathcal{F}_A$, che contiene tutti i sottoinsiemi che per qualche motivo consideriamo degni di nota, quelli che sono potenziali sottoinsiemi di soluzioni ottime. A sua volta, $\mathcal{F}_A$ contiene la regione ammissibile X. Nel caso dello zaino, X contiene tutti i sottoinsiemi di volume non superiore alla capacità, quindi anche l'insieme vuoto. In altri problemi, non è così. Per esempio, dato un grafo orientato, un nodo origine s e un nodo destinazione t, il problema del cammino minimo da s a t chiede di trovare un cammino, composto da archi del grafo, che parta da s e arrivi a t, e fra quelli che

soddisfano queste condizioni abbia costo totale minimo. La regione ammissibile X contiene i sottoinsiemi di archi che costituiscono cammini da s a t. L'insieme vuoto non è ammissibile, salvo che s coincida esattamente con t. Quale potrebbe essere lo spazio di ricerca $\mathcal{F}_A$? Deve considerare potenziali sottoinsiemi di cammini da s a t. Per esempio, potrebbero essere i cammini che partono da s e vanno in nodi qualsiasi (ma potrebbero essere anche insiemi più o meno generali: l'indice A sta proprio a specificare che spazi di ricerca diversi determinano algoritmi diversi). L'insieme delle soluzioni ammissibili X contiene l'insieme delle soluzioni ottime X^*. Lo scopo del problema è raggiungere una di queste soluzioni. Fra i vari sottoinsiemi esistono anche dei collegamenti, rappresentati dal fatto che è possibile aggiungere ad ogni sottoinsieme un elemento che ancora non vi appartiene, a patto di non uscire dallo spazio di ricerca. Si ottiene quindi un grafo, i cui nodi sono i sottoinsiemi di B, mentre gli archi collegano ogni sottoinsieme di $\mathcal{F}_A$ a ogni altro sottoinsieme di $\mathcal{F}_A$ che ha un elemento aggiuntivo. Un algoritmo *greedy*, quindi, corrisponde a spostarsi in questo grafo dall'insieme vuoto a sinistra, muovendosi di un livello per volta verso destra, restando nello spazio di ricerca, finché si arriva a un sottoinsieme che non ha archi uscenti. Dunque, un cammino lungo il grafo. Le tre figure mostrano tre diversi possibili cammini.



L'algoritmo visita una catena di sottoinsiemi $\emptyset = x^{(0)} \subset \ldots \subset x^{(k)}$

Può terminare

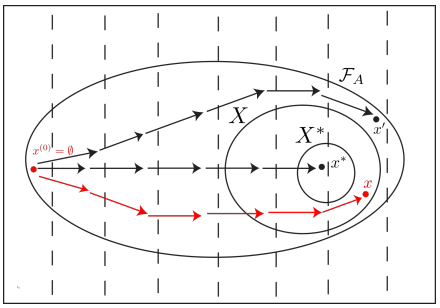
- in una soluzione ottima $x^* \in X^*$
- in una soluzione ammissibile non ottima $x \in X$
- in un sottoinsieme non ammissibile x'

Esempi: *albero ricoprente minimo, zaino unitario*

Figura 14.3: Un grafico

Se siamo molto fortunati (Figura 14.3), la struttura del problema permette di partire dall'insieme vuoto e muoversi allargando il sottoinsieme corrente fino a terminare in una soluzione ottima, cioè a centrare l'insieme X^*. È il caso del problema dell'albero ricoprente minimo (fra l'altro con due diverse definizioni di spazio di ricerca, e quindi due diversi algoritmi *greedy*).

In altri casi (Figura 14.4), siamo meno fortunati: partiamo dall'insieme vuoto e ci muoviamo arrivando a una soluzione certamente ammissibile, ma non certamente ottima. In questo caso, l'algoritmo *greedy* non sarà esatto, ma sarà euristico. È il caso del problema dello zaino, nel quale, in particolare, non solo uno, ma addirittura



L'algoritmo visita una catena di sottoinsiemi $\emptyset = x^{(0)} \subset \ldots \subset x^{(k)}$

Può terminare

- in una soluzione ottima $x^* \in X^*$
- in una soluzione ammissibile non ottima $x \in X$
- in un sottoinsieme non ammissibile x'

Esempio: zaino generico

Figura 14.4: Un grafico

tutti i sottoinsiemi attraversati sono ammissibili, e l'ultimo è il migliore.

Il caso più sfortunato è quello nel quale il problema è talmente complicato che, aggiungendo elemento dopo elemento, a un certo punto l'algoritmo si ferma perché diventa certo che tutti i sottoinsiemi successivi sono inammissibili (e quindi non portano a soluzioni ottime), ma in realtà anche i sottoinsiemi visitati sono rimasti sempre fuori da X, e l'algoritmo ha completamente fallito nel suo compito. Di questi casi non ci occuperemo in questa questa lezione.

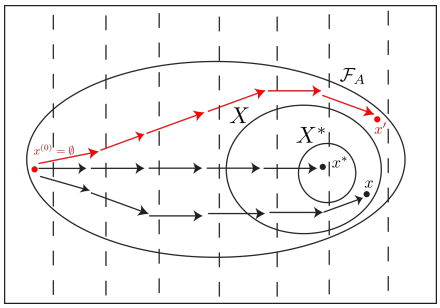
Ora semplifichiamo brutalmente, e passiamo dagli algoritmi *greedy* in genere a quello che viene detto l'algoritmo *greedy* (al singolare) nelle lezioni di teoria, e per il quale valgono le proprietà teoriche più interessanti. Questo comporta di aggiungere un'ipotesi, molto comune nelle applicazioni pratiche, sulla funzione obiettivo: che essa sia *additiva*.

Definizione 11 *Una funzione obiettivo $f : X \to \mathbb{N}$ si definisce* additiva *quando esiste una funzione ausiliaria $\phi : B \to \mathbb{N}$ tale che il valore dell'obiettivo per una soluzione x è pari alla somma dei valori della funzione ausiliaria sui singoli elementi della soluzione:*

$$f(x) = \sum_{i \in x} \phi_i$$

Un esempio già visto è il problema dello zaino: il valore della funzione obiettivo per una soluzione x è la somma dei valori ϕ_i dei singoli oggetti i che compongono la soluzione x. Sembra una cosa ovvia, ma vale solo per i problemi in cui la funzione obiettivo è additiva.

Richiediamo anche che la funzione ausiliaria sia non negativa, cioè che via via che si aggiungono elementi il valore della soluzione vada crescendo o, quanto meno, non decresca.



L'algoritmo visita una catena di sottoinsiemi $\emptyset = x^{(0)} \subset \ldots \subset x^{(k)}$

Può terminare

- in una soluzione ottima $x^* \in X^*$
- in una soluzione ammissibile non ottima $x \in X$
- in un sottoinsieme non ammissibile x'

Figura 14.5: Un grafico

L'algoritmo *greedy* più semplice è quello in cui

- la funzione obiettivo è additiva: $f(x) = \sum_{i \in x} \phi_i$
  e non negativa: $\phi_i \geq 0$ per ogni $i \in B$
- si sceglie l'elemento ammissibile che produce il sottoinsieme migliore

  $$i^* = \arg \max_{i \in B \setminus x : x \cup \{i\} \in \mathcal{F}_A} f(x \cup \{i\}) = \arg \max_{i \in B \setminus x : x \cup \{i\} \in \mathcal{F}_A} [f(x) + \phi_i]$$

  cioè quello di valore massimo: $i^* = \arg \max_{i \in B \setminus x : x \cup \{i\} \in \mathcal{F}_A} \phi_i$

Algorithm Greedy(I)
$x := \emptyset$;
While $\exists i \in B \setminus x : x \cup \{i\} \in \mathcal{F}_A$ *do*
    $i^* := \arg \max_{i \in B \setminus x : x \cup \{i\} \in \mathcal{F}_A} \phi_i$;
    $x := x \cup \{i^*\}$;
Return x; { La soluzione migliore visitata è l'ultima }

Figura 14.6: L'algoritmo greedy base

Se consideriamo l'algoritmo *greedy* che usa come criterio di scelta dell'elemento da aggiungere il valore della funzione obiettivo nel sottoinsieme allargato, che è la cosa più spontanea che possa venire in mente di fare (anche se non necessariamente la migliore), è facile vedere che, grazie alla sua additività, massimizzare $f(x \cup \{i\})$ corrisponde a massimizzare il valore iniziale $f(x)$ più quello aggiunto dal nuovo elemento, cioè ϕ_i. Siccome il primo non dipende dalla scelta di i, possiamo ignorarlo e massimizzare la funzione ϕ_i, cioè banalmente aggiungere ad ogni passo l'elemento di valore massimo.

Nel caso dello zaino, questo significa che si parte da un insieme vuoto e, fin tanto che esistono ancora elementi non appartenenti a x e tali che la loro aggiunta a x non ecceda la capacità dello zaino (abbiamo già visto che rimanere nello spazio di ricerca significa questo), sceglieremo l'elemento che ha valore intrinseco massimo e lo aggiungeremo a x. Quando tutti gli elementi esterni superano la capacità residua dello zaino, non è più possibile aggiungere elementi e l'algoritmo termina, restituendo l'ultimo sottoinsieme generato, che è anche il migliore. È un metodo molto spontaneo, ma anche molto miope, perché ad ogni passo guarda solo le scelte ottimali in quel momento. Si può sperare che il risultato finale vada bene.

Figura 14.7: Il problema dello zaino unitario
Si vuole scegliere da un insieme di oggetti di pari volume un sottoinsieme di valore massimo che possa stare in uno zaino di capacità limitata

In questo caso speciale del *KP* il vincolo di volume diventa di cardinalità

$\mathcal{F}_A$ coincide con la regione ammissibile $X = \{x \subseteq B : |x| \leq \lfloor V/v \rfloor\}$

Algorithm GreedyUKP(I)
$x := \emptyset$;
While $|x| < \lfloor V/v \rfloor$ *do* { si pone $\mathcal{F}_A = X$ }
    $i := \arg\max_{i \in B \setminus x} \phi_i$;
    $x := x \cup \{i\}$;
Return x;

Lo pseudocodice è semplificato dal fatto che x è estendibile

- per ogni x di cardinalità $|x| < \lfloor V/v \rfloor$
- aggiungendo qualsiasi elemento $i \in B \setminus x$

Consideriamo per un momento un problema dello zaino con ipotesi restrittive, cioè ipotizziamo che tutti gli oggetti abbiano lo stesso volume v. Sotto questa ipotesi, il vincolo di capacità diventa un vincolo di cardinalità: si può infatti definire il volume comune a tutti gli oggetti come unità di misura e usarla per misurare la capacità dello zaino. Per esempio, se lo zaino è in grado di contenere 4.5 oggetti, significa che ce ne stanno 4. Il vincolo sul volume massimo si traduce in un vincolo sul numero massimo di oggetti. Questo numero è il rapporto fra la capacità dello zaino e il volume di ogni singolo oggetto, arrotondato all'intero immediatamente inferiore. Abbiamo già visto che per il problema dello zaino è abbastanza naturale definire lo spazio di ricerca come l'insieme delle soluzioni ammissibili, perché finché c'è capacità residua, c'è speranza di allargare il sottoinsieme corrente, mentre quando non ce n'è, non c'è più alcun modo di ottenere soluzioni ammissibili.

Figura 14.8: Esempio: il problema dello zaino unitario

B	a	b	c	d	e	f
ϕ	7	2	4	5	4	1

$v_i = 1$ per ogni $i \in B$

$V = 4$

L'algoritmo esegue i seguenti passi:

1. $x := \emptyset$;
2. poiché $|x| = 0 < 4$, valuta $i := a$ e aggiorna $x := \{a\}$;
3. poiché $|x| = 1 < 4$, valuta $i := d$ e aggiorna $x := \{a, d\}$;
4. poiché $|x| = 2 < 4$, valuta $i := c$ e aggiorna $x := \{a, c, d\}$;
5. poiché $|x| = 3 < 4$, valuta $i := e$ e aggiorna $x := \{a, c, d, e\}$;
6. poiché $|x| = 4 \not< 4$, termina

Questo algoritmo trova sempre la soluzione ottima

Ma perché?

L'algoritmo *greedy*, quindi, parte dall'insieme vuoto e, finché la cardinalità attuale è strettamente inferiore a quella massima, cerca l'elemento esterno di valore massimo. Tutti gli elementi esterni sono accettabili perché hanno lo stesso volume, quindi sono tutti ammissibili o tutti inammissibili. L'elemento di valore massimo viene aggiunto alla soluzione corrente. È un algoritmo estremamente semplice.

Funziona questo algoritmo? È abbastanza intuitivo che funzioni. Diciamo che uno zaino può contenere 4 oggetti, e ce ne sono disponibili 6, con i valori indicati nella Figura **??**

24:00

7245 4 e uno è ragionevole prendere quello che vale di più Poi il secondo o il terzo il quarto è questo punto ci fermiamo e chiunque direbbe È ovvio che questa è la soluzione ottima però ovviamente in 0:24:10 Poi il secondo o il terzo poi quanto è questo punto ci fermiamo e chiunque direbbe È ovvio che questa è la soluzione ottima però ovviamente in matematica bisogna in qualche maniera dimostrarlo la cosa interessante e che il teorema Ti vado che vi verrà mostrato enunciato è dimostrato nella lezione di teoria da una motivazione solida del fatto che questo meccanismo funzioni non solo in questo banale problema ma in tutta la serie di problemi che hanno delle caratteristiche comuni con questo è questa è la grande grande interesse della teoria che c'è dietro algoritmo greedy a questo punto potremmo organizzarci e dire Benissimo Prendiamo il problema dello zaino generico oggetti di volume vario e diciamo che a questo punto un sottoinsieme ICS è allargata Bill estendibile con un elemento i quando Beh quando il volume totale e qui ho corretto il volume totale della attuale soluzione ICS più il volume del nuovo oggetto che stiamo aggiungendo non supera la capacità Beh allora in quel caso i Pooh e può essere effettivamente aggiunto a DX quindi l'algoritmo specializzato in questo modo diventa parto da un

insieme vuoto finché esistono oggetti esterni tali che il loro volume più quello attuale non supera la capacità oppure il loro volume minore uguale alla capacità residua dello zaino si va a prendere l'elemento che ha valore massimo fra quelli che godono di questa proprietà e lo si aggiunge e questo è l'idea del algoritmo greedy applicato il problema dello zaino possiamo provare ad applicarlo Alla distanza che già ben conosciamo e vediamo che si parte si prende l'elemento di valore massimo che ha che occupa 5 come volume poi andiamo a prendere il secondo elemento che è di che ha un premio pari a 5 e il a questo punto 5 + 3 fa 8 e non ci sono più elementi che abbiano volume minore uguale al valore residuo perché valore residuo la capacità residua e 8 di capacità meno 5 più 3 di volume cumulato fa 0 e non ci sono elementi che abbiano volume minore uguale a zero solo che questa soluzione arriva alle 12 e noi già sappiamo dalla lezione scorsa che la soluzione ottima di questa istanza Vale 15 ed è la soluzione AC e cioè 135 perché in questo caso il problema non è risolto dall'algoritmo che ridi mentre nell'altro caso lo è dimostrabilmente Ebbene tutto questo dipende dal fatto che le ipotesi di questo famoso teorema di rado che ho a cui ho all'uso sono rispettate per il problema dello zaino unitario mentre non lo sono per il problema dello zaino generico e come potete facilmente immaginare la chiave sta nel fatto che il volume in un caso è unitario cioè uniforme nell'altro ca invece è vario e l'algoritmo gridi non ne tiene conto sostanzialmente Tranne per quanto riguarda il garantire l'ammissibilità della soluzione alla prima di andare ad approfondire questo aspetto quello che possiamo però fare implementare questo algoritmo greedy che io chiamerò algoritmo banale tenete bene a mente che non è l'algoritmo che si adotta proprio perché qui abbiamo trovato una soluzione da 12 contro 15 ma In altri casi possiamo trovare soluzioni veramente molto cattive e quindi non è la strada migliore per ora per risolvere il problema possiamo ripartire dal fainaz a zero da qui siamo partiti con la corsa lezione e nel quale sono presenti tutte le strutture dati per rappresentare Le istanze di problema dello zaino le soluzioni c'è la solita funzione per caricare per interpretare la linea di comando caricando ci il nome del file dei dati c'è la funzione per carica metterli nella struttura dati dell'istanza e ci sono le funzioni per creare e distruggere stampare le soluzioni tutto regolare Che cos'è che vogliamo fare a questo punto molto semplice vogliamo andare a costruire questa funzione e gridi banale che risolve il problema dello zaino Quindi quello che faremo sarà andare a definirci una chiamare una funzione gridi banale KP tanto per darle un nome questa funzione prende come dato l'istanza i la prendo per indirizzo non è che la voglia cambiare al solito il telefilm Cenza è vestito isce la soluzione S Questa invece la passo per l'indirizzo perché la voglio modificare ora non ricordo se ho cose ho detto a proposito di questo passo credo di aver detto che si potrebbe anche restituire in uscita una soluzione ma siccome la soluzione è un oggetto una struttura di dimensioni anche 0:29:00 Si potrebbe anche restituire in uscita una soluzione ma siccome la soluzione è un oggetto una struttura di dimensioni anche cospicue somme contiene un vettore che un vettore dinamico si preferisce di solito queste strutture e passarle per indirizzo anche quando sono risultati una cosa che credo invece di non aver detto è che si si potrebbe chiedere se l'algoritmo debba non solo riempire la soluzione ma anche eventualmente allocare le sue strutture dati è una possibilità se un salto io le ho a Locate Prima ho chiamato la funzione che aveva soluzione prima di lanciare l'algoritmo per diversi motivi uno dei quali è che questo mi consente di avere questa funzione di definire questa funzione in maniera tale che riceva una soluzione vuota la elabori e restituisca in uscita una soluzione riempita questo può essere utile in situazioni nelle quali magari l'algoritmo viene soluzioni vuota mano soluzioni già parzialmente riempita E perché potrebbe bisogno di lavorare come soluzione parzialmente riempita per mille motivi alcuni motivi possono essere che ci sono dei vincoli che certi oggetti devono essere presi per forza altri potrebbero essere perché io ho qualche motivo matematico Per essere sicuro che certi oggetti ci siano c'è tutta una teoria dietro i problemi di ottimizzazione combinatoria che mi

consente di andare a dimostrare che certi oggetti devono e altri oggetti non devono esserci quindi diciamo è una buona idea anche se limitatamente a Questo esercizio di oggi non serve a niente è una buona idea che questo algoritmo Comunque non costruisca davvero una soluzione la riceva Dopodiché dobbiamo essere sicuri noi che la soluzione sia in partenza vuota e lo siamo perché l'abbiamo creata noi subito prima quindi questa funzione gridi banale Che cos'è che sostanzialmente farà al solito la dichiariamo e poi la definiamo sostanzialmente dobbiamo fare questo Cioè dobbiamo andare a risolvere in stanza del problema del Nat problem restituendo la soluzione s e questa istanza La risolviamo con l'algoritmo gridi banale Ti amo di questo nome non è ovviamente il nome tecnico è una funzione Void che riceve in ingresso i dati del NAS a problem un puntatore a una struttura dati e restituisce in uscita una un puntatore a una struttura del Nasdaq problem questo possiamo copiarlo quali sotto costruendo un file un file un corpo vuoto A questo punto quello che rimane da fare e implementare questo algoritmo di cui abbiamo lo pseudocodice che non è particolarmente complicato però presenta un po' di cose di aspetti interessanti e poi aspetta interessante è come si fa a gestire questo complesso insieme degli oggetti che non fanno parte della soluzione attuale e hanno volume minore o uguale a 1 dato valore il quale dato valore dipende dalla soluzione attuale dobbiamo prendere un solo un sottoinsieme questo sotto insieme al momento è definito con un vettore di incidenza perché vi ricordo Torniamo qua che la soluzione è definita come un vi boolean Quindi a un vettore di valori logici 01 Allora a questo punto in effetti si tratta di sostanzialmente di sapere intanto Quali oggetti sono dentro e fuori questo è facile ma bisogna anche sapere quali oggetti hanno un volume che non è eccessivo rispetto al capacità residua e questo è un pochino più complicato sostanzialmente quello che dovremmo fare e a conservare il valore del volume totale cumulato fino adesso dalla soluzione oppure valore della capacità visiva equivalentemente io ragionerò sul volume totale e tutte le volte che andiamo a cercare un nuovo elemento Dovremmo cercare di elemento di valore massimo fra quelli il cui volume non eccede come si fa a gestire questa cosa si potrebbe pensare banalmente di fare uno scorrimento tutte le volte Cioè tutte le volte io mi chiedo Qual è l'elemento che non sta in soluzione il cui volume Cetra Cetra perfetto si può fare ovviamente vuol dire scompensi n elementi tutte le volte per quante interazioni E mal che vada tutti gli elementi stanno nello zaino quindi viene un algoritmo di complessità ennequadro cercheremo di fare una cosa migliore è questa cosa migliore è evidentemente ispirata dà l'idea che sia tutto sommato 0:33:50 Cremo di fare una cosa migliore è questa cosa migliore è evidentemente ispirata dà l'idea che sia tutto sommato facile gestire trovare l'elemento massimo di rispetto a una certa funzione di un insieme perché Perché Abbiamo appena studiato studiando i brani di ordinamento la struttura dati max-heap che qui casca assolutamente a fagiolo cioè l'idea è se noi potessimo rappresentare con un Maxim questo insieme che ci interessa Ecco che potremo estrarre l'elemento massimo istantaneamente poi ovviamente dovremo aggiornare l'IP in tempo logaritmico per semplificarsi la vita in realtà aspettiamo un attimino le due condizioni cioè la condizione di essere insieme esterno e la condizione di avere volume non eccessivo quello che farò sarà una cosa di questo genere andremo a cercare non l'elemento Massimo all'interno di tutto il sottoinsieme manderemo cercare l'elemento Massimo quindi meno ICS che è molto facile da gestire perché è molto facile da gestire perché di passo in passo noi possiamo trovare il massimo e buttarlo via è via via questo insieme si va riducendo esattamente come abbiamo fatto nel algoritmo di ordinamento quicksort Però il problema è che se facciamo così questo elemento massimo potrebbe non rispettare questa condizione di avere volume non eccessivo Perfetto se la Viola questo oggetto evidentemente adesso non va bene e non andrà mai più bene perché il volume totale non fa altro che accrescere e se il volume attuale non sta nella capacità residua il volume attuale di questo oggetto non sta nella capacità residue in passi successivi la capacità residua sarà

più bassa e quindi questo volume non ci sarà neanche in futuro per cui L'idea è di andare strade tutte le volte il massimo di BMW X6 non ci sta nello zaino buttarlo via Se invece ci sta nello zaino aggiungerlo che è tutto sommato molto semplice e ci permette i rombi meno ICS che è più facile da sia da definire sia da gestire rispetto a questa insieme un pochino più complicato Quindi al solito Proviamo a scrivere uno pseudocodice dopodiché vi inviterò a realizzare l'algoritmo per i fatti vostri e vi darò la soluzione anche perché c'è un secondo problema che dovremo affrontare che vi descriverò nel corso dello pseudocodice Allora la prima cosa che dobbiamo fare è la seguente Finché ci sono elementi in B meno ics abbiamo questo ciclo all'interno del quale andremo a trovare trova l'elemento di valore massimo in B meno ics è veramente $B \setminus X$ No in realtà no perché per case va crescendo Ma alcuni elementi non li buttiamo via e non li mettiamo in INPS quindi dobbiamo dare un altro nome a questo insieme e potremmo chiamarlo il primo che all'inizio ebbi meno ics quindi all'inizio puniamo V primo uguale b1x E poi Finché ci sono elementi in di primo c'ho elementi che è potenzialmente sensato aggiungere trova l'elemento di valore massimo in di primo è una volta che l'ha trovato valuta se si possa aggiungere a ics oppure no Se a volume non superiore alla capacità residua lo aggiunge a ics Se invece non ce l'ha se volume superiore alla capacità di Silva lo toglie semplicemente in realtà in entrambi i casi Comunque lo toglie da di primo questo è ancora più semplice Allora a questo punto possiamo procedere e questo è il nostro scopo mi faccio notare una cosa ed è che se vogliamo trovare l'elemento di valore massimo possiamo tranquillamente mettere i valori in unità malip come l'abbiamo descritto fino adesso contiene i valori e quindi possiamo prendere questo vettore Fi possiamo trasformare in unip Guarda casa qui il 7 è già posto in primo elemento possiamo cancellare il 7 sopra scrivendolo con l'uno riapplicare la funzione Aggiorna Ip in maniera che vada tutto a posto Se non c'è problema che a questo punto stiamo ritirando i valori di Fi e non sappiamo più Quali sono i nomi i colori se non sappiamo più come si chiamano uno di 0:38:40 I valori di Fi e non sappiamo più Quali sono i nomi di questi valori se non sappiamo più come si chiamano 1-2-3-4-5-6-7 gli 123456 gli elementi del del nostro problema Quindi questa cosa non è in realtà fattibile quello che in effetti bisogna fare Sostanzialmente è dare intanto dei nomi ai diversi elementi e per esempio possono essere i valori numerici 123456 Dopodiché tutte le volte che andiamo a modificare in qualche modo Questo qua è quello che dovremmo fare è andare in parallelo a modificare i nomi e in realtà in parallelo modificare anche i volumi perché se no i vari indici vanno i vettori vanno a sposarsi l'uno rispetto all'alto ti faccio vedere che cosa ha in effetti vorremmo fare quello che dovevamo fare tenere un vettore di indici 123456 parallelo al vettore di valori Alberto e imporre una situazione una struttura di Max heap in realtà non solo Sufi Ma su tutte e tre questi oggetti per inciso se invece che fa il 3 vettori fate un unico vettore di terne va benissimo lo stesso ed è anche fosse un punto informatico sarebbe più elegante anche se non ti sta matematica è più complicato e vedremo che non seguiremo questa strada Comunque tanto per seguire l'ordine di idee che stiamo sviluppando Se vogliamo che fai diventi un max-heap cosa dobbiamo fare applicare la solita funzione creai per la quale dice che la metà seconda del vettore è già unip perché i figli di questi elementi sono sono delle elementi sono foglie quindi non hanno figli fino a partire dall'elemento di indice 320d indice troverà solo la foglia di sinistra il figlio di sinistra che è l'elemento di indice 6 e bisogna controllare se 4-6 valori di Fi è maggiore uguale al valore corri del figlio di sinistra che è uno lo Eh quindi non c'è niente da fare allora possiamo vederci vedere a 22 ci dice che il valore del di fi in corrispondenza l'indice dure dovrebbe essere maggiore uguale a quello dei suoi due figli che sono gli indici 4 e 5 ma due non è né maggiore uguale 5 né maggiore o uguale a 4 Quindi bisogna scambiarlo con il massimo dei due c'è bisogno di portare il 5 al posto del due delle mento di indice 4 al posto elemento indice 2 e scambiarlo ed è quello che viene fatto qua sotto qua

sotto abbiamo scambiato Clemente 24 i valori 25 e i volumi Tre Tre è che essendo uguali non si sono scambiati a questo punto dovremmo andare a vedere se per caso qui è tutto a posto Ma questa è una foglia ed è posto per definizione A questo punto possiamo tornare all'indice uno e scoprire che per fortuna 7 effettivamente è il massimo e quindi è maggiore sia di 5 sia di qua poi non c'è altro da fare ma questa è una strada possibile che non stata pesante perché vuol dire che ogni volta che c'è da fare uno scambio che sia da fare nella nella carriera IP iniziale che sia da fare via via che recuperiamo l'elemento di valore massimo e lo mettiamo nello zaino cancellandolo o eventualmente lo buttiamo via perché a volume eccessivo cancellandolo in entrambi i casi lo scambio comporta di andare a rigirare non uno ma tre vettori e in generale potrebbe portare a spostare grosse quantità di dati Come si fa a invitare questa cosa c'è una seconda possibilità la seconda possibilità è quella di effettuare gli scambi non Sufi non su tutti e tre i vettori ma soltanto sul vettore degli indici e precisamente l'idea è la seguente confrontiamo la situazione iniziale a questa situazione finale nella situazione finale io ho tenuto gli scambi effettuati sul vettore degli indici ma ho lasciato perfettamente intatti i due vettori e vi che sono identici a come vanno all'inizio cioè gli scambi che abbiamo visto lo scambio che abbiamo visto per la posizione 2 la posizione 4 è stato fatto solo a livello di indici perché questo c'è a risolve da solo il problema per un motivo molto semplice che la proprietà IPS Uffi non vale ma indirettamente se arriviamo a Fi passando attraverso ind in realtà Vale diamo un'occhiata in particolare guardiamo l'elemento due allora è chiaro che questo questo due non è maggiore uguale ai due figli 54 Ma se io invece di andare a prendere l'elemento due vado a prendere l'elemento ind di due che è 4 e i suoi due figli sono in d e d 4e indagati 5 quello che sto dicendo è che devo confrontare il quarto elemento del vettore dei valori e vedere se è maggiore uguale al secondo elemento del valore dell'IVA 0:43:30 Il quarto elemento del vettore dei valori e vedere se è maggiore uguale al secondo elemento del valore del vettore dei valori è al quinto elemento del vettore dei valori ora il quarto elemento del vettore dei valori è 5 ed è maggiore uguale al secondo ed è maggiore uguale al quinto cioè effettivamente la proprietà hip indirettamente filtrata attraverso questo vettore di indici Vale senza bisogno di spostare gli elementi dei vettori che contengono i dati Questa è la grande forza del meccanismo di indicizzazione mento indiretto questo diventa un indiretto e ha il grosso vantaggio di non richiedere scambi sui dati il secondo vantaggio è che se volete fare molti ordinamenti diversi su questi dati per esempio di volete ordinare non per valore crescente ma per volume non per valore decrescente ma per volume decrescente Vi basta aggiungere un secondo vettore di indici farci quello che volete e a questo punto avrete l'ordinamento anche dall'altro punto di vista che mi interessava e potete introdurre ordinamenti di qualsiasi genere senza toccare i dati chiaramente questo costa spazio ogni nuovo ordinamento costa un vettore di interi in più e qual è un altro costo risparmia anche il costo temporale degli scambi ma aggiunge un costo di cemento indiretto cioè io per poter andare a leggere il primo elemento non vado infidi uno vado in fi di ind di uno è qui guarda caso in the di una uguale a uno quindi indirizzamento indiretto coincide con quello diretto ma è perfettamente possibile che il massimo nel detto che si stia da un'altra parte semplicemente bisognerà per accedervi andare a leggere in questo rallenta leggermente ma è il vantaggio in termini di scambi è nettamente nettamente dominante Probabilmente è anche il vantaggio in termini di L'eleganza di possibilità di mantenere i dati originali e di gestire tanti ordinamenti diversi per cui è decisamente consigliabile a questo punto quello che vi consiglio di fare siccome già avete costruirmi il lettore degli indici torniamo indietro quindi dovremmo Per prima cosa costruire il vettore degli indici vettore che ovviamente al termine andrà poi deallocato dealloca il vettore degli indici poi andremo a costruire questo hip su tutto l'insieme degli oggetti che non stanno i Knicks mi faccio notare che si ipotizza sostanzialmente ipotizzare mo che il sia vuota all'inizio Quindi questa parte

qui me la sbrigo va facilmente dicendo che sono tutti gli elementi in linea di principio Bisognerebbe testarlo Dopodiché Finché ci sono elementi da prendere ancora in considerazione Procederemo in questo modo che è molto simile all'algoritmo di heapsort quindi chi volesse cimentarsi mette in pausa Che proceda Allora dicevamo si serve un vettore di indici Sarà ovviamente un vettore dinamico vinto che andiamo dal lo carci della Dimensione corretta Vabbè io continuo a usare la callo anche se poi in realtà inizia rideremo questi dati Quindi è un po' è un po' inutile In effetti e quanti ce ne servono se ne servono tanti quanti sono gli oggetti faccio più uno in maniera che vadano da1n e il singolo elemento sarà un intero Dopodiché Dobbiamo verificare che la locazione sia andata a buon fine con la solita verifica è il solito messaggio di errore qualora invece ci sia stato un problema un errore nella locazione del vettore degli indici l'ho fatto se c'è l'errore usciamo e qua in fondo Andiamo a liberarci di questo vettore ausiliario che a questo punto non ci serve più allora che altro bisogna fare inizializzarlo Cioè il vettore degli indici deve contenere questi valori 123456 Che significa che per i che va da 1 fino a pxn andiamo ad assegnare a indice di il valore di è così qui dovessimo buttare via qualche cosa 0:48:30 Buttare via qualche cosa Questo sarebbe il momento il posto opportuno per dire che alcuni elementi del nostro del nostro insieme non devono stare non devono essere presi in considerazione nel nell'algoritmo stesso a questo punto cosa facciamo Andiamo a dire che vi primo contiene tutti gli elementi ovviamente qui ci serve un cursore Come facciamo a dire a gestire ogni che va via via decrescendo Se vi ricordate avevamo un indice che ci diceva Qual è qual è la l'ultima coordinata la dimensione della tabella che va accorciando Sì potremmo per esempio chiamarlo n tanto per non inventarci nomi strani questo n Inizialmente è tutto tutto l'insieme degli oggetti c'ho fatto dobbiamo trasformare in un heap il vettore ma abbiamo bisogno di un IP Strano questo indiretto quindi invece di creare un ID come abbiamo fatto fino adesso vi ricordo si creare un heap passando il vettore v e la dimensione n qui dobbiamo intanto capire qual è il vettore su cui stiamo lavorando il vettore su cui stiamo lavorando e il vettore Fi che fa parte dei dati Quindi sarà P freccia figli ma noi vogliamo in realtà un vogliamo in realtà un indiretto e quindi utilizzerà gli indici per accedere noto che in realtà nella soluzione non ho preso n ho preso numero degli esterni Tanto per restare coerente con le soluzioni che vi ho dato correrei qntn Est e a questo punto qui Dobbiamo creare un hip hop questa volta indiretto che mi gestisca il vettore Fi attraverso indirettamente attraverso il vettore indice come fosse un heap che massimizzi il valore della della funzione Fi Allora a questo punto noi possiamo procedere Finché ci sono elementi nello heap cioè finché nn rimane maggiore di 0 e che cosa faremo andremo a cercare l'elemento di valore massimo e questo elemento di valore massimo lo andremo a testare per vedere se il volume è eccessivo è O è sufficiente abbastanza piccolo da stare nella soluzione nello zaino o se invece è eccessivo e va buttato Come si fa a recuperare l'elemento Massimo molto semplice si va a prendere l'elemento che sta in posizione 1 ma in posizione 1 non in ma in posizione 1 indice di conseguenza andiamo a recuperarsi un indice Tmax il massimo sarà uno ma sarà indice di uno al primo passo siccome sarà 7 coincideva con uno testiamo il volume di Max quindi andiamo a vedere se vi lì Imax è minore o maggiore minore uguale oppure Maggiore della capacità l'esigua e a questo punto per avere la capacità residua ci serve il volume cumulato fino adesso Quindi abbiamo bisogno del volume V cumulato fino adesso che inizialmente è Euro 0 e dobbiamo confrontare questo volume qua con che cosa con tutta la capacità di grande a cui si toglie il volume curato fino adesso lì se questa cosa vale Allora significa che il l'oggetto di valore massimo può effettivamente essere assegnato alla nostra soluzione Come si fa ad assegnare un elemento alla soluzione si va a prendere la soluzione si prende ICS e in posizione Imax si scrive tu si mette vero abbiamo finito lì no perché il volume cumulato cresce di quanto cresce cresce del volume di Imax e in realtà Vi ricordo che la soluzione Contiene anche un campo F che è il valore della soluzione è questo

quello della soluzione necessariamente deve crescere di figli di imacs in questo modo io sto aggiungendo a una soluzione un elemento Ecco Se volete questa coppia di punti operazioni potrebbe essere sostituita da una funzione Aggiungi elemento che lavora su soluzioni sarebbe più incapsulata non lo sto facendo Perché mi sembra tutto sommato superfluo e sicuramente ha meno efficiente 0:53:20 E sicuramente meno efficiente qui a Chiusa La quadra a questo punto Comunque sia l'elemento va tolto dal insieme va tolto dall'ip Come si fa a togliere e vi ricordo si tratta di copiarsi sul l'ultimo elemento dov'è che lavoriamo Dov'è Lip Lip è nel vettore degli indici quindi indice di uno viene sovrascritto da indice del totale della dimensione del nostro Iper quindi numero esterni e a questo punto il numero degli esterni va a semplicemente decrescere di uno abbiamo finito No perché in questo modo non è più Nip bisogna andare ad aggiornare Lip indiretto Vi ricordo che una volta si faceva così nel caso degli ci aveva avevamo il vettore vi avevamo la dimensione n e avevamo la posizione che era stata modificata che va la posizione 1 adesso non facciamo così dobbiamo andare a toccare Intanto un vettore in un IP indiretto che usa il vettore degli indici il valore il vettore rispetto al quale massimizziamo che consideriamo l'elite è freccia Fi la dimensione la mia chiamata numero esterni e questo uno di per sé invece rimane lo stesso perché comunque all'interno del dell'indice è l'elemento uno che corrisponde alla al massimo ed è quello che abbiamo cancellato qui indice di uno allora Tutto questo ci riconforta salvo il fatto che ovviamente abbiamo creato è aggiornato Unipd e che quindi queste cose vanno in qualche maniera recuperate Come facciamo a recuperarle una certa sveltezza io direi che ci ispiriamo abbastanza pesantemente per vedere che cosa è che cambia in effetti alla heapsort che abbiamo realizzato nella un paio di lezioni fatte nelle quali c'era la gestione è l'aggiornamento la creazione e l'aggiornamento di unip non di un IP indiretto avevamo crea hip e aggiorna hip ci serviranno un crea IP indiretto e aggiorna i più indiretto quindi andiamo a dichiarare definire prima vuote e poi piene queste funzioni la funzione che era hip indiretto ovviamente non fa altro che creare un heap indiretto attraverso indice sul vettore chiamiamolo pure B quindi abbiamo una funzione Void che riceve un vettore di indici interi e ricevi invece un vettore di valori che qui sono ancora interi e riceve un Lunghezza n Questo è quello che dobbiamo fare e analogamente Dobbiamo anche andare ad aggiornare se dobbiamo andare ad aggiornare Lip indiretto attraverso indice sul vettore di di Lunghezza n a partire dalla posizione i Equi Analogamente aggiungeremo il vettore indice Queste sono le due funzioni che ci tocca andare a definire le possiamo costruire inizialmente vuote e poi possiamo procedere a riempirle in parallelo a quello che si faceva prima cambiando ciò che c'è da cambiare cosa c'è da cambiare creai per quasi banale si tratta di scoprire dalla posizione mediana fino alla prima applicando Aggiorna hip Ed è esattamente la stessa identica cosa salvo il fatto che questo hip non è un Iper Ma è un indiretto quindi andremo dai nei mezzi giù fino uno chiamando Aggiorna hip indiretto e lavorando su indice vne di e questo è in effetti quanto vediamo invece l'aggiornamento cosa cambia nell' aggiornamento molto poco abbiamo sempre il figlio di sinistra il figlio di destra che Sono gestiti con il solito trucco della moltiplicare per due motivi cane per due sommare uno troviamo sempre valutare quale Fra indice indice Se l'indice di corrisponde al massimo quindi dobbiamo sempre valutare se SD sono valori ragionevoli minori e uguali ed n L'unica cosa che cambia sono questi confronti Poi per il resto c'è veramente poco che cambia quindi 0:58:20 Poi per il resto c'è veramente poco che cambia quindi procedo a copiare e incollare e a cambiare ciò che c'è da cambiare abbiamo sempre gli indici SD e l'indice Imax abbiamo sempre la definizione di SD come due per I e II per i più uno Imax si comincia Supponendo che sia il padre dopodiché se il figlio di sinistra che esiste ed ecco qua cosa che c'è da cambiare non dobbiamo accedere direttamente a v Ma dobbiamo accedere a v indirettamente passando attraverso il vettore indice applicato all'indice s e la stessa cosa faremo in D è la stessa cosa faremo in Imax Sì queste condizioni valgono Allora

Imax rispettivamente potrebbe diventare uguale essere o diventare uguale a dirti se abbiamo da cambiare qualche cosa c'è sei ma non è il padre andiamo a scambiare che cosa che non andiamo a scambiare i valori direttamente dei dati andremo a scambiare i valori degli indici e Attenzione qui non si scambia vidi indice perché i dati vi devono rimanere identici Qui si scambiano fisicamente gli indici Dopodiché andremo chiamare Aggiorna Lip indiretto al solito su indice vne Imax perché questa è la classica chiamata ricorsiva la quale non è poi cambiato sostanzialmente nulla allora se non abbiamo fatto pasticci L'unica cosa che manca e la funzione scambia Ma quale funzione scambia continua tranquillamente a scambiare due valori interi passati per indirizzo questo dovrebbe essere sufficiente a consentirci di compilare la nostra funzione abbiamo un Aggiorna IP indiretto dichiarata implicitamente la Viga 270 perché Aggiorna hip qui è indiretto e presumibilmente qua in alto Esatto Non era stata dichiarata quindi giustamente il compilatore non aveva trovato nessuna dichiarazione di questa funzione però a questo punto siamo sintatticamente a posto E potremmo persino provare a lanciare il nostro programma sulla prima soluzione Allora vi ricordo andiamo a vedere le soluzioni ricordo che nei dati le ultime due righe riportavano le soluzioni ottime Qui abbiamo una soluzione che vale 12 e che comprende gli elementi 14 Cioè hai di questa è l'esempio che abbiamo risolto a mano ed effettivamente non ci dal ottimo poi possiamo proseguire possiamo andare a caricare gli altri tre esempi di andare a confrontare le soluzioni euristiche che stiamo trovando con questa con questo algoritmo greedy troviamo una soluzione che vale 247 invece di 309 evidentemente parecchio più piccola parecchio meno valida questa qui che era 13 milioni e mezzo invece mi dà risultato 13 minuti e 140 mila che non è poi malissimo bisogna valutare in percentuale è questa da 50 elementi ci dà valore 981 conto 1166 che non è molto valida tutte queste soluzioni sono tenute Praticamente in maniera istantanea proprio perché la complessità di questo algoritmo che non è difficile valutare Qual è la complessità la complessità è data da questo ciclo sostanzialmente ciclo che viene ripetuto quante volte è che abbiamo un n Sterni che parte da n e che ad ogni passo si riduce di uno quindi questo ciclo viene eseguito rigorosamente n volte Dopodiché il ciclo viene eseguita n volte all'interno ci sono operazioni in tempo costante questo assegnamento quest'altro Qui c'è un operazione di aggiornamento del indiretto che abbiamo visto essere sostanzialmente la stessa del maggior knipe quindi è logaritmica in N mentre Queste sono operazioni ancora una volta in tempo costante quindi l'intera organizzazione intero algoritmo va in tempo e nello ghen è possibile migliorarlo non asintoticamente una cosa che si può fare e osservare che in effetti se noi a un certo punto abbiamo raggiunto la una soluzione che ha dimensione massima e non si può fare aumentare ulteriormente Beh un po' stupido continuare a esplorare e a volte fallire questo test e fintanto che non si eliminano gli elementi uno alla volta quello che sarebbe carino sarebbe vendersi con 1:03:10 Sarebbe carino sarebbe vendersi conto immediatamente che gli elementi ormai non ce ne sono di non ce ne sono più che siano inseribili nello zaino un modo molto Brutale che a volte funziona di farlo e quello di dire che noi proseguiamo fintanto che il volume attuale strettamente minore della capacità dello zaino cioè chiaro che se questa condizione è falsa c'è Se vi diventa uguale la capacità dello zaino via il volume totale cumulato Non ha senso cercare di aggiungerci altro si può migliorare questa cosa si potrebbe dire vi più valore minimo di volume dei gli elementi che sono ancora all'interno della hipp Come si fa a sapere il valore del volume minimo all'interno dello hip e bisognerebbe avere un altro hip organizzato per minimo che mi dia sempre il valore minimo Cioè sostanzialmente non sviluppo questa linea di pensiero ma è un esercizio che mi propongo di fare che è descritto nelle di accompagnamento quello che si potrebbe fare aggiungere un secondo hip con un secondo in di ispettore di indici che determini invece che si è relativa invece ai volumi e che gli consenta di andare a buttare via tutti gli elementi che hanno volume eccessivo rispetto alla capacità decidua questo dovrebbe accelerare ulterior-

mente chiaro che finché lavoriamo su istanze così piccole non ce ne rendiamo conto perché siamo già sotto il milionesimo di secondo Ma sui stanze da migliaia o decine di migliaia o milioni di oggetti effettivamente la cosa poi invece avere un senso allora quello che invece ci resta da fare fondamentale e capire se non si possa migliorare la prestazione di questo algoritmo specialmente basandosi sul fatto che per lo zaino in Italia Invece l'algoritmo funziona bene allora la risposta è un po' la seguente risposta è che no Scusate un attimo Eccoci qua se il nostro spazio non ha le proprietà che sono richieste dal famoso teorema di rado si può cercare di mettere una pezza e questo pezzo consiste in sostanza nel cambiare il criterio di scelta invece di massimizzare hi-fi si massimizza un vero e proprio criterio che dipende dal elemento aggiunto eventualmente anche dalla soluzione data tenendo conto non soltanto del premio del valore alla funzione obiettivo ma anche dei vincoli del problema ora qual è il problema effettivo il problema effettivo che abbiamo una capacità e che se la Riempiamo troppo con oggetti di valore alto ma di volume altrettanto alto è fastidioso in realtà rischiamo di non guadagnare tutto quello che si potrebbe guadagnare Quindi quello che noi vorremmo e trovare oggetti di valore alto ma al tempo stesso anche di volume basso È un modo grezzo ma non del tutto stupido di farlo e quello di andare a massimizzare in un Fi rapporto fra VII e VIII cioè il rapporto fra il valore e il volume di un oggetto che potremmo chiamare valore unitario del singolo getto perché questo algoritmo funziona molto meglio qui possiamo fare un esempio vediamo la solita istanza vediamo i rapporti Fi / B valori / volume e vediamo che se partiamo non prenderemo all'oggetto sa che al valore massimo ma anche un volume grosso prendere un oggetto è che va alle 4 ma un volume di 1 quindi è piccolino preso questo oggetto andavamo a prendere il secondo che è C E A questo punto il volume Pavia via crescendo da 1 a 3 ma c'è ancora molto spazio e guarda caso il 3 il terzo oggetto da prendere è l'oggetto a è complessivamente in questo modo riusciamo a riempire lo zaino costruendo la soluzione effettivamente ottima a cm 135 divano le 15 no scusa ti ho detto una stupidaggine e ho convinto di sta seguendo l'algoritmo perché in realtà il terzo passo giusto al terzo passo guardato a Non è vero c'è l'allenamento di che è migliore Esatto quindi la soluzione che si trova in realtà comprende e C e D con un volume totale di 1 + 2 + 36 e ci sta dentro anche la F Perfetto Quindi volume 7 e il valore è 4 + 14 fa 8 + 5 di di fa 13 + 1 fa 14 che non è ancora ottima però è una soluzione decisamente meglio di quella trovata in precedenza perché tutto questo ha un senso ora molto informalmente perché andare a valutare il valore unitario è un po' come se noi prendessimo ciascuno di questi oggetti e lo spaccassi mo 1:08:00 Valutare il valore unitario è un po' come se noi prendessimo ciascuno di questi oggetti e lo spaccassi Mo in oggetti di volume unitario è un po' come se prendessimo l'oggetto a e lo dividessimo un oggetto di volume 5 in 5 oggetti di volume 1 Ognuno di questi 5 oggetti a una parte del valore 140 e a questo punto noi avremmo tanti oggetti tutti identici tutti di valori 140 ovviamente avremo poi 3 oggetti corrispondenti AB di valore 067 due oggetti di valore due corrispondenti AC3 oggetti di valore 167 corrispondenti a di eccetera Questo è molto interessante Perché a questo punto ci troviamo in realtà in uno zaino unitario che possiamo applicare l'algoritmo giuridiche cui funzionerebbe e andrebbe a prendere l'oggetto corrispondente a e poi andrebbe a prendere i due oggetti corrispondenti a c poi andrebbe a prendere i 3 oggetti corrispondenti a di il singolo oggetto corrispondente qui non devo sbagliare l'abbiamo detto prende l'oggetto corrispondente ai i due oggetti corrispondenti aci3 oggetti corrispondenti a di' a questo punto ha un'occupazione di 6 i successivi sono i 5 oggetti di valore a in realtà non prendiamo questi cinque oggetti di valore perché non ci stanno tutti e cinque ma 1 + 2 + 3 fa 6 c'è ancora posto per due oggetti quindi possiamo prendere due oggetti di questo elemento di questo corrispondenti Ada e di conseguenza riusciremo a riempire perfettamente lo zaino con gli elementi di valore massimo questa soluzione non è arrivata accettabile Ma è possibile di trasformarla in una soluzione accettabile

andando a buttare via i due oggetti che costituiscono una porzione incompleta di uno degli oggetti di partenza e poi il risultato si può andare migliorare andando a prendere invece qualche cos'altro Cioè In questo caso l'oggetto F Cosa significa sostanzialmente che abbiamo applicato in un certo senso il problema dello zaino unitario fin dove possibile Poi ci siamo fermati e abbiamo dato un ulteriore piccolo miglioramento prendendo l'oggetto F Siccome il problema dello zaino in Italia si può risolvere all'ottimo e siccome la modifica che abbiamo fatto è ridotta Perché si tratta In sostanza sempre di prendere un pezzo di un oggetto e poi dire non riesco a prenderlo tutto lo butto via Quindi l'errore dei Guarda se vogliamo un oggetto soltanto e questo garantisce che presumibilmente la soluzione non sarà di qualità tanto cattiva Allora direi cerchiamo di applicare questo algoritmo e vedere che cosa cambia in sostanza nell'algoritmo che abbiamo già realizzato Se volessimo ragionare in termini non ti Massimo valori di Fi Ma Massimo valore del rapporto fini visuali Allora cos'è che si potrebbe In sostanza fare non cambierebbe quasi nulla Bisogna sempre lavorare su ogni indiretto solo che invece di lavorare Sufi Bisogna lavorare sul rapporto fili viso lì Qui ci sono due possibilità per muoversi Una è quella di passare a creare i PIN diretto e aggiorna I pin diretto sia sia sia p e b cioè sia il vettore dei valori e dei volumi Dopodiché nel gestire questi Cioè in particolare all'interno della funzione di aggiornamento non dovremmo lavorare sui vettori V Ma siccome ne arrivano due di vettori i valori Ei volumi dobbiamo lavorare sui rapporti questo modo di procedere però è abbastanza confuso e in elegante perché perché tutte le volte che cambia il tipo di dato che vogliamo gestire questa volta aveva porto di due vettori In altri casi potrebbe essere il prodotto la differenza una combinazione di qualunque genere tutte le completamente la funzione non sarebbe meglio se noi potessimo invece gestire unip che lavori con un indice intero e con un dato che poi è quello che dobbiamo massimizzare risposta sì L'unico problema è che questo dato e sembra un rapporto di due numeri interi non è più un numero intero ma il numero Nazionale Diciamo in generale reale e quindi l'unica cosa che dobbiamo fare trasformare questo IP indiretto in un che lavori su un vettore di valori reali In altre parole quello che ci interessa è avere un vettore Fi / V io lo chiamerò vivi che è un vettore B Double che 1:12:50 Fili viso vi io lo chiamiamo fivi che è un vettore B Double che se per caso come in effetti non c'è andiamo dichiararci il tipo di Double che ovviamente è solito puntatore al tipo base Dopodiché torniamo su avendolo dichiarato possiamo andare ad allocare questo vettore e andare a Calcola ce l'ho Quindi a questo punto usate però sto facendo una stupidaggine cerchiamo di andare con ordine quello che voglio fare è non applicare un gridi banale ma voglia applicare un gridi vero e proprio Quindi qua sotto mi vado a costruire una funzione grilli che applica l'algoritmo Drilli non banale quello con il rapporto rispetto a Fi / Fi questa funzione andiamo a dichiararla qua in alto ok ed è questa funzione che andiamo a chiamare per risolvere il nostro problema Quindi abbiamo la chiamata di Crilin abbiamo la definizione di Grilli che lavora su dati e soluzione e qua sotto adesso andiamo a costruirci la funzione che è praticamente identica alla precedente soltanto che va a lavorare su un dottore di Double fighi il quale vettore va ovviamente allocato e costruito inizialmente quindi dobbiamo allocare il nostro vettore di valori Double della lunghezza giusta è fatto di Double mondi interi questo vettore se non viene allocato dobbiamo dire che c'è stato un errore noi detto dei porti / v i a questo punto lo Riempiamo Con che cosa lo Riempiamo molto semplice per riempirlo dobbiamo andare a calcolare i rapporti del elementi Fi iesimo / l'elemento i-esimo del vettore i ricordandoci che questi due elementi sono dei numeri interi è che se lo scriviamo così questa diventa una divisione intera cioè quella che Butta via il resto non la vogliamo la divisione intera vogliamo Quindi prima trasformare uno dei due elementi in un numero reale prima di dividerlo per l'altro Quindi ci vuole questo cast esplicito A questo punto è solo a questo punto dentro nel vettore fivi abbiamo tutto quello che serve Dobbiamo ricordarci in uscita di The Halo Carlo

quindi di andare al locale VII dealloca il vettore degli indici e quello dei rapporti e a questo punto dobbiamo gestire questo nostro lettore tutto.it indiretto lo gestiamo passando gli vii-viii anziché il vettore Fi è la stessa cosa faremo qua sotto Ovviamente questo non funziona perché la funzione Aggiorna IP indiretto e crea IP indiretto richiedono dei vettori di interi Quindi vanno aggiornate trasformandoli in dati di tipo Double sia per l'aggiornamento sia per la creazione se non ho fatto grossi pasticci poi avendo fatto tutto un po' in fretta quello che succede a che a questo punto abbiamo un problema relativo alla vecchia funzione Eccola quando i di banale che ovviamente non è più accettabile che usa delle versioni di Crea IP diretto a cui passa un vettore di intero mentre abbiamo modificato la definizione Allora facciamo così le racchiudiamo all'interno disegni di compilazione condizionale dicendo che questa cosa funzionerà solo se definiamo banale Quindi il nostro compilatore salterà questa funzione e a questo punto non dovrebbero esserci particolari difficoltà Anche se ammetto che ho tirato un po' veloce Qui abbiamo conflitto Inter 1:17:40 Ammetto che ho tirato un po' veloce Qui abbiamo conflicting Type per crea hip indiretto 283 perché di sopra non ci siamo ricordati di aggiornare le definizioni 1 e 1 2 sembra che tutto funzioni possiamo provare una per una le nostre istanze e quello che ci aspettiamo è sull'istanza da 6 di vedere quel valore che abbiamo costruito a mano pari a 14 sull'istanza di ricordo che prima avevamo ottenuto invece soltanto 12 su distanza da 10 troviamo addirittura l'ottimo sull'istanza da 24 troviamo 13 milioni e 4 mezzo abbiamo ottenuto 13 milioni e uno è sull'istanza da 50 troviamo 1153 contro 1166 quindi siamo decisamente arrivati molto più vicino alla nostra soluzione ottima a questo punto posso fare un passettino 5 minuti concedermi una extra facendogli osservare che questa non è ancora la soluzione definitiva e in alcuni casi può essere decisamente mal impostata prendiamo questo esempio in cui abbiamo addirittura due oggetti di valore 10 e 90 di volume 110 e Supponiamo di avere uno zaino di capacità 10 È chiaro che la soluzione ottima consiste nel prendere l'oggetto B che perlomeno Vale 90 quindi qui litri di Panna le becca l'ottimo se applichiamo il gridi intelligente che considera il rapporto cioè solitario E siccome a Vale 10 e a volume uno ha un valore unitario di 10 B Vale 90 e ha un volume di 10 è un volontario di nome quindi inferiore apparentemente più promettente ha solo che se prendete a riempi te pochissimo lo zaino e l'altro getto non ci sta E non potete prenderlo di conseguenza ci sono in stanze come queste in cui si vede che è olistica può andare veramente male a piacere nel rapporto lo scarto il rapporto tra l'ottima il valore trovato questo è grande ha piacere Tuttavia si può fare una piccola modifica che qui viene anche trovata è dimostrata Ma questo lo lascio perché è assolutamente tutto extra e per il gusto di vedere il genere di ragionamenti con la quale si riesce a dimostrare che l'algoritmo può essere Addirittura due approssimato e vi dico cosa vuol dire vuol dire che il rapporto tra la soluzione ottima è quella che si trova Non può essere più grosso di due cerchi si trova almeno metà dell'ottimo metà non è un granché però è molto meglio che avere un risultato cattivo a piacere In cosa consiste l'algoritmo due approssimato molto semplice e consiste in pratica nel dire che ho si applica l'algoritmo che abbiamo visto cioè il rapporto fidi giovedì oppure banalmente si va alla ricerca dell' elemento di valore massimo e se si trova che l'elemento di valore massimo ci sta nello zaino quella è già una soluzione Siccome il problema si trova quando c'è un elemento di valore massimo che ha già anche un volume molto grosso quella soluzione li può essere un candidato da considerare In alternativa alla soluzione dell' algoritmo greedy la migliore delle due si può abbastanza facilmente dimostrare con un paio di passaggi algebrici essere almeno metà dell'ottimo e questo è un grosso vantaggio in realtà poi + ing Ale si può migliorare migliorare e migliorare gli algoritmi stessi aggravando né un pochino la complessità computazionale senza mai arrivare effettivamente un algoritmo ottimo polinomiale perché il Nasdaq problem non ha questa è uno di quei problemi per i quali non sono note anche ritmi polinomiali Ma è possibile avvicinarsi Aumentando il grado del

polinomio avvicinarsi sempre di più al ottimo in maniera assolutamente indefinita che è una bellissima proprietà di per sé e con questo direi che chiudo la prima parte della lezione Ben ritrovati Dunque nella prima parte di questa lezione abbiamo applicato la algoritmo greedy base al problema dello zaino e abbiamo scoperto con una serie di esempi che in pratica questo algoritmo non 1:22:30 Base al problema dello zaino e abbiamo scoperto con una serie di esempi che in pratica questo algoritmo non funziona invece lavorando su un esempio specifico molto piccolo ci siamo resi conto che sullo zaino unitario a questo modo di come funziona e diciamo abbiamo un'intuizione che potrebbe funzionare su qualunque stanza non solo su quel piccolo esempio abbiamo anche visto che modificando l'algoritmo gridi e utilizzando non il valore come criterio guida ma il valore unitario degli oggetti si possono migliorare le sue prestazioni ma ancora una volta non sia viva all'ottimo Allora questo punto può essere interessante li chiamasse un attimo alla teoria che vedrete trattata nel modulo di teoria Appunto e vedere per quale motivo questo algoritmo funzioni o non funzioni sul problema dello zaino o sul problema dello zaino unitario motivo sta nella struttura algebrica di questo problema sostanzialmente la algoritmo greedy se si ha un problema di ottimizzazione combinatoria quindi definito su un dato insieme base che nel caso dello zaino è l'insieme degli oggetti è un dato io l'ho chiamato spazio di ricerca nelle più correttamente si dovrebbe chiamare collezione di indipendenti in maniera che l'insieme dei due b ed F costituisca un sistema di indipendenza questo spazio di ricerca l'abbiamo proposto essere l'insieme di tutti i sottoinsiemi la collezione di tutti i sottoinsiemi che hanno volume totale non superiore alla capacità e quindi sono parti promettenti di soluzioni addirittura nel caso dello zaino Sono effettivamente soluzioni ammissibili bene definito un problema di ottimizzazione combinatoria In questo modo si può dimostrare con un famoso teorema che l'algoritmo gridi risolve questo problema Qualunque sia la funzione o purché additiva cioè purché data dalla somma degli valori di una funzione ausiliaria sugli elementi del sottoinsieme considerato Se solo se valgono alcune proprietà di base allora nel caso dello zaino tra parentesi la funzione obiettivo e additiva quindi sembrerebbe che siamo un po' nelle condizioni in cui si possano indagare queste proprietà proprietà che io rinuncerò in maniera molto informale semplicemente per verificarle applicando le ha lo zaino ero già in unitario sono le seguenti la prima proprietà dice che il sottoinsieme vuoto è un indipendente cioè fa parte di questa collezione F perché ci serve questa proprietà Sostanzialmente perché l'algoritmo che riparte dal sottoinsieme vuoto e in sostanza si muove rimanendo all'interno di questo insieme di indipendenti Quindi è una condizione tendenzialmente necessaria la seconda proprietà che ci ho imporre è che se un sottoinsieme è un indipendente allora tutti i suoi sottoinsiemi propri tutte le sue parti sono anch'esse indipendenti e anche questa cosa non l'ho detto la il sottoinsieme vuoto nel caso dello zaino è un indipendente perché ha un volume totale che è sicuramente minore uguale alla capacità torniamo alla seconda proprietà proprietà dice che qualunque indipendente deve ammettere come parti Proprio come i sottoinsiemi soltanto altri insiemi indipendenti perché questo perché sostanzialmente noi vogliamo poter arrivare a una soluzione con algoritmo greedy aggiungendo un elemento alla volta è l'ipotesi che stiamo facendo qui che è un'ipotesi di chiusura rispetto all'inclusione ed ipotesi molto forte in realtà ma che ci garantisce di poter costruire un sottoinsieme in qualunque modo si voglia seguendo aggiungendo elementi in qualunque ordine qui non è strettamente una condizione per non appare come una strettamente necessaria Ma è chiaro che in qualche modo ci aiuta questi sono ragionamenti che sto facendo informalmente ripeto dal punto di vista dello zaino Cosa vuol dire questa proprietà Vale beh sì perché se abbiamo un qualunque sottoinsieme di oggetti ammissibile cioè di volume totale non superiore alla capacità e ne prendiamo una parte proprio ne buttiamo via qualcuno è chiaro che il volume totale diminuirà non cresce certo E quindi rimane uguale alla capacità Quindi questa seconda proprietà Vale nel problema dello zaino

la terza proprietario abita più sofisticata e qui sarò ancora più vago e informale Supponiamo di avere due indipendenti uno un pochino più piccolo è un pochino più grosso con un elemento in più un elemento in più non vuol dire che è la stessa lo stesso insieme con un elemento in più vuol dire che la cardinalità è maggiore di 1 e allora si richiede come terza proprietà che sia sempre possibile prendere almeno 1:27:20 E allora si richiede come terza proprietà che sia sempre possibile prendere almeno uno degli elementi da sottoinsieme più grosso che non faccia parte su tensione più piccolo perché sennò sarebbe inutile e aggiungerlo al sottoinsieme più piccolo ingrandendolo e facendolo rimanere all'interno della collezione degli indipendenti questa una pulita decisamente più sofisticata e possiamo cercare di vedere se vale nel problema dello zaino Allora dobbiamo pensare di avere un insieme di due oggetti per esempio è un insieme di 3 oggetti e la proprietà dice che deve essere sempre possibile prendere uno dei tre oggetti e aggiungerlo al insieme da due se due insiemi inizialmente sono ammissibili l'insieme allargato dovrebbe rimanere ammissibile è chiaro che questo non funziona perché se voi avete due oggetti di volume 3 e la capacità e sei questo È ammissibile D'altra parte se da un'altra parte avete tre oggetti di volume 2 Anche questo è un sottoinsieme ammissibile perché la capacità e 6 e del maggiore è uguale a quella al volume totale di questo sotto insieme ma è anche chiaro che nessuno dei tre oggetti può essere preso è aggiunto al sotto insieme da due rimanendo inammissibilità perché si va a sfogare la capacità qualunque oggetto si aggiunga insieme insaturo la proprietà numero 3 nel caso del puledro zaino non vale ma si vede chiaramente che nel caso del problema dello zaino unitaria invece Vale perché Perché nel caso te lo zaino unitario il vincolo di capacità è diventato un vincolo di cardinalità e quindi se il sottoinsieme da tre ammissibile è quello da 2 ammissibile è chiaro che basta aggiungere qualunque elemento del sottoinsieme da 3 a quello da 2 per allargarlo in maniera ammissibile perché quel che qualunque sotto insieme da tre va bene di conseguenza nel caso dello zaino unitario Qualunque sia la funzione obiettivo additiva il l'algoritmo gridi fornisce una soluzione ottima per il problema di trovare il sottoinsieme di valore massimo questo è piuttosto interessante ed è alta interessante perché è estendibile a tanti altri problemi e la seconda parte della lezione dedicata a uno di questi problemi nei quali In effetti il l'algoritmo che ridi funziona esatta ed è un problema un pochino più interessante un pochino più sofisticato che non quello di prendere K oggetti che stiano in uno zaino perché nello zaino ce ne stanno al massimo K Allora questo problema è il problema dell' albero ricoprente minimo e lei voleva che in parte abbiamo visto nel senso che abbiamo parlato introducendo i graffi di alberi di sotto Gra fili coprenti eccetera eccetera che abbiamo un grafo non orientato come un insieme di vertici di un insieme di lati e diciamo che è connesso per il semplice motivo che stiamo cercando sottografi connessi e quindi è utile che il grafo sia connesso di per sé Praticamente il problema non avrebbe soluzione bene Che cos'è che stiamo poi abbiamo una funzione di costo questa funzione di costa una funzione che associa ad ogni lato del grafo un numero naturale che tu abbia semplicità in realtà si potrebbe tranquillamente stendere numeri negativi razionali eccetera è quello che stiamo cercando un sottografo di questo gruppo quindi un coppia contenente un insieme di vertici e un insieme di lati dove i vertici sono una parte dei vertici del grafo i lati anche e complessivamente insieme formano un grafo questa fotografia deve godere di te proprietà La prima è di essere ricoprente e questa è una qualità molto interessante semplice vuol dire che in realtà l'insieme dei vertici del fotografo coincide con quello del grafo Poi vogliamo che sottografo sia connesso Questo significa che qualunque coppia di vertici si prenda Ovviamente a questo punto nel drappo deve mettere un cammino all'interno del fotografo cioè dell'insieme di là TX e poi deve essere aciclico quindi non ci devono essere i camini chiusi che tornano indietro sullo stesso vertice fra tutti i fotografi di coprenti connessi aciclici e ricorderete che un sottografo un grafo in genere con esserci fico è un albero quindi fa tutti gli alberi ricoprenti si deve cercare un albero

che sia di costo totale minimo il costo di un albero è inteso come la somma dei costi dei lati che ne fanno parte domanda A cosa serve questo problema un sacco di cose per esempio a costruire delle reti elettriche di comunicazione eccetera eccetera potete facilmente immaginare una serie di applicazioni ci sono anche applicazioni più attratte comunque un modello molto semplice estremamente diffuso 1:32:10 Applicazioni ci sono anche applicazioni più attratte comunque un modello molto semplice estremamente diffuso Come facciamo a risolvere questo problema allora è chiaramente un problema di ottimizzazione combinatoria Perché Perché il numero di soluzioni è finito e si può anche vedere che le soluzioni sono dei sottoinsiemi e soluzioni sono dei sottografi Sì però siccome il sottoinsieme dei vertici e fisso possiamo dire che il La soluzione è il sottoinsieme dei lati e quindi insieme base di questo problema è l'insieme dei lati Cosa sono gli indipendenti cioè Quali sono i sottoinsiemi di lati che sono promettenti in quanto parti di un soluzione la risposta più semplice che si può dare non è l'unica in realtà ma è l'unica che vedremo in questo corso la risposta più semplice è che la ogni un indipendente un sottoinsieme ammissibile diciamo per la nostra ricerca è una foresta di coprente dove con festa si intende un fotografo aciclico non Richiediamo la connessione proprio perché se perdiamo un po' di lati non è più È vero che tutti i tutte le coppie di vertici sono connesse da cammini perché imponiamo Comunque la ciclicità perché se avete un sottoinsieme che chiude cicli è chiaro che nessun albero di corrente può contenere questo sottoinsieme di lati perché una volta introdotto un ciclo Non si può aggiungendo altri lati andare a romperlo Allora questo punto l'algoritmo Davide molto semplice si tratta di partire con un sottoinsieme di lati vuoto e poi di passo in passo andare a trovare il lato di costo minimo e * gli estremi O * invia * che abbia le seguenti proprietà cioè di essere non appartenente già alla soluzione di non essere già attaccato scattato in precedenza e di non chiudere cicli con la soluzione l'idea sostanzialmente Qual è l'idea è di come nel caso del dello zaino di andare a prendere gli elementi una volta e di andare a prendere solo quelli che non fanno uscire dallo spazio di ricerca che non violano i vincoli del sistema di indipendenza esattamente come nel caso dello zaino noi in realtà non andremo a cercare di passo in passo il lato di questo minimo che non chiude cicli perché sarebbe complicato Se vi ricordate nel caso dello zaino non siamo andati alla ricerca dell' elemento di valore massimo che rispettasse la capacità siamo andati a cercare elemento di costo massimo di valore massimo e poi abbiamo verificato se rispettava la capacità oppure no E nel caso in cui la aspettasse abbiamo aggiunto nel caso in cui non rispettasse l'abbiamo buttato via perché è abbastanza evidente che a questo punto non è più possibile usarlo per costruire una soluzione per caso dell'albero minimo l'idea Cioè andiamo in strada e non il lato di questo minimo che non chiude CTI con quelli li ha presi ma il lato di costo minimo tukur fra quelli non ancora considerati e se chiude cicli lo buttiamo Perché Perché non ci sarà nessun passo successivo in cui questo lato verrà buono Mentre invece se non più decidi lo prendiamo e la aggiungiamo Ecco perché l'algoritmo di kruskal che vedete qua sotto Quando scade il nome dell'autore di questo dello scopritore di questo algoritmo consiste nel partire con insieme XD lati vuoto con un insieme di lati e primo potenziali che è tutto l'insieme dei lati e finché ci sono ancora lati potenziale si va a estrarre con un admin il lato di questo minimo dai primo si elimina questo lato dai primo e se per caso il sottografo ICS Unito è asterisco e aciclico lo si aggiunge la soluzione altrimenti non si fa niente l'abbiamo semplicemente buttato via alla fine vestiti chiamo il sottografo formato da tutti i vertici e da lì insieme ics algoritmo è chiaramente l'algoritmo grilli con questa piccola differenza di andare a estrarre elementi minimi e poi testarli invece di andare strada elementi minimi da un sottoinsieme Ma è solo una questione di efficienza e di semplicità e si tratta di capire se funzionano se non funziona C'è una cosa da dire che qui il problema è di minimo non è di Massimo farla se avete già seguito la lezione di teoria saprete che l'algoritmo in realtà con le stesse condizioni

determina non solo il sottoinsieme massimo di valore massimo magri termina anche fra i sottoinsiemi massimali e cioè quelli col massimo numero di elementi determina quello di costo minimo è la funzione obiettivo non è di valore Ma eri costo si può usare anche in questa variante ed è in questa variante che noi lo stiamo effettivamente utilizzando Allora vediamo rapidamente esempio in questo esempio 1:37:00 Ed è in questa variante che non lo stiamo effettivamente utilizzando allora vediamo rapidamente un esempio in questo esempio abbiamo un grafo da 5 vertici e con dei costi sui lati stiamo applicando l'algoritmo di kruskal partiamo senza lati e andiamo di passo in passo e stare il lato di questo mio lato Il condominio è V1 V2 e costa uno chiaramente non chiude cicli perché un lato da solo non può fare cicli lo prendiamo poi abbiamo il lato V2 V3 V4 che costa 2 e insieme a B1 B2 non chiude cicli poi abbiamo il lato b1 b3 che connette i due lati precedenti ma non chiude cicli e quindi possiamo aggiungerlo e adesso salterebbe fuori il lato V2 V3 perché costa 5 questo lato chiude un ciclo V2 V3 V1 e quindi viene scattato Analogamente viene scartato di 24 viene scattato V1 V4 e finalmente possiamo passare agli altri agli ultimi tre lati cominciando con V3 v5 che non chiude nessun ciclo che viene aggiunto Invece di andare avanti ci fermiamo perché ci fermiamo perché una proprietà degli alberi che abbiamo già visto è che gli alberi hanno la caratteristica di avere un numero di lati che parli al numero dei vertici meno uno se il L'albero è ricoprente ha un numero didattiche Pavia il numero dei vertici del grafo meno 1 ma questo albero qui che abbiamo in mano adesso è fatto di 5 di Scusate di quattro lati cioè che esattamente è nemmeno uno lati e quindi non è possibile che ci siano alberi più grandi Cioè non è possibile che i lati rimanenti vadano aggiunti Quindi abbiamo scoperto anche eccezionalmente una condizione di terminazione anticipata rispetto algoritmo greedy generico che vale per tutti i problemi che terremo presente in nel caso dell'algoritmo dell' albero ricoprente minimo però a questo punto quello che dobbiamo porci il problema di come fare implementare tutto questo con l'efficienza migliore possibile come rappresentiamo insieme xe rappresentiamo l'insieme il primo in maniera da poter fare tutte queste operazioni nel modo più efficiente possibile vedete che ci sono delle aggiunte cancellazioni che sappiamo fare in tempo costante in un sacco di strutture dati diverse nei due operazioni più complicate sono la estrazione del minimo dell'atto di costo minimo dai primo e la valutazione di aciclicità dell'insieme ICS unito e * e Qui si tratta di capire un po' meglio come è possibile fare queste operazioni efficientemente Qui abbiamo a sinistra lo pseudocodice che abbiamo visto in precedenza c'è quello più astratto a destra Abbiamo la soluzione finale alla quale Stiamo arrivando che non è rigorosamente l'unica però in effetti è la più efficiente perlomeno nell'ambito di questo corso Ci sono algoritmi estremamente più sofisticati che vanno molto leggermente meglio allora la quello che si può fare per estrarre ad ogni passo il la questo minimo dai primo e ovviamente di gestire il primo comunica non con un maxi problema con un mini è una struttura ad albero binario quasi completo che ha una proprietà sui nodi esattamente complementare al maxi per cioè ogni nodo ha un etichettatura che è minore uguale a quella dei nodi figli di entrambi no di figli se ci sono entrambi dell'unico se ce n'è uno solo oppure se è una foglia siamo a posto così fatto questa cosa noi possiamo evidentemente costruire un hippie in anticipo e quindi sul versante destro abbiamo aggiunto un operazione di creazione di un hip ho segnato in rosso in maniera da far vedere che corrisponde all'estrazione del Carmine Dopodiché dobbiamo ad ogni passo andare a estrarre l'arco dico il lato di questo minimo che quel dolore e semplificato come admin ma è evidentemente un'operazione in tempo costante perché si tratta di andare a prendere l'elemento di indice 1 elip dopodiché dovremmo e cancellare questo elemento dall' e aggiornarlo e in Cancella minimo cosa ho fatto accorpato l'operazione di cancellazione Che vuol dire prendere l'ultimo elemento è sopra al kg al primo e V2 vera dimensione tempo costante e l'operazione di Aggiorna hip che sapete richiede un tempo logaritmico qui dovreste fermarmi e osservare che il

logaritmo è nella dimensione dell'heap e li contiene i lati quindi contiene m oggetti Ma siccome il logaritmo di m è minore uguale al logaritmo di n quadro Perché i lati sono minori uguali alle quadro Ecco quello che ritiene quadro e due volte il logaritmo di n quindi a meno di costanti moltiplicative abbiamo il tetano Gang è indicato è tutto qui no c'è il problema della ciclicità E allora qui 1:41:50 O c'è il problema della ciclicità Allora qui c'è un parte del corso che credo non abbiate ancora fatto E quindi che io però davo per scontata ed è la descrizione di e la descrizione la modellazione di una partizione di oggetti in sottoinsiemi per esempio della partizione dei vertici di un grafo in componenti connesse attraverso opportune strutture dati in maniera da fare certe operazioni nella modalità più una in modo più efficiente possibile Prima di tutto Fatemi osservare una cosa e torno indietro non avevo fatto salvare in precedenza ma adesso viene buona come si fa a dire che l'insieme di la TX unito e * non contiene cicli Allora l'idea è molto semplice se è* a due estremi che si chiamano si possono chiamare Lu * Ivi * se questi due vertici * sono già connessi da un cammino all'interno di ICS è chiaro che aggiungere il lato più di un ciclo ma se non sono connessi è chiaro che aggiungere il lato crea un unico possibile cammino da un asterisco* quindi la condizione case unito e * non contenga cicli equivale a condizione che gli scrivi * facciano parte di componenti connesse diverse all'interno DX Ed ecco che casca a fagiolo l'idea di avere una struttura dati che rappresenti le componenti connesse non come abbiamo fatto nel precedente esercitazione di laboratorio con un vettore che associa ad ogni elemento la l'indice della componente connessa di cui fa parte o l'indice del capo componente Ma come struttura più sofisticata che si chiama foresta eventualmente con bilanciamento e qui Vi darò una rapidissima idea in realtà ma che abbiate ripassato dato un'occhiata in precedenza questa parte qui abbiamo in oggetti che sono divisi in tre parti in tre sottoinsiemi e sono rappresentati con tre alberi Questi alberi non hanno nulla a che vedere con la topologia anche se gli oggetti sono ventralmente vertici quei lati lì quei quelle frecce quelle delle linee non rappresentano lati del grafo sono semplicemente dei collegamenti per dire che uno è 7 3768 eccetera fanno parte della stessa componente è messa in questo modo si vede che uno e tre sono appartenenti alla componente dominata da 76495 dalla componente dominata da 8 e due invece se ne sta per i fatti suoi una rappresentazione di questo tipo è facilmente descrivibile con il vettore padre che già abbiamo visto parlando della visita di grafi e questo vettore parallelo vedete indicato qua sotto l'unica differenza rispetto a ciò che abbiamo Pro utilizzato noi è che continueremo utilizzare il fatto che i vertici che non hanno un padre cioè detta vera 10789 invece di essere indicate da zero come le dispense di teoria noi indicheremo con il nome stesso del nodo cioè 7 avrà come padre 788 e 2^2 è del tutto equivalente Semplicemente ho voluto tenere la convenzione che abbiamo già usato in precedenza allora a questo punto quello che possiamo fare dire come si fa a gestire una cosa del genere con un vettore che cosa fa l'algoritmo di passo in passo Se invece all'inizio Noi abbiamo un certo numero di nodi per esempio 9 tutti disgiunti vuol dire che ognuno avrà come padre se stesso Dopodiché cominceremo a unirli E quand'è che gli uniremo molto semplice li uniremo nel momento in cui ci qua nel momento in cui la componente di cui fa parte o * e la componente di cui fa parte di * sono diverse se sono uguali saltiamo la riunificazione delle due componenti e l'aggiunta del lato V * di * alla soluzione in grande e non facciamo nulla andiamo a cercare il lato successivo se invece le due componenti sono diverse andiamo ad aggiungere il lato alla soluzione andiamo a unire le due componenti Come si fa a unire due componenti semplice sono alberi è proprio per quello che si usano alberi si aggancia la radice di una delle due alla radice dell'altra e qui c'è una regola di buon senso ma con una proprietà poi formale molto interessante e la regola dice che se prendete l'albero con meno elementi e lo attaccate all'albero con più elementi tendenzialmente otterrete un albero di profondità minore Che significa che quando esplorate l'albero stesso per capire a che componente fa parte la funzione Find che

abbiamo visto qui è che serve capire anche componente appartiene O * banalmente consigli 1:46:40 Mentre fa parte la funzione Find che abbiamo visto qui e che serve a capire anche componente appartiene O * banalmente consiste nel risalire l'albero stesso dal nodo che abbiamo D'Alberti CD Abbiamo indicato su fino alla radice nel momento in cui noi vogliamo fondere due alberi non facciamo altro che dire che il nodo 7 per esempio invece di avere se stesso come padre a 8 come padre questo in una singola operazione andando a cambiare una sola Celletta del vettore aggancia un albero all'altro C'è un teorema secondo cui questa cosa garantisce che la profondità di Questi alberi non supererà mai il logaritmo in base nei logaritmi in base 2 di n del del numero n di elementi del tempo insieme complessivo e questo evidentemente garantisce che quando facciamo una ricerca la ricerca non poteva mai via troppo tempo c'è oltretutto una possibilità ulteriore di migliorare questa cosa ma la trattiamo magari in seguito quando mettiamo le mani sul codice per non caricare troppo di concetti questa parte però l'idea sostanzialmente che a questo punto stiamo modificando ma stiamo ripetendo lo stesso pseudocodice astratto calando nella realtà dell'uso di opportune strutture dati e queste strutture dati ci consentiranno di ottenere che complessità Beh alcune operazioni in tempo costante All'inizio la creazione di un hip in tempo lineare nel numero dei lati di una foresta bilanciata in tempo lineare nei vertici perché si tratta di andare a riempire per ogni vertice assegnare lui stesso come padre di se stesso Dopodiché avremo un ciclo che viene eseguito quante volte non si sa perché c'è una condizione di terminazione anticipata però se siamo fortunati Potrebbe andare molto bene se siamo sfondo al limite ogni volta cancelliamo un lato lo eseguiremo m volte dentro il ciclo Noi abbiamo un'operazione in tempo costante per determinare il minimo il lato di corto minimo un'operazione delle operazioni logaritmi che per aggiornare il mini pco operazioni in tempo costante per fare l'unione di due alberi all'interno delle foreste con bilanciamento e per aumentare la soluzione queste operazioni defined sono un risalire le foreste stesse vi ho detto teta di Logan in realtà con la modifica che descriveremo in seguito si arriva a una funzione che è una funzione si crescente ma crescente in maniera talmente lenta che in realtà si può praticamente considerare il costante Ecco il perché di questo approssimativamente tenta di uno che chiaramente non ha un significato formale corretto ma da un punto di vista pratico lo potete considerare costante Allora se tutto avevo la complicità totale data da te tra MP3 3 n + m volte delle costanti più un logaritmo in totale abbiamo m per il logaritmo di n ed è questa la complessità dell'algoritmo di kruskal come lo andremo a implementare Ok allora passando all'implementazione che cosa abbiamo già pronto ho preparato già tutto il preparabile in maniera da concentrarmi sull'aspetto più algoritmico nonostante ciò un po' di disquisizioni di stile Più che altro con poca poca significato algoritmi con le fave li farò ne fa il 30 Noi abbiamo una serie di funzioni già pronte che serviranno al solito a capire dalla linea di comando Come si chiama il file del grafo che contiene i dati il file A questo Ovvio formato che è piuttosto simile a quello che abbiamo già usato in più precedentemente avevamo una semplice lista di lati Qui abbiamo invece scritti esplicitamente il numero dei nodi è il numero dei lati preceduti da n = DM = Dopodiché ciascun lato in realtà è seguito dal volume del costo e questo è quanto a questo punto avremo una funzione che carica Ida e li metto in una struttura grafo è che cos'è questa struttura grafo e digli a capire come implementare il grapo di implementazione del grafo ne abbiamo viste tre e vi ho detto ce ne sono molte altre che sono varianti di queste tre tendenzialmente una era la matrice di adiacenza tu non aveva la lista degli altri lati o degli archi e un'altra era invece il vettore delle liste di incidenza ho detto re delle favole Start quella più adatta al largo ritmo che stiamo cercando di implementare e qui noi Dobbiamo estrarre un elemento di costo minimo può sembrare che la ciclicità richieda di guardare le connessioni ma in realtà abbiamo visto che la realizzeremo con questi foreste con bilanciamento e quindi in effetti non ci serve sapere 1:51:30 Dare le connessioni ma

in realtà abbiamo visto che la realizzeremo con questi foreste con bilanciamento e quindi in effetti non ci serve sapere a che lati è incidente lati è adiacente Umberti o quali lati sono incidenti in un vertice Quindi in effetti il vettore del fuoco non è particolarmente utile e siccome in realtà Raddoppia l'occupazione in memoria e rende più difficile trovare l'elemento minimo perché divide i lati In tante liste diverse Beh allora forse non è la struttura adatta men che meno adatta alla matrice di adiacenza per il semplice motivo che Occupa un sacco di spazio è che se vogliamo trovare il minimo dobbiamo andare a correre n quadro Celle addirittura si potrebbe fare un Keep di n 4 Celle Ma allora tanto vale fare unip sugli insieme dei lati direttamente che sono m potrebbe essere molti di meno tutto questo ci porta a concludere che probabilmente rappresentazione a lista di archi è la migliore e infatti è quella che ho scelto dopodiché si potrebbe ancora discutere di cui vive su varianti dicendo Ma perché avere una lista di archi che in fondo dobbiamo semplicemente trovare un minimo dobbiamo mettere i lati In un heap perché non fare una tabella dirittura mi ricordo che le liste servono quando abbiamo bisogno di gestire un oggetto dinamico con aggiunte cancellazioni Qui non c'è quasi niente da aggiungere cancellare c'è da cancellare i lati dall'ip ma lo sappiamo già fare in un altro modo li cancelliamo da lì per non li cancelliamo dalla lista del grafo quindi si potrebbe dire tu a spingersi più avanti e dire no facciamo una tabella di latte però siccome non ho voluto introdurre la quarta è molto simile alla terza rappresentazione del grafo mi sono limitato a prendere la rappresentazione del grafo come lista di archi e a usarla così com'è Ho dovuto fare una modifica modifica In cosa consiste qui a un apparentemente non ci sono modifiche la modifica consiste nel fatto che se andiamo a vedere che cos'è un arco un arco Eh sì una coppia origine-destinazione con i suoi puntatori per andare a correre gli altri Archi ma anche un costo e quindi ho aggiunto questo elemento questo ha portato un po' di modifiche nella lettura degli archi che comporta la rottura di un costo nella scrittura di un arco e comporta Per quanto riguarda la libreria di grafico un porta che quando inseriamo un arco dobbiamo fornire non solo vergine destinazione ma anche un costo che quando andiamo a leggere gli estremi dell'Arco recuperiamo i due nodi ma abbiamo anche una funzione che legge il costo dell'Arco e lo ricava come valore intero e infine se proprio vogliamo andare a discutere tutte le modifiche che ho fatto nella stampa del grafo nella stampa del Grappa Eccola qua invece di stampare la coppia origine-destinazione tra parentesi tonde ho aggiunto anche il costo come richiesto del resto nella stampa finale anche se noi non faremo non useremo stampa grafo con questa funzione L'ho modificata solo per completezza e coerenza di tutto il di tutto il discorso quindi questo è ciò che già abbiamo e lo stiamo includendo nel nostro file e quindi abbiamo già presentato i dati in questo modo ripeto non è assolutamente detto che sia il modo migliore è un modo ragionevole e l'ho scelto in maniera da poter riutilizzare il più possibile con poche modifiche al codice che avevamo già pronto quindi più che altro era lo scopo di fare velocemente questa parte concentrarmi sulla angoli Allora dobbiamo rappresentare la soluzione Che cos'è la soluzione la soluzione è un sottoinsieme di dati Come si fa a piantare un sottoinsieme di lati e potremmo fare anche qui una lista è via ma che senso ha A noi la soluzione serve soltanto stamparla Quindi ci servono i due estremi E poi se vogliamo ci servono i costi no ci serve il costo totale della soluzione non c'è stato chiesto di scrivere il costo dei singoli lati quindi ho preferito a definire una tabella o meglio due tabelle una tabella di primi nodi una tabella di secondi no di di ciascuno degli archi è chiaro che avrei potuto definire la tabella di Archi ma questo avrebbe comportato la scrittura di una struttura Arco che contiene una coppia di nodi diversa da questa qui che contiene la coppia di no del costo e le posizioni e quindi ho preferito farla semplice e dare semplicemente questi due vettori Dopodiché trattandosi di una tabella c'è una dimensione allocata e c'è un valore corrente n l e ho aggiunto il valore della funzione obiettivo Questa è 1:56:20 Perimetrazione a locata e c'è un valore corrente n l e ho aggiunto il valore

della funzione obiettivo Questa è la descrizione della soluzione C'è poi una funzione che crea le soluzioni al locando tutto quello che si deve allocare e c'è una funzione che distrugge la soluzione The a Locarno tutto quello che c'è da dialogare astuccio mi sono detto Allora adesso possiamo procedere a implementare l'algoritmo e l'implementazione dell'algoritmo che cosa comporterà ti contiamo lo prima Dopodiché al solito vi inviterò farò uno schema come commento e poi vi inviterò a procedere più nel dettaglio comporta Chiaramente la chiamata di una funzione alla quale ho passato i dati ed alla quale recupero il risultato essendo risultato composto l'ho passato per indirizzo questa funzione kruskal in sostanza dovrà essere dichiarata e ho già scritto la dichiarazione e dovrà essere definita è la definizione l'ho scritta ma l'ho lasciata vuota A questo punto possiamo cominciare a entrare nella funzione stessa per discutere di che cosa sostanzialmente si tratta cioè Quali sono le parti di questo algoritmo è l'algoritmo dovrà chiaramente seguire la linea sequenza di idee che abbiamo visto in precedenza che cosa dobbiamo fare allora all'inizio abbiamo una soluzione vuota la soluzione volta ce l'abbiamo già perché la funzione crea soluzione crea una soluzione vuota dobbiamo inizializzare un libro che contiene tutti i lati del grafo Ovviamente questo punto si pone il problema Ma in che senso come abbiamo discusso nella scorsa lezione potrebbe essere uniti perché contiene i valori dei costi ma se lavoriamo soltanto sui costi non riusciamo a determinare Chi sono gli estremi dell'Arco potrebbe essere un fatto i lati Ma allora quando scambiamo due lati dovremmo muovere un sacco di strutture abbiamo parlato Insomma di indiretto Per farla breve Unipi indiretto abbiamo implementato in precedenza come vettore di interi perché vettore di interi perché gli indici interi ci consentivano poi di accedere del vettore dei dati che restava nel suo ordine iniziale e che forniva i dati stessi rimaneva intatto qui però i dati non sono in un vettore sono in una lista volutamente allora ho fatto questo secondo caso è chiaro che potremmo mettere i dati in un vettore e andare a fare un IP indiretto sul vettore ci sarebbe nulla di male salvo il fatto che occupiamo più spazio Ma allora tanto vale in realtà andare a costruirsi un indiretto fatto di puntatori puntatori ai lati del grappo Quindi quello che in sostanza faremo è costruisce costruire un heap prima costruiamo un vettore di puntatori ai lati del grafo Dopodiché questo simpatico elemento questo simpatico vettore lo andremo a trasf indiretto sui lati del grafo attraverso il vettore di puntatori E questa sarà una seconda cosa che dobbiamo fare poi che altro faremo andremo con lui nella foresta con bilanciamento Cosa vuol dire costruire una foresta con bilanciamento pianta visto che vuol dire avere un vettore in cui inizialmente ogni nodo punta se stesso Però speriamo in termini astratti costruisce una foresta con bilanciamento sui vertici del grafo Ok A questo punto dobbiamo sostanzialmente procedere e dobbiamo effettivamente entrare nella ritmo delle proprio per lui dice che fintanto che la soluzione non non è completata c'è fintanto che la soluzione non contiene esattamente nemmeno un elementi quindi finché la soluzione non contiene ennemenouno lati siamo sicuri che ci arriveremo a questa condizione cioè in sostanza il grafo a un albero ricoprente è connesso Beh Abbiamo ipotizzato di sì ma Mettiamoci al sicuro quindi che stiamo a 2:01:10 Devo ricoprente è connesso Beh Abbiamo ipotizzato di sì ma Mettiamoci al sicuro Quindi te Stiamo anche il fatto che nel Lip ci siano ancora elementi per Finché ci sono lati da esplorare è la soluzione non contiene n meno 1 lati Ecco finché valgono tutte queste cose qui che cosa faremo passo per passo la prima cosa che andremo a fare E andare a prendere trova il lato di costo minimo siamo anche Davide i nomi abbiamo chiamato e primo L'insieme in cui stiamo andando a cercare le cose abbiamo chiamato F la foresta stessa Quindi finché ci sono lati da esplorare in e primo trova il lato di costo minimo nel primo ho trovato il lato di questo minimo in il primo dobbiamo andare a vedere che cosa devo andare a vedere se i due estremi di questo lato dove dobbiamo andare a cancellarlo Sì potremmo anche dire cancella il lato dai primo e poi dobbiamo andare interrogarci 62 estremi stanno nella stessa componente o in com-

ponenti diverse quindi dobbiamo andare a chiederci se i due estremi del lato stanno in componenti diverse della foresta con bilanciamento Ecco che allora è solo allora potremo andare ad aggiungere il lato alla soluzione e a unire le due componenti quindi in questo caso andremo a aggiungere il lato alla soluzione e andremo a unire unisce le due componenti della foresta a questo punto abbiamo in realtà finito lago ritmo usciamo dal famoso ciclo e da un punto di vista informatico Siccome abbiamo usato un po' di strutture Dati dobbiamo liberarsene quindi andiamo ad allocare Lip indiretto e la foresta con bilanciamento Questo è quello che dobbiamo fare a questo punto io lascerei il compito di riempire queste cose e poi vi darò la mia soluzione l'unica cosa che forse non ho sottolineato a sufficienza e che per quanto riguarda la foresta è vero che ci serve un vettore padre ma per poter fare l'unione intelligente quella bilanciata abbiamo anche bisogno di sapere Quanti elementi contiene questo albero e Quanti elementi contiene quest'altro e Quanti elementi contiene il terzo Come si fa a fare questa cosa molto semplice si può costruire un altro vettore di valori numerici che è lungo Quanti vertici perché a priori non sappiamo quali vertici diventeranno radici e in corrispondenza ciascuna radice va a scrivere la cardinalità del carina età dell'albero che è radicato in quel vertice Ovviamente se il vertice non è una radice non abbiamo nessun dato di interesse in quella cella lì ok A questo punto fra poco vi darò la mia soluzione portiamo pure dall'inizio abbiamo bisogno di un heap indiretto è costituito da un vettore vettore che punta ai dati del grafo questo metodo e quindi è lungo m va da 1 m e lungo mp1 lati Che genere di vettore è un vettore di puntatori lati noi abbiamo il tipo puntatore al lato e precisamente questo tipo dentro Lee starkey si chiama posar co questo ci permette di essere anche come dire trasparenti di nascondere l'implementazione Nel senso che se per caso la lista fosse fatta con i vettori Allora il posarlo sarebbe un numero intero è tutto Dovrebbe funzionare lo stesso Non che sia particolarmente importante in questo caso allora io ho definito qua sopra un vettore riposare Un po' alla solita maniera come vettore dinamico c'è pure puntatori a posare gli faccio servare che un puntatore a posizione diventa un doppio puntatore a dark scritto come Arco doppio puntatore è un po' illeggibile scritto così Diciamo che non è assolutamente necessario farlo ma è un po' Più chiaro di che cosa stiamo parlando questo lettore gli va dato un nome potremmo chiamarlo h visto che è un ippopotamo che 2:06:00 Appuntamento è necessario farlo ma è un po' Più chiaro di che cosa stiamo parlando questo vettore viva dato un nome possiamo chiamarlo h visto che ogni poi potremmo chiamarlo e primo visto che quello è ciò è il nome che abbiamo usato nel nostro del nostro pseudocodice corrisponde esattamente a questa struttura e primo io qua lo chiamo posa e essendo un vettore devo banalmente andare a darlo Carlo lo alloco o con una Callo con una mano che solito discorso la lunghezza di questo vettore è data dal numero di lati m del grafo più uno perché indicizzato a partire da uno e singolo elemento è un po' d'Arco fatta la locazione dobbiamo ovviamente controllare che sia andata a buon fine e eventualmente lanciare il solito messaggio di errore errore nella locazione del per esempio dell'Iper indiretto dei lati qui Possiamo chiudere ed eventualmente uscire se c'è stato errori Fatto questo abbiamo un bel vettore pieno di puntatori a nulla ma no In realtà i puntatori dobbiamo fissarli a cosa dobbiamo fissarli dobbiamo consegnare ai lati del grafo Come si fa a scorrere i dati del grafo e 6 lati del grafo si scopano con le funzioni per scorrere i dati del grappo cioè con primo Arco finirà anche sul palco quindi ci serve una variabile cursore che chiameremo di Pippo posarlo che chiameremo Pia come abbiamo fatto tante volte questa variabile Pia partirà dal primo Arco del grafo che stiamo considerando che è il grafo g e noi stiamo passando il puntatore e andava avanti finché non siamo in fondo alla lista degli archi fine Archi del grafo PG la posizione Pia e ogni volta andiamo sostituire ad ogni Arco Pia la successione la successiva posizione nel grafo Pigi e questo è il la scrittura il ciclo che scorre tutti gli archi per ciascuno di questi anche noi in pose di Equi ci serve un numero intero a 15 è un numero che parte da uno che va

avanti allora questo numero intero Inizialmente è uno Questo è un modo per avere una Multipla inizializzazione basta separare con virgole i termi e analogamente possiamo andare a fare un multiplo aggiornamento con i più più quindi Così noi andiamo scrivere la posizione dell'Arco Pia se mi dà fastidio avere una Multipla inizializzazione una Multipla aggiornamento Basta che scriviate disposti a te una prima è che andiate a mettere l'aggiornamento all'interno del ciclo questo è solo un modo possibile di scrivere allora questo punto dobbiamo costruire un IP indiretto su questo benedetto vettore il che come dire non è banale ma si tratta semplicemente di andare a chiamare l'opportuna funzione non l'abbiamo ancora realizzata ma Noi procediamo top Town e quindi andremo a chiamare a creare un indiretto come funzionava indiretto nella lezione in cui L'abbiamo usato c'è nella lezione precedente Mi correggo nella lezione della prima parte nella prima parte di questa lezione Allora andiamo a prendere il naso e la funzione crea Eccola qua riceveva il vettore intero degli indici il vettore dei dati da ordinare e la dimensione Qui la situazione è simile ma un po' diversa io fatturo e poi ci aggiorniamo su un pochino la funzione crea hip indiretto no cominciamo a scrivere direttamente la chiamata ha bisogno sicuramente del vettore che costituisce lì perciò del vettore poze E questo va a sostituire il vettore degli indici Dopodiché ci sono i dati ma i dati non erano non sono un vettore erano un vettore ma adesso sono un grafo è una lista e d'altra parte abbiamo la dimensione Qual è la dimensione qua dentro Allora dimensione qui dentro lo sappiamo è data dal numero dei lati del grafo e i dati stanno da qualche parte all'interno del grappo cui potremmo scrivere PC freccia l volendo però se facessimo questo è alta no potremmo anche scriverlo Semplicemente io decido per semplicità di dire che l'IP lavora posizioni 2:10:50 No potremmo anche scriverlo Semplicemente io decido per semplicità di dire che l'IP lavora sulle posizioni di lunghezza m e va a cercarsi i suoi dati all'interno del grafo in qualche modo questo cosa vuol dire vuol dire che nel momento in cui andiamo a dichiarare e poi andremo a definire la nostra funzione col solito meccanismo tre fasi mi ha scritto la chiamata passiamo a scrivere la dichiarazione la dichiarazione crea un IP indiretto attraverso il vettore di puntatori pose Unipi indiretto di lunghezza m di lunghezza questa potremmo anche chiamarla n visto che questa è una funzione che lavora sul libro e non deve sapere di cosa si tratti attraverso il vettore di puntatori poso Su che cosa sui lati del grafo cugini in questo modo sappiamo tutto questa funzione intuisce un Void lavora su un vettore di posare Arco riceve un intero che abbiamo detto di chiamare n localmente che è la lunghezza attuale dell'hip e lavora su un contatore agrafo p e g quindi possiamo scendere e creare momentaneamente vuota questa funzione crea IP indiretto poi ci Scriveremo qua dentro e la riempiremo Ok creato l'IP Che cos'è che ci resta da fare ceretta da costruire la foresta Allora Cosa vuol dire costruire la foresta con bilanciamento come vi dicevo vuol dire andare a costruirsi questo vettore padre ma anche parallelamente un vettore di cardinalità detto Venom per esempio Allora ci servono due vettori di interi potremmo definire una struttura dati astratta potremmo fare Addirittura una libreria A parte sarebbe più bello però sto procedendo molto semplicemente e velocemente per il momento andiamo così mi faccio un vettore di interi padre è un vettore di interi num e questi due vettori vado a dallo Carmeli della Dimensione corretta Qual è la dimensione corretta Allora dimensione corretta intanto si tratta di un vettore di interi e la dimensione corretta è il numero dei nodi del grafo e siccome li indicizzo da1n se aggiungo uno al solito i singoli termini sono valori interi e c'è il solito controllo che la locazione sia andata a buon fine ovviamente la locazione di padre Errore nella locazione del vettore padre Analogamente Procederemo Per quanto riguarda il vettore delle cardinalità numa che anche quella è un vettore della stessa lunghezza fatto di interi e questo è il lettore num fatto questo dobbiamo riempirli inizializzare lì cioè costruire questa foresta sui vertici Cosa significa significa che dobbiamo scorrere i vertici intanto che sono numeri interi e per i che va da 1 fino a il numero dei nodi del grafo quello che

dobbiamo fare in pratica e andare ad assegnare un valore a padre di è un valore a nome di Qual è il valore del padre di easy come sono tutte radici ho detto che non userò lo 0-0 la convenzione che sei un no da una radice a se stesso come padre e d'altra parte non di è il numero di nodi dell'albero che stiamo rappresentando che è esattamente uno sono tutti i nodi degeneri tutti alberi costituiti soltanto dalla radice Siccome abbiamo avvocato un bel po' di cose fin da subito avremmo dovuto c'è il commento che ce lo ricorda ma tutto sommato è meglio farla immediatamente avremmo dovuto procedere ad locare pose other locale padre e infine other locale num e possiamo finalmente passare al ciclo deve proprio che cosa richiede è un ciclo while che a due condizioni di arresto di permanenza le due condizioni di permanenza corrispondono al fatto prima di tutto che ci sia ancora dei lati su cui vale la pena di ragionare questo significa la dimensione dell'heap ora la dimensione del libro noi qui abbiamo indicata Pavia PGM ma dentro il ciclo andrà a Pian pianino decrescendo quindi ci serve una variabile per andare a capire quanti lati ci sono 2:15:40 Dentro il ciclo andrà pian pianino dei crescendo quindi ci serve una variabile per andare a capire quanti lati ci sono ancora nel ciclo questa variabile io la chiamerò m est per dire che sono il numero dei lati esterno specie di m primo e ovviamente quando il tutto è appena cominciato questo m est coincide con Pigi 3 cm con numero dei lati dell'intero grafo fatto questo quand'è che ci fermeremo ci fermeremo o quando emmest è zero quindi la condizione di permanenza è che rimanga strettamente maggiore e l'altra condizione è che il la cardinalità della soluzione non sia troppo alta Com'è la candidata della soluzione andiamo a vederlo il ricordo che la soluzione a un campo nl che viene inizializzato a 0 e ti andrà vendo dovrà essere fatto crescere via via che si aggiunge un elemento quindi il la condizione cui è che PS freccia nl non superi il numero dei nodi del grafo meno 1 perché quando diventa uguale Abbiamo già finito di costruire il nostro albero ricoprente fatto questo dovremmo andare a trovare il lato di costo minimo molto semplice perché il lato di corto minimo corrisponde al primo elemento dell'ip corrisponde a poso di uno che cos'è Poldi 1 porzione un po' d'Arco un po' d'Arco è la posizione dell'Arco minimo potrei In effetti usare PDA a giorno aggiungo un'altra variabile in realtà è inutile per sottolineare che è l'arco minimo e vado a scrivere che pia Amin è la posizione uno del nostro hip a questo punto avendo la posizione avendo il telefono e potremmo andare ad accedere agli estremi qualcosa da fare Sara cancellare il lato possiamo cancellarlo Sì perché tanto abbiamo salvato in p Amin e tanto il lato sta dentro il drappo Come si fa a cancellare un lato da un heap lo si sovrascrive con l'elemento finale Cioè sostanzialmente si va a dire che pose di uno è uguale a pose di Quanti sono gli elementi attualmente nel bip m esterno e finalmente riduciamo di uno il numero dei lati esterni questo però distrugge la proprietà hip e quindi va a via giornata ed ecco che cacca la seconda funzione di gestione del Lip dobbiamo aggiornare li indiretto e stagionali indiretto nel nostro precedente esempio lavorava su un vettore di interi e adesso l'avevamo Invece sul vettore delle posizioni lavorava sui dati direttamente invece adesso lavoreremo sul Drappo lavorava sul numero di nodi sulla dimensione dell'heap quindi m esterno sul campo e sulla posizione uno che abbiamo attualmente modificato non te le devo scambiato dimensione e tati semplicemente perché in questo modo tengo insieme Lip e la sua dimensione e lascio un pochino a parte i dati che sono qui sono qualcosa di diverso Mentre qui tutto sommato I due vettori andavano in parallelo mi sembra più ragionevole scrivono in questo modo sono ok Cosa vuol dire questo vuol dire che dobbiamo andare ad aggiungere in alto la dichiarazione e poi dobbiamo andare in basso a scrivere la definizione che cosa fa questa funzione Aggiorna hip andiamo un attimino a ispirarci questa funzione è un fatto che aggiornare Unipi indiretto eccetera eccetera sarà un qualcosa di molto simile alla creazione Quest funzione va ad aggiornare un IP indiretto di Lunghezza n che è questo lato qua attraverso il vettore di puntatori pose sui lati del grafo g a partire dalla posizione o dall'indice Questo è quello che fa la funzione che è una funzione Void che riceve

in ingresso un Vibo salco che riceve la dimensione n dell'hip riceve il grafo * PC e riceve l'intero in sul quale si andrà a lavorare e qui poi dobbiamo vedere che cosa scrivere esattamente per il momento questa funzione la lasciamo tranquillamente vuota e ho finito No c'è tutta la parte da gestire dell'aggiunta e della valuta 2:20:30 E ho finito No c'è tutta la parte da gestire della giunta e della valutazione se il lato sia chiuso un ciclo non chiude un ciclo Che cos'è che bisogna sostanzialmente fare determinare le due componenti e poi andare a fare quest'operazione io qua sotto accorpato un po' ho chiamato F * FV * il risultato di queste due funzioni L'ho fatto per non allungare lo pseudocodice qui evidentemente invece dovremmo andare un pochino più espliciti quindi avrò bisogno di determinare la componente in cui sta una componente in cui stavi ma soprattutto ancora bisogno di determinare wavy perché io ho il puntatore Allora andiamo con calma la prima cosa da fare a determinare gli estremi dell'Arco e questo si fa Se vi ricordate la vostra libreria con la funzione legge estremi Arco ovviamente potrei andare a leggere direttamente i puntatori a chi spia freccia origine Pia freccia destinazione sto semplicemente cercando di tenere valido quello che ho detto in precedenza leggeremo gli estremi dell'Arco Pia andremo a scrivere inopportuni nodi o è Memoli uibi percorrenza conosciuto codice di due estremi ovviamente mi servono delle variabili nodo v e v e a questo punto ho letto i due estremi dell'Arco avendo letto i due estremi dell'Arco posso cercare le componenti le componenti Ti amo le potremmo scrivere che sono ancora dei nodo arbitrariamente decido che sono degli imperi tanto sono la stessa cosa e sono la componente di U e la componente DV l'albero di cui fa parte o l'altra di cui fa parte li dicevo potremmo decidere che siano dei nodi semplicemente perché sapete che ogni componente ha come indice il suo nodo capo la sua radice allora come facciamo a determinare queste due funzioni e qui bisogna andare ad applicare le funzioni di gestione di questa struttura qua che si chiamano in gergo funzione Find e funzione Union Come del resto qui abbiamo determinato la funzione Find applicata al nodo u dovrà lavorare sulla struttura questa struttura è costituita da due detto padre e dal vettore non ci potremmo chiedere se valga la pena di indicare Quanto sono lunghi questi vettori in realtà non serve perché la funzione Find non fa altro che risalire di elemento in elemento attraverso questo vettore quindi non ha veramente bisogno di sapere Quanto sono lunghi i vettori però è un dubbio che ci potrebbe giustamente venire Analogamente la funzione ccv viene determinata applicando Find al nodo vi possiamo testare se per caso ci è uguale o diverso da cd se è uguale non si fa nulla perché abbiamo già cancellato il lato e non ci interessa se invece è diverso Ecco che possiamo proseguire a fare le ultime operazioni che ci mancano queste ultime Premiazioni sono l'aggiunta del lato alla soluzione coerente e l'Unione delle due componenti per fare tutto questo però abbiamo aggiunto una chiamata quindi dobbiamo fare la dichiarazione in alto e la definizione in basso che cosa fa questa funzione Find Allora questa funzione find in sostanza trova nella componente la componente la componente a cui appartiene l'elemento u nella foresta con bilanciamento descritta da padre che num è una funzione che restituisce un intero che riceve un nodo che restituisce un wint è un altro vinto la Copiamo qua sotto e ci prepariamo poi A riempirla se vogliamo tenere Nessun commento qui Dobbiamo inventarci un risultato che per il momento metto a zero che deve stare bene attenti a non lasciare ovviamente a zero Quindi qua dentro dobbiamo finalmente aggiungere il lato alla soluzione corrente Come si fa ad aggiungere un lato e una soluzione andiamo a vedere come è fatta una soluzione una soluzione alla tabella e quindi è la solita meccanismo di aggiunta una tabella 2:25:20 Devo vedere come è fatta una soluzione una soluzione una tabella e quindi la solita meccanismo di aggiunta la tabella dobbiamo incrementare nl in teoria testando che la dimensione non sia eccessiva e poi andare a scrivere nei campi volevi due quello che viene effettivamente passato potremmo fare una funzione se in blu Massimo una soluzione in una libreria per la gestione di soluzioni concrete soluzioni distrugge soluzioni Aggiungi

elemento e stampa potremmo farlo ma ancora una volta vado per la soluzione più grezza è più semplice per raggiungere il lato alla soluzione coerente in pratica Che cosa devo fare prima cosa devo andare a incrementare la dimensione della soluzione stessa quindi PS freccia e nelle cresce Dopodiché all'interno di quella posizione in nel vettore di uno in posizione psnl io vado a scrivere il primo estremo che è quello che abbiamo chiamato u è la stessa cosa faccio con V sul secondo estremo quindi secondo estremo imposizioni PS BNL è uguale a via ho finito No la soluzione Contiene anche il valore della funzione obiettivo che va crescendo di quanto del costo dell'Arco il costo della non l'abbiamo ancora trovato ma c'è una funzione che ce l'ho da te e quindi è piuttosto semplice andare a caricarsi e lo andiamo a leggere il costo nel grafo PG dell'Arco Pia e siamo tutto sommato a posto anche da questo punto di vista cosa ci rimane ci rimane da fare la riunione delle due componenti la componente C una componente CV nella foresta descritta dai vettori padre e non vedete come si legge comodamente anche il codice e a questo punto la nostra dichiarazione che cosa dirà dirà che andiamo a unire le componenti radicate in c u e CV della foresta con bilanciamento descritta da padre e num questa funzione è una poid che riceve i due interi c u e c v e i due vettori di interi padre e num la prendiamo la Copiamo qua sotto e per il momento lasciamo vuota allora questo è un buon momento per vedere se abbiamo fatto delle battiture sbagliate si poteva fare anche prima Probabilmente ho un po' tagliando i tempi per quanto possibile e quindi possiamo andare a cercare kruskal zero e a vedere se il risultato funziona sto facendo un errore l'errore sta nel fatto C'è un piccolo Dov'è qui ma c'è una serie di funzioni che non sono riconosciute per il banale motivo che abbiamo incluso una bella serie di librerie Allora cominciamo a includere le librerie giuste cominciamo a dire che stiamo includendo la libreria drappo l a punto c e la libreria lista archi.co e se avessimo rappresentato Lip la soluzione piuttosto che la struttura foresta con bilanciamento tante librerie Qui abbiamo un lungo filotto di file da compilare cosa c'è un Warning di una funzione Pia min che è stata accettata ma non è stata usata e infatti ho fatto un errore un piccolo errore legato al fatto che qui ho letto il posto dell'Arco Pia non dell'Arco Pia min e anche qui ho letto gli estremi di P anziché pa1000 probabilmente avrei dovuto tenermi Pia e non aggiungere Questa seconda variabile Ma così fosse più chiaro A questo punto credo notate che comunque l'avvertimento è stato decisamente utile perché il devo non era di semantica quella è una variabile ci si può tranquillamente andare a scrivere su Maeva di sintassi era di sempre lei non era di sintassi ma era di semantica Ovviamente il nostro codice non fa nulla Salvo che controllare che lanieri comando sia corretta e salvo che stampare qualcosa ma non c'è una soluzione la soluzione vuota per il semplice motivo che in realtà le varie funzioni non fanno nulla e quindi qui succedono cose varie ed eventuali Ok Allora a questo punto potremmo anche indagare su cosa succede è Naturalmente quello che succede probabilmente è che entriamo nel ciclo Perché m est è alto è il numero di nld elementi della soluzione 0 Dopodiché gli elementi vengono presi dal famoso

2:30:10 Test è alto è il numero di nld elementi da soluzione 0 Dopodiché gli elementi vengono presi dal famoso hip che poi in realtà non è unip perché questa funzione non fa nulla ma è un vettore e vengono presi e vengono via via cancellati uno per uno dove In questa operazione qui che va sovrascrivere e che va a ridurre m est per cui dopo un certo numero di cicli si esce tranquillamente ed è tutto ragionevole ci mancano le funzioni a livello più basso la gestione del Lippi indiretto la gestione delle foreste con bilanciamento E allora possiamo procedere ispiriamoci all'indiretto che abbiamo già visto È chiaro che la creazione funziona molto in maniera simile come cioè chiamando l'aggiornamento da metà del vettore in giù che cosa vuol dire questo vuol dire che Procederemo da n Messi e poi giusto Mi ha chiamato è nella dimensione giù fino uno chiamando a giornali indiretto questa volta giornali indiretto lavora sul

vettore posizione sulla dimensione n lavora sul grafo PG e lavora sulla posizione E questo dovrebbe essere tranquillo e ragionevole compilata non ci sono particolari errori scendiamo a occuparsi della giornamento perché è decisamente più complicato come funzionava l'aggiornamento la linea esattamente la stessa determinare indice di sinistra è indice di destra ipotizzare che l'indice Massimo sia quello corrente e poi andare a vedere cosa succede quindi sicuramente queste parti Qui ci servono a questo punto quello che dobbiamo fare e dire ma l'elemento di sinistro e non potrebbe essere migliore il figlio di sinistra che non il padre e qui c'è una serie di considerazioni da fare punto primo che l'indice di sinistra non è un indice Ma è un po' Evidentemente non deve essere maggiore deve essere minore Perché questo è un mini non è un maxip comincio a copiare e poi qui ci sarà da fare diverse modifiche Allora intanto deve essere un minore poi abbiamo detto che non si chiama indice si chiama POS e questo vabbè è una cosa formale ma la cosa più importante è che non abbiamo un vettore B e questo non è un indice Questo è un contatore per cui non ha senso la scrittura avvii pose non esiste noi dobbiamo andare a trovare che cosa andiamo a trovare dobbiamo andare trovare il costo di questo arco qua ci serve il costo di può esse' che non è banalissimo come si fa a leggere il costo dell'Arco potesse l'abbiamo appena fatto poco fa in crusca Abbiamo appena letto il costo della Coppi Amin Quindi in realtà quello che dobbiamo effettivamente fare E andare a leggere Il costo nel grafo PG dell'Arco Boz S Questo potrebbe essere il peso è scritto tranquillamente qua dentro anche perché poi il costo non lo usiamo più il corso di quest'arco Quindi potremmo procedere in questo modo sostituendolo e potremmo fare esattamente la stessa cosa sulla posizione di Max perché dico potremmo perché c'è un insidioso problema insidioso problema sta nel fatto che noi dobbiamo essere sicuri che se esista che sia minore uguale n Allora in realtà non è sempre assolutamente garantito per sicurezza diciamo questo credo che dipenda un po' dalla macchina per sicurezza conviene annidare questi due test e quindi conviene andare a dire che se esiste un indice di sinistra allora vado a leggere il suo costo e lo vado a confrontare con il costo di Imax lo vado a confrontare con quest'altro costo qui nel caso se questa cosa vale andrò a sostituire Peccato che sia ci sia da cancellare una (nel caso vado a sangani che Imax è uguale a s questo qui che va Max Se non ho sbagliato e poi qui abbiamo chiamato il Max ma 2:35:00 Se non ho sbagliato e poi qui abbiamo chiamato i Max ma sarebbe più corretto Ricordarsi che stiamo parlando di un minimo Quindi non è che sia sbagliato punto-informatico ma la lettura non ne guadagna sicuramente a questo punto se non ho sbagliato facciamo qualcosa di leggermente diverso dei miei appunti Quindi è possibile anche che io abbia sbagliato Facciamo la stessa cosa sul figlio di destra quindi andiamo a confrontare la posizione dell'elemento destro con la con il costo del lago che sta nella posizione attualmente minima e nel caso andiamo a rimpiazzare I mean di tipo Ok A questo punto A questo punto dopo ti diamo che vada tutto che sia equivalente a quello che avevo preparato A questo punto ciò che dobbiamo vedere Semplicemente se uno dei due figli ha superato il padre Cioè ha un costo più basso di quello del padre Cioè se I mean è diverso dai perché in questo caso ciò che dovremo fare Sara scambiare all'interno del libro Le posizioni del figlio del padre quindi applicheremo con passaggio per indirizzo a posti i min e a post di una bella funzione scambia che sostituisca il ciascun valore con quello dell'altro e c'ho fatto andremo ricorsivamente richiamare la funzione che aggiorna lì indiretto suppose sulla dimensione n sul grafo Pigi e sulla posizione questa volta in in tutto questo funziona salvo il fatto che la funzione scambia non è stata dichiarata e quindi dobbiamo andare in alto ad aggiungere il solito prototipo e in basso a poi A riempirla Quindi la nostra funzione scambia Che cosa fa in sostanza non fa altro che andare a scambiare fra loro cambia i valori di due variabili che questa volta però sono di tipo posar co non sono degli interi quindi andremo con una funzione Void a cambiare un po' Sarco puntatore p a è un po' Sarco puntato i b e il meccanismo che lo so esattamente quello che già ben conosciamo tant'è che in effetti potremmo anche

direttamente implementare l'ha visto che si tratta solo di tre passaggi quindi si tratta di avere una variabile posar contemporanea di andare a salvarci il l'oggetto puntato da Pia di andare a sovrascrivere l'oggetto puntato da Pia con l'oggetto puntato da pipì e poi sovrascrivere l'oggetto puntato da pipì col solito oggetto temporaneo a questo punto in teoria Tutto ciò che riguarda l'IP è a posto dovremmo sarebbe consigliabile fermarsi compilare cercare di farsi stampare qualcosa Evidentemente non una soluzione perché la fai in della Union continuano a non funzionare ma potremmo In effetti farsi stampare per esempio gli archi con i loro costi per vedere che vanno via via progressivamente decrementando questo non costerebbe tanto ma siamo che procedo tanto la fai sono piuttosto elementari come implementazione Allora la funzione Find In che cosa consiste funzione Find Cominciamo con la Junior perché è più semplice In effetti la funzione Union consiste sostanzialmente nel andare a prendere una delle due radici e andare a assegnare come padre a quella va dice l'altra dice però dobbiamo anche andare a segnare a sistemare il vettore num e dobbiamo decidere quale delle due radici deve fare riferimento all'altra come padre abbiamo detto che tutto questo dipende dal cardinalità In pratica per inciso sto ipotizzando che ci vuoi ci vi siano diverse Tanto la Union viene chiamata in effetti viene chiamata subito dopo un test che garantisce che siano effettivamente diverse se trovate qualche discrepanza con le dispense è proprio perché le dispense fanno dei test prima che io faccio dopo di conseguenza che cosa stiamo dicendo Stiamo cercando di capire se la componente c u non se la componente CV se il 2:39:50 La componente c u non se la componente c u se il numero gli elementi della componente c u è più piccolo del numero di elementi della componente CV Allora quello che dobbiamo fare e appendere la componente più alla componente Civic Altrimenti dobbiamo fare esattamente il contrario come si fa ad appendere. semplice se dobbiamo appendere la componente CV alla cvd avremmo che il padre dcu è il nodo CV dopodiché la numerosità di più e dici mi cambia la vita dcw diventa nulla perché in sostanza non è più capo componente e però non me lo dite a 10 viva incrementarsi dcw Quindi prima bisogna aggiornare Questa seconda bisogna sommare la numerosità di più a stretto rigore non è veramente necessario azzerare questa non è necessario perché se andiamo a consultare il vettore num è soltanto su delle radici e quindi in effetti Siccome il padre di più non è più lo stesso è perfettamente inutile che andiamo però questa questo azzeramento sembravi pulisce la struttura stiamo perdendo un po' di tempo facciamo per niente ma semplicemente per Maggiore chiarezza qui facciamo esattamente il contrario cioè tanti ci andiamo a dire che il padre di CVC che la numerosità dcw cresce perché prende tutti gli elementi decisivi e che numero sta dici di invece diventa zero e a questo punto abbiamo sistemato la Union la funzione Find mi diceva è più complicata In realtà nella sua forma elementare È più semplice perché si tratta molto banalmente di risalire di padre in padre fino a che non si arriva un nodo che ha come padre se stesso quindi la forma più semplice si potrebbe scrivere in questo modo finché coincide con un Anzi finché è diverso Continuiamo Ci fermiamo quando diventa uguale E come continuiamo semplice andando a sostituire a un suo padre e questo sarebbe quanto se qualcuno fosse inquietato dal fatto che sto usando il dato tu e lo sto continuamente modificando Tenete conto che questa è una copia non sto andando a cambiare la u che c'è fuori nelle dispense non è così perché le dispense si usa un linguaggio che passa i parametri per indirizzo però questa è la versione base alla fine della quale si va a restituire il valore di questo nodo Che è padre di se stesso e siamo a posto così però dovreste sapere a Valle delle lezioni di teoria che è possibile rendere più efficiente questa meccanismo e in particolare schiacciare l'altezza la prof di Questi alberi usando la cosiddetta nel suddetto meccanismo di compressione dei cammini Eccoli qua il quale meccanismo dice andiamo che qualcuno abbia indagato Come funzione Find Chi è il capo componente di di nel farlo abbiamo vi salito l'intera catena cve a questa cosa ci ha dato una soluzione ma in realtà c'è anche detto che

non solo di alla Fin della fiera obbedisce Ada ma c e b obbediscono Ada E allora perché non tornare di nuovo a capo e ognuno di questi nodi appenderlo direttamente a da come avviene qui sulla destra è chiaro che in questo modo perdiamo del tempo perché ripetiamo due volte il cammino per le chiaro che se in seguito ci toccherà di nuovo di accedere a do e co AB e chiaramente ci metteremo molto di meno l'accesso a di in un passo solo ci dirà Chi è suo padre lo stesso perche' lo stesso per di mentre Prima avevamo bisogno di tre passi per di due passi per ci è soltanto B soltanto per me non cambierà nulla quindi l'idea In sostanza è quella di fare questa cosa e allora cui può valere la pena di cominciare linguere un pochino non usare la stessa variabile a Tony scopo e quindi andremo introdurre una variabile per esempio iota per dire qual è il vertice corrente è una variabile R per dire chi è la radice è questo che cosa significa che io parto con i j = u e finché il padre di yota è diverso dai J non faccio altro che continuare a salire salire salire e questo è a posto A questo punto in cui ho lasciato il nodo iniziale che mi serve perché devo tornarci per il secondo giro a questo punto ho trovato la radice quando esco da questo ciclo ho trovato la radice e so che si chiama è il valore corrente di yota posso ritornare di nuovo in basso e posso nuovamente scorrere questa operazione qui posso di nuovo usare la stessa condizione perché evidentemente la condizione non è cambiata potrei anche dire no vado su 2:44:40 Qui posso di nuovo usare la stessa condizione perché evidentemente la condizione non è cambiata potrei anche dire no vado su finché io ho te Diverso da R finché non sono arrivato di nuovo alla radice è la stessa cosa potete scrivere niente rambi modi la scrivo così giusto per farvi vedere entrambi e a questo punto che cos'è che bisogna fare in sostanza quello che bisognerebbe fare e appendere ciascun nodo padre ciascun nodo J fino a prenderlo direttamente alla radice Il problema è che nel far questo poi io devo passare da J a suo padre e chiaramente questo punto padre R l'ho rovinato non posso scrivere padre DJ qua di cosa devo fare Devo sostanzialmente in precedenza salvarmi questo valore e quindi ho bisogno di un ulteriore variabile ea posteriori andare a recuperare il valore che ho salvato quando abbiamo finito non restituirà un perché abbiamo detto che alla fine devo restituire la radice questo Uh derivava da quando l'ha dalla forma semplice del procedura in cui continua a essere giornata e quindi Qui restituiremo invece il valore della radice stessa Ok A questo punto se non ho fatto troppe differenze rispetto alle appunti che mi sono fatto dobbiamo intanto andare a correggere un po' di errori Qui c'è un po' d'Arco ti ha scritto due volte che evidentemente è posar copiapo d'Arco PB e probabilmente c'è lo stesso problema anche qui alto a questo punto abbiamo un PJ che potrebbe essere inizializzato perché ho semplicemente invertito la l'uso di PJ PJ deve salvarsi il valore del padre e a questo punto è possibile che il nostro algoritmo CDA la soluzione corretta che in effetti è la soluzione corretta Questo è il l'esempio di albero problema dell'albero ricoprente minimo che riportano le dispense di teoria L'unica cosa che ho fatto e cambiare i nomi da renderli dei numeri anziché delle lettere come vedete la soluzione comprende 14 che sarebbe a di' che vale 145 che sarebbe di e che vale due Poi c'è 136 che è questo che vale 212 che va alle 3 abbiamo 78 che vale 368 che vale 469 che va alle 5 abbiamo 47 che vale 47 che va alle 7 abbiamo saltato il i due archi di costo 5:02 anche di questo sei Se non sbaglio Dopodiché concludiamo no compriamo col 47 E se si somma il risultato viene effettivamente 27 Ok Direi a questo punto abbiamo tutto sommato concluso l'argomento della lezione L'unica cosa che mi resta da dire è un considerazione sul progetto finale il progetto uscirà fra una decina di giorni intorno a metà maggio in maniera da consentire a voi alcune settimane di svolgimento e a me alcuni giorni di correzione sono volutamente vago perché ci vuole del tempo a scriverlo del tempo a risolverlo da parte mia e del tempo a valutare nella difficoltà D'altra parte non so quanti studenti si presenteranno all'esame quindi non so quanti giorni durerà la correzione stessa ma questo è normale In che cosa consiste il progetto che lo ricordo una relazione è un codice nella relazione valuterò intanto la completezza che ci sia

scritto tutto quello che è degno di nota del progetto stesso vi rimando alle prime lezioni di laboratorio nelle prime quelle intermedie diciamo nelle quali ho cercato di scrivere in che maniera sarebbe potuto essere una relazione di progetto sugli esercizi Quindi niente dettagli informatici ma l'idea di prima è tutto il problema poi la trasformazione in un modello matematico poi le idee algoritmiche descritte in modalità top down come ho 2:49:30 Tutto il problema poi la trasformazione in un modello matematico poi le idee algoritmiche descritte in modalità top down come ho sempre cercato di fare in tutti questi esercizi e Dawn vuol dire che c'è una descrizione a grande livello il primo passo e poi eventualmente ci sono delle sezioni dedicate alle singole parti Qui sarebbe stato andiamo a costruire l'albero ricoprente minimo Dopodiché c'è una divisione in Esiste un problema di determinare una l'atto di questo minimo e Esiste un problema di determinare una aggiunta un test di aciclicità della soluzione corrente questi due sotto problemi potrebbero militare delle sezioni ad hoc con indicazione di quali strutture dati si sono usate con le strutture dati vengono in gioco nel momento in cui è chiaro che cosa bisogna fare quindi non all'inizio ma a metà avanza della relazione e a questo punto ci permettono le strutture dati di capire qual è la complessità di queste operazioni per cui descritta a singola funzione è possibile ricavarne data alla implementazione della struttura dati è possibile ricavarne la complessità e quindi alla fine tirare le somme sulla complessità totale Come del resto Abbiamo fatto anche in questo questa serie di luci di abbiamo descritto in cosa consisteva il nostro algoritmo dopodiché abbiamo detto in che maniera implementa Vamo le singole parti e ne abbiamo derivato la complessità totale dell'algoritmo Allora complessi completezza vi dicevo è un aspetto fondamentale e fatevi una specie di checklist guardando il codice e guardando la relazione Chi legge il codice della velazione in parallelo come me dovrebbe trovare più o meno tutto quello che non sia un dettaglio brutalmente programmatorio vuoi totalmente informatico dopo per correttezza che è duplice la correttezza vuol dire che il codice è la relazione dicano la stessa cosa è che questa cosa sia una cosa sensata cioè Non ditemi che avete fatto un algoritmo che Ordina un vettore in tempo logaritmico perché non è possibile Non ditemi che l'avete fatto in tempo n log n Se invece l'avete fatto in tempo e nel quadro perché è possibile ma non è corretto Dopodiché c'è una questione di dicevo struttura ed eleganza che è riferito ciò che dicevo prima riguardo al all'ordine problema modello algoritmo quindi strutture dati quindi complessità e si può passare Quindi al codice il codice ho cercato in ogni punto di questo corso di darvi un'idea di come costruire le maniere che sia leggibile corretto chiaro e via dicendo poi ognuno ha chiaramente il suo stile e ci sono 10 soluzioni per ogni possibile problema Certamente conta la correttezza perché risultato sia quello che deve essere dopodiché la correttezza si declina ovviamente in sintattica e semantica Cioè ci saranno giustamente errori di puntatori allocazione e deallocazione che hanno il loro peso perché pesano anche sulla durata della correzione e quindi l'invito il più possibile a cercare di evitare pasticci in termini di sintassi e quindi a fare tutti i controlli del caso su questo dico qualcosa poi dopo dal punto di vista della efficienza e cercare di usare sempre le strutture dati e gli algoritmi più adeguati ci sono algoritmi per calcolare l'albero del cammino minimo c'è c'è un altro algoritmo che la guida di prima che vi verrà descritto E a seconda dei casi è meglio uno è meglio un altro lo stesso per l'ordinamento lo stesso per i cammini minimi lo stesso per un po' tutti gli algoritmi del caso questo viene valutato e anzi tipicamente là criterio rispetto al quale c'è il massimo ventaglio di valutazione e poi c'è un aspetto di eleganza e pulizia formale bellezza modularità del codice che è un po' più imponderabile Ma che ha il suo peso meno importante per quanto riguarda l'acqua Nettezza e io scrivo nel documento che vi invito a leggere sulle istruzioni sui consigli per lo svolgimento del progetto scrivo una serie di consigli che sono quelli di utilizzare tutti queste opzioni di compilazione certe che te lo Come avete visto non garantiscono assolutamente la correttezza altri modi di procedere sono

un meno come deve meno consigli meno universali quello che vi posso dire che io personalmente quando correggeva i vostri codici userò un ambiente che è il Visual

2:54:20 Consigli meno universali quello che vi posso dire che io personalmente quando correggeva i vostri codici userò un ambiente che è il Visual studio Express di Microsoft questo evidentemente non è accessibile a tutti perché funziona sotto Windows credo e d'altra parte non mi sembra neanche il caso di forzare l'uso di uno strumento commerciale Delta gratuito ma di una ditta per cui non l'ho detto all'inizio e non lo richiedo assolutamente Quello che noto è che questo strumento è estremamente pignolo e che veramente metà dei progetti che mi arrivano e che sul computer degli studenti girano e funzionano sul mio computer non girano e non funzionano e regolarmente sempre c'è un errore quindi può essere magari utile per chi ha questa la possibilità di dargli una un tentativo di dargli una prova di vedere se se funziona sia effettivamente tutto Io continuerò a presidiare il forum saltuariamente non più due ore al mercoledì mattina due ore a giovedì pomeriggio ma collegandomi un giorno ogni tanto due tre volte la settimana per vedere se ci sono domande ovviamente si possono essere domande anche via mail quelle di carattere più generale interessante magari mi fa senso che vengano spedite in tutti e due i modi O perlomeno che sul forum figuri No è per il resto direi che non posso che augurarvi un buon successo È un buon esame e qui direi che chiudo il corso per quest'anno in bocca al lupo

Figura 14.9: Il problema dello zaino
Si vuole scegliere da un insieme di oggetti di vario volume un sottoinsieme di valore massimo che possa stare in uno zaino di capacità limitata

La differenza fondamentale è che si complica la definizione di $\mathcal{F}_A$ dato che non tutti gli elementi di $B \setminus x$ estendono x in modo ammissibile

$$x \cup \{i\} \in \mathcal{F}_A = X \Leftrightarrow \sum_{j \in x} v_j + v_i \leq V$$

Algorithm GreedyKP(I)
$x := \emptyset$;
While $\exists i \in B \setminus x : v_i \leq V - \sum_{j \in x} v_j$ *do*
    $i := \arg \max_{i \in B \setminus x : v_i \leq V - \sum_{j \in x} v_j} \phi_i$;
    $x := x \cup \{i\}$;
Return x;

Problema

Affrontiamo lo stesso problema della scorsa lezione: scrivere un programma `knapsack.c` che risolva il problema dello zaino. Questa volta, la soluzione non sarà esatta, ma euristica, cioè priva di garanzia di ottimalità. Gli algoritmi che realizzeremo sono *greedy*, cioè basati sull'idea di partire da un sottoinsieme vuoto e aumentarlo di un elemento alla volta, scelto come il più "promettente" fra quelli che non violano il vincolo di capacità. I formati di ingresso e uscita rimangono quelli già definiti.

Figura 14.10: Esempio: il problema dello zaino

B	a	b	c	d	e	f
ϕ	7	2	4	5	4	1
v	5	3	2	3	1	1

$$V = 8$$

L'algoritmo esegue i seguenti passi:

1. $x := \emptyset$;
2. poiché $v_i \leq V - \sum_{j \in x} v_j \ \forall i \in B \setminus x$, sceglie $i := a$ e aggiorna $x := \{a\}$;
3. poiché $v_i \leq V - \sum_{j \in x} v_j \ \forall i \in B \setminus x$, sceglie $i := d$ e aggiorna $x := \{a, d\}$;
4. poiché $v_i > V - \sum_{j \in x} v_j \ \forall i \in B \setminus x$, termina

Questo algoritmo non ha trovato la soluzione ottima $x^* = \{a, c, e\}$

Ma perché?

Figura 14.11: Correttezza dell'algoritmo greedy

Dato un problema di Ottimizzazione Combinatoria con

- insieme base B
- spazio di ricerca ("collezione di indipendenti") $\mathcal{F}_A \subseteq 2^B$

l'algoritmo greedy lo risolve per ogni funzione obiettivo additiva $f(x) = \sum_{i \in x} \phi_i$ se e solo se

1. il sottoinsieme vuoto è un indipendente: $\emptyset \in \mathcal{F}_A$
2. ogni sottoinsieme proprio di un indipendente è un indipendente: se $x \in \mathcal{F}_A$ e $y \subset x$ allora $y \in \mathcal{F}_A$
3. ogni indipendente si può allargare con un opportuno elemento di qualsiasi altro indipendente di cardinalità superiore:
   per ogni $x, y \in \mathcal{F}_A$ con $|x| = |y| + 1$, $\exists i \in x \setminus y : y \cup \{i\} \in \mathcal{F}_A$

Queste condizioni

- valgono per il KP unitario
- non valgono per il KP generico

Figura 14.12: Algoritmi greedy euristici: il *KP*

Se lo spazio di ricerca $\mathcal{F}_A$ non ha le proprietà adatte, si può adottare

- una definizione sofisticata del criterio di scelta:

$$i = \arg \max_{i \in B \setminus x : x \cup \{i\} \in \mathcal{F}_A} \phi_i$$

diventa

$$i = \arg \max_{i \in B \setminus x : x \cup \{i\} \in \mathcal{F}_A} \varphi_A(i, x)$$

dove $\varphi_A(i, x)$ dipende sia dall'obiettivo sia dai vincoli del problema

Questo consente risultati efficaci, pur se non dimostrabilmente ottimi

Siccome l'algoritmo greedy base per il *KP* fallisce a causa del volume degli oggetti, si cercano oggetti di valore alto e volume basso

- anziché il valore ϕ_i, si usa il valore unitario $\varphi_A(i, x) = \dfrac{\phi_i}{v_i}$

L'algoritmo risultante tipicamente funziona molto meglio

Figura 14.13: Esempio: il *KP*

B	a	b	c	d	e	f
ϕ	7	2	4	5	4	1
v	5	3	2	3	1	1
ϕ/v	1.40	0.67	2.00	1.67	4.00	1.00

$$V = 8$$

L'algoritmo esegue i seguenti passi:

1. $x := \emptyset$;
2. sceglie $i := e$ e aggiorna $x := \{e\}$;
3. sceglie $i := c$ e aggiorna $x := \{c, e\}$;
4. sceglie $i := d$ e aggiorna $x := \{c, d, e\}$;
5. sceglie $i := f$ e aggiorna $x := \{c, d, e, f\}$; (*l'oggetto a non ci sta*)
6. poiché $v_i > V - \sum_{j \in x} v_j$ per ogni $i \in B \setminus x$, termina

La soluzione trovata vale 14, quella ottima è $x^* = \{a, c, e\}$ e vale 15

Figura 14.14: Istanze critiche del KP per l'algoritmo greedy
Nota: la parte che segue è fuori del programma del corso

Ci sono ancora casi critici

B	a	b
ϕ	10	90
v	1	10
ϕ/v	10	9

$$V = 10$$

L'algoritmo esegue i seguenti passi:

1. $x := \emptyset$;
2. sceglie $i := a$ e aggiorna $x := \{b\}$;
3. poiché $v_i > V - \sum_{j \in x} v_j$ per ogni $i \in B \setminus x$, termina

La soluzione vale 10, quella ottima vale 90

L'errore diventa grande a piacere quando
il primo oggetto scartato ha volume grande e valore grande

Figura 14.15: Un algoritmo 2-approssimato per il *KP*
Con una piccola modifica, l'algoritmo diventa 2 – *approssimato*
Il suo risultato è almeno metà dell'ottimo

1. Si parte con un sottoinsieme vuoto: $x^{(0)} = \emptyset$
2. Si trova l'oggetto i di valore unitario massimo in $B \setminus x$
3. Se non eccede il volume, si mette in soluzione e si torna al punto 2

$$x^{(t-1)} = \{i_1, i_2, \ldots, i_{t-1}\} \rightarrow x^{(t)} = \{i_1, i_2, \ldots, i_t\}$$

4. Altrimenti, si costruisce una soluzione col solo oggetto

$$x' = \{i_t\}$$

5. Si restituisce la soluzione migliore fra x e x': $f_A = \max\left[f(x), f(x')\right]$

È facile dimostrare che

- la somma delle due soluzioni è una stima per eccesso dell'ottimo

$$f(x) + f(x') = \sum_{\tau=1}^{t} \phi_{i_\tau} \geq f^*$$

- la migliore delle due soluzioni è almeno metà della somma

$$f_A = \max\left[f(x), f(x')\right] \geq \frac{f(x) + f(x')}{2} \geq \frac{1}{2} f^*$$

Figura 14.16: Il problema dell'albero ricoprente minimo

Dati

- un grafo non orientato connesso $G = (V, E)$
  con $n = |V|$ vertici e $m = |E|$ lati
- una funzione di costo $c : E \to \mathbb{N}$ definita sui lati

si trovi un sottografo $T^* = (U^*, X^*)$

1. ricoprente: U^* contiene tutti i vertici ($U^* = V$)
2. connesso: X^* include un cammino fra ogni coppia di vertici u e v
3. aciclico: X^* non contiene cicli
4. di costo totale minimo:

   $c_{X^*} \leq c_X$ per ogni $T = (U, X)$ che soddisfa le proprietà 1, 2 e 3

   dove $c_X = \sum_{e \in X} c_e$

Traccia della risoluzione

Per cominciare, si possono scorrere i lucidi che ricordano la definizione di Ottimizzazione Combinatoria e di algoritmo esaustivo, e poi tratteggiano lo schema generale di algoritmo *greedy* e introducono il caso specifico del problema dello zaino unitario (con oggetti di volume uniforme), osservando che in esso l'algoritmo *greedy* sembra funzionare, mentre chiaramente non funziona per il problema dello zaino generico.

L'esercizio parte dal file `knapsack0.c` da cui già comcinciava la scorsa lezione. Siamo quindi in grado di caricare istanze del problema dello zaino, stampare soluzioni, ecc...

Prima fase (`knapsack-greedy1.c` Questa fase dell'esercizio dovrebbe realizzare l'algoritmo *greedy* base, che

1. parte da un sottoinsieme vuoto
2. valuta se esistono elementi aggiungibili al sottoinsieme corrente nel rispetto del vincolo di capacità;
3. determina l'oggetto di valore massimo fra quelli aggiungibili;
4. lo aggiunge alla soluzione corrente e torna al punto 2.

In realtà, gestire l'insieme degli oggetti aggiungibili non è la soluzione più efficiente, perché comporta di scorrere ad ogni passo l'insieme $B \setminus x$ degli oggetti esterni per sfrondarlo degli elementi che sono diventati inammissibili dopo l'ultima estensione di x. È in fondo più semplice gestire direttamente $B \setminus x$ e valutare di volta in volta se l'oggetto di valore massimo è aggiungibile o no: nel primo caso, si procede come sopra, nel secondo, ci si limita ad eliminarlo. Quindi l'algoritmo che realizzeremo:

1. parte da un sottoinsieme vuoto

2. determina l'oggetto di valore massimo fra quelli esterni;
3. se tale elemento è aggiungibile, procede ad aggiungerlo alla soluzione corrente; altrimenti, lo elimina; in entrambi i casi, torna al punto 2.

Il problema fondamentale di questo algoritmo è determinare in modo efficiente l'oggetto di valore massimo. Per farlo, si possono mettere i premi in un *max-heap*. Ma questa struttura restituisce solo il valore massimo, mentre a noi serve conoscere l'indice dell'oggetto corrispondente, per aggiungerlo al sottoinsieme corrente. Una prima soluzione è costruire un vettore di "oggetti", cioè di strutture contenenti indici, valori e volumi, e ordinare tale vettore per valori crescenti. Si noti che gli indici sono necessari perché, una volta scambiati gli oggetti rispetto all'ordinamento iniziale, non si ha più modo di indicarli per nome. Alternativamente, si possono gestire tre vettori di interi (uno per gli indici, uno per i valori e uno per i volumi), e ordinare il secondo, mantenento però gli altri due ordinati in parallelo, attraverso l'esecuzione di scambi coordinati con quelli eseguiti sul secondo.

Per esempio, supponendo di partire dai vettori:

```
ind = [ 1 2 3 4 5 6 ]
phi = [ 7 2 4 5 4 1 ]
v   = [ 5 3 2 3 1 1 ]
```

la costruzione del *max-heap* comporta di scorrere le posizioni 3, 2 e 1 confrontando ciascuna con i due figli:

- la posizione 3 è già corretta, dato che `phi[3] > phi[6]` e non esiste `phi[7]`;
- la posizione 2 non è corretta, dato che `phi[2] < phi[5] < phi[4]`, per cui bisogna scambiare 2 e 5 (e operare ricorsivamente su 5, dove non si fa nulla);
- la posizione 1 è corretta, dato che `phi[1] > phi[2] = phi[3]`.

ottenendo i vettori

```
ind = [ 1 4 3 2 5 6 ]
phi = [ 7 5 4 2 4 1 ]
v   = [ 5 3 2 3 1 1 ]
```

Scambiare gli elementi di tutti i vettori (o gli elementi dell'unico vettore di strutture) può essere pesante, e in più distrugge l'ordinamento iniziale dei dati, che potrebbe essere utile conservare per altri motivi.

È però possibile, e preferibile, scambiare solo le posizioni nel vettore `ind`

```
ind = [ 1 4 3 2 5 6 ]
phi = [ 7 2 4 5 4 1 ]
v   = [ 5 3 2 3 1 1 ]
```

osservando che la proprietà *max-heap* vale **indirettamente** sul vettore `phi`: basta che per identificare i nodi figli si usino, al posto degli indici diretti, gli indici indiretti contenuti in `ind`. Per esempio, non diciamo più che l'elemento di indice 2 ha come figli gli elementi di indice 4 e 5 e ha valore non inferiore ad essi, ma che l'elemento di indice `ind[2]` (cioè 4) ha come figli gli elementi di indice `ind[4]` (cioè 2) e `ind[5]` (cioè 5) e ha valore non inferiore ad essi: `phi[ind[2]] >= phi[ind[4]]` (cioè $\phi_4 = 5 \geq 2 = \phi_2$) e `phi[ind[2]] >= phi[ind[5]]` (cioè $\phi_4 = 5 \geq 4 = \phi_5$). Per quanto riguarda la ricerca del massimo, la non diciamo più che la radice corrisponde all'indice 1, così che `phi[1]` è il valore massimo, ma che la radice corrisponde all'indice `ind[1]` (qui casualmente ancora pari ad 1) e quindi `phi[ind[1]]` è il

valore massimo. Si tratta cioè di accedere ai vettori con i dati attraverso gli indici del vettore `ind`. È ovvio che sarebbe del tutto equivalente avere invece un vettore di puntatori, anche se questo permetterebbe l'accesso solo a uno dei due vettori con i dati (valori e volumi); sarebbe allora preferibile raccogliere i dati in un solo vettore di strutture, come già proposto più sopra.

Questo dà luogo a un "*heap* indiretto", ma si tratta di un'idea del tutto generale, che può essere facilmente adattata a tutti gli algoritmi di ordinamento visti nel corso, anche se non usano *heap*:

- il vettore che contiene i dati è intatto, e resta disordinato;
- gli scambi avvengono sul vettore degli indici;
- i confronti fra elementi non si fanno direttamente sul vettore dei dati, ma indirettamente sul vettore dei dati passando per il vettore degli indici (`phi[ind[...]]` anziché `phi[...]`).

Dal punto di vista della complessità, l'ordine asintotico non cambia; la lettura è più complicata, e leggermente più lenta, ma i dati non subiscono modifiche e gli scambi sono più veloci.

Inoltre, introducendo più vettori `ind` (di interi o di puntatori) si possono ottenere ordinamenti diversi degli stessi dati. Per esempio, si potrebbero ordinare gli oggetti sia rispetto ai valori sia rispetto ai volumi. Questo consentirebbe di ridiscutere l'algoritmo, consentendo ad ogni aggiunta la cancellazione degli elementi esterni a x che risultano non più aggiungibili: essi infatti sarebbero gli elementi di volume massimo. Si può trattare di un buon esercizio, che comporta la gestione di due *max-heap* sullo stesso insieme e l'individuazione dell'elemento di valore massimo, ma talvolta anche di quello di volume massimo, e la sua cancellazione da entrambi gli *heap*.

Realizzato l'algoritmo, si può facilmente constatare che le sue prestazioni sono piuttosto cattive dal punto di vista della qualità della soluzione (il tempo di calcolo è ovviamente rapidissimo, dato che la complessità nel caso pessimo è facilmente stimabile in $\Theta(n \log n)$).

Seconda fase (`knapsack-greedy2.c`) Il motivo principale del cattivo funzionamento dell'algoritmo *greedy* banale è che non tiene conto in alcun modo del vincolo di capacità, per cui aggiunge al sottoinsieme x corrente oggetti di valore alto, ma anche di volume eccessivo, che esauriscono in fretta la capacità dello zaino. Per ovviare, si può cambiare il criterio di scelta dell'elemento da aggiungere, usando il *valore unitario* ϕ_i/v_i anziché il valore ϕ_i. Questo corrisponde a cercare contemporaneamente di massimizzare il valore e minimizzare il volume. Corrisponde anche a cercare di risolvere il problema di zaino unitario ottenuto rompendo ogni oggetto in "porzioni" di volume unitario, di valore pari al valore dell'oggetto diviso il suo volume (cioè il numero delle porzioni). Questo è ovviamente vietato dai vincoli del problema originario, ma consente di trovare una soluzione inammissibile, che comprende tutte le "porzioni" degli oggetti di valore unitario massimo, più alcune porzioni di un solo oggetto "limite". Scartando queste porzioni, si ottiene una soluzione euristica, che in pratica si dimostra di valore decisamente migliore di quella fornita dall'algoritmo *greedy* banale.

Per realizzare questo algoritmo, basta sostituire nel precedente il vettore di interi `phi` con un vettore di numeri reali `phi_v` che contiene i dati usati nell'ordinamento.

Null'altro cambia. Ovviamente, si potrebbe anche passare alla procedura di ordinamento tutti i dati e modificare le operazioni di confronto, sostituendo i rapporti ϕ_i/v_i al valore ϕ_i.

Problema

La seconda parte della lezione affronta il problema dell'albero ricoprente minimo, per il quale l'algoritmo *greedy* determina la soluzione ottima. Si tratta di scrivere un programma `kruskal.c` che carichi un grafo non orientato e ne determini un albero ricoprente di costo totale minimo. Il nome del programma viene dal fatto che l'algoritmo *greedy* è noto in in letteratura (e viene indicato nelle dispense) come *algoritmo di Kruskal*, dal nome del suo scopritore.

Formato di ingresso e di uscita Il formato di ingresso è un po' più sofisticato di quello utilizzato nelle lezioni 9 e 10 per il calcolo del sottografo indotto, ma molto simile. Il file si compone di tre righe: la prima riporta il numero dei vertici del grafo, preceduto dalle parole chiave `n =`, la seconda riporta il numero dei lati del grafo, preceduto dalle parole chiave `m =`, la terza riporta l'elenco dei lati stessi, con gli indici dei nodi estremi racchiusi fra parentesi tonde e separati da virgole, seguiti dal costo del lato stesso. Per esempio:

```
n = 9
m = 16
(1,2) 3 (1,4) 1 (1,5) 6 (2,3) 8 (2,5) 5 (3,5) 8 (3,6) 2 (4,5) 2 (4,7)
7 (5,6) 10 (5,7) 8 (5,8) 9 (6,8) 4 (6,9) 5 (7,8) 3 (8,9) 6
```

rappresenta l'esempio riportato nelle dispense.

La soluzione va stampata a video su due righe: la prima riga riporta il costo totale della soluzione, preceduto dalle parole chiave `f* =`; la seconda riporta l'elenco dei lati dell'albero minimo, nello stesso formato usato per il file dei dati, ma preceduto dalle parole chiave `x* =`.

Traccia della risoluzione

La risoluzione ricalca l'algoritmo di Kruskal descritto nelle dispense. L'unici aspette su cui si può discutere è l'implementazione del grafo. È abbastanza ovvio che la matrice di adiacenza è inefficiente dal punto di vista spaziale, ma anche da quello temporale (l'algoritmo non richiede di accedere direttamente a un lato dati gli estremi, mentre richiede di trovare il lato di costo minimo, e questo comporterebbe di scorrere $\Theta\left(n^2\right)$ posizioni). Il vettore delle liste di incidenza (*forward star*) è utile per visitare il grafo, ma non per gestire l'insieme dei lati trovandone il minimo, e non è particolarmente efficiente per determinare se un lato chiude cicli o no con la soluzione corrente. Alla fine, la lista dei lati è la struttura più efficiente, specialmente se i lati sono organizzati in un *min-heap*, cioè in un *heap* dove la proprietà sulle informazioni ai nodi viene invertita rispetto a quella che abbiamo usato per il *KP*.

In breve, procederemo come segue:

- useremo la libreria `grafo-la.c`, arricchendo gli archi con un campo intero `c` che riporti il costo dell'arco stesso;

- rappresenteremo le soluzioni con due tabelle di interi `v1` e `v2`, di lunghezza `nl` pari al numero dei lati dell'albero (dunque il numero dei vertici del grafo meno uno), che riportino gli estremi di ciascun lato;
- per trovare il lato di costo minimo, useremo un *min-heap* indiretto costituito da un vettore di `posarco`;
- per gestire il test di aciclità che distingue i lati da aggiungere alla soluzione da quelli da ignorare, useremo la struttura dati astratta *foresta con bilanciamento*, implementata con un vettore come descritto nelle dispense di teoria.

Ovviamente molte altre soluzioni sono possibili, ed eventualmente preferibili da diversi punti di vista. La lista dei lati potrebbe essere una tabella (risparmiando i puntatori e la sentinella, che in effetti non servono a nulla nell'algoritmo), oppure tre tabelle (due per gli indici dei nodi estremi e una per i costi). Oppure si potrebbe mantenere la lista degli archi e aggiungere un vettore a parte per i costi (con il vantaggio di non modificare la libreria e di tener separate topologia e pesatura del grafo, ma lo svantaggio che l'*heap* indiretto dovrebbe indirizzare in qualche modo sia gli archi sia i costi). La foresta con bilanciamento potrebbe essere gestita con una libreria a parte. Le scelte sopra elencate sono ragionevoli, ma non ovviamente migliori, e sono essenzialmente volte ad accelerare il più possibile la risoluzione dell'esercizio, facendo modifiche minime a codice già scritto.