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.

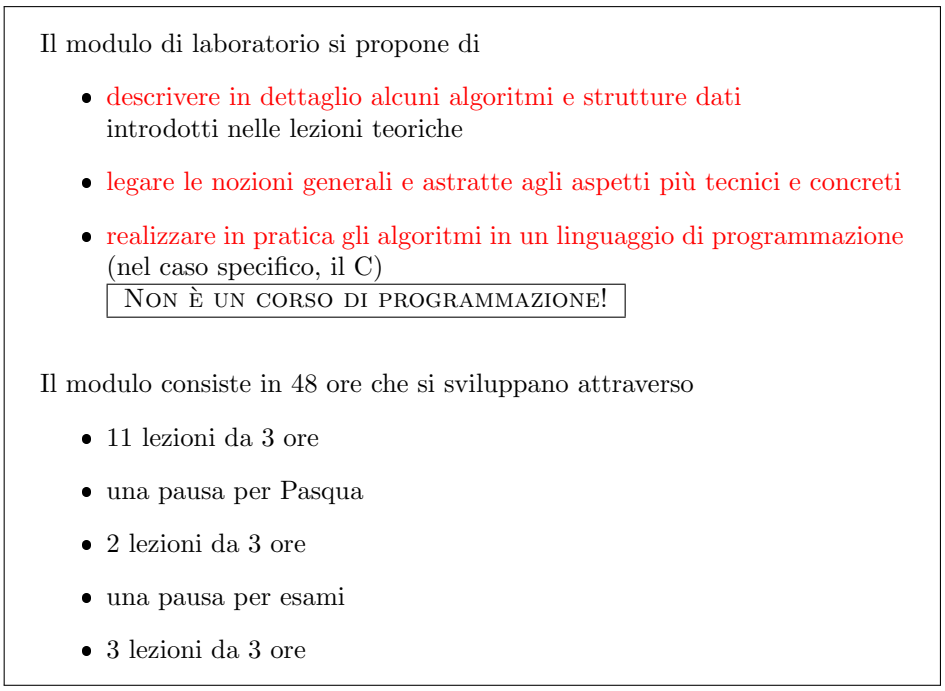


Figura 1: Obiettivi e struttura del modulo di laboratorio

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?

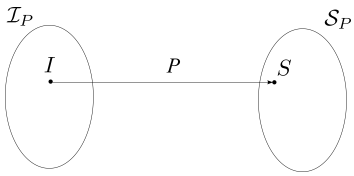
Definiremo

- 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 $\mathbb{N}$ $(12\,514, 12\,645, \ldots)$

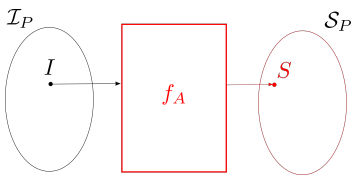
I problemi sono isomorfi a funzioni da numeri naturali a numeri naturali

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

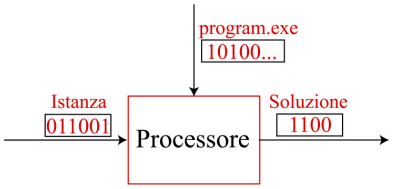


Figura 1.13: Processore

(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

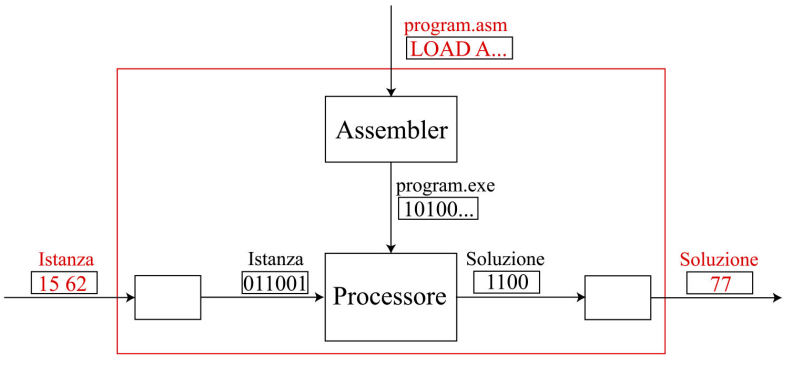


Figura 1.15: Assembler

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.

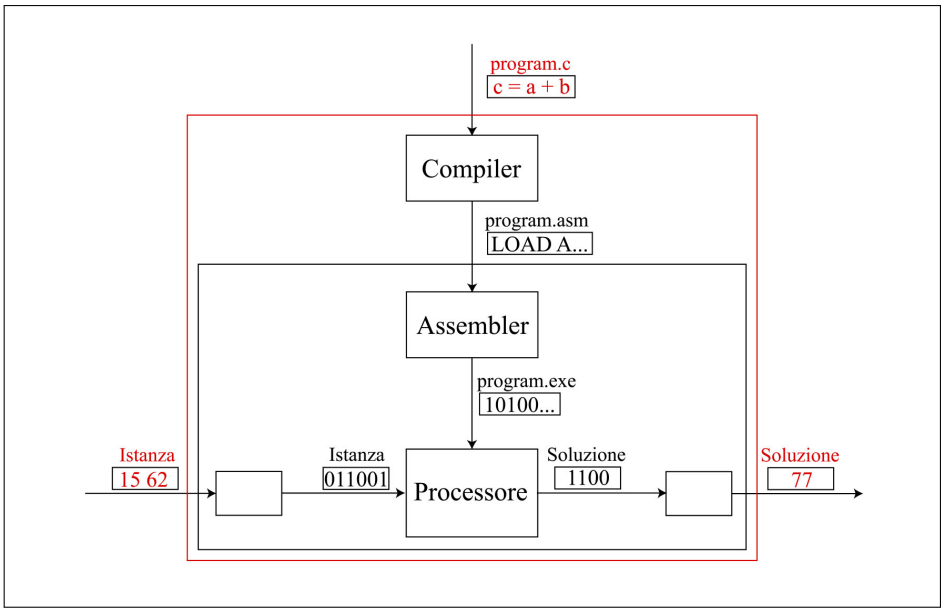


Figura 1.17: Compilatore

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.

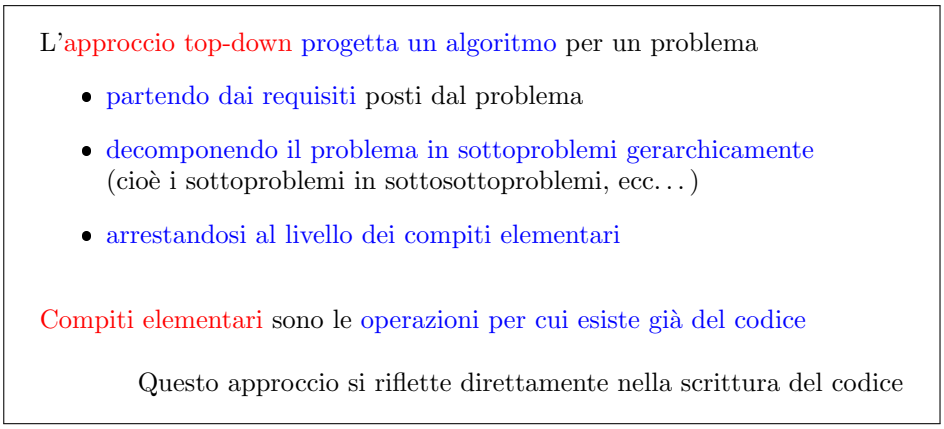


Figura 1.18: Approccio top-down

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

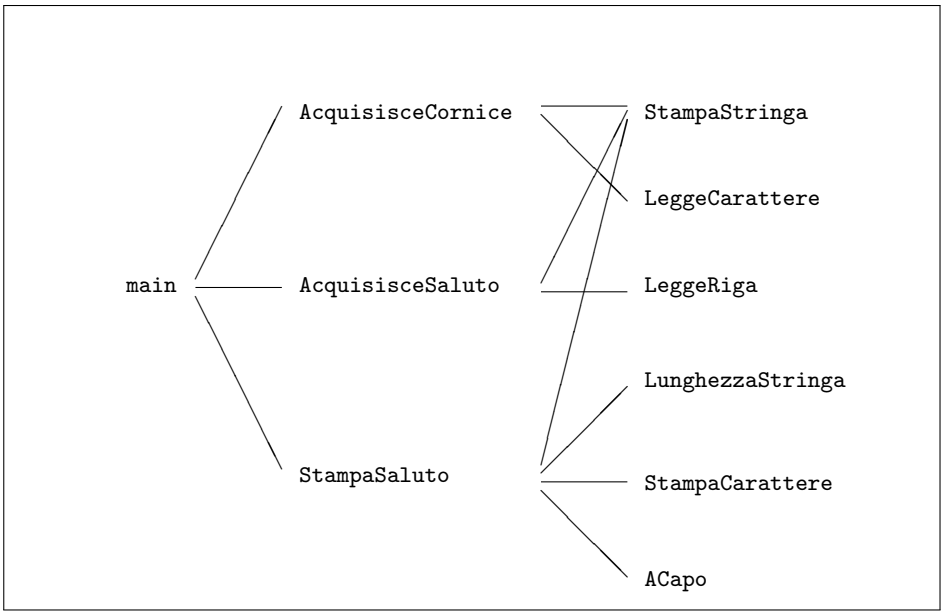


Figura 1.22: Esempio di approccio top-down (4)

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

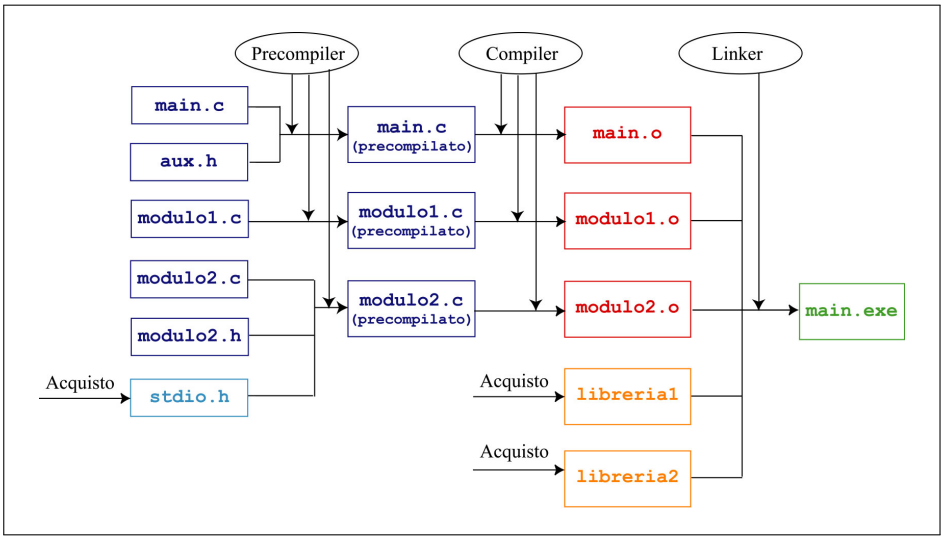


Figura 1.26: Schema del processo di compilazione

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`

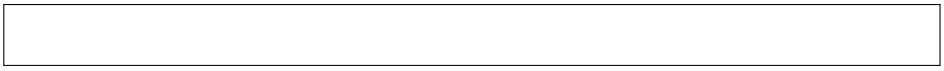


Figura 1.37: Struttura: parte dichiarativa

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'	vale ''pro''
0	1	2	3	4	5	6	

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`. Questo 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 Invece di passarlo da linea di comando, si chieda all'utente di fornire il nome del file contenente il catalogo, valutando la differenza tra leggerlo con la funzione `fscanf` e la funzione `fgets` riguardo l'a capo finale e l'esistenza di spazi bianchi nel nome del file.

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

Esercizio 3 Si stampi il singolo articolo in versione formattata come un'unica stringa combinando le funzioni `fputs` e `sscanf` anziché `fscanf`.

Esercizio 4 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);
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: $E[T_A(I)] \leq E[T_{A'}(I)]$
   - 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

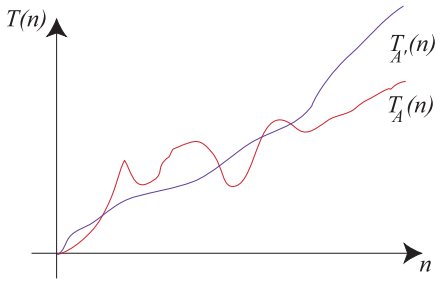


Figura 3.5: Complessità e dimensione (2)

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 $n_0 \in \mathbb{N}$

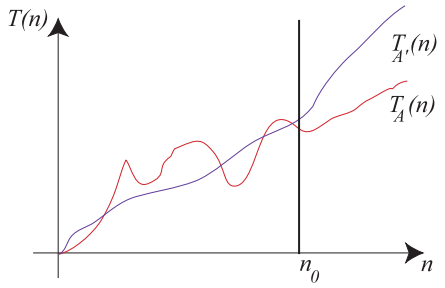


Figura 3.6: Complessità asintotica

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 $f(n)$

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

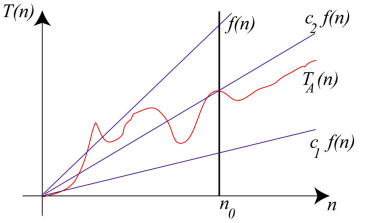
significa formalmente che

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

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 $f(n)$

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

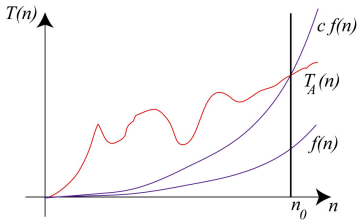
significa formalmente che

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

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 $f(n)$

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

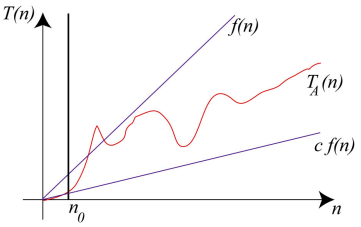
significa formalmente che

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

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

$$(\log n)^r \in o\left(n^s\right) \quad \text{per ogni } r \geq 0 \text{ e } s > 0$$
$$n^r \in o\left(s^n\right) \quad \text{per ogni } r > 0 \text{ e } s > 1$$

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

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

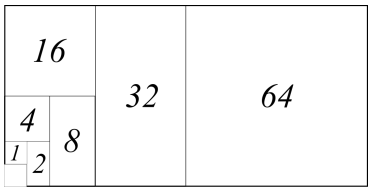
Questo risolve anche la piccola tecnicalità che la sommatoria viene valutata da 0 a n anziché da 1 a n: semplicemente andrebbe sottratto un termine costante, pari a 1, che non cambia l'espressione asintotica. La sommatoria sostanzialmente si comporta come l'ultimo suo termine $f(n)$, salvo qualche costante 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

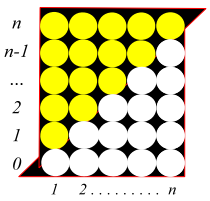
che crescono abbastanza lentamente da rendere tutti i termini degni di nota, e quindi la sommatoria approssimabile con $nf(n)$.

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

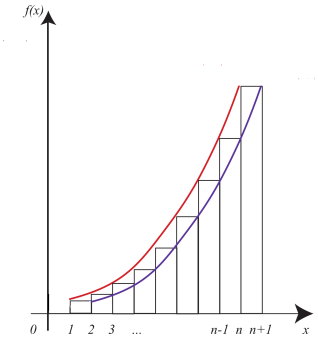
Figura 3.26: Approssimante lineare: somma aritmetica

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];
```

V[0][0]	...	V[0][9]	V[1][0]	...	V[1][9]	...	V[4][0]	...	V[4][9]

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.

La cosa si può ripetere ricorsivamente (vedi Figura 4.14): un record può conte-

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)

nere 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++)
  }
{
```

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 `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
```

[10] Questo andrebbe spostato nella sezione sul progetto, eventualmente lasciando solo un breve richiamo qui.

[11] Potrebbe essere anche un *record* contenente i due indici.

```
    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 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 */
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.

[12] Disquisizione poco chiara, forse, da cancellare, riscrivere o spostare.

```
/* 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 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).

[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.

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]

[14] Ne andrebbe aggiunto uno apposta.

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 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

[15] Bisognerebbe aggiungere i messaggi di errore e qualche commento.

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

#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

****

4.11.1 Analisi della complessità

QUESTA SEZIONE VA ASSOLUTAMENTE FATTA SPARIRE, SE NO GLI STUDENTI TONTI NELLA RELAZIONE FANNO UNA SEZIONE DI ANALISI SE-

PARATA DALLA DESCRIZIONE DELL'ALGORITMO E RENDONO ILLEGGIBILE LA RELAZIONE. OPPURE LA LASCIAMO COME CARTINA DI TORNASOLE PER DISTINGUERE CHI HA CAPITO E DIVARICARE I VOTI.

Abbiamo descritto la modellazione del problema specificando che una scacchiera è descritta da una matrice statica perché la stragrande maggioranza delle operazioni richieste dall'algoritmo sono di lettura e scrittura, e queste sono particolarmente efficienti su vettori. Inoltre, la dimensione della scacchiera è data una volta per tutte dal gioco stesso, per cui non occorre che la matrice sia dinamica (come vedremo invece in seguito per altre strutture dati).

Ora stendiamo alcune righe per l'analisi della complessità di questo algoritmo. Questo va assolutamente fatto in parallelo alla descrizione dell'algoritmo, via via, in modo che sia motivata e non cada dall'alto.

Consideriamo l'esecuzione delle mosse. Questa consiste in un ciclo che scorre le mosse. Per ciascuna, si valuta se la mossa è ammissibile o no e, in caso positivo, che effetto. La valutazione comporta per ogni mossa di verificare in ciascuna delle 8 direzioni (verticali, orizzontali e diagonali) se in quella direzione compare un filotto di celle avversarie contigue che termina in una cella del colore del giocatore. L'analisi richiederebbe di determinare il numero di mosse m e il costo di verifica e valutazione di ciascuna, che comporta di considerare 8 direzioni, e per ciascuna le celle di un filotto nella matrice. Ogni cella richiederà un numero finito di operazioni elementari. A stretto rigore, tutti questi numeri sono finiti, perché la scacchiera ha 8 righe, 8 colonne e contiene 64 celle, e ogni mossa lecita riempie una cella. Per rendere l'esercizio un po' più interessante, rendiamo parametrica l'analisi rispetto alla dimensione della scacchiera, ipotizzando un numero n di righe e di colonne, non costante. Abbiamo quindi m mosse, e per ciascuna la visita di 8 direzioni (questo valore rimane costante, a meno di introdurre un gioco su d dimensioni, con d generico). Ogni filotto, però, si compone di al massimo $n-1$ celle, per cui il ciclo di esecuzione delle mosse ha costo temporale $\Theta(mn)$. Volendo, potremmo anche osservare che le mosse lecite aggiungono una pedina, e quindi non possono essere più del numero di celle. Ipotizzando m_i mosse illecite, avremmo costo temporale $\Theta\left(\left(m_i + n^2\right) n\right)$. Ipotizzando di terminare dopo una mossa illecita, la procedura costerebbe $O\left(n^3\right)$. Il costo spaziale è invece ovviamente $\Theta\left(n^2\right)$ a causa della matrice.

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.

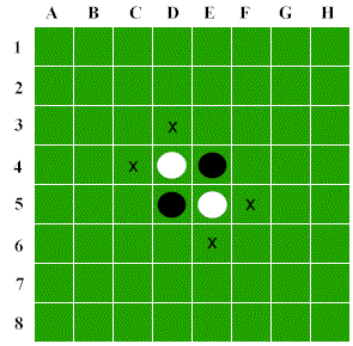


Figura 4.16: Configurazione iniziale: le crocette indicano le mosse lecite per il giocatore Nero, cioè *c4*, *d3*, *e6*, *f5*

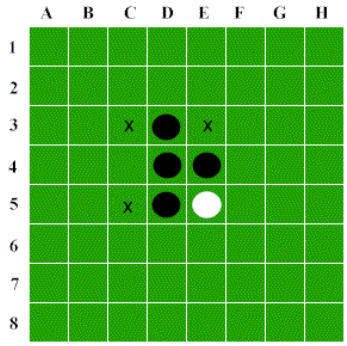


Figura 4.17: Configurazione della scacchiera dopo che il giocatore Nero ha eseguito la mossa *d3*: le crocette indicano le mosse lecite per il giocatore Bianco, cioè *c3*, *c5*, *e3*

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

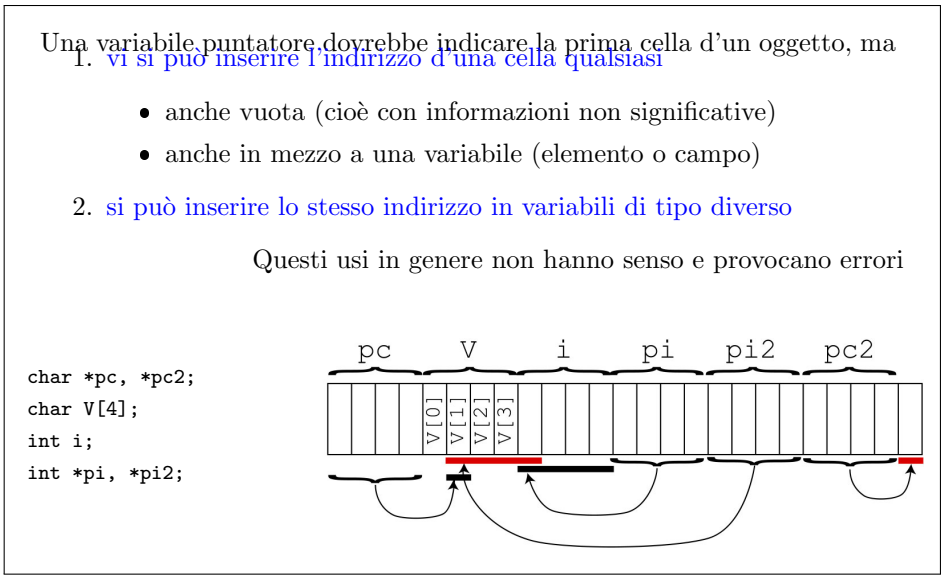


Figura 5.3: Esempi di puntatori corretti e scorretti

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.

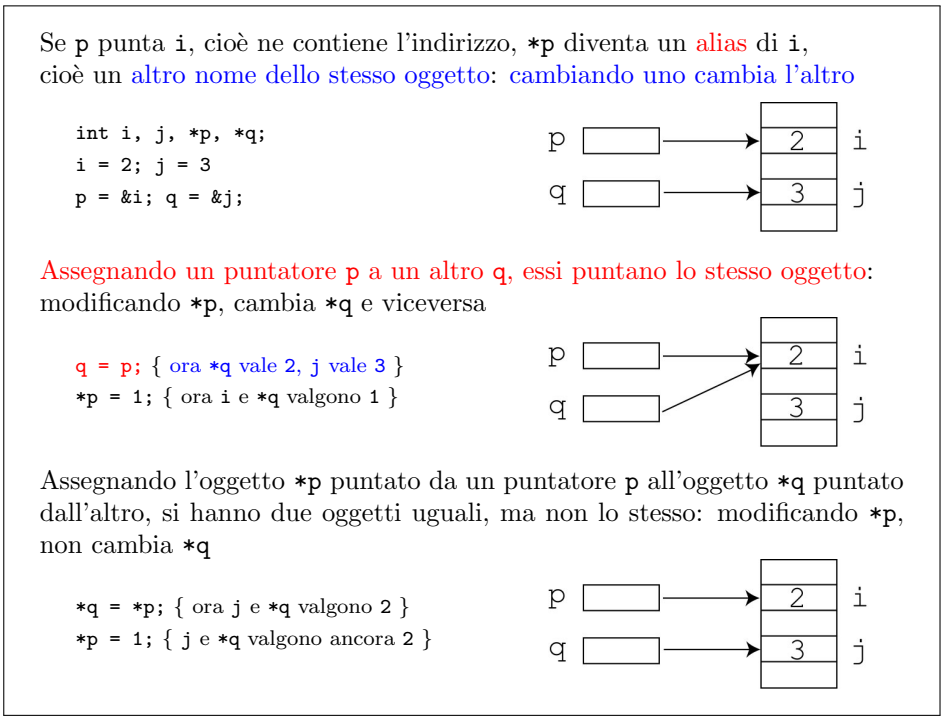


Figura 5.5: Assegnamento fra puntatori

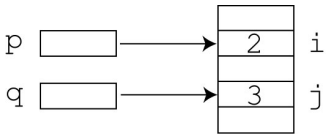
5.1.2 Aritmetica dei puntatori

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.

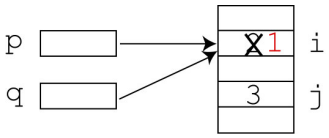
Se `p` punta `i`, cioè ne contiene l'indirizzo, `*p` diventa un alias di `i`, cioè un altro nome dello stesso oggetto: cambiando uno cambia l'altro

```
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 }
```

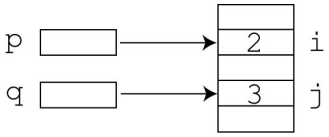
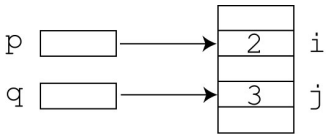


Figura 5.6: Assegnamento fra puntatori

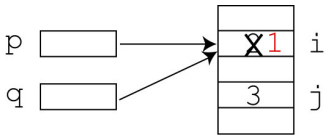
Se `p` punta `i`, cioè ne contiene l'indirizzo, `*p` diventa un alias di `i`, cioè un altro nome dello stesso oggetto: cambiando uno cambia l'altro

```
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 }
```

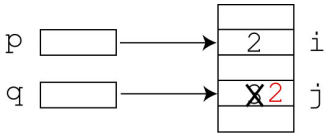


Figura 5.7: Assegnamento fra puntatori

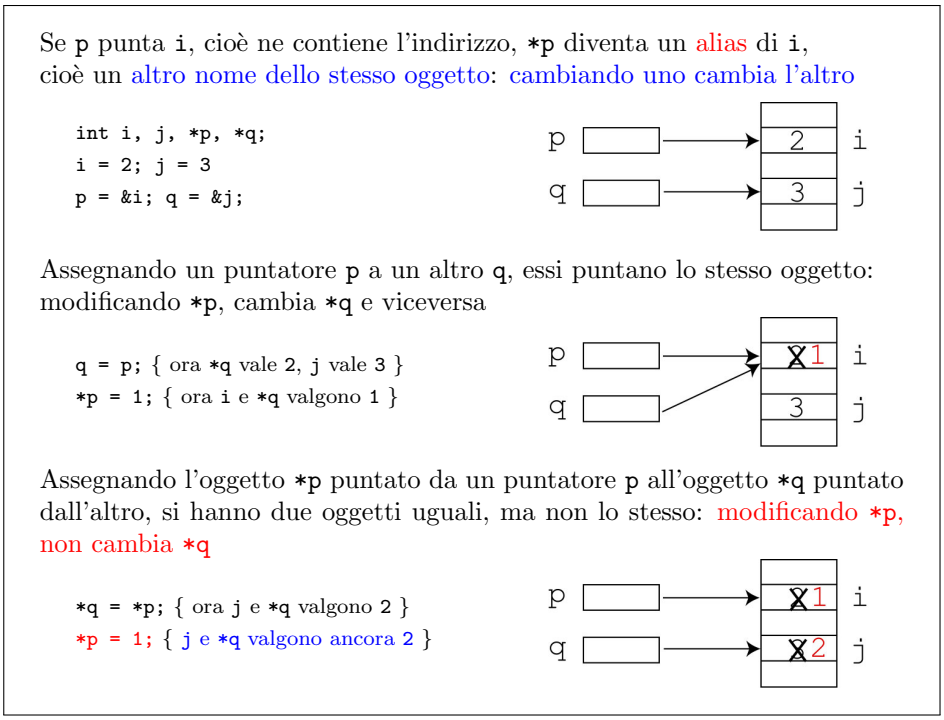


Figura 5.8: Assegnamento fra puntatori

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);
```

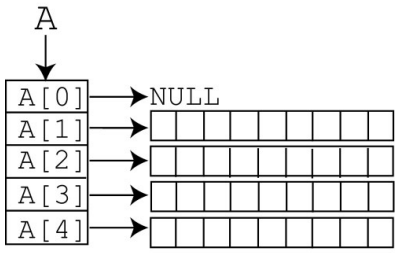


Figura 5.16: Matrici dinamiche

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

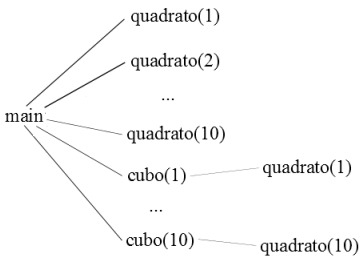


Figura 5.23: Struttura modulare

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.

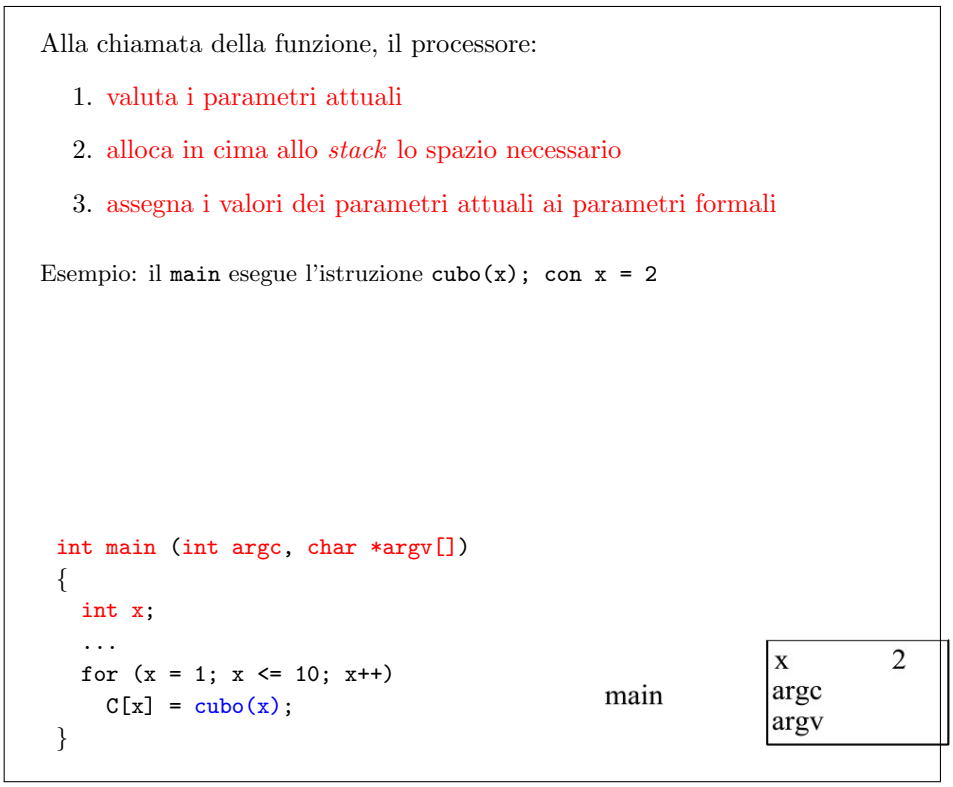


Figura 5.25: Gestione dello *stack*: esempio

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;
}
```

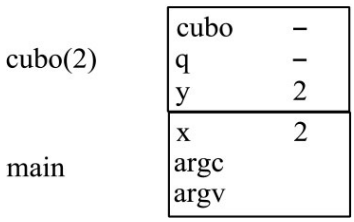


Figura 5.26: Gestione dello *stack*: esempio

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;
}
```

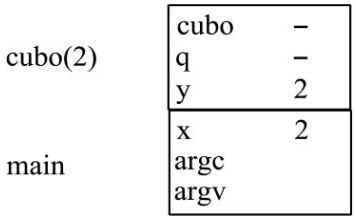


Figura 5.28: Gestione dello *stack*: esempio

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;
}
```

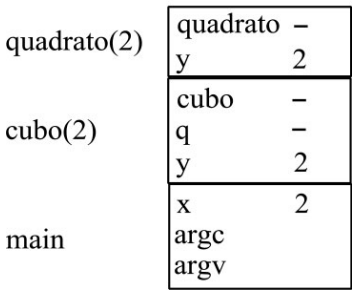


Figura 5.29: Gestione dello *stack*: esempio

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;
}
```

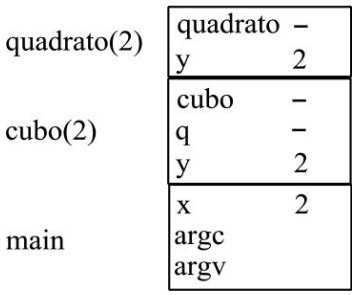


Figura 5.30: Gestione dello *stack*: esempio

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;
}
```

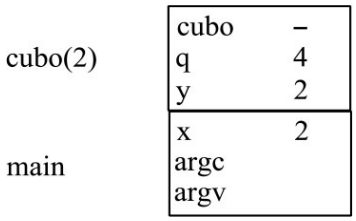


Figura 5.31: Gestione dello *stack*: esempio

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;
}
```

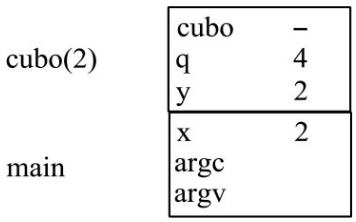


Figura 5.32: Gestione dello *stack*: esempio

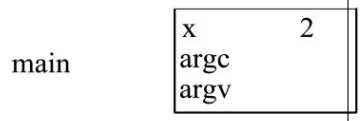
Al termine della funzione, il processore:

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);
}
```



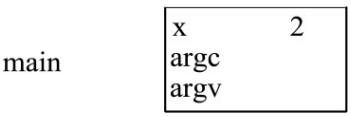


Figura 5.33: Gestione dello *stack*: esempio

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, int **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`.

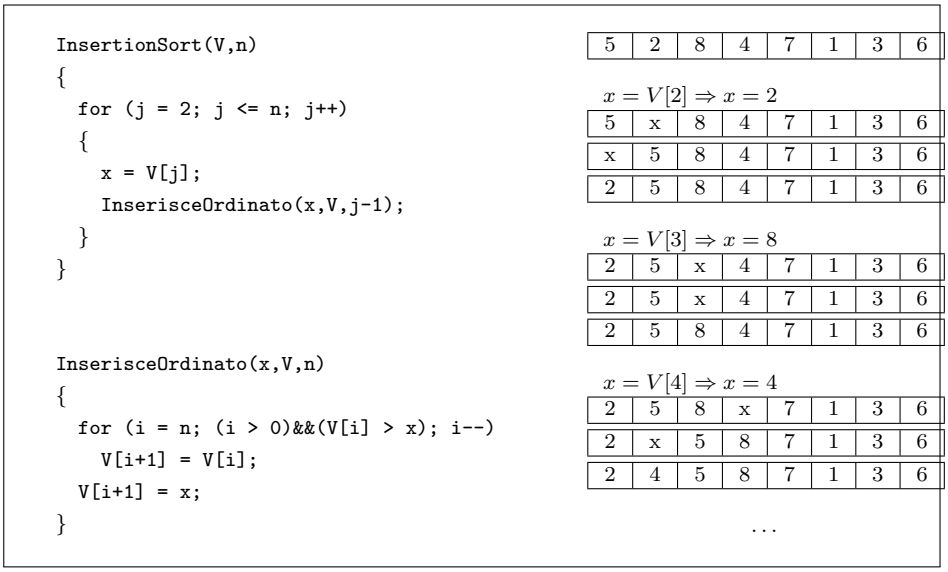


Figura 6.11: InsertionSort: pseudocodice ed esempio

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

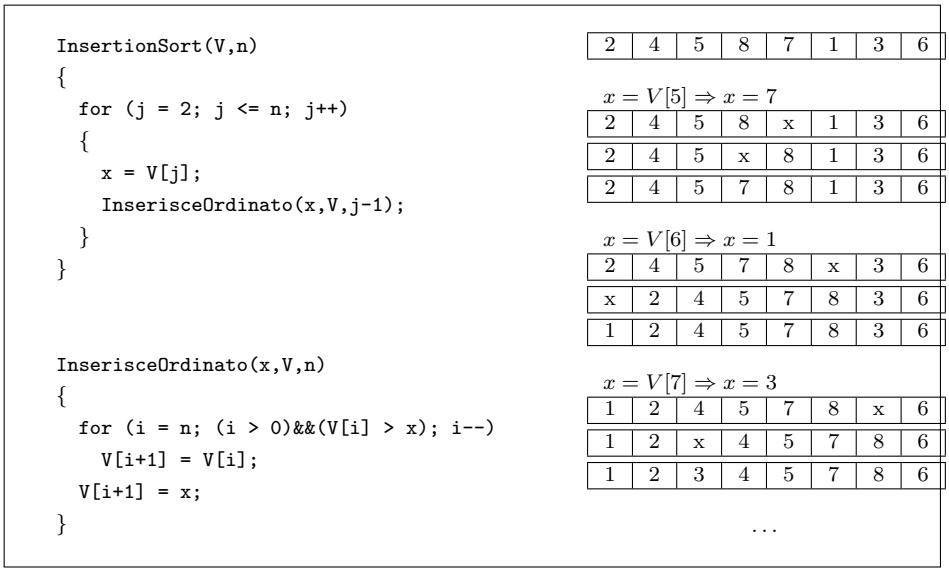


Figura 6.12: InsertionSort: pseudocodice ed esempio

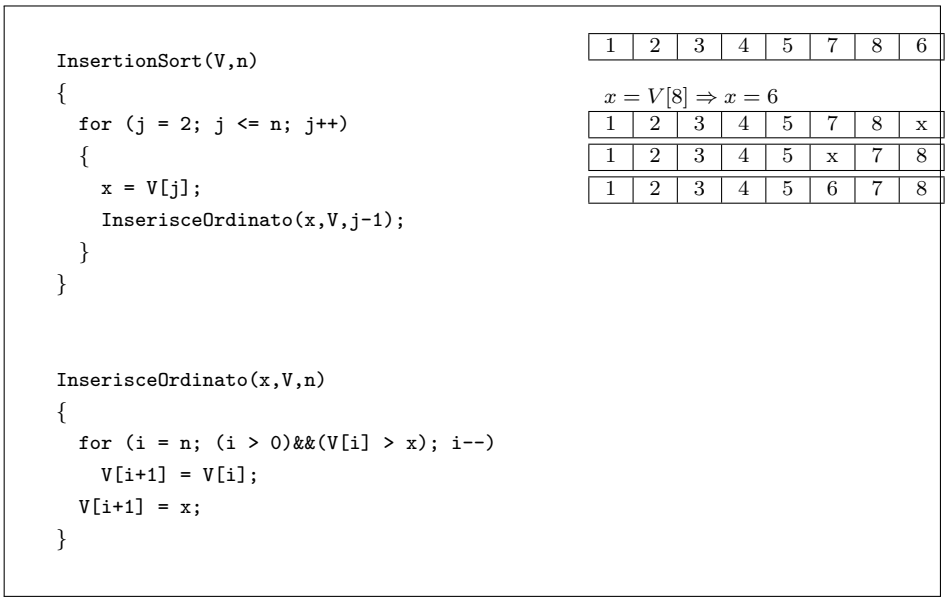


Figura 6.13: InsertionSort: pseudocodice ed esempio

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 5.4 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).

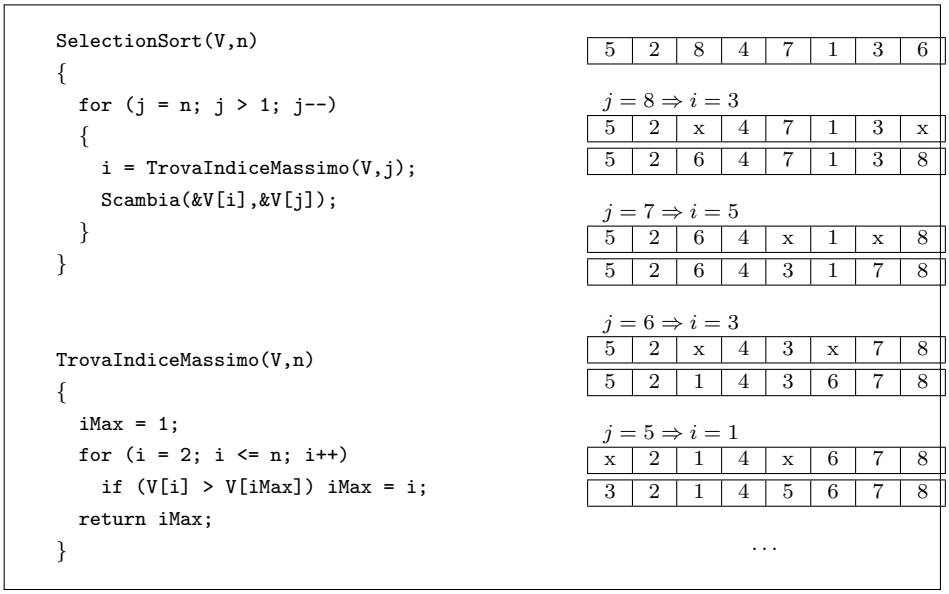
Le Figure 6.18 e 6.19 riportano un esempio numerico di esecuzione passo per passo della procedura stessa.

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);
```

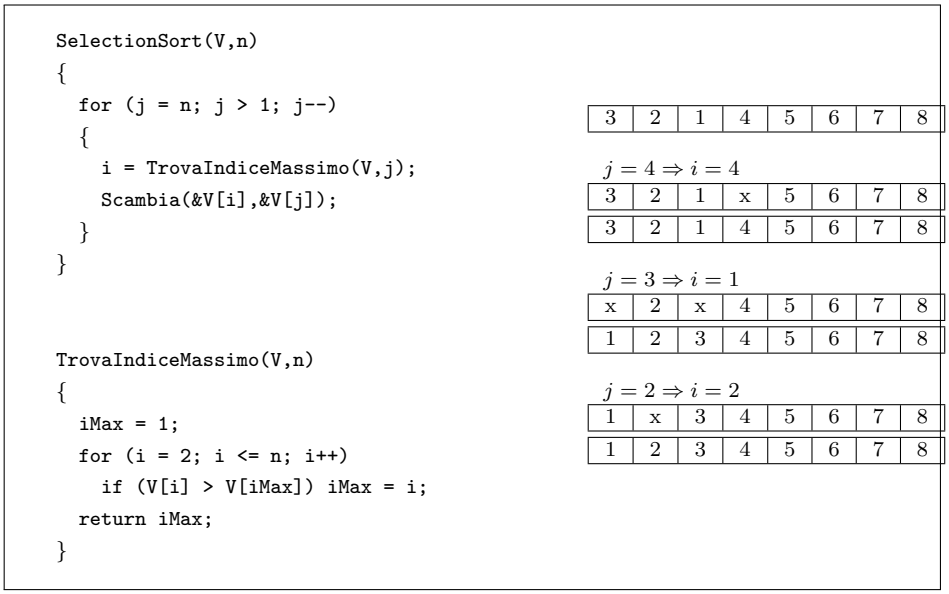
e quindi anche una dichiarazione e definizione praticamente identiche:



```
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--)                           n
  {                                                 Σ (...)
    i = TrovaIndiceMassimo(V,j);                    j=2
    Scambia(&V[i],&V[j]);                           f(j)
  }                                                 Θ(1)
}


TrovaIndiceMassimo(V,n)                             f(j) = ...      (con n = j)
{                                                   
  iMax = 1;                                         Θ(1)
  for (i = 2; i <= n; i++)                          n
    if (V[i] > V[iMax]) iMax = i;                   Σ (...)
  return iMax;                                      i=2
}                                                   Θ(1)
```

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.

6.5.5 Esercizio

Esercizio sulla ricerca binaria in una tabella ordinata?

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*:

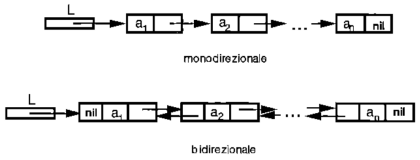
- una lista *monodirezionale* consente di andare da un elemento solo al successivo;
- una lista *bidirezionale* consente di andare da un elemento al successivo o al precedente.

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

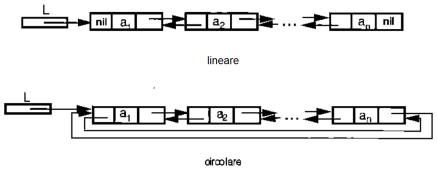
Figura 7.8: Varianti delle liste a puntatori

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.

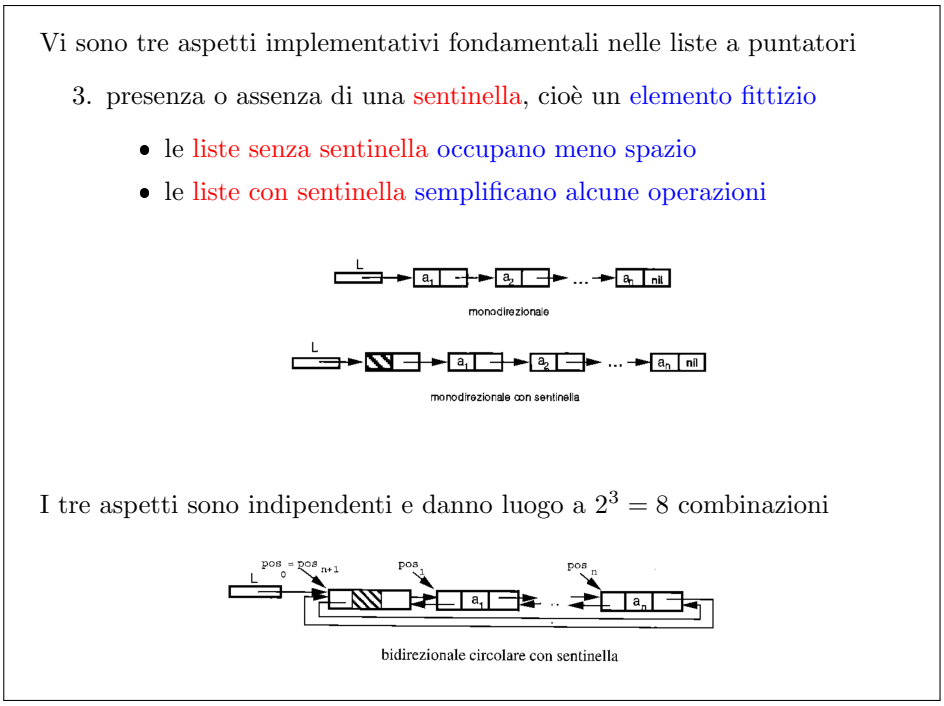


Figura 7.10: Varianti delle liste a puntatori

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`

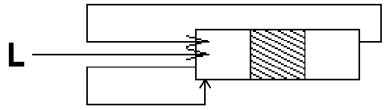


Figura 7.11: Lista vuota

```
  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]

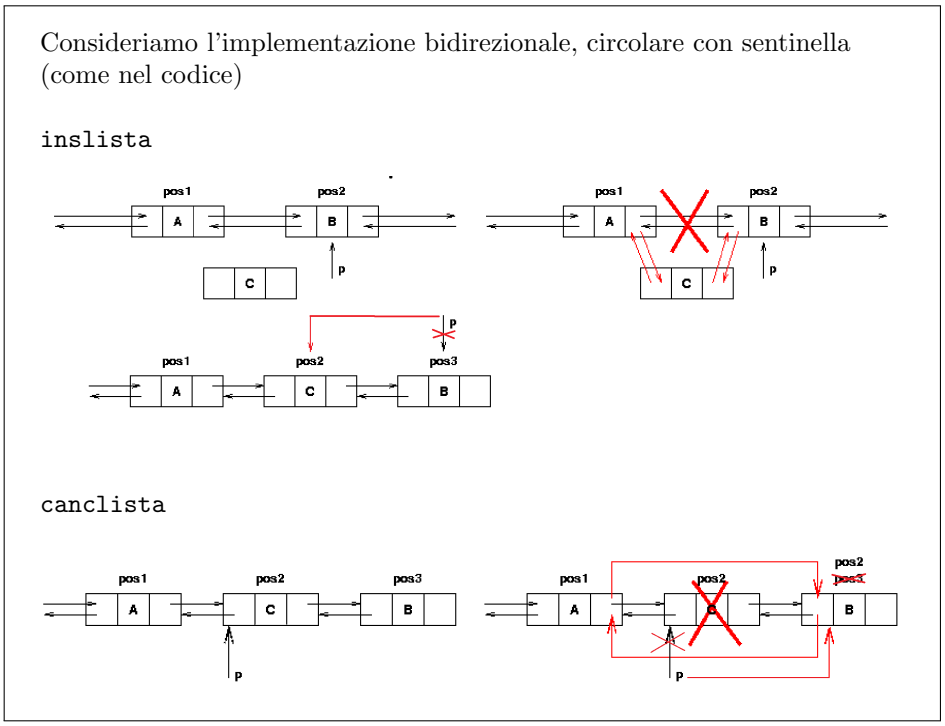


Figura 7.12: Inserimento e cancellazione

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 Analisi della complessità

Come nell'esercizio di gestione delle matrici statiche, concludiamo con un'analisi di complessità, che in una relazione andrebbe fornita direttamente insieme alla descrizione dell'algoritmo.

Consideriamo la procedura di esecuzione delle mosse. Ancora una volta, non sappiamo a priori quante saranno, per cui ne indichiamo il numero con m, che è una delle grandezze che definiscono la dimensione di dati. Le m mosse sono p prestiti di libri disponibili e r rese di libri restituiti, oltre alla mossa terminale che indica la conclusione di tutta la sequenza di operazioni. Questa richiede banalmente tempo costante.

Il prestito cerca un libro nello scaffale, nel caso peggiore scorrendolo tutto. Diciamo che i libri sullo scaffale al principio della giornata sono n, e poi non fanno altro che calare via via. Se il libro cercato viene trovato, va aggiunto alla lista dei resi, in tempo costante.

[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.

Le restituzioni, invece, consistono nel cercare il libro nella lista dei prestiti, ed eventualmente rimuoverlo da essa. I libri in prestito sono al massimo n, dato che nella lista compaiono solo i libri prestati nel giorno corrente. Che il libro fosse stato prestato il giorno stesso o un giorno precedente, viene comunque aggiunto in tempo costante alla lista dei resi.

Nel complesso, l'esecuzione delle mosse comporta $\Theta(mn)$ confronti fra i libri oggetto delle mosse e quelli delle due liste. A questo punto, bisogna definire se il confronto di due libri, cioè di due stringhe richiede tempo costante, perché si assume un limite superiore alla lunghezza del loro identificativo (titolo e autore), oppure tempo proporzionale alla lunghezza l delle stringhe stesse. Quindi, il risultato potrebbe essere $\Theta(mn)$ come $\Theta(mnl)$.

7.6 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).

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

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: 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.

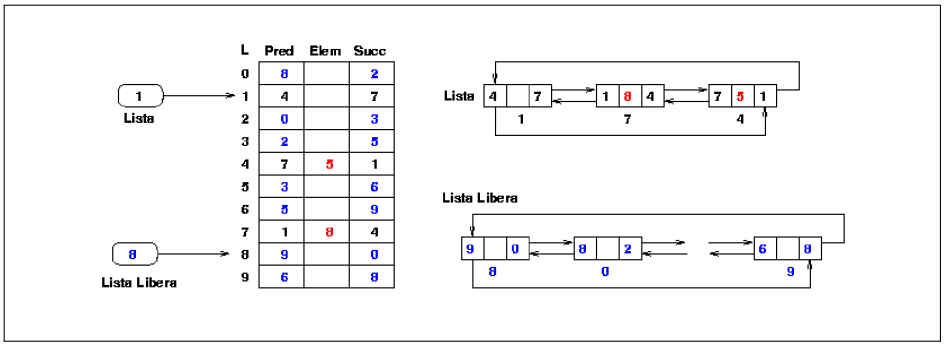


Figura 7.14: Liste: implementazione con vettori e cursori

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 parti-

colare 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.

```
#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

[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.

7.6.1 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.

[29] Si potrebbe pensare di riallocare il vettore con dimensioni superiori, raffinamento che qui non discutiamo, dato che comporta diverse questioni tecniche.

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;
  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.

[30] Ma forse in effetti potremmo usare le informazioni che sono appena state scritte nel nuovo elemento.

7.7 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

[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.

`\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.

7.8 Esercizi

7.8.1 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).

7.8.2 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.

7.8.3 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.

7.8.4 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.

7.8.5 Ricerca di un elemento in una lista

Aggiungere la funzione `TrovaLibro` alla libreria e usarla.

7.8.6 Altre implementazioni

Ovviamente, ciascuna delle altre 14 implementazioni presentate in precedenza (7 a puntatori e 7 a cursori) costituisce un utile esercizio.

7.8.7 Implementazione con indirizzi separati del primo e ultimo elemento

7.8.8 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

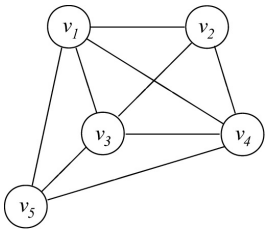


Figura 8.1: Grafi

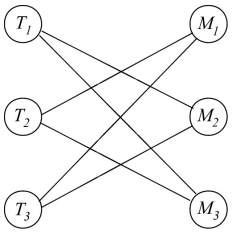
Un *grafo non orientato* descrive una relazione binaria simmetrica tra elementi di un insieme base finito $V = \{v_1, v_2, \ldots, v_n\}$, i cui elementi vengono convenzionalmente chiamati *vertici*.

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

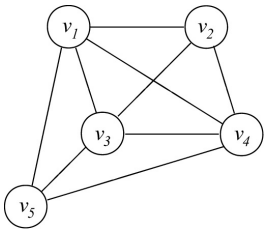
Non si può eseguire un task su un task, né una macchina su una macchina

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.

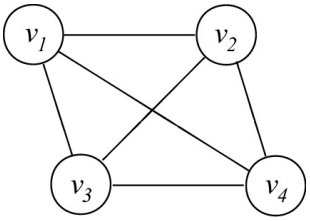
Un grafo che ha tutti i possibili lati si dice *grafo completo*, e siccome i lati sono tutte le coppie possibili, il numero dei lati di un grafo completo coincide con il numero delle combinazioni di n oggetti a due a due, $C_{n,2}$. Se non ammettiamo le ripetizioni, questo numero è espresso del coefficiente 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

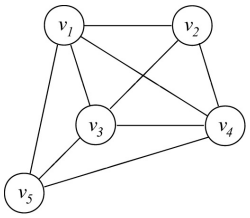
$H = (U, X)$ è un sottografo di $G = (V, E)$ se

- è un grafo
- $U \subseteq V$ e $X \subseteq E$

È un sottografo ricoprente quando $U = V$

È un sottografo indotto quando $X = E_U = \{\{u, v\} \in E : u, v \in U\}$

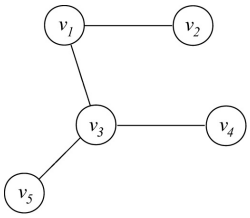
$G = (V, E)$



$V = \{v_1, v_2, v_3, v_4, v_5\}$

$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_1, v_5\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}, \{v_3, v_5\}, \{v_4, v_5\}\}$

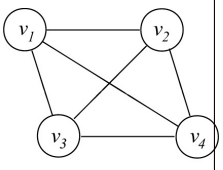
$H_1 = (U_1, X_1)$



$U_1 = \{v_1, v_2, v_3, v_4, v_5\} = V$

$X_1 = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_3, v_4\}, \{v_3, v_5\}\}$

$H_2 = (U_2, X_2)$



$U_2 = \{v_1, v_2, v_3, v_4\}$

$X_2 = \{\{v_1, v_2], \{v_1, v_3], \{v_1, v_4], \{v_2, v_3], \{v_2, v_4], \{v_3, v_4]\} = E_{U_2}$

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.

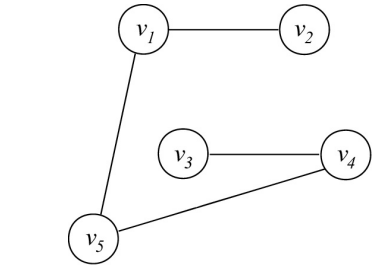
- Un cammino è una sequenza di lati, in cui ognuno condivide un estremo col lato precedente e l'altro col lato successivo (*se esistono*)

$$P = \left(\{v_{\pi_0}, v_{\pi_1}\}, \{v_{\pi_1}, v_{\pi_2}\}, \ldots, \{v_{\pi_{k-1}}, v_{\pi_k}\}\right)$$

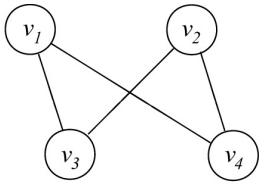
I vertici estremi v_{π_0} e v_{π_k} sono connessi

- 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.

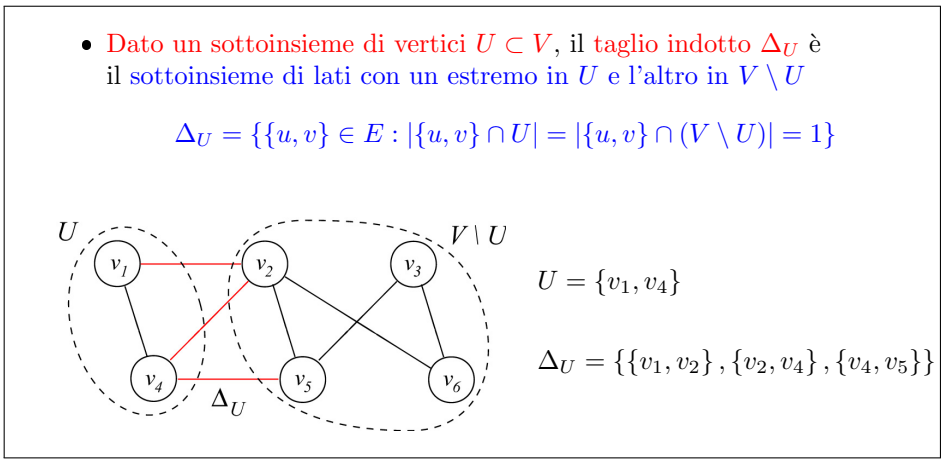


Figura 8.7: Tagli

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. I lati che hanno un estremo nel sottoinsieme indicato e l'altro nel sottoinsieme complementare costituiscono il così detto *taglio indotto*, indotto da ciascuno dei due sottoinsiemi (la definizione è simmetrica.

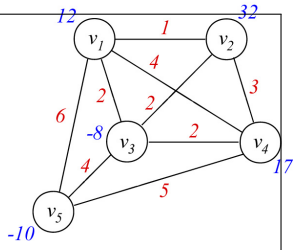
I tagli sono interessanti perché determinano le vie per le quali bisogna necessariamente passare se si vuole andare da un insieme di posizioni del grafo a quello complementare, per esempio vie di passaggio in caso di guerra, di commercio, di flussi idraulici, elettrici, ecc...

Nelle applicazioni, è tipico che ai vertici o ai lati si associno informazioni (per esempio, valori numerici). In tal caso, si dice che il grafo è *pesato* sui vertici o

sui lati, o su entrambi (vedi Figura 8.8). Per esempio, se il grafo rappresenta una rete stradale, i lati tipicamente hanno delle lunghezze associate, o dei tempi di percorrenza, o dei flussi di traffico (quante automobili percorrono la strada in un'unità di tempo). Nel caso dei giochi, una posizione può essere vincente per uno dei due giocatori o per l'altro, o che sia tendenzialmente più forte per uno o per l'altro dei due giocatori.

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

Figura 8.8: Grafi pesati

Le Figure 8.9, 8.10 e 8.11 riportano una carrellata di modelli di problemi basati su grafi che non commentiamo in dettaglio perché non hanno nessun legame con l'esercizio pratico di questo capitolo e pochi con quelli dei capitoli seguenti, ma illustrano molto efficacemente la pervasività del concetto di grafo in matematica applicata.

8.2 Definizioni per i grafi orientati

Il concetto di grafo orientato è molto simile a quello di grafo non orientato, con l'unica differenza che descrive relazioni binarie asimmetriche, cioè nelle quali l'ordine dei due elementi appartenenti ad ogni coppia è significativo: se a è in relazione con b, non è detto che b sia in relazione con a.

Alcuni testi differenziano nettamente la nomenclatura dei grafi orientati da quella dei grafi non orientati, mentre altri le confondono assolutamente. Qui seguiremo l'approccio più rigoroso, avvertendo che però la letteratura lo è molto meno.

Un *grafo orientato* (vedi Figura 8.12) sarà quindi una coppia di insiemi $G = (N, A)$, dove N è un insieme di oggetti elementari, detti *nodi*, e A è un insieme di coppie ordinate di nodi, dette *archi*. In quanto oggetti elementari, nulla distingue veramente i nodi dai vertici, salvo il nome. Gli archi, invece, differiscono dai lati perché l'ordine dei due elementi di ogni coppia è fondamentale. La Figura 8.12 mostra la rappresentazione grafica, nella quale i nodi compaiono come cerchi e gli archi come linee che collegano coppie di nodi, dotate di una freccia che indica la

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 8.9: Modelli basati su grafi (1)

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 8.10: Modelli basati su grafi (2)

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 8.11: Modelli basati su grafi (3)

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

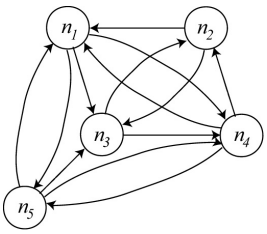


Figura 8.12: Grafi orientati

direzione della relazione. Si può vedere che n_1 è in relazione con n_1, mentre n_1 non lo è con n_2, perché i due nodi sono legati da una sola linea, con la freccia in una sola direzione. In altri casi, come fra n_1 e n_5 ci sono due archi con direzioni opposte. Infatti, il grafo ha 5 nodi, corrispondenti ai 5 vertici del grafo non orientato della Figura 8.1, ma, invece di avere 9 lati, ha 13 archi, con alcune relazioni in entrambi i versi e altre in uno solo.

- 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 8.13: Cammini, cicli e tagli orientati

Proseguendo con la nomenclatura (vedi Figura 8.13), i due nodi che appartengono a un arco e lo definiscono si dicono *estremi* dell'arco, ma in particolare il primo viene detto *coda* e il secondo *testa*, come se si trattasse di un animale allungato dal primo al secondo nodo. Si dice anche che l'arco esce dal primo nodo ed entra nel secondo, per cui il *grado* del nodo, cioè il numero degli archi incidenti in esso, va diviso in *grado entrante* (numero degli archi entranti) e *grado uscente* (numero degli archi uscenti), la cui somma è il *grado totale*.

Avremo ancora dei cammini, che tecnicamente dovrebbero chiamarsi *cammini orientati*, nei quali non basta più che che ogni arco abbia un estremo in comune con l'arco successivo e l'altro estremo col precedente, ma si richiede anche che la sequenza rispetti la direzione, cioè la coda di ciascun arco sia la testa del precedente e la testa di un arco sia la coda del successivo. Una sequenza di archi orientati in maniera alternata non è un cammino orientato (sarebbe come seguire una strada e a un certo punto trovarsi di fronte a un senso unico vietato: la strada si interrompe).

Qualunque coppia di nodi lungo un cammino orientato in cui il primo nodo sia precedente al secondo è *fortemente connessa*. La connessione forte non è una relazione di equivalenza, perché non è simmetrica.

Può avvenire che l'ultimo nodo di un cammino coincida col primo. In tal caso, si parla di *ciclo orientato*, ovvero *circuito*. Infine, dato un sottoinsieme di nodi, si può individuare il sottografo indotto, che è formato da tutti gli archi che hanno entrambi gli estremi in quei nodi, determinare il sottoinsieme complementare con il relativo sottografo indotto e identificare gli archi che sono a cavallo dei due sottografi indotti. Tali archi hanno un estremo nel primo sottografo e l'altro nel secondo. Nel complesso formano la *sezione indotta* (che corrisponde al taglio dei grafi non orientati). Una sezione è a sua volta composta dall'unione della *sezione uscente*, composta dagli archi che vanno dal sottoinsieme dato al suo complemento, e della *sezione entrante*, composta dagli archi che tornano dal sottoinsieme complementare a quello dato. Questo concetto è particolarmente importante per i problemi di flusso, dei quali questo corso non si occupa.

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...)

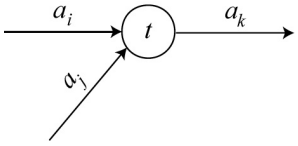
Nelle reti stradali urbane, molte strade sono a senso unico

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

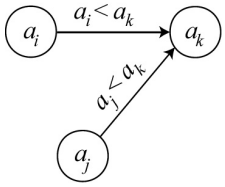


Figura 8.14: Modelli basati sui grafi orientati

I grafi orientati modellano una quantità di situazioni pratiche (vedi Figura 8.14). Per esempio, in alcune reti sociali esiste il concetto simmetrico di "amicizia", mentre in altre vale quello di asimmetrico di coppia ("follower","leader"). In molti giochi, le mosse che collegano le posizioni sono irreversibili, e quindi inducono un chiaro orientamento. Per esempio, negli scacchi i pedoni vanno solo in avanti. In molte reti stradali ci sono strade a senso unico. Nella descrizione dei progetti, può capitare di rappresentare certi eventi importanti (detti "milestone", che segnano la fine di un'attività e l'inizio di un'altra, come nodi e le attività (dato che collegano due di questi eventi) come archi (modelli "activity on arc", AOA). Altri modelli di rappresentazione di progetti, nei quali si privilegia la relazione di precedenza fra attività, le attività sono rappresentate come nodi e la precedenza fra due attività come un arco che collega in corrispondeti nodi (modelli "activity on node", AON). Nella figura, abbiamo tre attività (i, j e k), delle quali la terza può cominciare solo quando le prime due sono terminate. Il modello AOA, a sinistra, introduce un evento t, che segna il passaggio dalle prime due attività alla terza, descritto come un nodo nel quale entrano i due archi corrispondenti alle attività i e j e dal quale esce l'arco corrispondente all'attività k. A destra, invece, il modello AON descrive

le tre attività con nodi e descrive la precedenza di i rispetto a k e quella di j rispetto a k come due archi.

8.3 La struttura dati astratta grafo

Nel seguito parleremo solo di grafi orientati, perché, come accennato in precedenza, una relazione simmetrica equivale a due relazioni asimmetriche uguali e contrarie. Legare n_1 e n_5 con un lato, ovvero formare con essi una coppia non ordinata $\{n_1, n_5\}$ oppure legarli con due archi opposti, ovvero formare le due coppie ordinate (n_1, n_5) e (n_5, n_1) è praticamente lo stesso. Quindi, esiste una relazione biunivoca fra grafi non orientati e grafi orientati simmetrici. Non esiste invece in generale per grafi non orientati e i grafi orientati generici.

Questo significa che una rappresentazione valida per grafi orientati generici vale in particolare per quelli simmetrici e si può direttamente estendere ai grafi non orientati. Risolvendo un problema più ampio, si evita di occuparsi del problema più ristretto. Concentriamoci quindi sui grafi orientati.

Come per tutte le strutture dati astratte, dobbiamo definire l'insieme degli oggetti che la compongono e le operazioni che si possono compiere su tali oggetti. Faremo una semplificazione: anziché considerare l'insieme di tutti i grafi orientati, considereremo l'insieme dei grafi orientati che hanno un dato insieme N di nodi, che poi significa un dato numero $n = |N|$ di nodi, dato che i nodi sono oggetti elementari privi di attributi. Lo scopo di avere un insieme fissato di nodi è evitare di dover definire operazioni come l'aggiunta e la cancellazione di un nodo, che sono perfettamente fattibili, ma non aggiungono poco alla comprensione logica dell'argomento, complicano abbastanza l'implementazione e molto spesso non sono necessarie, dato che l'insieme su cui è definita la relazione binaria rappresentata dal grafo è dato una volta per tutte. Qualora non lo sia, la definizione va raffinata di conseguenza, in modo analogo a quando siamo passati dai vettori, che hanno dimensione fissa, alle tabelle o alle liste, che hanno dimensione libera, cioè ammettendo la creazione di grafi vuoti e l'aggiunta e la cancellazione di nodi.

Teniamo invece variabile l'insieme degli archi, per cui ipotizzeremo di introdurre tre operazioni fondamentali: l'aggiunta a un grafo dato di un arco fra due nodi dati, la cancellazione da un grafo dato di un arco fra due nodi dati e il test di esistenza in un grafo dato di un arco fra due nodi dati (vedi Figura 8.15).

La procedura `insarco`, quindi riceve in ingresso un grafo sull'insieme N e una coppia ordinata di elementi di N. Restituisce in uscita un altro grafo, coincidente con quello di partenza a cui è stato aggiunto un arco dal primo al secondo nodo. È spontaneo chiedersi che fare se tale arco esiste già. Al solito, si tratta di decidere una convenzione: si può considerare l'errore inaccettabile e terminare il programma, limitarsi a restituire un codice di errore oppure ragionare in termini di unione di insiemi e concludere che, aggiungendo un arco a un insieme che lo conteneva già, non cambia niente. Infine, si può ignorare il problema scaricando sull'utente il compito di sincerarsi che l'arco non esista prima di chiamare la funzione (e ovviamente segnalando chiaramente questa necessità nella documentazione). Per efficienza, seguiremo quest'ultima strada. Un'altra questione da definire è se l'operazione effettivamente produce un nuovo grafo oppure modifica il precedente. Come in tutte le strutture discusse finora, per efficienza sceglieremo di realizzare la procedura sostituendo il grafo originale con il nuovo.

La procedura `cancarco` funziona in maniera analoga. Riceve un grafo sull'insieme N e una coppia ordinata di nodi e restituisce un grafo identico al primo, ma privo dell'arco corrispondente alla coppia di nodi data (cioè, in effetti, modificando

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 8.15: Grafi: operazioni

il grafo iniziale). Anche qui, il caso dell'eliminazione di un arco che non esiste può essere risolto in vario modo, e la nostra scelta sarà di imporre all'utente che non lo faccia.

La necessità di questa verifiche suggerisce che un utente dovrebbe in qualche modo poter sapere se una data coppia ordinata di nodi ordinata corrisponde o no a un arco del grafo. La procedura `esistearco` riceve un grafo sull'insieme N e una coppia ordinata di nodi e restituisce il valore logico vero se la coppia corrisponde a un arco e il valore falso in caso contrario.

Sono tre procedure banalissime. Nel seguito ne introdurremo altre più interessanti, suggerite via via dalle esigenze dell'esercizio pratico che svilupperemo.

E se il grafo fosse pesato? Viene spontaneo definire delle funzioni di accesso, che ricevano il grafo e un nodo (o un arco) e restituiscano il valore del peso di tale nodo (o arco). Analogamente, si potrebbero introdurre procedure di accesso in scrittura per modificare i pesi stessi.

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 8.16: Grafi: operazioni

Come sempre, per tener conto del fatto che i grafi occupano memoria sarà necessario definire procedure per gestire tale memoria, dunque per creare e per distruggere grafi (vedi Figura 8.16). La procedura `creagrafo` riceve l'insieme di nodi N (cioè il loro numero n), alloca la memoria dinamica necessaria e restituisce il grafo con insieme di nodi N e insieme di archi vuoto. La procedura `distruggegrafo` riceve un grafo generico e lo riduce a un grafo vuoto, privo sia di nodi sia di archi.

8.4 Implementazioni dei grafi

Le implementazioni che descriveremo (vedi Figura 8.17) non sono perfettamente identiche a quelle presentate nelle dispense di teoria, per motivi di convenienza pratica. Le differenze sono comunque minime. Considereremo grafi non pesati, specificando poi in che modo sia possibile adattare l'implementazione alla presenza di pesi.

Tutte le implementazioni trattano l'insieme dei nodi nel modo più efficiente possibile: essendo un insieme finito di oggetti privi di caratteristiche ulteriori, i nodi sono in corrispondenza biunivoca con gli interi positivi fra 1 e $n = |N|$ e si possono rappresentare con tali numeri. Dunque, l'insieme corrisponde semplicemente con il numero n.

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 8.17: Grafi: implementazioni

In linea di principio, i nodi potrebbero avere informazioni associate, e si potrebbe trovare più conveniente rappresentarli nel grafo come puntatori a strutture che contengano tali informazioni. Volendo rendere la definizione astratta, si può ricorrere alla definizione di un tipo `nodo` con l'istruzione `typedef`. Va detto che, però, l'uso degli indici interi consente di conservare eventuali informazioni ausiliarie in vettori e l'accesso ad esse attraverso tali vettori, per cui questa astrazione ha vantaggi abbastanza limitati. A noi qui interessa solo l'aspetto topologico del grafo, per cui non applicheremo questa idea.

8.4.1 Lista degli archi

La prima implementazione, indicata come *lista degli archi* è banalmente la definizione del grafo: all'insieme dei nodi aggiunge un insieme di archi, che sono coppie di nodi, dunque di numeri interi. Rappresentiamo gli archi come una lista, dato che (in questa sede) vogliamo tenere una completa dinamicità. La Figura 8.18 mostra un grafo di 6 nodi e 9 archi. Il grafo è rappresentabile con il numero 6, che indica quanti sono i nodi, e con una lista di 9 coppie di numeri interi (da $(1, 2)$ a $(5, 6)$).

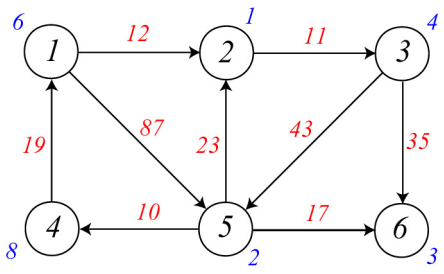
Si vede subito il vantaggio principale di questa rappresentazione: la compattezza. L'occupazione è ridotta all'essenziale (puntatori e sentinella della lista si potrebbero eliminare se il grafo non dovesse essere totalmente dinamico sugli archi). L'occupazione spaziale asintotica è quindi $\Theta(m)$, lineare nel numero degli archi $m = |A|$.

L'aggiunta e l'eliminazione di archi corrisponde all'aggiunta e all'eliminazione di elementi da una lista o da una tabella, dunque (conoscendone la posizione) a un tempo costante ($\Theta(1)$), se non occorre introdurre un ordinamento.

Lo svantaggio fondamentale di questa implementazione riguarda la verifica di esistenza. Questa operazione è analoga alla ricerca di un libro nella lista dei prestiti o dei resi nel Capitolo 7: il tempo richiesto è lineare nel numero di elementi della lista, dunque $\Theta(m)$, dato che nel caso pessimo bisogna guardarli tutti (per esempio, quando l'arco non esiste). Questo è piuttosto inefficiente.

Se il grafo fosse pesato, ci troveremmo a dover anche rappresentare una funzione definita su un insieme finito. In generale, il modo più efficiente di rappresentare una funzione è di usare un vettore in cui gli indici corrispondono agli elementi dell'insieme di definizione. Siccome i vertici corrispondono ai numeri da 1 a n, basta un vettore con gli stessi indici: il peso del vertice 2 è l'elemento di indice 2 del vettore w.

Analogamente, il costo degli archi si può rappresentare con un vettore, a patto di assegnare ad ogni arco un indice numerico, che naturalmente è l'indice progressivo di scorrimento della lista. In alternativa, l'informazione sul costo potrebbe invece essere conservata nell'arco stesso, che quindi non sarebbe una coppia di valori (gli indici dei nodi), ma una terna (gli indici dei nodi e il costo).



$N \to \{1, 2, 3, 4, 5, 6\}$

$w \to [\ 6\ \ 1\ \ 4\ \ 8\ \ 2\ \ 3\ ]$

$A \to \big((1,2), (1,5), (2,3), (3,5), (3,6), (4,1), (5,2), (5,4), (5,6)\big)$

$c \to [\ 12 \quad 87 \quad 11 \quad 43 \quad 35 \quad 19 \quad 23 \quad 10 \quad 17\ ]$

- Vantaggio: rappresentazione compatta ($\Theta(|A|)$)
- Svantaggio: ricerca inefficiente di un arco dato ($\Theta(|A|)$)

Figura 8.18: Lista degli archi

8.4.2 Matrice di adiacenza

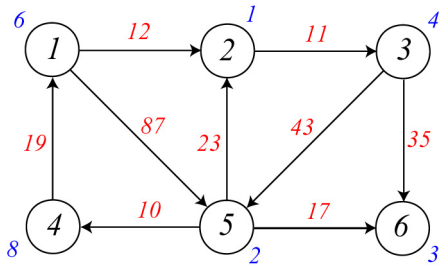
La seconda implementazione, indicata come *matrice di adiacenza* è una delle implementazioni discusse nelle dispense di teoria: gli archi corrispondono a un vettore di incidenza bidimensionale, indicizzato sul primo e sul secondo nodo di ogni arco potenziale, dunque una matrice quadrata. La Figura 8.19 mostra questa rappresentazione per il grafo già considerato.

Questa rappresentazione è una risposta all'inefficienza nella ricerca di un arco, dati i nodi estremi. Il modo più efficiente di determinare se un elemento appartiene o no a un insieme è di associare gli elementi dell'insieme a indici numerici e avere un vettore indicizzato su tali indici e contenente due valori logici, rispettivamente corrispondenti al fatto che l'elemento appartenga (1) o non appartenga (0) all'insieme. Nel caso del grafo, conviene avere due indici, dunque una matrice, con le righe corrispondenti al nodo di origine e le colonne al nodo di destinazione degli archi. Qualora il grafo fosse pesato, è possibile sostituire al valore logico 1 il valore del peso, ovviamente a patto che il peso 0 non sia possibile, o che vi sia un altro

valore impossibile che rappresenti l'assenza dell'arco (in figura, il trattino contenuto in diverse celle della matrice). Nella figura, la coppia $(4, 1)$ corrisponde alla cella della riga 4 e della colonna 1, che contiene il valore 19, indicando che l'arco $(4, 1)$ esiste e che il suo costo è 19. La matrice contiene valori logici se ci interessa solo la topologia, valori interi se abbiamo anche una funzione di costo.

Il vantaggio di questa rappresentazione è che il test di esistenza è istantaneo, ovvero richiede tempo costante, $\Theta(1)$. Lo stesso vale per l'aggiunta e la cancellazione di archi, dato che basta accedere direttamente in scrittura alla cella corrispondente della matrice, ancora in tempo costante.

Lo svantaggio fondamentale è che l'occupazione di memoria è enorme, perché è in $\Theta(n^2)$. Per un grafo completo, la cosa ha senso: anzi, è persino meglio che usare la lista degli archi, che richiede due numeri interi per ogni arco, anziché un singolo valore logico. Ma per grafi meno densi, in particolare se grandi, la differenza fra il numero di archi potenziale e quello effettivo diventa rapidamente enorme: un grafo stradale da 1 000 nodi raramente ha più di 4 000 archi, ma la sua matrice di adiacenza ha 1 000 000 di celle, con un'occupazione centinaia di volte maggiore del necessario.



$$N \rightarrow \{1, 2, 3, 4, 5, 6\} \qquad w \rightarrow [\,6\ 1\ 4\ 8\ 2\ 3\,]$$

$$(A, c) \rightarrow \begin{bmatrix} - & 12 & - & - & 87 & - \\ - & - & 11 & - & - & - \\ - & - & - & - & 43 & 35 \\ 19 & - & - & - & - & - \\ - & 23 & - & 10 & - & 17 \\ - & - & - & - & - & - \end{bmatrix}$$

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 8.19: Matrice di adiacenza

8.4.3 Vettore di *forward-star*

La terza implementazione, indicata come *forward star*, è equivalente all'implementazione indicata come *vettore delle liste di adiacenza* nelle dispense di teoria, ma mostra qualche piccola differenza. Questa rappresentazione è in qualche modo intermedia fra le altre due. L'idea è di non avere una singola lista per tutti gli archi, ma una lista per ogni nodo, contenente gli archi uscenti dal nodo stesso. L'espressione inglese *forward-star* indica che si tratta della "stella" degli archi (sono archi con un'origine comune, come i raggi di una stella), che escono (vanno avanti, cioè "forward") da uno stesso nodo origine.

La Figura 8.20 mostra un vettore, il cui primo elemento è la lista degli archi uscenti dal nodo 1, cioè $(1, 2)$ e $(1, 5)$. Volendo evitare ridondanze, ci si può limitare a indicare le destinazioni degli archi, dato che l'origine è sempre la stessa (1) ed è necessariamente nota per poter scorrere la lista. Al nodo di destinazione si può aggiungere il peso dell'arco, se il grafo è pesato, come in questo caso. Lo stesso vale per gli elementi successivi del vettore, fino all'ultimo, che è una lista vuota, dato che il nodo 6 non ha archi uscenti. Quindi, le singole liste del vettore possono avere come elementi interi archi, coppie contenenti il nodo di destinazione e il costo di un arco, oppure semplicemente gli indici dei nodi di destinazione.

Il vantaggio di questa rappresentazione è che per determinare se un arco fra due nodi dati esiste basta raggiungere la corrispondente cella del vettore e scorrere la lista degli archi uscenti. La complessità di questa ricerca comporta un tempo costante per arrivare all'elemento giusto del vettore e un tempo proporzionale al grado uscente del nodo per scorrere gli elementi della lista, dunque $\Theta(\delta_v^+)$. Se non c'è un limite superiore al grado uscente (alcuni grafi ce l'hanno per natura), nel caso pessimo la complessità è $\Theta(n)$, che è in genere minore di $\Theta(m)$. Per esempio, in un grafo da 1 000 nodi, gli archi potrebbero anche essere un milione, ma in ogni lista ce ne saranno al massimo 1 000 (contando anche gli autoanelli). Certo, non è un'operazione in tempo costante, ma l'occupazione in spazio richiede un elemento per ogni arco più un vettore indicizzato sui nodi, dunque $\Theta(m + n)$. È una complessità superiore a quella della semplice lista di archi, ma molto inferiore a quella della matrice di adiacenza.

Un vantaggio che discuteremo meglio nel seguito è che per conoscere gli archi uscenti da un nodo dato, questa struttura fornisce esattamente l'informazione necessaria nel tempo strettamente indispensabile ($\Theta(\delta_v^+)$), mentre la matrice di adiacenza richiederebbe di scorrere tutta una riga (in tempo $\Theta(n)$) e la lista degli archi richiederebbe di scorrere tutti gli archi (in tempo $\Theta(m)$).

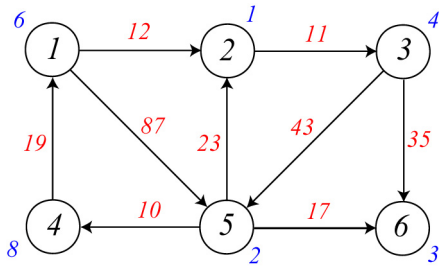
D'altra parte, per conoscere gli archi entranti in un nodo bisogna ricorrere a uno scorrimento di tutte le liste (in tempo $\Theta(n + m)$). Per ovviare a questo limite, potremmo aggiungere un secondo vettore, che fornisca per ogni nodo la lista degli archi entranti. Si tratterebbe della *backward-star*. Porterebbe però ad un'occupazione doppia di memoria, dato che ogni arco comparirebbe in due liste.

Tutte le considerazioni svolte sinora suggeriscono che non esiste una rappresentazione perfetta per i grafi: ognuna ha vantaggi e svantaggi, e la rappresentazione più adatta dipende strettamente dalle operazioni che si intendono svolgere sui grafi rappresentati e dalla disponibilità pratica di memoria rapportata alla dimensione dei grafi da manipolare.

8.5 Laboratorio: determinazione del sottografo indotto

Passiamo ora a un esercizio pratico. Riprendiamo il problema di individuare il sottografo indotto da un certo sottoinsieme di nodi e lo risolviamo con un programma in C, utilizzando tutte e tre le rappresentazioni sopra descritte, in modo da implementare le funzioni fondamentali (introducendone altre suggerite dal problema da risolvere).

Inoltre, risolveremo il problema con tre algoritmi diversi tra loro, valutando per ciascuno la complessità ottenuta utilizzando ciascuna delle tre implementazioni. Combinando tre algoritmi a livello astratto con tre implementazioni a basso livello otterremo 9 possibili soluzioni del problema, ognuna con la sua analisi di complessità temporale e spaziale.



$$N \rightarrow \{1, 2, 3, 4, 5, 6\} \qquad w \rightarrow [\,6\ 1\ 4\ 8\ 2\ 3\,]$$

$$(A, c) \rightarrow \left[\begin{array}{l} (2,12) \rightarrow (5,87) \perp \\ (3,11) \perp \\ (5,43) \rightarrow (6,35) \perp \\ (1,19) \perp \\ (2,23) \rightarrow (4,10) \rightarrow (6,17) \perp \\ \quad \perp \end{array}\right]$$

dove $\perp$ indica una lista vuota

- (S)Vantaggi: ricerca di efficienza intermedia di un arco dato ($\Theta\left(\delta_v^+\right)$)
  occupazione di memoria intermedia ($\Theta\left(|N| + |A|\right)$)
- Svantaggio: nessuna informazione sugli archi entranti
  (a meno che sia accompagnato dal vettore delle *backward star*)

Figura 8.20: Vettore delle liste di incidenza (*forward star*)

Vediamo l'esempio della Figura 8.20: se dei 6 nodi consideriamo solo il sottoinsieme $U = \{1, 2, 4, 5\}$, ci interessano gli archi che hanno entrambi gli estremi in U, cioè $A\left(U\right) = \{(1,2), (1,5), (4,1), (5,2), (5,4)\}$. Il problema è abbastanza semplice da enunciare, ma ammette diversi semplici algoritmi che ci consentiranno di osservare in che modo la complessità di ciascuno dipenda dall'implementazione delle strutture dati, oltre che, ovviamente, alle operazioni che l'algoritmo prevede.

Il concetto di struttura dati astratta implica che qualunque algoritmo possa essere eseguito su qualunque implementazione che consenta le operazioni richieste dall'algoritmo. D'altra parte, alcune implementazioni si prestano particolarmente bene per alcuni algoritmi, perché in esse le operazioni che richiedono sono particolarmente efficienti, nonché facili da esprimere in forma di codice. Questo è talmente vero che nel momento in cui si concepisce un algoritmo spesso si pensa già all'implementazione più opportuna, anche se le due cose dal punto di vista astratto sono del tutto indipendenti. Non è una contraddizione: sono le due facce della stessa medaglia.

8.5.1 Avvio dell'implementazione

Il problema considerato non è un problema pratico, e quindi non abbiamo una fase di modellazione, in cui decidiamo quali strutture dati astratte (quali oggetti matematici) impiegare per rappresentare gli oggetti concreti del problema affrontato. Qui abbiamo la richiesta esplicita di lavorare su grafi, dunque oggetti matematici.

Partiamo dal file dei dati, `grafo1.txt`, che adotta un formato autoesplicativo per rappresentare lo stesso grafo usato come esempio nelle figure precedenti: si tratta di un banale elenco di archi, rappresentati come coppie di numeri interi separati da

virgole e racchiusi fra parentesi tonde.

```
(1,2) (1,5) (2,3) (3,5) (3,6) (4,1) (5,2) (5,4) (5,6)
```

Il file non indica esplicitamente il numero dei nodi. Questo è un problema, e potenzialmente anche un errore, dato che evidentemente 6 nodi compaiono esplicitamente, dato che sono toccati da archi, ma nulla esclude che ve ne siano tanti altri senza archi incidenti. Faremo la semplificazione di ipotizzare che non vi siano grafi con nodi isolati (anche perché i nodi isolati non giocano alcun ruolo nella determinazione dei sottografi indotti). Quindi, possiamo concludere che il numero dei nodi è pari al numero dei nodi osservati, e se ipotizziamo anche che i loro indici partano da 1, possiamo concludere che il numero dei nodi n coincide con il massimo indice osservato nel file.

Dovremo caricare e inserire in un'opportuna struttura in memoria il grafo. Poi riceveremo un sottoinsieme. Questo potrebbe essere contenuto in un altro file di testo, lo riceveremo invece come elenco di indici sulla linea di comando, per fare esercizio anche su questo argomento. Quindi, il programma richiede di ricevere come primo argomento il nome del file di testo che contiene il grafo e come successivi argomenti gli indici numerici dei nodi del sottografo. Questo consente di eseguire il programma molte volte sullo stesso grafo, semplicemente fornendogli diversi elenchi di nodi.

Il file listato `sottografo0.c` fornisce le solite direttive, la solita dichiarazione di lunghezza massima convenzionale per le stringhe (`ROW_LENGTH`) e una definizione di tipo per dichiarare vettore dinamici di interi (il tipo `vint`, che è un puntatore a intero). Nel programma principale è predisposto lo schema che decompone in modalità *top-down* il problema generale in sottoproblemi.

```
/* Programma principale */
int main (int argc, char *argv[])
{

  /* Legge da linea di comando il nome del file che contiene il grafo
     e gli indici dei nodi del sottografo */

  /* Carica il grafo da file */

  /* Stampa a video il grafo */

  /* Stampa a video il sottografo indotto */

  return EXIT_SUCCESS;
}
```

La prima fase dell'implementazione consisterà quindi nel sostanziare i commenti con chiamate a opportune funzioni, da dichiarare e definire, inizialmente vuote. Farlo richiederà di definire le strutture dati da utilizzare, in particolare il grafo e l'elenco dei nodi che identificano il sottografo. Partiremo rappresentando il grafo `G` nella forma che utilizza la matrice di adiacenza, dunque includendo la relativa libreria `grafo-ma.h`. Per gli indici del sottografo useremo una tabella, ovvero un un vettore dinamico `U` di interi e una variabile intera `n` che ne indichi la lunghezza. Dovremo poi discutere se il sottografo indotto sia rappresentato a sua volta come un grafo, oppure se ci limiteremo a stamparlo sfruttando il grafo e il sottoinsieme di nodi. Per efficienza, dato che non occorre fare altro che stamparlo, seguiremo la seconda opzione. Questo evita il costo temporale e spaziale di costruire le relative strutture, ma anche un'altra questione spinosa. Un sottografo è anche un grafo, ma

contiene un sottoinsieme dei nodi del grafo di partenza: come vanno numerati tali nodi? Se conserviamo la numerazione originale, violiamo la convenzione che in ogni grafo i nodi corrispondano ai primi numeri interi positivi; se la cambiamo, i nomi degli archi del sottografo cambiano rispetto a quelli del grafo, che sono gli stessi, renderendo inconfrontabile il risultato con i dati.

Scorriamo anche rapidamente la libreria `grafo-ma.h`. In base a tale rappresentazione, (vedi Figura 8.21 un grafo è un *record* contenente il numero dei nodi n, il numero degli archi m (questo l'abbiamo aggiunto) e una matrice di valori logici che per ogni riga e per ogni colonna (dall'indice 1 all'indice n) contiene valore 1 se c'è un arco dal nodo corrispondente alla riga a quello corrispondente alla colonna e 0 se non c'è.

```
typedef struct _grafo grafo;
struct _grafo
{
   int n;                                    (numero dei nodi)
   int m;                                    (numero degli archi)
   boolean **MatriceAdiacenza;               (matrice di adiacenza)
};
```

Figura 8.21: Matrice di adiacenza: implementazione in C

Dunque, due interi e un doppio puntatore a valori logici.

```
typedef int nodo;
typedef struct _grafo grafo;

struct _grafo
{
  int n;
  int m;
  boolean **MatriceAdiacenza;
};
```

Vi sono poi procedure per creare e distruggere grafi e le tre procedure fondamentali per inserire nuovi archi, cancellare archi e valutare se una data coppia di nodi effettivamente corrisponde a un arco nel grafo dato. Useremo pesantemente la prima per caricare il grafo in memoria e la terza per risolvere il problema, mentre la seconda sarà completamente inutile.

8.5.2 Prima fase (`sottografo1.c`)

Cominciamo, al solito, convertendo i singoli passaggi dello schema in chiamate a procedure che li realizzeranno e che andremo poi a implementare

```
/* Programma principale */
int main (int argc, char *argv[])
{
  char filedati[ROW_LENGTH];
  vint S;
  int ns;
  grafo G;
```

```
  /* Legge da linea di comando il nome del file che contiene il grafo
     e gli indici dei nodi del sottografo */
  InterpretaLineaComando(argc,argv,filedati,&S,&ns);

  /* Carica il grafo da file */
  CaricaGrafo(filedati,&G);

  /* Stampa a video il grafo */
  stampagrafo(&G);

  /* Stampa a video il sottografo indotto */
  StampaSottografoIndotto(&G,S,ns);

  return EXIT_SUCCESS;
}
```

Oltre alle semplifi chiamate, abbiamo aggiunto un po' di interfaccia per rendere la stampa dei risultati più leggibile: dato che stamperemo sia l'intero grafo sia il sottografo indotto nello stesso formato, converrà premettere a ognuna delle due stampe un'introduzione (rispettivamente, `G =` e `G_S =`) e aggiungere a capi che separino meglio tra loro le stampe.

La procedura `InterpretaLineaComando` riceve al solito gli argomenti `argc` e `argv`, che permettono di accedere alla linea di comando e ne ricava la stringa in cui conservare il nome dei file del grafo, il vettore dinamico in cui conservare l'elenco dei nodi e il loro numero. Mentre la stringa è il classico vettore statico di caratteri, il vettore dinamico `S` e la cardinalità `ns` sono entrambi passati per indirizzo, perché vanno modificati internamente. Il vettore, in particolare, va allocato, dato che a priori non sappiamo quanto sarà lungo.

```
/* Legge da linea di comando il nome del file che contiene il grafo
   e gli indici dei nodi del sottografo */
void InterpretaLineaComando (int argc, char *argv[], char *filedati,
    vint *pS, int *pns);
```

La procedura `CaricaGrafo` carica il grafo dal file di testo in una struttura `G` di tipo `grafo`, che possiamo dichiarare avendo incluso l'apposita libreria. Essendo il risultato, lo passiamo per indirizzo.

```
/* Carica il grafo G dal file filedati */
void CaricaGrafo (char *filedati, grafo *pG);
```

Per stampare il grafo a video realizzeremo una funzione che opera su un grafo. Siccome tale funzione appare utile, la aggiungeremo alla libreria `grafo-la.h`. Scriveremo quindi il prototipo non in alto nel file corrente, ma nello stesso file intestazione, e il corpo nel file listato `grafo-la.c`. Un vantaggio di questo approccio è che nel realizzare la procedura potremo accedere direttamente a tutte le strutture interne del grafo, senza doverci limitare alle funzioni di interfaccia. Seguendo la convenzione adottata in tutto il corso, useremo le lettere minuscole per il nome della procedura `stampagrafo`, dato che è una funzione di libreria a basso livello. Si potrebbe passarle il grafo per copia, dato che la stampa non modifica il grafo, ma continueremo a passarlo per indirizzo, per limitare il numero di argomenti e snellire il passaggio dei parametri.

Per concludere, manca la procedura `StampaSottografoIndotto`, che determina gli archi del sottografo indotto e li stampa a video. La procedura riceve il grafo (per indirizzo per efficienza), il vettore `S` (che è già un indirizzo, dunque non occorre compattarlo) e la lunghezza (che è un semplice numero).

```
/* Stampa a video il sottografo indotto sul grafo G dal sottoinsieme
    S di ns nodi */
void StampaSottografoIndotto (grafo *pG, vint S, int ns);
```

Ora occorre riempire le singole procedure, partendo da quelle ausiliarie.

Siccome l'interpretazione della linea di comando è abbastanza sofisticata, prepariamone prima uno schema. Prima dobbiamo verificare che numero degli argomenti sia sensato. Poi assegneremo il primo argomento come nome del file dei dati. Quindi, determineremo quanti siano gli argomenti successivi, che sono i nodi del sottografo indotto. Conoscerne il numero permette di allocare il vettore. Dopo di che, potremo riempire il vettore convertendo in numeri gli argomenti residui.

```
/* Legge da linea di comando il nome del file che contiene il grafo
   e gli indici dei nodi del sottografo */
void InterpretaLineaComando (int argc, char *argv[], char *filedati,
    vint *pS, int *pns)
{

  /* Controlla il numero degli argomenti */

  /* Determina il nome del file dei dati */

  /* Determina il numero di nodi del sottografo indotto */

  /* Alloca il vettore dei nodi del sottografo indotto */

  /* Riempie il vettore dei nodi del sottografo indotto */
}
```

Scendendo nei dettagli, il numero sensato di argomenti è almeno pari a 2, cioè il nome del programma chiamato e il nome del file dei dati. Questo è il caso limite, in cui il sottoinsieme di vertici dati è vuoto, e di conseguenza anche il sottografo indotto lo è.

Fatto questo, il numero dei nodi del grafo indotto è pari ad `argc-2`, perché i primi due argomenti sono il nome del programma e quello del file dei dati. Questo determina la lunghezza del vettore allocato e il valore dell'ultimo risultato. Gli argomenti che seguono dovrebbero essere tutti numeri interi compresi fra 1 e $|N|$. In effetti, però, gli elementi del vettore `argv` sono stringhe. Prima di assegnare gli argomenti al vettore `S`, bisogna convertirli da stringa a numero con la funzione `atoi` oppure con la funzione `sscanf`, che consente anche di verificare che le stringhe rappresentino davvero numeri interi. Siccome gli indici vanno da 2 ad `argc-1` in `argv` e da 1 a `*pns` in `S`, bisogna anche scalarli.

```
/* Legge da linea di comando il nome del file che contiene il grafo
   e gli indici dei nodi del sottografo */
void InterpretaLineaComando (int argc, char *argv[], char *filedati,
    vint *pS, int *pns)
{
  int p;

  /* Controlla il numero degli argomenti */
```

```
  if (argc < 2)
  {
    fprintf(stderr,"Errore nella linea di comando!\n");
    exit(EXIT_FAILURE);
  }

  /* Determina il nome del file dei dati */
  strcpy(filedati,argv[1]);

  /* Determina il numero di nodi del sottografo indotto */
  *pns = argc-2;

  /* Alloca il vettore dei nodi del sottografo indotto */
  *pS = (vint) calloc(*pns+1,sizeof(int));
  if (*pS == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione del vettore dei nodi del
        sottografo indotto!\n");
    exit(EXIT_FAILURE);
  }

  /* Riempie il vettore dei nodi del sottografo indotto */
  for (p = 1; p <= *pns; p++)
    if (sscanf(argv[p+1],"%d",&(*pS)[p]) != 1)
    {
      fprintf(stderr,"Errore nel formato della linea di comando!\n");
      exit(EXIT_FAILURE);
    }
}
```

La procedura `CaricaGrafo` riceve il nome del file di testo e restituisce il grafo, ovviamente passato per indirizzo. Deve aprire il file e creare il grafo. Per far questo, la funzione `creagrafo` richiede il numero dei nodi del grafo, che è quindi la prima cosa da determinare nel file. Abbiamo già detto che si tratta di scorrere i lati determinando il massimo indice dei nodi che vi compaiono. Creato il grafo, occorre ripetere una seconda volta la scansione degli archi, leggendoli uno per volta e aggiungendoli al grafo con la funzione `insarco`. Questa decomposizione del problema consente di usare le funzioni fondamentali di gestione del grafo elencate nel file di intestazione. Alla fine, dovremo chiudere il file.

```
/* Carica il grafo G dal file filedati */
void CaricaGrafo (char *filedati, grafo *pG)
{

  /* Apre il file */

  /* Conta i nodi del grafo: legge gli archi e salva il massimo
      indice */

  /* Torna all'inizio del file */

  /* Crea un grafo vuoto di n nodi */

  /* Scorre gli archi inserendoli nel grafo */

  /* Chiude il file */
}
```

Steso lo schema, entriamo nel dettaglio dell'implementazione. Per leggere gli archi, applichiamo un ciclo di lettura che sfrutta la struttura ripetitiva del formato.

Ogni arco si apre con una parentesi tonda aperta, seguita da un numero intero, una virgola, un altro numero intero e una parentesi tonda chiusa. Introduciamo qualche elemento di flessibilità, ammettendo che fra l'uno e l'altro di questi termini ci possano essere uno o più caratteri separatori qualsiasi. La funzione `fscanf` consente di realizzare tutto questo in modo molto semplice, gestendo automaticamente i dettagli con un'opportuna stringa di formato, e fornendo al tempo stesso una condizione di permanenza nel ciclo di lettura, cioè il fatto di riuscire a identificare in ogni blocco di lettura relativo a un arco i due indici interi dei nodi estremi dell'arco. La stringa di formato `" (%d ,%d )"` mantiene tutta la flessibilità richiesta, dato che gli spazi bianchi corrispondono a sequenze qualsiasi (anche vuote) di separatori prima delle parentesi e della virgola, mentre le specifiche `"%d"` fanno lo stesso prima di ciascuno dei due numeri interi (aggiungere altri spazi non farebbe danno, ma non cambierebbe nulla). I due numeri interi letti determinano i nodi di origine e destinazione dell'arco e vanno confrontati con l'attuale indice massimo per aggiornare il numero totale di nodi. Per caricare gli archi, occorre rieseguire la lettura, tornando al principio del file. Quindi, si esegue un ciclo assolutamente analogo al primo, nel quale il corpo inserisce archi nel grafo invece di determinare l'indice massimo. Per semplicità, non verifichiamo che gli archi siano univoci.

Compilando il codice, si può verificare che quanto meno la sintassi sia corretta. Il passo seguente, come già fatto più volte, sarà implementare la libreria, in modo da verificare che le operazioni sin qui compiute siano corrette (per esempio, stampando il grafo caricato da file).

8.5.3 Seconda fase (`grafo-ma.c`): implementazione della matrice di adiacenza

Vogliamo ora riempire le procedure vuote del file `grafo-ma0.c`, cioè creare e distruggere un grafo, inserirvi un nuovo arco o cancellarne uno vecchio, valutare se esista un arco fra due nodi dati e stampare un grafo, ipotizzando che il grafo sia descritto da una matrice di adiacenza.

Partiamo con la creazione del grafo. Bisogna assegnare un valore corretto a ognuno dei tre campi della struttura `grafo`: il numero di nodi è dato come argomento, quello degli archi è nullo, la matrice va allocata e annullata, per cui la funzione `calloc` è particolarmente adatta.

```
/* Crea un grafo *pG di n nodi senza archi */
void creagrafo (int n, grafo *pG)
{
  int o;

  pG->n = n;
  pG->m = 0;
  pG->MatriceAdiacenza = (boolean **) calloc(pG->n+1,sizeof(boolean
      *));
  if (pG->MatriceAdiacenza == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione della matrice di
        adiacenza!\n");
    exit(EXIT_FAILURE);
  }
  for (o = 1; o <= pG->n; o++)
  {
    pG->MatriceAdiacenza[o] = (boolean *)
        calloc(pG->n+1,sizeof(boolean));
    if (pG->MatriceAdiacenza[o] == NULL)
    {
      fprintf(stderr,"Errore nell'allocazione della riga %d della
          matrice di adiacenza!\n",o);
```

```
      exit(EXIT_FAILURE);
    }
  }
}
```

La distruzione di un grafo procede nella sequenza opposta, prima deallocando le singole righe, poi la "costola" che le punta, e infine azzerando il numero di archi e nodi. Azzerare anche il puntatore alla matrice (cosa che la deallocazione non fa!) è utile per creare una condizione facilmente testabile che distingua i grafi distrutti dagli altri ed evitare che un utente maldestro cerchi di usare il puntatore per accedere a un'area di memoria che non è più riservata.

```
/* Distrugge il grafo *pG */
void distruggegrafo (grafo *pG)
{
  int o;


  for (o = 1; o <= pG->n; o++)
    free(pG->MatriceAdiacenza[o]);
  free(pG->MatriceAdiacenza);
  pG->MatriceAdiacenza = NULL;

  pG->m = 0;
  pG->n = 0;
}
```

L'inserimento di un arco in un grafo rappresentato come matrice di adiacenza è quasi banale: basta assegnare il valore `TRUE` alla cella corrispondente e incrementare il numero totale di archi.

```
/* Aggiunge l'arco (o,d) al grafo G
   N.B.: non verifica che l'arco sia nuovo! */
void insarco (nodo orig, nodo dest, grafo *pG)
{
  pG->m++;
  pG->MatriceAdiacenza[orig][dest] = TRUE;
}
```

Notiamo che, come anticipato, non stiamo verificando che l'arco inizialmente non esistesse. In tal caso, la matrice sarebbe comunque corretta, ma il numero di archi sarebbe eccessivo. Non pare complicato modificare il codice in modo da evitare questo errore, scegliendo quindi la soluzione che ignora l'inserimento. Non lo facciamo perché nelle altre implementazioni il controllo sarebbe decisamente più complicato e aggraverebbe pesantemente la complessità dell'operazione.

La cancellazoine di un arco da un grafo si riduce ad assegnare il valore `FALSE` alla cella corrispondente e decrementare il numero totale di archi (con la stessa osservazione fatta sopra sul fatto che si introduce un errore qualora l'arco non esista).

```
/* Cancella l'arco (o,d) dal grafo G
   N.B.: non verifica che l'arco sia presente! */
void cancarco (nodo orig, nodo dest, grafo *pG)
{
  pG->m--;
```

```
  pG->MatriceAdiacenza[orig][dest] = FALSE;
}
```

Quindi, decidere se un arco esiste richiede banalmente di consultare la cella corrispondente nella matrice e restituirne il valore.

```
/* Determina se l'arco (orig,dest) appartiene al grafo *pG */
boolean esistearco (nodo orig, nodo dest, grafo *pG)
{
  return pG->MatriceAdiacenza[orig][dest];
}
```

La banalità e l'efficienza di questa operazione in questa implementazione indurrà generazioni di studenti a farne largo uso nei progetti adottando le altre implementazioni, producendo quindi codice inefficiente che riceverà valutazioni basse. Fra l'altro, il fatto in sé di usare una procedura che alloca un *record* di attivazione e vi copia tre argomenti solo per accedere a un campo di una struttura pone qualche domanda. L'unica giustificazione è che vogliamo una struttura dati astratta per poter cambiare l'implementazione in seguito senza dover modificare il codice che ne fa uso.

Concludiamo con la stampa del grafo. Vogliamo scorrere gli archi e stampare a video la loro origine e destinazione. Viene naturale scorrere la matrice, valutare per ogni cella se l'arco esista o no (qui possiamo accedere direttamente alla matrice, dato che la procedura è interna alla libreria) e, nel caso l'arco esista, procedere alla sua stampa nel formato richiesto.

```
/* Stampa a video il grafo G */
void stampagrafo (grafo *pG)
{
  int o, d;


  for (o = 1; o <= pG->n; o++)
    for (d = 1; d <= pG->n; d++)
      if (pG->MatriceAdiacenza[o][d]) printf("(%d,%d) ",o,d);
}
```

8.5.4 Algoritmi per il calcolo del sottografo indotto

Ora introdurremo tre algoritmi per la determinazione del sottografo indotto su un grafo da un sottoinsieme di nodi. Realizzeremo ciascuno dei tre algoritmi su ciascuna delle tre implementazioni del grafo. Inoltre, anche il sottoinsieme dei nodi può essere rappresentato in vari modi, e discuteremo l'effetto che questo può avere sulla complessità finale. Dato che ogni algoritmo è compatibile con ogni rappresentazione dell'una e dell'altra struttura dati, il numero di possibili combinazioni è potenzialmente esplosivo. Un'analisi teorica a priori ci dovrebbe guidare, restringendo il numero delle combinazioni che ha senso esplorare. Qui ne implementeremo invece molte, per passarle in rassegna e fare un esercizio di analisi, ma anche per sottolineare fisicamente le differenze tra le varie versioni e mostrare come in pratica capiti spesso di rendersi conto a posteriori che certe operazioni si possono fare in

modo migliore e sviluppare varianti più efficienti a partire da quelle meno efficienti. Al termine, scopriremo che non esiste una versione che sia vincitrice assoluta.

Abbiamo già sviluppato un'implementazione per il grafo con la matrice di adiacenza e un'implementazione per il sottoinsieme dei nodi con un vettore che contiene l'elenco dei loro indici. Entrambe sono state scelte per semplicità.

Dato un grafo $G = (N, A)$ e un sottoinsieme di nodi $S \subseteq N$, vogliamo determinare gli archi del sottografo $G_S = (S, A_S)$ indotto da S su G e stamparli a video. Consideriamo i seguenti tre algoritmi.

Algoritmo 1 In base al principio per cui cerchiamo archi e gli archi sono particolari coppie di nodi, scorriamo tutte le coppie ordinate (o, d) di nodi di N e valutiamo per ciascuna tre condizioni: se il nodo origine o appartenga al sottoinsieme S, se il nodo destinazione d vi appartenga e se la coppia corrisponda a un arco di A. Se la coppia soddisfa tutte le condizioni, la stampiamo. 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
```

Quanto costa questo algoritmo? Prima facciamo una stima a priori, senza scegliere l'implementazione delle strutture dati. Ragionevolmente, ipotizziamo che si possano scorrere i nodi impiegando tempo costante per ognuno. Rimangono i tre test di appartenenza (dell'origine o e della destinazione d a S e della coppia (o, d) ad A) e la stampa dell'arco, che indichiamo con procedure astratte, `appartienetabella` e `esistearco`, alla cui complessità attribuiamo espressioni simboliche. I due cicli annidati corrispondono a sommatorie:

$$T_1 = \sum_{o=1}^{n} \sum_{d=1}^{n} (2T_S + T_A + T_{\mathrm{pr}}) \tag{8.1}$$

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 $T_{\mathrm{pr}}(\cdot)$ il tempo per la stampa di un arco. La stampa di un arco ragionevolmente richiederà ancora tempo costante.

Il costo dei test di appartenenza dipende dall'implementazione. Cominciamo a implementare e analizzare l'algoritmo con le strutture semplici che abbiamo già definito: S è un vettore di indici di lunghezza n_S, mentre A è la matrice di adiacenza del grafo.

Implementazione di Algoritmo 1 con la matrice di adiacenza La procedura `StampaSottografoIndotto` realizza pedissequamente lo pseudocodice che abbiamo visto sopra, con lo scorrimento doppio dei nodi di N, la valutazione di appartenenza di entrambi all'insieme dei nodi e della coppia all'insieme degli archi, e la stampa.

```
/* Stampa a video il sottografo indotto sul grafo G dal sottoinsieme
   S di ns nodi */
void StampaSottografoIndotto (grafo *pG, vint S, int ns)
{
```

```
  nodo o, d;

  /* Algoritmo 1 (scorrimento doppio dei nodi di N e valutazione di
       appartenenza
     di o e d a S e di (o,d) ad A) */
  for (o = 1; o <= pG->n; o++)
    for (d = 1; d <= pG->n; d++)
      if ( appartienetabella(o,S,ns) && appartienetabella(d,S,ns) &&
           esistearco(o,d,pG) ) printf("(%d,%d) ",o,d);
}
```

Mentre il test di appartenenza di una coppia all'insieme degli archi è già disponibile e implementato nella libreria, quello di appartenenza di un nodo a una tabella va ancora realizzato con il solito procedimento: chiamata, dichiarazione e definizione.

```
/* Determina se l'intero o appartiene alla tabella S di lunghezza ns
   */
boolean appartienetabella (int o, vint S, int ns);
```

La funzione `appartienetabella` riceve il nodo cercato e la tabella, cioè il vettore S e la lunghezza n_S e restituisce un valore logico. Per farlo, scorre la tabella confrontando il valore cercato con quello contenuto nelle varie posizioni della tabella stessa, e uscendo non appena lo trova, oppure al termine quando ha dimostrato la non appartenenza del valore alla tabella.

```
/* Determina se l'intero o appartiene alla tabella S di lunghezza ns
   */
boolean appartienetabella (int o, vint S, int ns)
{
  int i;

  for (i = 1; i <= ns; i++)
    if (o == S[i]) return TRUE;

  return FALSE;
}
```

Per inciso, questa è un'operazione di quelle su cui si scrivono interi libri di algoritmica, cioè la ricerca di un elemento in un insieme. L'abbiamo discussa nel capitolo sulle liste e ne discuteremo ancora introducendo strutture più sofisticate dette *dizionari*. Le dispense di teoria descrivono un'alternativa possibile detta *ricerca binaria*, che è più efficiente, ma richiede che l'insieme sia ordinato. L'insieme S non è ordinato, perché l'abbiamo semplicemente caricato da linea di comando, ma un'idea potrebbe anche essere di rendere il caricamento più raffinato per poi sfruttare l'ordinamento di S.

Le abbiamo attribuito un nome con caratteri tutti minuscoli perché somiglia molto a una funzione di libreria che potrebbe essere inclusa in un librera di gestione di tabelle, che potremmo includere nell'intero progetto.

Compilato il tutto, si può eseguire per verificare che la stampa del grafo sia corretta e che la determinazione e stampa del sottografo indotto lo sia anch'essa.

Analisi di Algoritmo 1 con la matrice di adiacenza Ora ragioniamo sulla complessità dell'algoritmo, avendo definito dettagliatamente l'implementazione di tutte le sue strutture dati.

Se S è rappresentato con una tabella non ordinata, la valutazione di appartenenza di ciascuno dei due nodi richiede tempo $T_S \in \Theta(n_S)$, dato che comporta nel caso pessimo lo scorrimento di tutto il vettore s che elenca gli indici dei nodi[1].

Se adottiamo l'implementazione del grafo con la matrice di adiacenza, la valutazione di esistenza dell'arco richiede tempo costante: $T_A \in \Theta(1)$, perché è una semplice consultazione della matrice di adiacenza.

Infine, la stampa dell'arco richiede tempo costante: $T_{\text{pr}} \in \Theta(1)$.

Sommando questi termini sugli indici o e d, dato che sono tutti indipendenti da o e d, basta moltiplicarli per n^2 e ottenere

$$T_1(n, n_S) = \sum_{o=1}^{n} \sum_{d=1}^{n} (2\Theta(n_S) + \Theta(1) + \Theta(1)) \in \Theta\left(n^2 n_S\right) \tag{8.2}$$

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 per definizione $n_S \leq n$, 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.

La complessità spaziale dell'Algoritmo 1 è dominata dallo spazio occupato dal grafo: $S_1(n) \in \Theta\left(n^2\right)$, che domina nettamente lo spazio $\Theta(n_S)$ occupato dalla tabella.

Nell'ipotesi di avere una tabella ordinata, dovremmo aggiungere un tempo $\Theta(n_S \log n_S)$ per ordinarla (con algoritmi più efficienti di quelli che abbiamo già discusso, algoritmi che tratteremo in dettaglio in un capitolo successivo). In compenso, ridurremmo il tempo della ricerca nella tabella da lineare a $\Theta(\log n_S)$, e quindi l'intero algoritmo a $\Theta\left(n^2 \log n_S\right)$. Vedremo che questa strada, per ora promettente, in realtà non porta vantaggi sufficienti.

Algoritmo 2 L'Algoritmo 1 ha un'inefficienza ovvia e macroscopica: scorre tutte le coppie (o, d) anche per elementi o che non appartengono a S. Vedendola da un altro punto di vista, la procedura `appartienetabella` viene eseguita n volte su ogni nodo o, mentre basterebbe eseguirla una volta sola, nel ciclo esterno anziché in quello interno.

L'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
```

[1] 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.

```
7:         end for
8:     end if
9: end for
```

Più che un nuovo algoritmo, è una correzione dell'Algoritmo 1.

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. Quindi, la sommatoria che corrisponde al ciclo esterno su o può essere divisa in due sommatorie, relative ai due casi possibili, e noi sappiamo quanti termini cadono in una o nell'altra. In tutti i termini compare l'addendo T_S, in quelli corrispondenti agli elementi $o \in S$ compare anche il ciclo sul nodo d. Possiamo rappresentare questi casi in cui o cade in S attraverso lo scorrimento dell'indice i_o nella tabella da 1 a n_S:

$$T_2 = \sum_{o=1}^{n} T_S + \sum_{i_o=1}^{n_S} \sum_{d=1}^{n} (T_S + T_A + T_{\text{pr}}) \tag{8.3}$$

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 $n_S \ll n$. Se per caso abbiamo un'istanza in cui un sottoinsieme di vertici molto piccolo induce un sottografo su un grafo molto grande, questa banale modifica migliora fortemente la complessità.

Implementazione di Algoritmo 2 con la matrice di adiacenza L'algoritmo 2 scorre i nodi origine e per ciascuno chiama `appartienetabella`. In caso positivo, scorre i nodi di destinazione e valuta la loro appartenenza a S e l'appartenenza della coppia all'insieme degli archi del grafo.

```
/* Stampa a video il sottografo indotto sul grafo G dal sottoinsieme
    S di ns nodi */
void StampaSottografoIndotto (grafo *pG, vint S, int ns)
{
  nodo o, d;

  /* Algoritmo 2 (scorrimento dei nodi origine di N e valutazione di
       appartenenza di o a S
     scorrimento dei nodi destinazione di N e valutazione di
         appartenenza di d a S e di (o,d) ad A */
  for (o = 1; o <= pG->n; o++)
    if (appartienetabella(o,S,ns))
      for (d = 1; d <= pG->n; d++)
        if ( appartienetabella(d,S,ns) && esistearco(o,d,pG) )
            printf("(%d,%d) ",o,d);*/
}
```

La modifica è minima. Rimane l'idea che una tabella ordinata consentirebbe di ridurre il termine n_S a $\log n_S$, concludendo con una complessità in $\Theta(nn_S \log n_S)$. Teniamo ancora in sospeso l'idea di migliorare l'implementazione della tabella.

Algoritmo 3 Il terzo algoritmo cambia completamente approccio. Infatti, evita completamente la ricerca dei nodi nella tabella scorrendo direttamente il vettore che li contiene. È un principio di economia generale, quando si vuole scorrere un sottoinsieme trovare il modo di scorrerlo direttamente, anziché scorrere un insieme più ampio che lo contiene e valutare l'appartenenza dell'elemento generico al sottoinsieme specifico. Questo principio varrebbe anche se il test di appartenenza fosse efficientissimo, cosa che qui, tra l'altro, non vale[2].

L'Algoritmo 3, dunque, scorre tutte le coppie di posizioni nella tabella. Per chiarire che stiamo scorrendo una tabella, useremo gli indici numerici `io` e `id`, che faremo variare da 1 alla cardinalità `ns`. Ogni volta, recupereremo il nodo corrispondente all'indice corrente attraverso un accesso in lettura alla tabella (`s[io]` oppure `s[id]`). Determinati i due nodi, valutiamo se la coppia di nodi corrisponda a un arco in A, e in tal caso lo stampiamo.

```
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
```

Analisi di Algoritmo 3 con la matrice di adiacenza La complessità dell'algoritmo è data da un ciclo esterno che scorre la tabella con n_S iterazioni. In ciascuna, si ricava il nodo corrispondente a un dati indice, con una proiezione, quindi in tempo costante, indicato con il simbolo $T_{\text{find}}(\cdot)$, cioè il tempo richiesto per trovare il nodo in N dato un elemento del vettore che rappresenta S. Inoltre, ad ogni iterazione del ciclo esterno si esegue un ciclo interno, ancora sugli elementi della tabella, con n_S iterazioni. Dentro quest'ultimo ciclo, c'è un'altra proiezione, il test di appartenenza di una coppia di nodi all'insieme degli archi e la stampa.

$$T_3 = \sum_{i_o=1}^{n_S} \left(T_{\text{find}} + \sum_{i_d=1}^{n_S} \left(T_{\text{find}} + T_A + T_{\text{pr}} \right) \right) \tag{8.5}$$

Dati i tempi costanti delle operazioni elementari, il risultato è in $\Theta\left(n_S^2\right)$. Questo è certamente meglio del $\Theta\left(nn_S^2\right)$ dell'Algoritmo 2, che era meglio del $\Theta\left(n^2 n_S\right)$ di Algoritmo 1. Quindi, il terzo algoritmo è il migliore. La complessità spaziale, invece, rimane sempre la stessa. Ma queste conclusioni dipendono dall'implementazione che abbiamo scelto.

Se sostituissimo la tabella con una tabella ordinata, non cambierebbe nulla, perché abbiamo eliminato ogni necessità di una ricerca nella tabella. Ma se cambiassimo l'implementazione del grafo? Si può ragionare direttamente sulle espressioni astratte. In effetti, 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. È leggermente più difficile vedermo, ma il terzo algoritmo domina il secondo. Infatti, test di appartenenza all'insieme degli archi e stampa (T_A e T_{pr} sono contenuti in due sommatorie da 1 a n_S, anziché in una da 1 a n_S e una da 1 a n, dunque un numero maggiore di iterazioni. Per il resto, abbiamo una chiamata a T_{find} in una doppia sommatoria da 1 a n_S e un'altra racchiusa in una sommatoria

[2]Nota di importanza capitale per i progetti d'esame.

semplice. L'Algoritmo 2, invece, ha una chiamata a T_S racchiusa in una sommatoria da 1 a n_S e una da 1 a n e un'altra racchiusa in una sommatoria semplice da 1 a n_S. Cioè l'Algoritmo 2 esegue una ricerca in S almeno altrettanto spesso, ed eventualmente più spesso, di quanto l'Algoritmo 3 esegua la proiezione da indice a nodo (che è un'operazione di lettura di un vettore). In qualsiasi implementazione sensata, T_{find} non sarà mai più lenta di T_S, e non viene chiamata più spesso. Al limite, per $n \approx n_S$, i due algoritmi potrebbero giocarsela sulle costanti moltiplicative. Nel seguito, quindi, abbandoneremo i primi due algoritmi.

D'altra parte, questi non sono i soli algoritmi possibili. Ne introdurremo infatti un quarto e discuteremo un'altra implementazione del sottoinsieme dei nodi S che consenta di ottenere un test di appartenenza più rapido. Ovviamente, è possibile combinare i quattro algoritmi, 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.

Implementazione di Algoritmo 3 con la matrice di adiacenza L'Algoritmo 3 compie uno scorrimento doppio degli indici della tabella S e, determinati i due nodi corrispondenti, si esegue una valutazione di appartenenza della coppia ad A.

```
/* Stampa a video il sottografo indotto sul grafo G dal sottoinsieme
    S di ns nodi */
void StampaSottografoIndotto (grafo *pG, vint S, int ns)
{
  int io, id;
  /* Algoritmo 3 (scorrimento doppio degli indici in S, traduzione in
      nodi
     e valutazione di appartenenza di (o,d) ad A */
  for (io = 1; io <= ns; io++)
    for (id = 1; id <= ns; id++)
      if ( esistearco(S[io],S[id],pG) ) printf("(%d,%d)
          ",S[io],S[id]);
}
```

8.5.5 Quarta fase (grafo-la.c): implementazione della lista di archi

Ora proviamo a cambiare l'implementazione del grafo, conservando l'Algoritmo 3. Proviamo a utilizzare l'implementazione come lista di archi, descritta nella Figura 8.22.

Servirà una libreria `grafo-la.h`, nella quale un grafo è dichiarato come un *record* che contiene (come prima) il numero dei e il numero degli archi, ma ha una lista di archi al posto della matrice di adiacenza.

```
typedef struct _grafo grafo;

struct _grafo
{
  int n;
  int m;
  listaarchi L;
```

```
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 8.22: Lista degli archi: implementazione in C

};

L'idea non è assurda, perché risponde al principio generale enunciato più sopra, in base al quale conviene scorrere direttamente un sottoinsieme di oggetti (gli archi), anziché scorrere un insieme che lo contiene (le coppie di nodi) eseguendo su ciascun elemento un test di appartenenza al sottoinsieme. Per farlo, occorre che la rappresentazione degli archi consenta di scorrerli. Dato che il grafo include una lista, occorrerà includere nella libreria un'altra libreria `listaarchi.h` che gestisca questa lista. Tale libreria è già disponibile, completamente implementata nella variante bidirezionale, circolare e con sentinella. Non occorre dirne molto, in quanto è assolutamente identica alla libreria realizzata nel Capitolo 7, salvo per il fatto che l'informazione contenuta negli elementi delle liste non è una stringa di caratteri, ma una coppia di indici interi (gli indici dei nodi).

```
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;
};
```

La libreria `grafo-la.h` presenta le consuete 6 procedure fondamentali: creazione, distruzione, test di esistenza, aggiunta, cancellazione e stampa del grafo. Presenta però anche altre 4 procedure aggiuntive, che discuteremo nel seguito.

```
/* Crea un grafo G di n nodi, senza archi */
void creagrafo (int n, grafo *pG);

/* Distrugge il grafo G */
void distruggegrafo (grafo *pG);

/* Aggiunge l'arco (o,d) al grafo G
   N.B.: non verifica che l'arco sia nuovo! */
void insarco (nodo o, nodo d, grafo *pG);

/* Cancella l'arco (o,d) dal grafo G (non fa nulla se non esiste) */
void cancarco (nodo o, nodo d, grafo *pG);

/* Determina se l'arco (o,d) appartiene al grafo G */
boolean esistearco (nodo o, nodo d, grafo *pG);

/* Stampa a video il grafo G */
void stampagrafo (grafo *pG);

/* Restituisce la posizione del primo arco del grafo G */
posarco primoarco (grafo *pG);

/* Restituisce la posizione dell'arco successivo a pa nel grafo G */
posarco succarco (grafo *pG, posarco pa);

/* Determina se la posizione pa e' fuori della lista degli archi del
    grafo G */
boolean finearchi (grafo *pG, posarco pa);

/* Restituisce gli estremi *po e *pd dell'arco in posizione pa nel
    grafo G */
void leggeestremiarco (grafo *pG, posarco pa, nodo *po, nodo *pd);
```

Procediamo a implementare le procedure, partendo con quelle fondamentali, comuni anche all'implementazione già discussa. La creazione del grafo, a parte l'ovvia inizializzazione del numero dei nodi e degli archi, sfrutta la creazione di una lista di archi vuota, anziché della matrice di adiacenza.

```
/* Crea un grafo *pG di n nodi, senza archi */
void creagrafo (int n, grafo *pG)
{
  pG->n = n;
  pG->m = 0;
  pG->L = crealistaarchi();
}
```

Questo richiede tempo costante, $\Theta(1)$, con un notevole calo rispetto al caso della matrice di adiacenza.

Anche la distruzione del grafo sfrutta la distruzione della lista di archi.

```
/* Distrugge il grafo *pG */
void distruggegrafo (grafo *pG)
{
  pG->n = 0;
  pG->m = 0;
  distruggelistaarchi(&pG->L);
```

```
}
```

Come sappiamo dal capitolo sulle liste, la complessità di questa è lineare nel numero di archi, $\Theta(m)$, dato che dealloca tutti gli elementi, sentinella compresa.

L'aggiunta di un nuovo arco consiste, a parte l'incremento del numero di archi, nell'aggiunta dell'arco alla lista. Per analogia con l'esercizio sulle liste, aggiungeremo gli archi in coda, ma qualsiasi posizione sarebbe corretta. Ricordiamo che non faremo alcun controllo del fatto che l'arco non esista già.

```
/* Aggiunge l'arco (o,d) al grafo G
   N.B.: non verifica che l'arco sia nuovo! */
void insarco (nodo o, nodo d, grafo *pG)
{
  posarco pa;

  pa = succlistaarchi(pG->L, ultimolistaarchi(pG->L));
  inslistaarchi(pG->L,pa,o,d);
  pG->m++;
}
```

Il tutto richiede ovviamente tempo costante, $\Theta(1)$. Un eventuale controllo che l'arco non esista richiederebbe lo scorrimento di tutta la lista, per confrontarne gli elementi con quello da aggiungere, e quindi un forte aggravio nel tempo di esecuzione.

La cancellazione di un arco è nettamente più complessa, perché la procedura che cancella un arco da una lista richiede di conoscere la posizione dell'arco stesso, mentre al momento conosciamo solo gli indici dei nodi estremi. Per trovare la posizione di un arco a partire dagli estremi, sarà necessario scorrere la lista usando un cursore, leggendo ogni singolo elemento e confrontandone gli estremi con quelli dell'arco cercato, fino a trovarlo e uscire dal ciclo. La posizione trovata viene usata per cancellare l'arco (passandola per indirizzo, come richiede la libreria) e ovviamente si decrementa il numero di archi. Abbiamo dato per scontato che l'arco da cancellare esista. Se non esiste, il ciclo termina restituendo la condizione di fine della lista. Volendo, si potrebbe verificarla, prima di procedere a cancellare l'arco. Aggiungere il controllo qui e non nell'inserimento è un po' incoerente, ma va qui il costo della verifica è costante e impatta poco sull'efficienza della procedura.

```
/* Cancella l'arco (o,d) dal grafo *pG (non fa nulla se non esiste) */
void cancarco (nodo o, nodo d, grafo *pG)
{
  posarco pa;
  nodo orig, dest;

  for (pa = primolistaarchi(pG->L); !finelistaarchi(pG->L,pa); pa =
      succlistaarchi(pG->L,pa))
  {
    leggearco(pG->L,pa,&orig,&dest);
    if ( (o == orig) && (d == dest) ) break;
  }

  if (!finelistaarchi(pG->L,pa))
  {
    canclistaarchi(pG->L,&pa);
    pG->m--;
  }
}
```

Nel complesso, a causa dello scorrimento, il tempo richiesto dalla procedura è in $\Theta(m)$.

La funzione che valuta l'esistenza di un arco dati gli estremi esegue anch'essa uno scorrimento della lista, molto simile alla prima parte della procedura di cancellazione, salvo che si limita a restituire il valore logico vero se trova l'arco e quello falso se arriva in fondo alla lista senza trovarlo. Di conseguenza, la complessità è ancora lineare, $\Theta(m)$.

```
/* Determina se l'arco (o,d) appartiene al grafo *pG */
boolean esistearco (nodo o, nodo d, grafo *pG)
{
  posarco pa;
  nodo orig, dest;


  for (pa = primolistaarchi(pG->L); !finelistaarchi(pG->L,pa); pa =
      succlistaarchi(pG->L,pa))
  {
    leggearco(pG->L,pa,&orig,&dest);
    if ( (o == orig) && (d == dest) ) return TRUE;
  }

  return FALSE;
}
```

Concludiamo con la stampa del grafo. Qui emerge invece un netto vantaggio dell'implementazione a lista di archi: anziché scorrere tutte le coppie di nodi e valutare l'esistenza dell'arco corrispondente, possiamo sfruttare la struttura per limitarci a considerare gli archi esistenti. Il ciclo scorre gli archi, ne legge gli estremi e direttamente li stampa.

```
/* Stampa a video il grafo G */
void stampagrafo (grafo *pG)
{
  posarco pa;
  nodo o, d;


  for (pa = primolistaarchi(pG->L); !finelistaarchi(pG->L,pa); pa =
      succlistaarchi(pG->L,pa))
  {
    leggearco(pG->L,pa,&o,&d);
    printf("(%d,%d) ",o,d);
  }
}
```

La complessità cala quindi da $\Theta\left(n^2\right)$ a $\Theta(m)$, che può essere lo stesso (nei grafi completi) o molto meno (in quelli sparsi). Qualcuno potrebbe ovviamente pensare di copiare la procedura di stampa usata per la matrice di adiacenza, cioè il doppio ciclo sui nodi con il test di esistenza dell'arco: sarebbe una completa assurdità, dato che andrebbe contro la logica della struttura dati e, infatti, costerebbe niente meno che $\Theta\left(n^2 m\right)$: per un grafo completo da 1 000 nodi si salirebbe da 10^6 a 10^{12} operazioni (come dire, da 1 secondo a 11 giorni e mezzo)[3].

Un aspetto interessante di tutto questo è che, esattamente come quando abbiamo sostituito la libreria che implementava le liste a puntatori con quella che le

[3] La cosa è d'altronde piuttosto comune nei progetti d'esame.

implementava con vettori e indici, sostituendo la nuova libreria alla vecchia nelle direttive di inclusione e nella linea di comando della compilazione genera un programma che funziona correttamente senza bisogno di alcuna modifica al codice del programma principale.

Analisi di Algoritmo 3 con la lista degli archi L'uso della nuova implementazione nell'Algoritmo 3 è però tutt'altro che promettente. Infatti, possiamo recuperare l'analisi astratta e osservare che in essa il test di appartenenza di una coppia di nodi all'insieme degli archi era molto efficiente: $T_A \in \Theta(1)$. Se conserviamo l'Algoritmo 3 e ci limitiamo a cambiare l'implementazione del grafo, l'espressione della complessità computazionale rimane la stessa, salvo sostituire l'espressione di T_A col nuovo valore, dato che stampa e ricerca di un nodo nel sottoinsieme S rimangono invariate.

Ma con una lista il test di appartenenza può essere realizzato solo in un modo: scorrendo la lista. Quindi, la complessità diventa $T_A \in \Theta(m)$. Adattando l'espressione (8.5), si ottiene la stima:

$$T(n, m, n_S) = \sum_{i_o=1}^{n_S} \left(\Theta(1) + \sum_{i_d=1}^{n_S} (\Theta(1) + \Theta(m) + \Theta(1)) \right) \in \Theta\left(m n_S^2\right) \qquad (8.6)$$

che è peggiore di un fattore m di quella ottenuta con l'altra implementazione. Un grafo da 1 000 nodi può avere 1 000 000 di archi e l'algoritmo di conseguenza diventa un milione di volte più lento. Considerazioni analoghe valgono per l'Algoritmo 1 e l'Algoritmo 2, che del resto sono intrinsecamente dominati dall'Algoritmo 3: le loro complessità sono $\Theta\left(n^2(m + n_S)\right)$ per l'Algoritmo 1 e $\Theta(n n_S(m + n_S))$ per l'Algoritmo 2.

È vero d'altra parte che la complessità spaziale migliora in alcuni grafi, mentre peggiora in altri. Il grafo occupa infatti spazio $\Theta(m)$, avendo una lista di m coppie di interi e puntatori ausiliari, invece di una matrice di n^2 valori logici, e $m \leq n^2$. In assenza di autoanelli, si ha $m \leq n(n-1)$. Ora, in un grafo sparso da 1 000 nodi potremmo avere qualche migliaio di archi invece di una matrice di 1 000 000 di celle. È praticamente il fenomeno opposto a quanto avviene per il tempo. Al contrario, per i grafi completi (o molto densi), l'occupazione è asintoticamente la stessa, e ha costanti moltiplicative più alte, per la presenza di due dati interi e uno o due puntatori al posto di un valore logico, senza contare poi la sentinella.

Cominciamo a vedere il fatto che un algoritmo può essere meglio di un altro dal punto di vista temporale, ma peggio da quello spaziale. Inoltre, anche considerando un solo aspetto (per esempio quello spaziale) un algoritmo può essere meglio per alcune istanze del problema (quelle sparse) e peggio per altre (quelle complete o quasi).

8.5.6 Algoritmo 4

La disponibilità di una struttura per scorrere direttamente gli archi, anziché le coppie di nodi suggerisce (come abbiamo visto nel caso della stampa) un approccio alternativo, complementare, alla riduzione del tempo di calcolo. L'idea è, anziché scorrere tutte le coppie di nodi del sottoinsieme S e determinare se corrispondono ad archi scorrere gli archi e determinare se i nodi estremi appartengano entrambi al sottoinsieme S.

```
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
```

Cominciamo a valutare la complessità astratta di tutto questo, nell'ipotesi (ragionevole, se pensiamo di rappresentare gli archi con una lista) che lo scorrimento di A richieda tempo costante per ogni arco:

$$T = \sum_{a=1}^{m} (2T_S + T_{\text{extr}} + T_{\text{pr}}) \tag{8.7}$$

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.

Per confermare queste ipotesi fondamentali, conviene completare l'implementazione della libreria di gestione del grafo come lista di archi.

8.5.7 Implementazione della lista di archi: procedure di scorrimento

L'Algoritmo 4 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 `grafo-la.h` 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.

```
/* Restituisce la posizione del primo arco del grafo G */
posarco primoarco (grafo *pG);

/* Restituisce la posizione dell'arco successivo a pa nel grafo G */
posarco succarco (grafo *pG, posarco pa);

/* Determina se la posizione pa e' fuori della lista degli archi del
   grafo G */
boolean finearchi (grafo *pG, posarco pa);

/* Restituisce gli estremi *po e *pd dell'arco in posizione pa nel
   grafo G */
void leggeestremiarco (grafo *pG, posarco pa, nodo *po, nodo *pd);
```

Si potrebbe valutare l'ipotesi di realizzare anche le altre funzioni di scorrimento della lista degli archi (la ricerca dell'ultimo arco, del precedente, ecc...) e di scrittura degli archi. Tutto dipende dallo scopo ultimo, che qui è di realizzare l'Algoritmo 4, che di tali funzioni non ha bisogno. Quindi, al momento non lo facciamo.

La realizzazione è ovviamente basata sulla disponibilità della libreria `listaarchi.h` e si rivela praticamente banale. La ricerca del primo arco del grafo coincide con la ricerca del primo elemento della sua lista di archi.

```
/* Restituisce la posizione del primo arco del grafo *pG */
posarco primoarco (grafo *pG)
{
  return primolistaarchi(pG->L);
}
```

La ricerca dell'arco successivo del grafo coincide con la ricerca dell'elemento successivo nella lista degli archi.

```
/* Restituisce la posizione dell'arco successivo a pa nel grafo *pG */
posarco succarco (grafo *pG, posarco pa)
{
  return succlistaarchi(pG->L,pa);
}
```

Il test di terminazione degli archi chiama la corrispondente procedura di test della lista.

```
/* Determina se la posizione pa e' fuori della lista degli archi del
    grafo *pG */
boolean finearchi (grafo *pG, posarco pa)
{
  return finelistaarchi(pG->L,pa);
}
```

Infine, la lettura degli estremi dell'arco in una posizione data chiama la funzione corrispondente per la lista.

```
/* Restituisce gli estremi *po e *pd dell'arco in posizione pa nel
    grafo *pG */
void leggeestremiarco (grafo *pG, posarco pa, nodo *po, nodo *pd)
{
  leggearco(pG->L,pa,po,pd);
}
```

A questo punto, è lecito chiedersi perché non usare direttamente le funzioni di gestione della lista di archi, anziché passare attraverso questo filtro ridondante, che aggiunge operazioni non necessarie di gestione dello *stack*. Perché usare queste funzioni nel programma principale equivarrebbe ad aprire la scatola nera "grafo" e svelare l'implementazione effettivamente usata. Quindi, stiamo usando un approccio con qualche inefficienza allo scopo di mantenere una struttura dati astratta, una scrittura più chiara e una possibile flessibilità futura. È il solito compromesso. Va ricordato che esistono anche strumenti tecnologici per far sì che queste non siano delle vere chiamate di funzione, e quindi l'inefficienza introdotta sia limitata.

Si noti anche 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. Ci limitiamo a degli accenni. Per prima cosa, bisognerebbe definire in qualche modo la posizione di un arco in A. Per esempio, potrebbe essere un puntatore di tipo `boolean *` alla cella della matrice). Si dovrebbe poi implementare come funzioni le operazioni di ricerca del primo arco (per esempio, scorrere la matrice in un opportuno ordine, fermarsi alla prima cella pari a vero e restituirne l'indirizzo), di passaggio all'arco successivo (un altro scorrimento della matrice, fino alla successiva cella pari a vero) e di verifica che lo scorrimento sia terminato (ad es., con la verifica di essere usciti dall'area di memoria della matrices[4]). Il tutto potrebbe essere un esercizio istruttivo, ma sarebbe una soluzione piuttosto barocca e inefficiente (la maggior parte delle procedure non richiederebbero tempo costante). Purtroppo, aggiungendo via via funzioni alle diverse librerie di gestione dei grafi e utilizzando le nuove funzioni nelle versioni successive degli algoritmi, violeremo in pratica il principio che sia possibile usare una qualsiasi delle tre librerie con ogni versione dell'algoritmo. Concettualmente dovrebbe essere vero, ma per semplicità non lo garantiamo.

8.5.8 Implementazione dell'Algoritmo 4

A questo punto, possiamo realizzare l'Algoritmo 4 usando le nuove funzioni aggiunte alla libreria di gestione del grafo. L'algoritmo scorre gli archi usando il solito ciclo che parte dal primo e procede sino alla condizione di termine passando via via all'arco successivo. Occorre un cursore di tipo `posarco`, definito in `listaarchi.h` e accessibile quindi indirettamente, grazie al fatto che includiamo `grafo-la.h`, che a sua volta include `listaarchi.h`. Per ogni arco, si applica la determinazione degli estremi e la valutazione della loro appartenenza al sottoinsieme S.

8.5.9 Complessità di Algoritmo 4 con l'implementazione come lista di archi e tabella

Riprendendo l'espressione (8.7), possiamo arrivare a una forma dettagliata usando le complessità relative ai singoli passi elementari. La complessità di determinare gli estremi di un arco dato è costante. Il test di appartenenza di un nodo al sottoinsieme S costa $\Theta(n_S)$. Infine, la stampa richiede tempo costante.

$$T(m, n_S) = \sum_{a=1}^{m} (\Theta(1) + 2\Theta(n_S) + \Theta(1)) \in \Theta(mn_S) \tag{8.8}$$

Ne deriva un valore finale di in $\Theta(mn_S)$. Confrontato con il valore $\Theta\left(n_S^2\right)$ dell'Algoritmo 3 con la matrice di adiacenza, risulta dominato quando $n_s \ll m$, cioè quando i nodi del sottoinsieme (che sono al più tanti quanti i nodi totali) sono meno degli archi. Questo è molto frequente[5]. Sembrerebbe un altro buco nell'acqua.

[4] E questo potrebbe porre qualche problema tecnico

[5] A rigore, avendo escluso la possibilità di nodi isolati, abbiamo anche imposto che vi siano almeno $n/2$ archi.

Ma tutto questo dipende sostanzialmente dal fatto che l'appartenenza al sottoinsieme S richieda necessariamente una ricerca lineare, e già sappiamo che potremmo quanto meno farne una binaria, allargando molto la nicchia dei grafi nei quali l'Algoritmo 4 funziona meglio dell'Algoritmo 3. In realtà si può fare anche di meglio.

8.5.10 Il vettore di incidenza

Un *vettore di incidenza* è un vettore usato per descrivere sottoinsiemi di un insieme finito dato, che nel nostro caso sarà l'insieme N dei nodi del grafo. Il vettore di incidenza V_S che descrive il sottoinsieme $S \subseteq N$ è allocato con tanti elementi quanti quelli dell'insieme complessivo N (dunque da 1 a n) e contiene valori logici che indicano con il valore vero il fatto l'elemento appartiene al sottoinsieme e con il valore falso il fatto che non vi appartenga.

Esistono quindi 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.

Dal punto di vista spaziale, la tabella occupa n_S valori interi, mentre il vettore di incidenza richiede n valori binari, con $n \geq n_S$. A meno di rappresentare i valori logici in modo più efficiente (guadagnando un fattore moltiplicativo costante), la tabella è quindi in genere più compatta.

Dal punto di vista del tempo richiesto dalle operazioni:

- scorrere il sottoinsieme implementato come tabella richiede tempo $\Theta(|S|)$, mentre scorrerlo implementato come vettore di incidenza richiede $\Theta(|N|)$;
- valutare l'appartenenza di un elemento all'insieme richiede $\Theta(|S|)$ con l'implementazione come tabella, $\Theta(1)$ con quella come vettore di incidenza, perché si tratta semplicemente di valutare il valore del vettore nella posizione corrispondente all'elemento cercato.

8.5.11 Algoritmo 4 (S come vettore di incidenza)

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. Nell'Algoritmo 4, la situazione è esattamente complementare: non si scorre S, ma si eseguono continuamente test di appartenenza. Modifichiamo quindi l'Algoritmo 4 in modo che costruisca al principio il vettore di incidenza V_S del sottoinsieme $S \subseteq N$:

```
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
```

Il vettore viene allocato, inizializzato tutto a falso e poi modificato a vero negli elementi corrispondenti ai nodi di S scorrendo la tabella. A questo punto, può partire l'Algoritmo 4 già noto, che scorrere tutti gli archi e ne determina gli estremi, ma questa volta per valutare se gli estremi appartengono a S ci limitiano a leggere il vettore di incidenza.

8.5.12 Complessità di Algoritmo 4 con l'implementazione come lista di archi e vettore di incidenza

Possiamo quindi aggiornare l'espressione astratta (8.7) della complessità dell'Algoritmo 4. Abbiamo due termini aggiuntivi che descrivono la costruzione e l'inizializzazione del vettore V_S: in quanto scorrimenti di vettori, il primo richiede tempo $\Theta(n)$ e il secondo tempo $\Theta(n_S)$. La parte rimanente è invariata, ma bisogna specificare la valutazione di T_S come termine in $\Theta(1)$:

$$T(m,n) = \Theta(n) + \Theta(n_S) + \sum_{a=1}^{m} (\Theta(1) + \Theta(1) + \Theta(1) + \Theta(1)) \in \Theta(n+m) \quad (8.9)$$

dove teniamo esplicitamente n e m per ammettere il caso di grafi sparsissimi con meno archi che nodi.

Notiamo, di passaggio, che la tabella s è ancora 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'utent: se l'utente avesse fornito direttamente il vettore di incidenza, la tabella non sarebbe necessaria.

A questo punto, il confronto fra Algoritmo 3 e Algoritmo 4 diventa più interessante, perché fra $\Theta(n+m)$ e $\Theta\left(n_s^2\right)$ nessuno dei due domina chiaramente l'altro: per sottografi grandi di grafi sparsi sarà meglio l'Algoritmo 4, mentre per sottografi piccoli di grafi densi sarà meglio l'Algoritmo 3, ovviamente ciascuno con l'implementazione a cui siamo arrivati.

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.

8.5.13 Implementazione di Algoritmo 4 con l'implementazione come lista di archi e vettore di incidenza

Per procedere all'implementazione, dobbiamo aggiungere la dichiarazione del vettore di incidenza `VS`, del tipo `vboolean` definito come puntatore a `boolean`, la sua allocazione (con `calloc` per procedere simultaneamente all'inizializzazione a falso) e l'inizializzazione a vero dei soli nodi appartenenti a S. Il resto del codice è praticamente identico all'implementazione con tabella, salvo che sostituisce la chiamata ad `appartienetabella` con la lettura del vettore di incidenza (ovviamente, la sostituzione potrebbe avvenire dentro la procedura, ma comporterebbe di cambiare il significato dei suoi argomenti, o di definire in modo più astratto la struttura dati S, in modo da nascondere il fatto stesso che si tratti di tabella o di vettore di incidenza). Al termine, si deallocherà il vettore di incidenza per renderlo nuovamente disponibile al processore.

```
/* Stampa a video il sottografo indotto sul grafo G dal sottoinsieme
    S di ns nodi */
void StampaSottografoIndotto (grafo *pG, vint S, int ns)
{
  nodo o, d;
  int io;/*, id;*/
  posarco pa;
  vboolean VS;
  /* Algoritmo 4 (scorrimento degli archi in A, determinazione degli
       estremi
     e valutazione di appartenenza di o e d a S)
     con S implementato come vettore di incidenza */

  VS = (vboolean) calloc(pG->n+1,sizeof(boolean));
  if (VS == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione del vettore di incidenza
        di S!\n");
    exit(EXIT_FAILURE);
  }
  for (io = 1; io <= ns; io++)
    VS[S[io]] = TRUE;

  for (pa = primoarco(pG); !finearchi(pG,pa); pa = succarco(pG,pa))
  {
    leggeestremiarco(pG,pa,&o,&d);
    if ( VS[o] && VS[d] ) printf("(%d,%d) ",o,d);
  }

  free(VS);
}
```

8.5.14 Sesta fase (`grafo-fs.c` e `sottografo-fs.c`): vettore di *forward-star*

Passiamo alla terza implementazione, per valutare se per caso consenta qualche ulteriore miglioramento. L'implementazione a vettore di liste di adiacenza consiste nel gestire gli archi non in un'unica lista, ma in tante liste raccolte in un vettore, una per ogni nodo di origine (vedi Figura 8.23).

Ancora una volta, si tratta di abbandonare la libreria utilizzata e crearne un'altra, la libreria `grafo-fs.h`. Anche questa è già fornita, e contiene come le altre il

```
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
```

Figura 8.23: Forward star: implementazione in C

numero di nodi e di archi, ma anche un vettore dinamico di liste di archi. Di conseguenza, dovremo includere anche questa volta la libreria ausiliaria `listaarchi.h` per gestire tali liste.

```
typedef struct _grafo grafo;

struct _grafo
{
  int n;
  int m;
  listaarchi *FS;
};
```

Per inciso, in alcune applicazioni potrebbe valere la pena di avere anche un vettore che fornisca per ogni nodo il grado uscente, ovvero la cardinalitè della lista degli archi uscenti.

Dovremo poi implementare le solite funzioni per creare e distruggere grafi, inserire e cancellare archi, ecc..., in modo da poter far funzionare gli algoritmi già implementati. La particolare struttura di gestione degli archi suggerirà però, come già successo per la lista degli archi, di aggiungere nuove funzioni che ne possano sfruttare le caratteristiche.

La creazione del grafo richiede di inizializzare il numero di nodi e archi, ma soprattutto di allocare il vettore delle liste e crearle una per una sfruttando l'apposita procedura offerta dalla libreria per le liste di archi.

```
/* Crea un grafo *pG di n nodi, senza archi */
void creagrafo (int n, grafo *pG)
{
  nodo o;
```

```
  pG->n = n;
  pG->m = 0;
  pG->FS = (listaarchi *) calloc(pG->n+1,sizeof(listaarchi));
  if (pG->FS == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione del vettore delle forward
        star!\n");
    exit(EXIT_FAILURE);
  }

  for (o = 1; o <= pG->n; o++)
    pG->FS[o] = crealistaarchi();
}
```

La complessità sarà lineare nel numero dei nodi ($\Theta(n)$), dato che ogni lista va allocata indipendentemente.

La distruzione è relativamente semplice e speculare: per ogni elemento del vettore chiamiamo la procedura per distruggere la relativa lista, poi deallochiamo il vettore (e lo azzeriamo per evitare che il puntatore venga usato surrettiziamente) e infine azzeriamo il numero degli archi e quello dei nodi.

```
/* Distrugge il grafo *pG */
void distruggegrafo (grafo *pG)
{
  nodo o;


  for (o = 1; o <= pG->n; o++)
    distruggelistaarchi(&pG->FS[o]);
  free(pG->FS);

  pG->n = 0;
  pG->m = 0;
}
```

Qui la complessità è invece $\Theta(n+m)$, perché ogni arco viene distrutto e per ogni nodo (anche privo di archi uscenti) va deallocata la sentinella.

L'inserimento di un nuovo arco è praticamente identico a quello dell'implementazione a lista di archi, perché l'unica differenza è di scegliere correttamente la lista nella quale l'arco viene inserito, cioè quella del suo nodo origine.

```
/* Aggiunge l'arco (o,d) dal grafo *pG
   N.B.: non verifica che l'arco sia nuovo! */
void insarco (nodo o, nodo d, grafo *pG)
{
  posarco pa;

  pa = succlistaarchi(pG->FS[o],ultimolistaarchi(pG->FS[o]));
  inslistaarchi(pG->FS[o],pa,o,d);
  pG->m++;
```

La complessità è quindi ancora in $\Theta(1)$. Ancora una volta, per efficienza, non verifichiamo che l'arco sia effettivamente nuovo.

La cancellazione di un arco richiede di determinarne la posizione a partire dai due nodi estremi. Rispetto al caso della lista semplice di archi, abbiamo il vantaggio

di poter cercare l'arco nella specifica lista degli archi uscenti dal nodo di origine, che è noto. Il resto della procedura è sostanzialmente identico.

```
/* Cancella l'arco (o,d) dal grafo *pG (non fa nulla se non esiste) */
void cancarco (nodo o, nodo d, grafo *pG)
{
  posarco pa;
  nodo orig, dest;

  for (pa = primolistaarchi(pG->FS[o]);
       !finelistaarchi(pG->FS[o],pa); pa =
       succlistaarchi(pG->FS[o],pa))
  {
    leggearco(pG->FS[o],pa,&orig,&dest);
    if ( (o == orig) && (d == dest) ) break;
  }

  if (!finelistaarchi(pG->FS[o],pa))
  {
    canclistaarchi(pG->FS[o],&pa);
    pG->m--;
  }
}
```

La compessità deriva essenzialmente dalla ricerca dell'arco, e quindi trae profitto dal fatto che la lista è più corta, contenendo solo il numero di archi uscenti dal nodo origine. Possiamo esprimerla come $\Theta(n)$, dato un nodo non può avere più di $n-1$ archi uscenti. Volendo introdurre un nuovo parametro di dimensione, che può giocare un ruolo, possiamo indicare con $\delta^+_{\max} = \max_{o \in N} \delta^+_o \leq n$ il massimo grado uscente su tutti i nodi del grafo. Avremmo allora che la cancellazione di un arco richiede tempo $\Theta(\delta^+_{\max})$. Si noti che sarebbe scorretto scrivere $\Theta(\delta^+_o)$ perché non si tratta di una funzione di un parametro quantitativo: $\delta^+_{\max}$ un numero, δ^+_o una funzione.

Per il test di esistenza, abbiamo ancora uno scorrimento della lista del nodo origine, ancora con la stessa complessità $\Theta(\delta^+_{\max})$.

```
/* Determina se l'arco (orig,dest) appartiene al grafo *pG */
boolean esistearco (nodo o, nodo d, grafo *pG)
{
  posarco pa;
  nodo orig, dest;


  for (pa = primolistaarchi(pG->FS[o]);
       !finelistaarchi(pG->FS[o],pa); pa =
       succlistaarchi(pG->FS[o],pa))
  {
    leggearco(pG->FS[o],pa,&orig,&dest);
    if ( (o == orig) && (d == dest) ) return TRUE;
  }

  return FALSE;
}
```

Infine, la stampa del grafo cambia leggermente, perché, non avendo più una sola lista, lo scorrimento di tutti gli archi richiede un meccanismo su due livelli: prima lo scorrimento dei nodi origine e poi, per ciascuno, lo scorrimento della sua lista di archi. L'unico dettaglio tecnico è che leggendo il nodo origine abbiamo il suo indice

sia fra i dati sia fra i risultati: se gli archi sono corretti, non dovrebbe costituire un problema, ma si può decidere di cautelarsi usando due variabili diverse (`orig` e `o`).

```
/* Stampa a video il grafo *pG */
void stampagrafo (grafo *pG)
{
  posarco pa;
  nodo o, d, orig;


  for (orig = 1; orig <= pG->n; orig++)
    for (pa = primolistaarchi(pG->FS[orig]);
         !finelistaarchi(pG->FS[orig],pa); pa =
         succlistaarchi(pG->FS[orig],pa))
    {
      leggearco(pG->FS[orig],pa,&o,&d);
      printf("(%d,%d) ",o,d);
    }
}
```

Ne deriva una complessità pari a $\Theta(m+n)$, dato che anche i nodi privi di archi uscenti vengono quanto meno presi in considerazione.

L'Algoritmo 3 non ha bisogno di altre funzioni di libreria, per cui si può anche subito valutarne la correttezza e compatibilità anche con la terza implementazione.

8.5.15 Complessità dell'Algoritmo 3 con l'implementazione come vettore di *forward-star* e tabella

Rispetto alla forma astratta dell'Algoritmo 3, cioè all'espressione (8.6), e alle precedenti implementazioni, cambia il tempo necessario al test di appartenenza all'insieme degli archi, che era poi il termine dominante. Nell'implementazione con la matrice di adiacenza, essendo $T_A \in \Theta(1)$, il risultato finale era $\Theta\left(n_S^2\right)$. Nell'implementazione con la lista degli archi, c'era un brusco aumento, con $T_A \in \Theta(m)$ e l'intero algoritmo in $\Theta\left(n_S^2 m\right)$.

Il caso presente è intermedio, dato che la funzione `esistearco` richiede tempo $\Theta\left(\delta^+_{\max}\right)$, e quindi l'Algoritmo 3 in questa implementazione costa tempo $\Theta\left(n_S^2 \delta^+_{\max}\right)$.

$$T(n, n_S) = \sum_{i_o=1}^{n_S} \left(\Theta(1) + \sum_{i_d=1}^{n_S} \left(\Theta(1) + \Theta\left(\delta_o^+\right) + \Theta(1) \right) \right) \in \Theta\left(n_S^2 \delta^+_{\max}\right) \qquad (8.10)$$

Anche il costo spaziale è intermedio. Infatti, la memoria totale occupata è in $\Theta(m+n)$, a causa delle liste di archi e del vettore che le raccoglie (trascurando, in quanto dominata, la tabella). A rigore, per un grafo completo, questo non è intermedio, ma peggiore di entrambe le implementazioni alternative.

Nel complesso, il risultato non è soddisfacente, se non dal punto di vista spaziale.

8.5.16 Algoritmo 4 con l'implementazione come vettore di *forward-star* e vettore di incidenza

Passando all'Algoritmo 4, ignoriamo l'implementazione con tabella, dato che abbiamo notato l'efficacia nettamente superiore del vettore di incidenza sulla ricerca sequenziale in una tabella nel verificare se un nodo appartenga o no a un sottoinsieme. Il tempo aggiuntivo necessario a costruire il vettore di incidenza è ampiamente riguadagnato con il suo uso.

Teoricamente, bisognerebbe implementare le funzioni per lo scorrimento della lista globale degli archi, che l'Algoritmo 4 richiede. Tuttavia, una lista globale di archi non esiste. Potremmo forse simularla per tenerci stretti al paradigma delle strutture dati astratte[6], ma probabilmente conviene limitarsi a sostituire (come fatto nella stampa del grafo) lo scorrimento semplice di tutti gli archi con lo scorrimento doppio dei nodi e, per ciascuno, degli archi uscenti da questo.

Quindi, non realizziamo le funzioni `primoarco`, `succarco` e `finearchi`, ma le corrispondenti funzioni che operano sulla specifica lista di archi uscenti da un nodo dato. Ne deriva un parametro aggiuntivo (l'indice del nodo di origine).

```
/* Restituisce la posizione del primo arco della forward star del
    nodo o del grafo *pG */
posarco primoarcoFS (grafo *pG, nodo o);

/* Restituisce la posizione dell'arco successivo a pa nella forward
    star del nodo o del grafo *pG */
posarco succarcoFS (grafo *pG, nodo o, posarco pa);

/* Determina se la posizione pa e' fuori della forward star del nodo
    o del grafo *pG */
boolean finearchiFS (grafo *pG, nodo o, posarco pa);
```

Un'altra differenza minore è che, se accediamo a un arco non attraverso e una posizione, ma attraverso un nodo origine e una posizione, non ha molto senso farsi restituire origine e destinazione: la destinazione sarà sufficiente. Ed essendo un dato semplice può essere restituita direttamente senza usare il trucco del passaggio per indirizzo di dati finti.

```
/* Restituisce la dinazione dell'arco in posizione pa nella forward
    star del nodo o del grafo *pG */
nodo leggedestarco (grafo *pG, nodo o, posarco pa);
```

Aggiorniamo di conseguenza lo pseudocodice:

```
 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
```

Implementiamo direttamente la nuova versione dell'algoritmo 4: anche nel co-

[6] Volendo si 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, se ci rassegniamo a scorrere l'intero insieme degli archi combinando uno scorrimento dei nodi con uno scorrimento degli archi uscenti da ciascun nodo.

dice, l'unica differenza è la sostituzione di un ciclo semplice con uno doppio e la lettura della destinazione dell'arco anziché di entrambi gli estremi.

```
/* Stampa a video il sottografo indotto sul grafo G dal sottoinsieme
    S di ns nodi */
void StampaSottografoIndotto (grafo *pG, vint S, int ns)
{
  nodo o, d;
  int io;/*, id;*/
  posarco pa;
  vboolean VS;
  /* Algoritmo 4 (scorrimento degli archi in A, determinazione degli
      estremi
     e valutazione di appartenenza di o e d a S)
     con S implementato come vettore di incidenza */

  VS = (vboolean) calloc(pG->n+1,sizeof(boolean));
  if (VS == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione del vettore di incidenza
        di S!\n");
    exit(EXIT_FAILURE);
  }
  for (io = 1; io <= ns; io++)
    VS[S[io]] = TRUE;

  for (io = 1; io <= ns; io++)
  {
    o = S[io];
    for (pa = primoarcoFS(pG,o); !finearchiFS(pG,o,pa); pa =
        succarcoFS(pG,o,pa))
    {
      d = leggedestarco(pG,o,pa);
      if ( VS[d] ) printf("(%d,%d) ",o,d);
    }
  }

  free(VS);
}
```

Rispetto all'implementazione con la lista degli archi, rimane l'allocazione e inizializzazione del vettore di incidenza, in tempo $\Theta(n)$. Lo scorrimento degli archi diventa uno scorrimento dei nodi e degli archi, dunque si aggiunge un altro termine $\Theta(n)$, che non cambia molto. Le operazioni sugli archi, però, non cambiano praticamente, quindi il risultato finale rimane lo stesso.

Le funzioni di scorrimento degli archi nelle liste di incidenza sono facili da implementare usando la libreria `listaarchi` e richiedono tutte tempo costante. Discutiamole metodicamente.

La ricerca della posizione del primo arco uscente da un dato nodo consiste nel recuperare il primo arco della corrispondente lista.

```
/* Restituisce la posizione del primo arco della forward star del
    nodo o del grafo *pG */
posarco primoarcoFS (grafo *pG, nodo o)
{
  return primolistaarchi(pG->FS[o]);
}
```

La ricerca dell'arco successivo consiste nel prendere l'arco successivo nella lista degli archi uscenti dal nodo dato.

```
/* Restituisce la posizione dell'arco successivo a pa nella forward
    star del nodo o del grafo *pG */
posarco succarcoFS (grafo *pG, nodo o, posarco pa)
{
  return succlistaarchi(pG->FS[o],pa);
}
```

Per determinare se siamo alla fine della lista degli archi uscenti dal nodo corrente applicheremo l'apposita funzione alla lista stessa.

```
/* Determina se la posizione pa e' fuori della forward star del nodo
    o del grafo *pG */
boolean finearchiFS (grafo *pG, nodo o, posarco pa)
{
  return finelistaarchi(pG->FS[o],pa);
}
```

Infine, per leggere la destinazione dell'arco, la cosa è poco più complicata: leggeremo l'intero arco e ignoreremo l'origine, che già conosciamo, restituendo invece solo la destinazione.

Compilato il codice, possiamo eseguire l'algoritmo e verificare che sia corretto.

8.5.17 Un ulteriore miglioramento

Il fatto è che la divisione, in buona parte involontaria, dello scorrimento degli archi in tanti scorrimenti relativi agli archi uscenti da ciascun nodo, consente di applicare di nuovo un trucco che abbiamo già applicato all'inizio della vicenda. Si tratta di osservare che il test di appartenenza dell'origine di un arco al sottoinsieme S non va ripetuto su ogni arco uscente, ma si può eseguire una volta sola. Anticipando il test, non ci limitiamo ad eseguirlo meno spesso (in fondo, si tratta di una sola operazione, molto veloce). In effetti, evitiamo anche di scorrere l'intera lista di archi uscenti da un nodo che non fa parte di S.

Il ciclo esterno viene eseguito n volte, ma non richiede più di eseguire il ciclo interno (sugli archi) ogni volta. Solo gli n_S nodi di S richiederanno l'esecuzione del ciclo interno. Quindi gli archi considerati per la stampa non saranno tutti gli archi, ma solo quelli uscenti da nodi di S. Di questi, stamperemo quelli che sono anche entranti in nodi di S e ignoreremo gli altri.

Si può fare ancora meglio: anziché scorrere tutti i nodi, valutare se appartengono a S e, in caso positivo, proseguire con lo scorrimento della lista degli archi uscenti, possiamo sfruttare la tabella per scorrere direttamente solo i nodi di S. È il principio enunciato al principio, secondo il quale non si scorre un insieme più ampio facendo un test ad ogni elemento per valutare se è fra quelli desiderati, quando è possibile scorrere direttamente i soli elementi dell'insieme desiderato.

1: **for** $i := 1$ **to** n **do**
2: $\quad V_S[i] :=$ **false**;
3: **end for**
4: **for all** $o \in S$ **do**
5: $\quad V_S[o] :=$ **true**;
6: **end for**
7: **for all** $i_o = 1$ **to** n_S **do**
8: $\quad o := s[i_o]$;

```
 9:     for all a ∈ Δ_o^+ do
10:         (o, d) := LeggeEstremiArco(a);
11:         if V_S[d] = true then
12:             StampaArco(o, d);
13:         end if
14:     end for
15: end for
```

Si noti che in questo modo stiamo usando entrambe le rappresentazioni del sottoinsieme S, ognuna per lo scopo al quale è più adatta: la tabella per scorrere il sottoinsieme, il vettore di incidenza per valutare l'appartenenza di un nodo ad esso. Qui le rappresentazioni sono necessarie entrambe ai fini dell'algorimo, e permettono di migliorarne il costo temporale, anche se aggravano leggermene quello spaziale.

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 al massimo m, eventualmente molto meno. Si noti che in questo caso δ_o^+ compare in una sommatoria, e quindi il risultato è una costante, non una funzione, per cui può essere lecitamente usato in una valutazione di complessità.

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.

In effetti, entrambi gli algoritmi lasciano (almeno teoricamente) un margine di miglioramento, dato che il minimo tempo indispensabile a stampare gli archi del sottografo indotto è proporzionale al loro numero, e sia n_S^2 sia $m_S = \sum_{o \in S} \delta_o^+$ sono sovrastime di tale numero. Va detto che, se si tiene conto del fatto che il grafo va comunque caricato in memoria e questo richiede un tempo almeno proporzionale al numero di archi, e non solo del tempo richiesto a estrarne il sottografo indotto, siamo già alla complessità asintotica ottimale.

Abbiamo visto che alcuni algoritmi sono dominati da un punto di vista teorico, indipendentemente dal modo in cui sono implementati, perché fanno operazioni aggiuntive inutili. D'altra parte, le implementazioni hanno un effetto determinante sugli algoritmi, e a volte un'implementazione che peggiora le prestazioni di un algoritmo può migliorare quelle di un altro. Abbiamo anche visto che spesso peggioramenti nel costo spaziale consentono miglioramenti in quello temporale. Abbiamo visto che gli algoritmi sono in teoria realizzabili più o meno con qualsiasi implementazione, a patto di introdurre le funzioni necessarie, ma che alcune implementazioni sono fondamentalmente inadatte a eseguire certe operazioni, che si possono tutt'al più simulare goffamente e in modo inefficiente. Il risultato è un algoritmo poco promettente può diventare molto valido se i suoi limiti vengono risolti con una buona implementazione. Infine, abbiamo visto che algoritmi diversi si rivelano più o meno adatti a seconda delle specifiche istanze del problema, per cui la dimensione pura e semplice non è abbastanza per caratterizzare le proprietà dell'algoritmo. E ancora abbiamo visto che implementazioni combinate possono essere utilmente sfruttate per ovviare ai limiti di ciascuna delle implementazioni che le compongono.

Un'altra considerazione che merita fare è che, anche se non abbiamo discusso un esercizio pratico, il meccanismo di descrizione dell'algoritmo (eventualmente con pseudocodici), da cui derivare con riferimenti chiari alle singole operazioni l'analisi di complessità è senz'altro utile anche a svolgere questo aspetto della prova d'esame[7].

[7]Purtroppo per esigenza didattica descrizione dell'algoritmo e analisi sono state qui spesso

L'idea è descrivere l'algoritmo, dedurne la scelta delle strutture dati, quindi la complessità delle operazioni fondamentali, quindi mostrare che le operazioni scelte hanno un certo costo.

8.6 Esercizi

8.6.1 Esercizio 1

Si modifichi il programma di calcolo del sottografo indotto in modo che carichi l'elenco dei nodi da un secondo file di testo, passato da linea di comando, anziché direttamente dalla linea di comando.

8.6.2 Esercizio 2

Si modifichino gli algoritmi realizzati nel capitolo in modo che la tabella S sia ordinata e il test di appartenenza di un nodo ad essa sfrutti la ricerca binaria, anziché quella sequenziale.

8.6.3 Esercizio 3

Si modifichi la libreria `grafo-fs.h` in modo che contenga anche un vettore che fornisca il grado uscente di ogni nodo e una funzione per comunicare tale informazione all'utente.

8.6.4 Esercizio 4

Si modifichi la libreria `grafo-fs.h` in modo che contenga anche un vettore di *backward-star*, cioè di liste di archi entranti in ogni nodo.

tenute in sequenza anziché immediatamente legate nelle stesse sezioni (per non parlare degli intermezzi di commento al codice), per cui questa nota avrà probabilmente effetti devastanti nella stesura delle relazioni, istigando molti a tenere separate due cose che invece vanno portate avanti insieme.

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

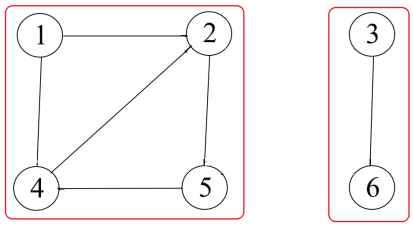


Figura 9.1: Componenti connesse di un grafo

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 $\Theta(1)$

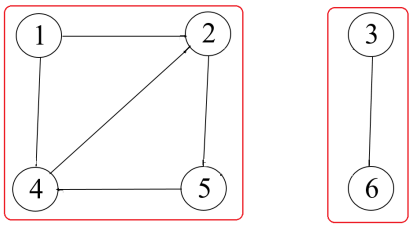
Altre operazioni sono meno efficienti, ma non sono richieste

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 ($C = [\,2\,2\,1\,1\,2\,2\,1\,]$), 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

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

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 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

[1] Su questo devo ragionare meglio: in fondo pile e code somigliano molto a delle tabelle.

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 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.

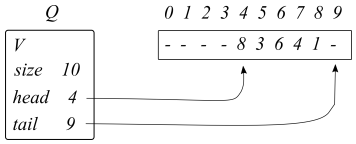
Una coda può essere facilmente rappresentata da un vettore che contiene l'informazione e da due indici numerici, che determinano le posizioni estreme. Convenzionalmente, si indica con `tail` la posizione della coda e con `head` quella della testa. Per essere precisi, `tail` non è l'ultima posizione occupata 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.

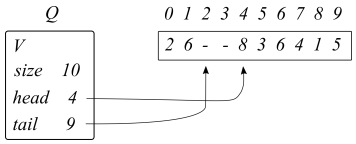
Una possibile implementazione in C è riportata nella Figura 9.14. Una coda di interi si può rappresentare con un vettore di interi, l'indicazione della sua dimensione allocata e due indici interi che determinano la testa (posizione occupata ed estratta per prima) e la coda (prima posizione libera, disponibile per il primo nuovo elemento).

9.4.3 Seconda fase: visita in ampiezza

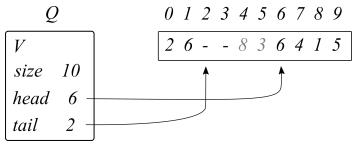
Procediamo ora a implementare la visita in ampiezza di un grafo, ipotizzando di avere la libreria `intqueue` che permette di definire e gestire code di interi. In particolare, la libreria fornisce due funzioni per creare e distruggere la coda, una funzione per valutare se la coda è vuota, una per leggere 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



`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

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 */
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))
```

```
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

```
    {
      /* 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);
      }
    }
  }

  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 elemento 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 assolutamente 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 sostan-

zialmente 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

9.4.5 Rappresentazioni di alberi con il vettore padre

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 ft$3 = 3$ 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 fortemen-

te 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 questa 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 vetto-

re 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:

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;
}
```

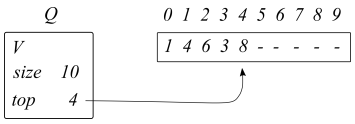
Lo spazio è esaurito quando `S.top == S.size`

La cancellazione sposta un passo indietro l'ultima posizione occupata

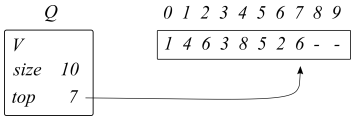
```
Pop(S)
{
  S.top = S.top-1;
}
```

La gestione della pila non richiede l'uso dell'aritmetica modulare, perché inserimenti ed estrazioni avvengono dalla stessa parte del vettore

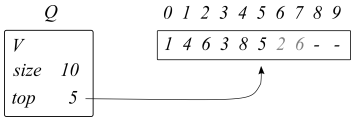
Figura 9.20: Esempio



`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);
      }
  }
}
```

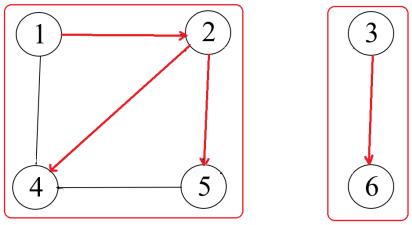
È equivalente alla precedente purché si scorra Adj(v) in ordine inverso

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

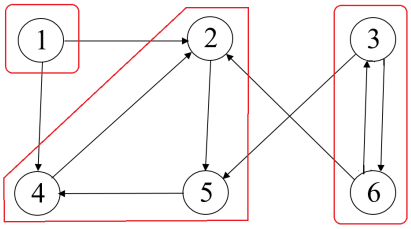


Figura 9.24: Componenti fortemente connesse e visita di un grafo
Definiamo nodi co-raggiungibili da v i nodi da cui v è raggiungibile

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

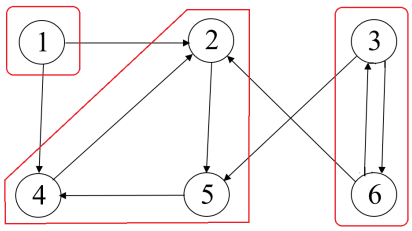
In un grafo orientato, la visita determina solo i nodi raggiungibili da v, ma invertendo il verso degli archi, la visita determina i nodi co-raggiungibili

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

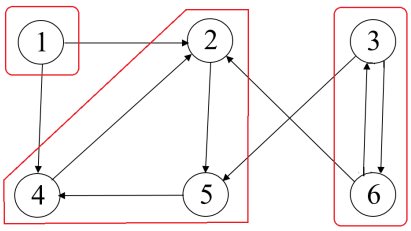
Perciò nel seguito discutiamo solo le componenti connesse

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

- 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.

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.

[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 $\Theta\left(n \max_{s \in V}(n_s + m_s)\right)$.

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.

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:

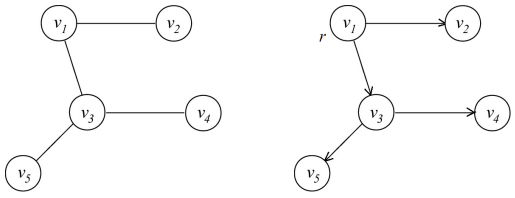
1. *connessi*: ogni coppia di vertici del grafo stesso è legata da un cammino;
2. *aciclici*: nessuno dei cammini costituiti dai lati del grafo si richiude su sé stesso ciclicamente.

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.

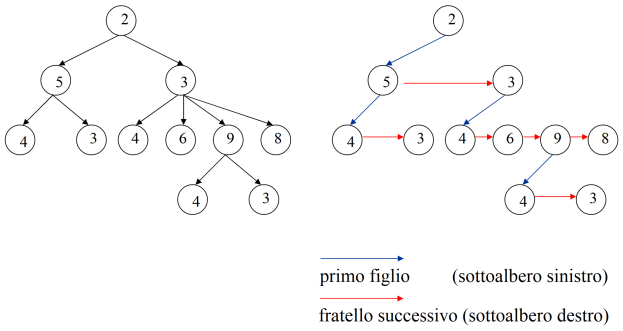
Se fissiamo un limite massimo al numero degli archi uscenti da ogni nodo, fissiamo la cosiddetta *arità* dell'albero (orrido anglicismo, etimologicamente insensato). L'unico aspetto che ci interessa a riguardo è il concetto di *albero binario*. Un albero binario è un albero che ha al massimo due figli 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.

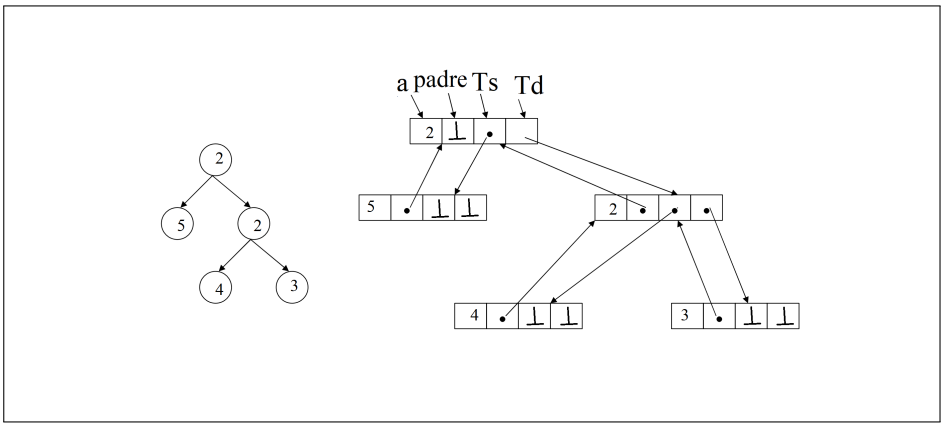


Figura 10.10: Alberi binari: implementazione con puntatori

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

Svolgereremo un esercizio su una possibile applicazione degli alberi binari per rappresentare strutture gerarchiche. Si tratta di realizzare una specie di 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 rappresentano le classiche quattro operazioni aritmetiche che si imparano alle scuole elementari: la somma, la differenza, il prodotto e la divisione. L'espressione combina numeri e operatori in vario modo, con parentesi che impongono le precedenze desiderate, come nelle espressioni che si risolvono alle scuole medie.

Da un punto di vista matematico, le espressioni aritmetiche sono delle strutture intrinsecamente gerarchiche. Infatti, hanno una definizione ricorsiva; un'espressione aritmetica può essere:

- *semplice*, cioè un numero reale;
- *composta*, cioè formata da una terna costituita da un *operatore* (somma, differenza, prodotto o rapporto) e due *operandi*, che sono a loro volta espressioni aritmetiche (semplici o composte).

Come al solito, la cosa ha senso grazie all'esistenza del caso base, che consente alla ricorsione di arrestarsi dopo un numero di livelli finito, anche se non determinato a priori.

Il programma `calcola.c` dovrà caricare ogni espressione da un file di testo, dunque rappresentarla in qualche modo all'interno della memoria. Quindi, dovrà calcolarne il valore e stamparlo a video. Poi dovrà convertire l'espressione dal formato iniziale, che discuteremo più sotto, nel classico formato in cui gli operatori sono singoli caratteri in posizione intermedia fra gli operandi a cui vengono applicati e stampare l'espressione a video in questo formato. Infine, vogliamo determinare una proprietà dell'espressione stessa detta profondità, che è ovviamente legata alla sua rappresentazione come albero.

Le espressioni sono conservate nel file dei dati in un formato abbastanza dal sapore abbastanza informatico, ispirato dalle chiamate a funzione. Corrispondentemente, la codifica nel file di testo di un'espressione aritmetica consiste:

- per le espressioni semplici, in un numero reale;
- per le espressioni composte, nella stringa *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.

Nell'esempio seguente:

```
prod(somma(5,3),div(10,diff(4,2))
```

si moltiplica la somma di 5 e 3 per il rapporto fra 10 e la differenza di 4 e 2. In realtà, questa notazione somiglia moltissimo alla cosiddetta *notazione prefissa*, o *notazione polacca*, che è comunemente usata in matematica. In essa, gli operatori precedono gli operandi, esattamente come in programmazione il nome di una procedura precede i suoi argomenti. Le differenze fondamentali sono che la notazione polacca usa simboli di operatori elementari anziché nomi di funzione e che non usa le parentesi e le virgole. Queste, in effetti, non sono necessarie nel momento in cui si sa che ogni operatore ha esattamente due operandi. Le abbiamo aggiunte per dare un tono più informatico e per rendere l'espressione un po' più leggibile per un essere umano, sottolineando dove comincia e dove termina ciascun operando.

Nell'esempio specifico, l'espressione è composta: l'operando prodotto (rappresentato dalla stringa `prod`) viene applicato alle espressioni composte `somma(5,3)` e `div(10,diff(4,2))`. A sua volta, la prima delle due espressioni è composta dall'operando somma e dalle espressioni semplici 5 e 3, che sono numeri interi. La seconda, invece, è composta dall'operando rapporto, dall'espressione semplice 10 e dall'espressione composta `diff(4,2)`, che finalmente si divide nell'operatore differenza e nei due operandi semplici 4 e 2, terminando la ricorsione.

10.3.1 Modello del problema e decomposizione in sottoproblemi

Per quanto riguarda il modello del problema, è abbastanza intuitivo che un'espressione si può rappresentare come un albero nel quale i nodi interni corrispondono agli operatori e le foglie agli operandi semplici, cioè a numeri reali. L'albero è binario perché tutti gli operatori considerati sono binari. Quindi non occorre introdurre la convenzione dei puntatori al primo figlio e al fratello successivo. Siccome alcuni degli operatori non hanno la proprietà commutativa, le espressioni possono essere asimmetriche. Gli alberi saranno quindi ordinati, per poter gestire le espressioni asimmetriche, oltre alle più semplici espressioni simmetriche.

Il file dei dati `input01.txt` contiene alcune di queste espressioni, una per riga, per semplicità con una dimensione massima di 256 caratteri.

```
prod(somma(5,3),div(10,diff(4,2)))
div(10.0,3.0)
somma(1.0,somma(2.0,somma(3.0,somma(4.0,somma(5.0,6.0)))))
2.5
```

La Figura 10.11 riporta le rappresentazioni ad albero delle espressioni contenute nel file.

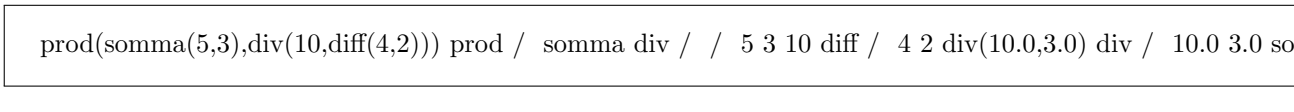


Figura 10.11: Esempi di rappresentazioni ad albero di espressioni aritmetiche

La notazione prefissa consente di identificare facilmente in ogni espressione l'operatore, che corrisponde a un nodo dell'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, letta da sinistra verso destra, corrisponde esattamente a una visita dell'albero in quello che viene definito *preordine*. Nel primo esempio, si parte con il prodotto, a cui segue (nel figlio di sinistra) la somma poi l'operando 5 (sempre nel figlio di sinistra). A questo punto siamo in una foglia, per cui proseguiamo con il figlio di destra, che è l'operando 3, dunque ancora una foglia. Risaliamo al primo nodo che ha ancora figli inesplorati a destra: raggiungiamo la divisione, poi l'operando 10 e il nodo differenza con i due figli 4 e 2.

C'è una perfetta corrispondenza fra la lettura dell'espressione da sinistra a destra e la visita dell'albero in preordine, cioè leggendo prima il nodo padre, poi il figlio di sinistra, infine quello di destra. Quindi, se avessimo già costruito l'albero e volessimo stamparlo in notazione polacca, basterebbe farne una visita in preordine. Noi invece abbiamo l'espressione e dobbiamo costruire l'albero.

Costruito l'albero, dovremo valutare l'espressione. Se questa è semplice, è facile farlo, perché si riduce a un numero, e il valore dell'espressione coincide col numero stesso. Se invece si tratta di un'espressione composta, bisogna prima capire quanto valgono i due operandi e poi applicare l'operatore ai loro due valori. Per esempio, se stiamo considerando l'espressione `diff(4,2)`, prima valuteremo 4 e 2 e poi applicheremo la differenza. Questo corrisponde ancora a una visita, nella quale però prima si opera sui figli e poi sul padre. Questo è tipico della visita in *postordine*. Infatti,

nell'espressione `prod(somma(5,3),div(10,diff(4,2)))` prima di poter calcolare il valore del prodotto dobbiamo sapere quanto valgono la somma e il rapporto. La somma richiede di conoscere i due addendi, 5 e 3, e vale 8, mentre per conoscere il valore del rapporto dobbiamo prima calcolare la differenza di 4 e 2. Insomma, i passaggi si svolgono in questo modo:

$$(5+3)*[10/(4-2)] = 8*(10/2) = 8*5 = 40$$

A nessuno sarà sfuggito che questo corrisponde alle parentesi tonde o quadre che si usano nella notazione tipica delle scuole medie e alle classiche regole di precedenza.

Calcolato il valore, dovremo scrivere l'espressione nella notazione nota come *infissa*, cioè con l'operatore riportato fra i due operandi: il `+` viene messo fra 5 e 3, il `*` fra la somma di 5 e 3 e il rapporto di 10 con la differenza di 4 e 2, eccetera. Per ottenere questa espressione eseguiremo un'altra visita dell'albero.

Analogamente, la profondità dell'espressione corrisponde all'altezza dell' albero

Tutto questo suggerisce che adottare un albero binario ordinato come modello del problema pratico sia corretto, dato che consente naturalmente di eseguire tutte le operazioni che saranno necessarie a risolvere il problema.

Inoltre, abbiamo diviso il problema in sottoproblemi. 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.

10.3.2 Implementazione: punto di partenza

Il file di partenza, `calcola0.c`, al solito contiene qualche operazione già predisposta. La procedura `InterpretaLineaComando` semplicemente legge il nome del file delle espressioni. La procedura `ElaboraEspressioni` che riceve il nome del file e fa tutto il resto.

```
/* Programma principale */
int main (int argc, char *argv[])
{
  /* Parte dichiarativa */
  char file_espressioni[ROW_LENGTH];

  /* Parte esecutiva */
  InterpretaLineaComando(argc,argv,file_espressioni);

  /* Elabora le espressioni nel file */
  ElaboraEspressioni(file_espressioni);

  return EXIT_SUCCESS;
}
```

Perché non abbiamo una procedura che carichi i dati in un'opportuna struttura in memoria (per esempio un vettore di stringhe, oppure già subito un vettore di alberi binari) e poi una procedura che operi sulla struttura in memoria per risolvere il problema? Questo consentirebbe di chiudere subito il file e operare direttamente in memoria. Semplicemente perché l'esercizio è già abbastanza sofisticato, ma potrebbe essere un'idea assolutamente valida.

La procedura `ElaboraEspressioni` non è completa, ma si limita ad aprire il file in lettura, eseguire un ciclo che carica le espressioni una alla volta e chiudere il file alla fine.

```
void ElaboraEspressioni (char *file_espressioni)
{
  FILE *fp;
  char Espressione[ROW_LENGTH];


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

  /* Legge il file una riga alla volta */
  while (fscanf(fp,"%[^\n]\n",Espressione) == 1)
  {
    /* Carica l'espressione corrente dalla stringa in un albero */

    /* Calcola il valore dell'espressione rappresentata dall'albero */

    /* Stampa in notazione infissa l'espressione rappresentata
        dall'albero */

    /* Calcola la profondita' dell'albero */

    /* Distrugge l'albero */

    printf("\n");
  }

  fclose(fp);
}
```

Finora per leggere una riga di un file abbiamo sempre usato la funzione `fgets`. Potremmo continare a usarla, ma per mostrare un'alternativa possibile adotteremo un'altra modalità, che ha un possibile vantaggio (tutto da dimostrare), cioè di non caricare al termine della stringa l'a capo finale. Questa modalità applica la classica funzione `fscanf` al file con una stringa di formato piuttosto complessa e scrive ciò che è stato eletto in un vettore di caratteri `Espressione`, dichiarato a monte in modo statico con una lunghezza che in base al testo è sufficiente a contenere qualsiasi espressione.

La stringa di formato parte con il solito `%` legato al fatto che dobbiamo riconoscere qualcosa, valutarne il valore e scriverlo in una variabile. Segue una coppia di parentesi quadre. Queste racchiudono un insieme di caratteri che sono accettabili oppure non accettabili nella stringa. Per esempio, `%[acd]` indicherebbe una stringa di qualsiasi lunghezza, composta solo dai caratteri `a`, `c` e `d`, fermandosi al primo carattere diverso. Nel nostro caso, vogliamo invece procedere per esclusione, cioè accettando tutti i caratteri tranne uno, l'a capo. Per indicarlo, premetteremo all'elenco dei caratteri esclusi un accento circonflesso (`^`). Siccome qualsiasi carattere è accettabile, tranne l'a capo, scriveremo `%[^\n]` e leggeremo e caricheremo nella stringa `Espressione` tutti i caratteri della riga corrente fino all'a capo escluso. Ora un dettaglio: la testina di lettura è ferma sull'a capo, che sarà quindi il prossimo carattere letto. Se vogliamo liberarcene, possiamo concludere la stringa di formato con un separatore, che potrebbe anche essere uno spazio bianco, ma è probabilmente

più chiaro indicare con un a capo esplicito. Va detto che la prossima operazione di lettura probabilmente avrà una stringa di formato che comincia con un `%`, e quindi salterà l'a capo e tutti gli eventuali separatori che lo seguono prima di fare qualsiasi altra operazione. In conclusione:

```
fscanf(fp,"%[^\n]\n",Espressione)
```

Qualora potessimo essere sicuri che l'espressione non contenesse spazi, potremmo anche limitarci a leggerla con la solita stringa `%`.

10.3.3 Prima fase (`calcola1.c`)

A questo punto, possiamo procedere alla prima fase, completando lo scheletro della funzione principale, come sempre in modalità *top-down*. Questo richiede di:

1. leggere l'espressione corrente, che corrisponde a una riga del file;
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`) e operandi fittizii (`NO_VAL`), dato che un nodo lascia sempre uno dei due campi informativi inutilizzato e può essere utile sapere quale (a rigore, il valore fittizio per gli operatori è chiaramente riconoscibile come tale, mentre quello per gli operandi è un numero, e quindi non lo è; fortunatamente, basta uno dei due). Inoltre, avremo costanti simboliche per gli alberi vuoti (`NO_TREE`) e le posizioni fittizie (`NO_NODE`) che possono rappresentare situazioni di errore. Il file `albero0.c` fornisce solo i corpi vuoti delle funzioni; quelle che devono restituire un valore ne restituiscono uno fittizio convenzionale.

L'unica differenza tra la struttura dati albero descritta in precedenza e quella che assumiamo di possedere è il fatto che la nostra ha due campi informativi anziché uno solo, dato che vogliamo poter rappresentare nell'albero sia operatori sia numeri. È chiaro che in questo modo uno dei due campi viene sempre sprecato, cioè si occupa

memoria ridondante, ma lo spreco non è enorme. Esistono alternative sofisticate (in C esiste il concetto di tipo `union`), ma complicherebbero il codice con il problema di dover capire per ciascun elemento dell'albero a quale dei due tipi di dato appartenga. Si perderebbe probabilmente in tempo quello che si guadagna forse in spazio, e in generale preferiamo fare il compromesso opposto.

La struttura dati albero è implementata a puntatori, cioè con alberi e posizioni descritti entrambi da puntatori a nodo e in ogni nodo de campi informativi e due posizioni corrispondenti al figlio di sinistra e a quello di destra e una corrispondente al nodo padre.

```
#define NO_NODE   NULL
#define NO_TREE   NULL

typedef struct _nodo nodo;
typedef nodo *albero;
typedef nodo *posizione;

struct _nodo
{
  Operatore op;
  Operando  val;
  posizione padre;
  posizione Ts;
  posizione Td;
};
```

Vi sono poi le classiche funzioni di gestione introdotte nella discussione teorica:

- `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.

Al momento, le funzioni sono tutte definite nel file `albero0.c` come corpi vuoti. Ce ne preoccuperemo in seguito.

Cominciamo quindi a caricare l'espressione corrente dalla stringa `Espressione` in un albero `T`, di cui bisogna aggiungere la dichiarazione, e per poterlo fare dobbiamo includere il corrispondente file di intestazione. Lo faremo chiamando una funzione `ConverteEspressione`, che prende in ingresso la stringa e la converte in un albero. Seguiamo il solito triplice meccanismo di chiamata, dichiarazione e definizione (vuota).

```
/* Carica l'espressione corrente dalla stringa nell'albero */
albero ConverteEspressione (char *Espressione);
```

Quindi, possiamo calcolare il valore dell'espressione rappresentata dall'albero per poi stamparlo in un opportuno formato. Il valore sarà un numero reale `v`, ottenuto chiamando una procedura `CalcolaValore` sull'albero `T`.

```
/* Calcola il valore dell'espressione rappresentata dall'albero */
double CalcolaValore (albero T);
```

Quindi, possiamo stampare l'espressione in notazione infissa con la funzione `StampaEspressione`, seguita da un a capo, che potrà essere incluso o meno nella stampa a seconda dei gusti (qui non lo faremo, per consentire di accodare più stampe se desiderato).

```
/* Stampa in notazione infissa l'espressione rappresentata
   dall'albero */
void StampaEspressione (albero T);
```

Infine, dobbiamo calcolare la profondità dell'albero, che è ovviamente un numero intero ed è il risultato di una funzione `Profondita` applicata all'albero. Ancora una volta, dovremo stampare a video il risultato.

```
/* Calcola la profondita' dell'albero */
int Profondita (albero T);
```

Prima di passare all'iterazione seguente, dobbiamo distruggere l'albero con l'apposita funzione di libreria, che riceve l'albero per indirizzo, dato che deve modificarlo.

In conclusione, otteniamo un codice che compila, salvo alcuni messaggi di avvertimento dovuti al fatto che alcune funzioni dovrebbero restituire dei valori, ma (essendo ancora vuote) non lo fanno. Il codice funziona anche, ovviamente stampando valori casuali.

```
void ElaboraEspressioni (char *file_espressioni)
{
  FILE *fp;
  char Espressione[ROW_LENGTH];
  albero T;
  double v;
  int p;


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

  /* Legge il file una riga alla volta */
  while (fscanf(fp,"%[^\n]\n",Espressione) == 1)
  {
    /* Carica l'espressione corrente dalla stringa nell'albero */
    T = ConverteEspressione(Espressione);

    /* Calcola il valore dell'espressione rappresentata dall'albero */
    v = CalcolaValore(T);
    printf("valore      = %f\n",v);

    /* Stampa in notazione infissa l'espressione rappresentata
        dall'albero */
    StampaEspressione(T);
    printf("\n");

    /* Calcola la profondita' dell'albero */
    p = Profondita(T);
    printf("profondita = %d\n",p);

    /* Distrugge l'albero */
    distruggealbero(&T);
    printf("\n");
  }

  fclose(fp);
}
```

Può non essere una cattiva idea far temporaneamente stampare la stringa `Espressione`, allo scopo di verificare che quanto meno la lettura sia andata a buon fine.

10.3.4 Seconda fase (`albero.c`)

La fase successiva consiste nel completare le funzioni della libreria di gestione dell'albero, con la consueta interruzione temporanea dell'approccio *top-down* per passare all'approccio *bottom-up*.

Lo scopo è duplice: da un lato avere il più presto possibile dei risultati per correggere eventuali errori, dall'altro sfruttare la recente esposizione teorica per non dover tornare a rivedere i concetti astratti. Va anche detto che in pratica molto spesso librerie di basso livello come queste sono già disponibili fin dal principio, e quindi questa interruzione non ha luogo. D'altra parte, si potrebbe anche procedere rigorosamente in modalità *top-down* e lasciare per ultima la realizzazione della libreria stessa.

Le funzioni sono quasi tutte banali. Per realizzarle è in genere utile avere un supporto grafico, che illustri chiaramente le singole operazioni.

Creazione di un albero vuoto Un albero vuoto è un albero che non ha nodi. Siccome nell'implementazione a puntatori, un albero è un puntatore a nodo, se l'albero è vuoto, la cosa più ragionevole è che il puntatore abbia valore `NO_TREE`[2].

```
albero creaalbero ()
{
  return NO_TREE;
}
```

Rimandiamo la distruzione perché abbiamo osservato che distruggere un albero o cancellare un sottoalbero sono compiti molto simili, per cui ridurremo il primo al secondo, eseguendolo a partire dalla radice.

Test di vuotezza Per sapere se un albero è vuoto, basta semplicemente verificare se il puntatore che lo rappresenta vale `NO_TREE` oppure punta a un nodo radice.

```
boolean alberovuoto (albero T)
{
  return (T == NO_TREE);
}
```

Lettura di un nodo Per poter leggere l'informazione associata a un nodo, dobbiamo conoscere l'albero e la posizione del nodo nell'albero. Sono entrambi puntatori, anche se definiti con tipi astratti diversi. Trovato il nodo, si deve accedere al campo informativo desiderato. È una situazione molto simile a quella delle liste. Il fatto che i campi siano due può portare o a scrivere due funzioni (una per campo) o a scriverne una che li legga entrambi e li assegni agli argomenti passati per indirizzo. Questa è la strategia che abbiamo scelto.

```
void leggenodo (albero T, posizione p, Operatore *pop, Operando *pval)
{
  *pop = p->op;
  *pval = p->val;
}
```

Si può anche osservare che, come nelle liste, l'albero `T` non compare nella funzione, dato che la posizione espressa come puntatore è sufficiente da sola a ottenere il dato desiderato. Teniamo comunque entrambi i parametri perché una diversa implementazione (un vettore con posizioni descritte da indici interi) ne avrebbe invece bisogno e vogliamo ancora un'implementazione che nasconda la struttura effettiva e simuli la struttura dati astratta.

[2] L'esempio delle liste potrebbe far pensare che usare una sentinella possa essere utile. In effetti, però, i vantaggi che la sentinella procura nella gestione delle liste non si ritrovano negli alberi.

Scrittura di un nodo La scrittura di dati nei nodi è assolutamente analoga: albero e posizione determinano il nodo, nei cui campi è possibile copiare i valori dei due parametri (anche qui, usiamo una sola funzione per entrambi i campi anziché una per ciascuno).

```
void scrivenodo (albero T, posizione p, Operatore op, Operando val)
{
  p->op = op;
  p->val = val;
}
```

Determinazione della radice Partendo dal puntatore che determina un albero, si vuole il puntatore che determina la posizione della sua radice. Qui siamo quasi all'assurdo, nel senso che questo secondo puntatore coincide esattamente col primo. Il senso di chiamare una funzione, spendendo tempo e spazio, per riottenere ciò che già si ha è al solito l'astrazione: altre implementazione richiederebbero operazioni più complicate (anche se poco).

```
posizione radice (albero T)
{
  return T;
}
```

Ci si potrebbe chiedere se sia necessario, o almeno opportuno, trasformare esplicitamente tale puntatore dal tipo `albero` al tipo `posizione` con un'operazione esplicita di *cast* (`return (posizione) T;`). La conversione in pratica non fa nulla. Inoltre, l'operazione è contenuta in una libreria, a basso livello, dove l'uguaglianza dei due tipi è esplicitamente nota, dato che è dichiarata nell'intestazione. Quindi, pare lecito sfruttarla esplicitamente e risparmiarsi operazioni inutili.

Determinazione dei sottoalberi e del nodo padre Per determinare le posizioni delle radici del figlio sinistro e del figlio destro è sufficiente determinare i campi puntatore interni alla posizione corrente.

```
posizione figliosinistro (albero T, posizione p)
{
  return p->Ts;
}


posizione figliodestro (albero T, posizione p)
{
  return p->Td;
}
```

Lo stesso vale per la posizione del nodo padre.

```
posizione padre (albero T, posizione p)
{
  return p->padre;
}
```

Ancora una volta, passiamo l'albero anche se in questa implementazione non serve.

Costruzione di un albero Ricordiamo che questa funzione deve costruire un albero partendo dall'informazione che vogliamo conservare nel nodo radice (un operatore o un operando, in effetti entrambi, con l'accortezza che uno dei due sarà fittizio) e da due alberi che diventeranno i sottoalberi sinistro e destro di quello nuovo. Il nuovo albero complessivo sarà ottenuto creando un nuovo nodo con l'informazione desiderata e appendendo ad esso i due alberi dati.

Dobbiamo quindi allocare la memoria per un nodo `r`, che diventerà la radice, verificare che l'allocazione sia andata a buon fine e assegnare ai due campi informativi i valori desiderati. Restano da indicare i tre puntatori. Il nodo padre per un nodo radice non esiste, per cui lo indicheremo con la posizione fittizia `NO_NODE`. I due sottoalberi sembrano dover essere banalmente i due alberi passati come argomento, ma questo non è sufficiente. Nel momento in cui si decide che `Ts` e `Td` non sono più alberi indipendenti, ma sottoalberi di `T`, le loro radici devono acquisire un nodo padre, che è `r`. D'altra parte, non possiamo banalmente assegnare `r` a `Ts->padre` e `Td->padre`, perché potrebbe essere che questi due alberi siano vuoti. In tal caso, è corretto assegnarli ai due campi del nuovo nodo radice, ma non è corretto scrivere nel campo `padre`, che non esiste.

```
albero costruiscealbero (Operatore op, Operando val, albero Ts,
    albero Td)
{
  nodo *r;

  /* Alloca l'elemento radice */
  r = (nodo *) malloc(sizeof(nodo));
  if (r == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione di un albero!\n");
    exit(EXIT_FAILURE);
  }

  r->op = op;
  r->val = val;
  r->padre = NO_NODE;
  r->Ts = Ts;
  if (Ts != NULL) Ts->padre = r;
  r->Td = Td;
  if (Td != NULL) Td->padre = r;

  return r;
}
```

Un'altra osservazione, che vale anche per la successiva operazione di cancellazione, è che l'operazione restituisce un albero, che è il risultato dell'operazione. D'altra parte, l'operazione modifica l'albero di partenza, che non esiste quindi più in quanto tale. A questo punto, non ci sarebbe veramente bisogno di passarlo anche in uscita. Lo facciamo per adattare il più possibile la forma della funzione alla sua definizione astratta. Avevamo fatto lo stesso per le liste.

Nel caso della costruzione, l'albero che viene restituito è il puntatore al nuovo nodo radice, da cui l'ultima istruzione `return r;`.

Cancellazione di un sottoalbero Per cancellare un sottoalbero, dobbiamo conoscere l'albero di cui fa parte e la posizione del nodo radice del sottoalbero stesso.

Questa posizione viene passata per indirizzo, perché, eseguita l'operazione, vogliamo che sia correttamente annullata senza lasciare in giro puntatori ad aree di memoria deallocate (un po' come abbiamo passato per indirizzo la posizione di un elemento da cancellare da una lista). Non dobbiamo però cancellare solo il nodo nella posizione indicata, ma anche tutti i suoi discendenti. Apparentemente, il compito è immane, ma la struttura ricorsiva degli alberi consente di portarlo a termine facilmente. La potenza della ricorsione, infatti, consente di ridurre il problema alla cancellazione di alberi più piccoli, per esempio dei due sottoalberi. Se ipotizziamo di saperlo fare (e basterà chiamare ricorsivamente la funzione che ancora non abbiamo scritto), rimarrà solo da cancellare il nodo di partenza, cosa piuttosto banale.

Al solito, occorre definire un caso base. In linea di principio, conviene quasi sempre che questo sia il caso di albero vuoto. Lo è anche qui: se l'albero è vuoto, cioè se la posizione fornita di partenza è quella fittizia, non occorre fare nulla. Nel caso ricorsivo, invece, basta cancellare con chiamate ricorsive i due sottoalberi sinistro e destro, deallocare il nodo considerato e aggiornare il suo puntatore al valore fittizio. Raggiungere i due sottoalberi richiede delle espressioni un po' contorte perché non partiamo dalla posizione del nodo, ma da un puntatore ad essa. Quindi, il nodo ha posizione `*pp` e il suo sottoalbero sinistro è `(*pp)->Ts`, dove le parentesi tonde sono necessarie per evitare che le precedenze fra operatori diano un risultato scorretto: l'asterisco deve essere applicato a `pp` prima che la freccia di dereferenziazione vada a raggiungere il campo `Ts`. Infine, questo sottoalbero va passato per indirizzo, per cui occorre premettere a tutto l'operatore `&`.

```
albero cancsottoalbero (albero T, posizione *pp)
{
  /* Visita il sottoalbero in postordine deallocandone i nodi */
  if (*pp != NO_NODE)
  {
    cancsottoalbero(T,&(*pp)->Ts);
    cancsottoalbero(T,&(*pp)->Td);
    free(*pp);
    *pp = NO_NODE;
  }

  return T;
}
```

Osserviamo che l'intera operazione somiglia molto a una visita ricorsiva del sottoalbero, cosa naturale, dato che per cancellarlo completamente si dovrà scorrerlo. La visita è ovviamente compiuta in postordine, dato che deallocare la radice prima di uno dei due sottoalberi (o di entrambi) impedirebbe poi di accedervi.

Come la costruzione, anche la cancellazione restituisce un albero per aderire il più possibile alla forma astratta dell'operazione. In questo caso, però, l'albero restituito è quello di partenza, `T`.

Distruzione di un albero La distruzione di un albero richiede un puntatore `pT` all'albero stesso, dato che si tratta di una modifica, che comporta un passaggio per indirizzo. Come già osservato, distruggere un albero coincide, in sostanza, con il cancellarlo a partire dalla sua radice. Quindi, l'albero da distruggere è `*pT` e la posizione da usare per cancellarlo è ancora `*pT`, ma il suo indirizzo da fornire alla procedura di cancellazione è `pT`.

Il codice risultante è molto stringato e un po' criptico.

```
void distruggealbero (albero *pT)
{
  cancsottoalbero(*pT,pT);
}
```

Complessità Quanto sopra ci consente di utilizzare gli alberi senza pensare ai puntatori che li realizzano.

Dalle considerazioni svolte è ovvio che quasi tutte le operazioni fondamentali sono in tempo costante $\Theta(1)$, purché le informazioni associate ai nodi abbiano dimensione costante. Fanno eccezione la cancellazione e la distruzione, che richiedono tempo lineare nel numero n degli elementi dell'albero ($\Theta(n)$).

10.3.5 Terza fase (`calcola2.c`)

A questo punto, possiamo tornare allo svolgimento in modalità *top-down* e realizzare la procedura `ConverteEspressione`, che converte un'espressione letta da file in un albero binario.

Le espressioni cominciano con un operatore, seguito da una parentesi tonda aperta, dal primo operando, da una virgola, dal secondo operando e da una parentesi chiusa. La struttura è molto rigorosa ed è ricorsiva, dato che ognuno dei due operandi, a sua volta, è costituito da un operatore, una parentesi aperta, un operando, una virgola, un altro operando e una parentesi chiusa. Fanno eccezione solo le espressioni semplici, che sono numeri reali (o interi, come caso particolare). Quindi, per convertire una stringa che rappresenta un'espressione dobbiamo per prima cosa riconoscere se si tratta di un'espressione semplice o composta. È facile farlo, dato che si tratta di un numero reale, anziché di una delle quattro parole chiave che rappresentano gli operatori aritmetici. Costruire l'albero corrispondente a un'espressione semplice è anch'essa un'operazione facile: si tratta del solo nodo radice. Per le espressioni composte, la cosa diventa più complicata, ma non particolarmente difficile: bisogna dividere la stringa in tre parti e trasformare la prima parte (l'operatore) nella radice dell'albero e le altre due parti (gli operandi) nei sottoalberi. Ipotizzando ricorsivamente di saper trasformare le stringhe che descrivono gli operandi in alberi, non è difficile ricombinare questi ultimi con il nodo radice per ottenere l'albero complessivo: abbiamo appena discusso e realizzato la procedura `costruiscealbero` per farlo. Tutte le funzioni che lavorano su alberi tendono ad avere una struttura ricorsiva che riflette quella dei dati su cui operano. È vero che tutte le procedure ricorsive si possono ridurre in forma iterativa, e conviene anche farlo, perché si risparmia tutta la meccanica delle chiamate, dell'allocazione dei record di attivazione sullo *stack* e della restituzione dei valori. D'altra parte, questa trasformazione è facile nel caso di ricorsione terminale, mentre gli alberi binari tendono ad avere due chiamate ricorsive (una per sottoalbero), e quindi almeno una delle due chiamate non è terminale e la trasformazione richiede di gestire esplicitamente una pila. Tutto questo eccede i limiti del corso, per cui considereremo solo la forma ricorsiva.

Per realizzare la procedura `ConverteEspressione` manca solo il riconoscimento di espressioni semplici e la divisione delle stringhe nelle tre componenti. Cominciamo a stendere uno schema costituito da commenti.

```
/* Converte un'espressione da stringa ad albero */
albero ConverteEspressione (char *Espressione)
```

```
{
  /* Se l'espressione consiste in un numero reale, e' un'espressione
     semplice */
  /* Se invece comincia con il nome di un operatore, seguito da
     parentesi, e' composta */
    /* Identifica le tre parti dell'espressione: operatore, operando1
       e operando2 */

  return T;
}
```

Per riconoscere se un'espressione è semplice possiamo sfruttare la capacità di `fscanf` di riconoscere i numeri reali. In questo modo, oltre a restituire il valore 1 quando l'espressione è davvero semplice, la funzione assegna il valore dell'espressione a una variabile `val` di tipo `Operando` (cioè numero reale).

```
if (sscanf(Espressione,"%lf",&val) == 1)
```

In caso positivo, dobbiamo valutare il numero reale (cosa che `fscanf` ha già fatto) e costruire un semplice albero `T` fatto di un nodo radice che contiene il numero `val`, l'operatore fittizio e due sottoalberi vuoti.

```
T = costruiscealbero(NO_OP, val, NO_TREE, NO_TREE);
```

Se l'espressione invece è composta, dobbiamo dividere la stringa nelle tre sottostringhe relative all'operando (`s_op`) e ai due operatori (`s_op1` e `s_op2`). Questo è un problema abbastanza complicato da volerlo affidare a un'opportuna procedura `ScomponeEspressione`, come di consueto nell'approccio *top-down*. Per semplicità, diamo per scontato che le stringhe in ingresso siano ben formate e non occorra controllarlo.

```
/* Identifica le tre parti dell'espressione: operatore, operando1
   e operando2 */
ScomponeEspressione(Espressione,s,s_op1,s_op2);
```

Poi dobbiamo identificare l'operatore `op` a partire dalla stringa (altra procedura, `IdentificaOperatore`) e costruire i due sottoalberi `Ts` e `Td` relativi ai due operatori con chiamate ricorsive a `ConverteEspressione`, applicate alle due stringhe `s_op1` e `s_op2`. Infine, costruiremo l'albero `T` componendo il valore fittizio, l'operatore identificato e i due sottoalberi.

```
Ts = ConverteEspressione(s_op1);
Td = ConverteEspressione(s_op2);
op = IdentificaOperatore(s);
T = costruiscealbero(op,NO_VAL,Ts,Td);
```

E nel complesso otteniamo la soluzione al problema:

```
/* Converte un'espressione da stringa ad albero */
albero ConverteEspressione (char *Espressione)
{
  char s[ROW_LENGTH];
  char s_op1[ROW_LENGTH];
  char s_op2[ROW_LENGTH];
  Operando val;
  Operatore op;
  albero T, Ts, Td;


  /* Se l'espressione consiste in un numero reale, e' un'espressione
     semplice */
  if (sscanf(Espressione,"%lf",&val) == 1)
    T = costruiscealbero(NO_OP, val, NO_TREE, NO_TREE);
  else
  /* Se invece comincia con il nome di un operatore, seguito da
     parentesi, e' composta */
  {
    /* Identifica le tre parti dell'espressione: operatore, operando1
       e operando2 */
    ScomponeEspressione(Espressione, s, s_op1, s_op2);

    Ts = ConverteEspressione(s_op1);
    Td = ConverteEspressione(s_op2);
    op = IdentificaOperatore(s);
    T = costruiscealbero(op, NO_VAL, Ts, Td);
  }

  return T;
}
```

che richiede di aggiungere dichiarazioni e definizioni vuote di due nuove procedure. `ScomponeEspressione` riceve una stringa e ne restituisce tre, `IdentificaOperatore` riceve una stringa e restituisce un `Operatore`, cioè un carattere.

```
/* Scompone Espressione nelle sue componenti s, s_op1 e s_op2 */
void ScomponeEspressione (char *Espressione, char *s, char *s_op1,
    char *s_op2);

/* Identifica l'operatore rappresentato dalla stringa s */
Operatore IdentificaOperatore (char *s);
```

Possiamo verificare la correttezza sintattica compilando il tutto, ma ancora non abbiamo operazioni che forniscano risultati utili a valutarne la correttezza semantica. Per arrivarci, dobbiamo quanto meno completare le due procedure e arrivare a costruire l'albero che rappresenta l'espressione.

10.3.6 Quarta fase (`calcola3.c`)

Procediamo quindi a realizzare le funzioni ancora vuote. `IdentificaOperatore` è una banale serie di confronti in cascata fra la stringa che rappresenta l'operatore e le quattro stringhe predefinite, condotti con la funzione `strcmp`. In ciascun caso si restituisce il carattere corrispondente.

```
/* Identifica l'operatore rappresentato dalla stringa s */
Operatore IdentificaOperatore (char *s)
{
```

```
  if (strcmp(s,"somma") == 0)
    return '+';
  else if (strcmp(s,"diff") == 0)
    return '-';
  else if (strcmp(s,"prod") == 0)
    return '*';
  else if (strcmp(s,"div") == 0)
    return '/';
  else
  {
    fprintf(stderr,"Errore nell'identificazione dell'operatore
        %s!\n",s);
    exit(EXIT_FAILURE);
  }
}
```

`ScomponeEspressione` è decisamente più complicata, perché deve dividere la stringa in tre parti individuando i punti in cui ciascuna parte comincia e finisce. Assumendo che la stringa sia scritta correttamente, l'operatore è tutto ciò che precede la prima parentesi aperta[3]. Non è difficile scrivere una funzione che trova la sottostringa dal principio alla prima parentesi tonda.

Il primo operando è più complicato: comprende tutto ciò che segue la prima parentesi tonda e precede la virgola che separa il primo operando dal secondo. Tuttavia, questa virgola non è in genere la prima. Nell'espressione `prod(somma(5,3),div(10,diff(4,2)))`, il primo operando è `somma(5,3)`, non `somma(5`. La virgola corretta è la seconda, perché la prima è contenuta all'interno di un'espressione, cioè di una coppia di parentesi. Per identificare la virgola giusta, bisogna contare le parentesi che sono state aperte e non ancora chiuse: quando tutte le parentesi aperte dell'espressione sono state bilanciate da parentesi chiuse, la prima virgola segna la fine dell'operando. Il secondo operando è costituito da tutto ciò segue la virgola identificata e precede la parentesi che chiude l'intera espressione, che in generale non è la prima. Nell'esempio già discusso, il secondo operando è `div(10,diff(4,2))`, non `div(10,diff(4,2)`. Ancora una volta, bisogna garantire il bilanciamento fra parentesi aperte e chiuse.

A prima vista, riconoscere le tre parti dell'espressione sembrano tre compiti diversi. In realtà, si possono svolgere in modo abbastanza elegante, riconoscendo che i tre compiti sono molto simili fra loro. In tutti e tre i casi, si tratta di scorrere la stringa fino a un opportuno carattere terminale (una parentesi aperta, una virgola o una parentesi chiusa), ma sotto la condizione che il numero di parentesi aperte e chiuse sia perfettamente bilanciato. In breve, si cerca la prima occorrenza di un carattere terminale che non sia annidata all'interno di una coppia di parentesi. Per risolvere il problema realizzeremo una funzione `TrovaCarattereNonAnnidato`, che riceva una stringa e un carattere terminale e restituisca la prima posizione del carattere che non sia annidata fra parentesi tonde. Dunque, si tratta di scorrere la stringa tenendo conto del numero di parentesi aperte che non siano ancora state chiuse: il carattere terminale va accettato solo se questo numero è nullo.

```
int TrovaCarattereNonAnnidato (char ch, char *Espressione);
```

Nota la posizione, possiamo copiare nella stringa `s` tutto ciò che è nella stringa `Espressione` dal principio fino alla posizione individuata. Per farlo, basta chiamare

[3]Ovviamente, se cominciano ad esserci spazi bianchi, questo si complica potenzialmente molto. Quindi, lo escludiamo.

la funzione `strncpy`, che differisce dalla basica funzione `strcpy` in quanto copia al massimo un numero di caratteri `i` dato come argomento. Infatti, noi sappiamo esattamente quanti caratteri copiare. I caratteri copiati occupano le posizioni di indice compreso fra `0` e `i-1`. In corrispondenza all'indice `i` andiamo a scrivere il terminatore di stringa `'\0'`, in modo che `s` sia una stringa ben formata.

```
i = TrovaCarattereNonAnnidato('(',Espressione);
strncpy(s,Espressione,i);
s[i] = '\0';
```

Un approccio alternativo richiederebbe di modificare la stringa `Espressione`: invece di gestire tre stringhe ausiliarie, che occupano memoria e richiedono tempo per la copia, si potrebbero gestire le tre sottostringhe direttamente come pezzi della stringa data. In questo caso, dovremo comunque trovare la posizione del carattere terminale, ma potremmo semplicemente azzerarla senza dover copiare la sottostringa altrove.

Riconosciuto e copiato l'operatore, possiamo passare agli operandi. Per il primo, partiamo dalla posizione `i+1`. Possiamo sfruttare l'equivalenza fra puntatori e vettori per creare virtualmente sulla stringa `Espressione` una sottostringa che parta in un'altra posizione: è sufficiente assegnare a un puntatore l'indirizzo di tale posizione, cioè `&Espressione[i+1]`. Ricordandoci che l'argomento `Espressione` non è il vettore statico allocato e riempito nella funzione `ElaboraEspressioni`, ma semplicemente un puntatore nello spazio di memoria locale della funzione `ConverteEspressione`. Così infatti funziona il passaggio di vettori alle funzioni C. Quindi si può tranquillamente modificare il valore di tale puntatore, senza alterare il vettore statico originale (che, in quanto vettore statico, non potrebbe proprio essere usato a sinistra in un assegnamento) rimane intatto nello spazio di memoria della funzione chiamante. Il puntatore aggiornato descrive la sottostringa che comincia con il primo operatore e prosegue fino al termine dell'espressione originale. Possiamo passarla alla procedura `TrovaCarattereNonAnnidato`, cercando questa volta non una parentesi aperta, ma una virgola non annidata. Il risultato sarà la posizione (nella sottostringa) della virgola e si può usare per copiare la seconda sottostringa in `s_op1` e terminarla.

```
Espressione = &Espressione[i+1];
i = TrovaCarattereNonAnnidato(',',Espressione);
strncpy(s_op1,Espressione,i);
s_op1[i] = '\0';
```

Infine, determiniamo la posizione iniziale della terza sottostringa, che segue immediatamente il carattere terminale appena individuato, determiniamo il suo carattere terminale (cioè la parentesi tonda chiusa), copiamo la sottostringa in `s_op2` e la terminiamo.

```
Espressione = &Espressione[i+1];
i = TrovaCarattereNonAnnidato(')',Espressione);
strncpy(s_op2,Espressione,i);
s_op2[i] = '\0';
```

Il tutto porta alla soluzione seguente.

```
/* Scompone Espressione nelle sue componenti s, s_op1 e s_op2 */
void ScomponeEspressione (char *Espressione, char *s, char *s_op1,
    char *s_op2)
{
  int i;


  i = TrovaCarattereNonAnnidato('(',Espressione);
  strncpy(s,Espressione,i);
  s[i] = '\0';

  Espressione = &Espressione[i+1];
  i = TrovaCarattereNonAnnidato(',',Espressione);
  strncpy(s_op1,Espressione,i);
  s_op1[i] = '\0';

  Espressione = &Espressione[i+1];
  i = TrovaCarattereNonAnnidato(')',Espressione);
  strncpy(s_op2,Espressione,i);
  s_op2[i] = '\0';
}
```

Ancora una volta, la nuova funzione in realtà non fa nulla. Per poter vedere qualcosa dobbiamo arrivare a riempire `TrovaCarattereNonAnnidato`. Questo non è complicatissimo: basta scorrere la stringa tenendo traccia della differenza fra parentesi aperte e chiuse individuate. Quando il carattere desiderato viene letto e il conteggio delle parentesi è bilanciato, ci fermiamo e restituiamo la posizione corrente. Val la pena di ragionare sul caso in cui il carattere non esiste o è sempre annidato (non succede quando le espressioni sono corrette, ma potremmo voler riciclare altrove la funzione). Potremmo decidere che in tal caso si restituisce la posizione del terminatore di stringa, che è oltre le posizioni lecite, e quindi è riconoscibile.

Dichiarato un cursore `i`, si scorre con esso l'intera stringa, dall'indice `0` all'indice in cui compare il terminatore[4]. Per tenere conto delle parentesi, dichiariamo anche una variabile `cont`, che misura la differenza (inizialmente nulla) fra parentesi aperte e chiuse.

Durante lo scorrimento, ad ogni carattere verifichiamo se si tratta del carattere terminale e il conteggio è bilanciato. In caso positivo, usciamo dalla procedura restituendo la posizione corrente. Altrimenti, verifichiamo se è una parentesi aperta o chiusa, e corrispondentemente aggiorniamo il conteggio, rispettivamente incrementando o decrementando la variabile `cont`. Al termine, restituiamo la posizione corrente, che è quella del terminatore, dato che siamo usciti dal ciclo.

```
int TrovaCarattereNonAnnidato (char ch, char *Espressione)
{
  int i, cont;

  cont = 0;
  for (i = 0; Espressione[i] != '\0'; i++)
  {
    if ( (Espressione[i] == ch) && (cont == 0) ) return i;
    if (Espressione[i] == '(') cont++;
    if (Espressione[i] == ')') cont--;
  }

  return i;
```

[4]Non si usa assolutamente `strlen` per conoscere l'indice finale, dato che questo corrisponde a scorrere la stringa un'altra volta, o addirittura una volta per ogni carattere, se la funzione viene chiamata nella condizione del ciclo.

}

Ancora una volta, possiamo fare una verifica più che altro sintattica. Volendo, potremmo far stampare le tre sottostringhe per verificare che la decomposizione avvenga correttamente.

Complessità 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.

10.3.7 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.

Qual è il caso base? Si è detto che in genere il caso base quando si opera su alberi è quello di albero vuoto. Questo è in effetti vero, ma nella specifica situazione non è l'unico caso base. Avremo in effetti due casi base:

1. il caso dell'*albero vuoto*, per il quale si restituisce il valore fittizio `NO_VAL`;
2. il caso dell'albero che rappresenta un'espressione semplice, cioè dell'*albero ridotto alla sola radice*, che ha un operando corretto e un operatore fittizio `NO_OP`); in questo caso, si restituisce il valore dell'operando.

Ovviamente, c'è poi il caso ricorsivo, nel quale occorre prima valutare i due operandi descritti dai sottoalberi sinistro e destro e poi applicare l'operatore ai loro valori.

Procediamo all'implementazione.

```
/* Calcola il valore dell'espressione rappresentata dall'albero */
double CalcolaValore (albero T)
{
  Operatore op;
  Operando val, vals, vald;

  if (alberovuoto(T))
    return NO_VAL;
  else
  {
    leggenodo(T, radice(T),&op,&val);
    if (op == NO_OP)
      return val;
    else
    {
      vals = CalcolaValore(figliosinistro(T, radice(T)));
      vald = CalcolaValore(figliodestro(T, radice(T)));
      if (op == '+')
        return vals + vald;
      else if (op == '-')
        return vals - vald;
      else if (op == '*')
        return vals * vald;
      else if (op == '/')
        return vals / vald;
```

```
      else
      {
        fprintf(stderr,"Errore nell'operatore!\n");
        exit(EXIT_FAILURE);
      }
    }
  }
}
```

Il primo caso base usa la funzione di libreria `alberovuoto` e restituisce il valore fittizio `NO_VAL`. Il secondo richiede di leggere con l'apposita funzione di libreria l'operatore e l'operando associati al nodo radice, determinato a sua volta con una funzione di libreria. Se l'operatore è fittizio, l'albero rappresenta un'espressione semplice, e quindi si restituisce l'operando. Altrimenti, per valutare i valori `vals` e `vald` dei due operandi si eseguono chiamate ricorsive sui due alberi figli del nodo radice dell'albero `T`. I figli e il nodo radice sono ricavati con le apposite funzioni di sistema. Infine, una cascata di costrutti di selezione in base al carattere `op` determina quale delle quattro operazioni compiere sui due valori per ottenere quello da restituire. Finalmente, possiamo verificare anche eseguendo il programma che il codice è stato realizzato correttamente.

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.

10.3.8 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.

Avremo ancora due casi base (l'albero vuoto e l'albero ridotto alla sola radice), dato che corrispondono a operazioni concettualmente diverse. Nel primo non si fa nulla (quindi, nel codice questo caso apparentemente non comparirà). Nel secondo caso base, si stampa il valore dell'operando. Ci sarà poi un caso ricorsivo, nel quale si tratta di stampare prima una parentesi aperta, l'espressione di sinistra, l'operatore, l'espressione di destra e infine una parentesi chiusa. Quindi, si tratta di manipolare prima il sottoalbero sinistro, poi il nodo radice e infine il sottoalbero destro. Il tutto corrisponde ancora a una visita dell'albero, questa volta in in-ordine. Di conseguenza, la complessità è ancora $\Theta(n)$.

Procedendo all'implementazione, partiremo come sempre dal caso base di albero vuoto, che però non comporta operazioni. Quindi, per distinguere il secondo caso base dal caso ricorsivo leggiamo dal nodo radice operatore `op` e operando `val`. Qualora l'operatore fosse fittizio, saremmo nel secondo caso base e procederemmo a stampare l'operando. Nel caso ricorsivo, invece, abbiamo la stampa della parentesi aperta, la chiamata ricorsiva di `StampaEspressione` sul nodo radice del sottoalbero

sinistro, la stampa dell'operatore, la chiamata ricorsiva di `StampaEspressione` sul nodo radice del sottoalbero destro e la stampa della parentesi chiusa.

```
/* Stampa in notazione infissa l'espressione rappresentata
    dall'albero */
void StampaEspressione (albero T)
{
  Operatore op;
  Operando val;

  if (!alberovuoto(T))
  {
    leggenodo(T,radice(T),&op,&val);
    if (op == NO_OP)
      printf(" %f ",val);
    else
    {
      printf(" ( ");
      StampaEspressione(figliosinistro(T,radice(T)));
      printf(" %c ",op);
      StampaEspressione(figliodestro(T,radice(T)));
      printf(" ) ");
    }
  }
}
```

Il risultato non è leggibilissimo, a causa delle troppe parentesi e dei troppi spazi bianchi, ma si può raffinare investendo ulteriore tempo e pazienza. Concettualmente, non aggiungerebbe nulla.

10.3.9 Settima fase (`calcola6.c`)

L'ultima fase dell'esercizio realizza il calcolo della profondità dell'albero. Questa è definita come il numero massimo di nodi compresi in un cammino dalla radice a una delle foglie dell'albero. La profondità viene calcolata ancora con un algoritmo ricorsivo. Questa volta è sufficiente considerare il caso base in cui l'albero è vuoto, dato che la distinzione tra operatori e operandi non ha effetto sul calcolo della profondità, mentre ne aveva per la valutazione e la stampa. Nel caso base, la profondità è per definizione nulla. Nel *caso ricorsivo*, invece, che include la possibilità di avere un albero ridotto alla sola radice, 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.

Procediamo quindi all'implementazione.

```
/* Calcola la profondita' dell'albero */
int Profondita (albero T)
{
  int ps, pd;

  if (alberovuoto(T))
    return 0;
  else
  {
    ps = Profondita(figliosinistro(T,radice(T)));
    pd = Profondita(figliodestro(T,radice(T)));
```

```
    return 1 + ( (ps > pd) ? ps : pd);
  }
}
```

Nel caso di albero vuoto, la procedura restituisce valore nullo. In ogni altro caso, determiniamo le due profondità `ps` e `pd` con chiamate ricorsive sui due sottoalberi relativi alla radice. Per determinare la maggiore delle due e incrementarla di 1, possiamo usare un costrutto selettivo classico, oppure quello compatto a tre argomenti.

10.3.10 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, ma questo va oltre i limiti del corso. Merita però discutere le differenze fondamentali:[5]

- 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 dovrebbe richiedere nessuna modifica al file `calcola.c`: potenza delle strutture dati astratte[6].

10.4 Esercizi

10.4.1 Esercizio

Si modifichi l'esercizio svolto in laboratorio in modo che il programma carichi le espressioni in memoria in un vettore di stringhe e la procedura che risolve il problema operi su quest'ultimo.

[5] Quanto segue è stato riadattato dalla lezione sulle liste senza procedere all'effettiva implementazione: si prega di segnalare eventuali errori o incongruenze.

[6] In realtà, probabilmente non è esattamente così, per una serie di tecnicalità, legate al fatto che tutti gli alberi che vengono manipolati dall'algoritmo devono essere ospitati sullo stesso vettore per poter essere combinati, e questo complica le cose. Al momento, non ho una soluzione completa della difficoltà da proporre.

10.4.2 Esercizio

Si modifichi l'esercizio svolto in laboratorio in modo che il programma carichi le espressioni in memoria in un vettore di alberi binari e la procedura che risolve il problema operi su quest'ultimo.

10.4.3 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.

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 (vedi Figura 11.1). 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` (vedi Figura 11.2). 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

[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

struttura interna 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. Possiamo notare che entrambe le funzioni in teoria ricevono un albero e ne restituiscono un altro. Come già avvenuto per le liste, vedremo che in pratica converrà modificare l'albero originale, ma manterremo comunque l'idea che queste funzioni restituiscono il risultato.

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.

Si è già detto che in generale si assume l'esistenza di una relazione di ordine totale sull'insieme universo U, dato che (in caso contrario) l'unica ricerca possibile è quella sequenziale, che richiede troppo tempo per dizionari grandi, e questo influenza anche le operazioni di inserimento e cancellazione descritte in precedenza. Altre due operazioni sfruttano l'ordinamento totale dell'insieme universo U, che automaticamente si riflette su tutti i suoi sottoinsiemi, e quindi in particolare su tutti i dizionari possibili. Le due funzioni sono la ricerca dell'elemento minimo e dell'elemento massimo del dizionario (vedi Figura 11.3). Le corrispondenti funzioni `minABR` e `maxABR` ricevono un albero e restituiscono in uscita, rispettivamente, il valore del suo elemento minimo e di quello massimo. Al solito, occorre definire che cosa accada nel caso particolare di albero vuoto. Evidentemente, non esiste un valore minimo o massimo, per cui occorre restituire un valore fittizio, che indicheremo

come $+\infty$ per il minimo peggiore in assoluto e $-\infty$ per il massimo peggiore. A parte questo caso un elemento minimo e uno massimo esistono sempre perché l'insieme rappresentato è ordinato e finito.

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 11.3: Dizionari: operazioni

Infine, al solito, siccome parliamo di strutture dati da calare nella realtà di un processore, avremo bisogno di funzioni per la creazione di un albero binario di ricerca vuoto e la distruzione di un albero binario di ricerca qualsiasi (vedi Figura 11.4).

Sottolineiamo ancora il fatto che negli alberi binari di ricerca, contrariamente a tutte le strutture trattate finora, non esiste un concetto di posizione: quando l'utente inserisce un elemento, quell'elemento va in una posizione decisa dalla struttura stessa in base all'ordinamento totale e che è tenuta nascosta all'utente

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 11.4: Dizionari: operazioni

Ora apriamo la "scatola nera" della struttura dati astratta per determinarne la forma un po' più in dettaglio. In realtà, non arriviamo ancora al livello dell'im-

plementazione, ma ci fermiamo a un livello di astrazione intermedio, nel senso che dal generico *dizionario* passiamo all'*albero binario di ricerca*, che è uno dei modi di realizzare un dizionario (un altro è la *tabella hash*, che non tratteremo). D'altra parte, l'albero binario di ricerca ha a sua volta diverse implementazioni possibili.

L'idea è rappresentare un sottoinsieme come un albero binario (vedi Figura 11.5). Ogni nodo dell'albero è un elemento del sottoinsieme. Come sempre, useremo i numeri interi come elementi di questo insieme, che sarà quindi totalmente ordinato. Un albero binario di ricerca è un albero binario (quindi, un albero nel quale ogni nodo ha un figlio sinistro e un figlio destro, che sono anch'essi alberi binari, a meno che l'albero non sia vuoto). In più, valgono alcune proprietà sull'informazione associata ai nodi. Per prima cosa, ogni nodo è associato a un'informazione diversa: non ci sono doppioni, contrariamente ai vettori, alle tabelle, alle liste e agli alberi binari. Questo perché l'operazione principale su un albero binario di ricerca è rispondere a un test di appartenenza e da questo punto di vista i doppioni semplicemente occupano spazio per niente. Per questo motivo, quando si inserisce un elemento che c'è già, in realtà non si fa nulla. La seconda proprietà fondamentale è che l'informazione associata a un nodo deve essere strettamente successiva a tutte le informazioni associate ai nodi del sottoalbero sinistro e strettamente precedente a tutte le informazioni associate al sottoalbero destro. Nell'esempio in Figura 11.5, il valore 6 conservato nella radice è strettamente maggiore dei valori 2, 1, 4 e 3 conservati nel sottoalbero sinistro e strettamente minore del valore 8 conservato nel sottoalbero destro. Analogamente, il valore 2 è strettamente maggiori del valore 1 e strettamente minore dei valori 4 e 3. Infine, il valore 4 è strettamente maggiore del valore 3, ma nel suo sottoalbero destro non ci sono valori strettamente maggiori di esso. Sulle foglie non sono imposte condizioni, dato che i loro sottoalberi sono vuoti.

Queste condizioni impongono abbastanza facilmente una forte struttura sull'insieme $\{1, 2, 3, 4, 6, 8\}$, nel senso che i suoi elementi sono distribuiti in modo non casuale nell'albero. Le regole che rispettano verranno sfruttate per rispondere in modo efficiente al test di appartenenza, per inserire e cancellare elementi, per calcolare il valore minimo e massimo. Il principio di base è abbastanza intuitivo, ma lo indagheremo in dettaglio nel seguito.

11.1 Implementazione a puntatori

Prima di tutto, discutiamo una possibile implementazione. Trattandosi di un albero binario, con regole aggiuntive sulla distribuzione dell'informazione fra i nodi, qualsiasi implementazione di un albero binario è valida anche per gli alberi binari di ricerca. Vale l'implementazione a puntatori come quella a vettori e indici. Considereremo solo l'implementazione a puntatori (vedi Figura 11.6). Questa definisce un albero binario di ricerca (che definiremo di tipo `ABR` per distinguerlo dall'albero binaro puro e semplice) come un puntatore al suo nodo radice, dunque di tipo `nodo *`. Non esiste un concetto di posizione, quindi non esistono un tipo `posizione` e una costante simbolica `NO_NODE`, mentre esiste la costante `NO_TREE` che rappresenta l'albero vuoto. Un `nodo` è una struttura che contiene un campo informazione di qualsiasi tipo ammetta un ordinamento totale e tre puntatori a nodo che sono il puntatore al nodo padre e quelli alle radici del sottoalbero sinistro e del sottoalbero destro. Non sfuggirà che la definizione è esattamente identica a quella dell'albero binario introdotta nel capitolo precedente. Quello che cambia sono le operazioni eseguibili sulla struttura.

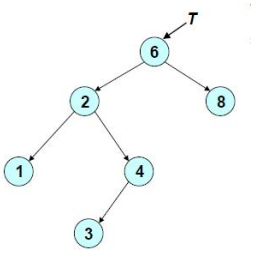
Un albero binario di ricerca (ABR) T su un insieme ordinato U è un

- 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 11.5: Albero binario di ricerca

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;
};
```

Figura 11.6: ABR: implementazione con puntatori

11.2 Esercizio implementativo: un gestore di insiemi numerici

Si vuole realizzare un gestore di sottoinsiemi di numeri interi, che chiameremo `ricerca.c`. Questo programma carica da un file di testo un insieme di numeri interi. I numeri saranno tutti diversi tra loro e semplicemente riportati in sequenza, 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
```

Il programma permetterà all'utente di manipolare interattivamente l'insieme, digitando da tastiera delle istruzioni, che il programma eseguirà via via sull'insieme corrente. Le operazioni da realizzare sono le seguenti:

- `member` i: chiede al programma di indicare se il numero intero i fa parte dell'insieme oppure no; il programma risponderà stampando a video i `in T` oppure i `non in T`, a seconda del risultato (i è il numero indicato dall'utente);
- `insert` i: chiede al programma di aggiungere il numero intero i all'insieme; come abbiamo già discusso, un numero già contenuto nell'insieme non viene duplicato;
- `delete` i: chiede al programma di cancellare il numero intero i dall'insieme; come abbiamo già discusso, un numero non contenuto nell'insieme non viene eliminato;
- `print`: chiede al programma di stampare l'insieme per valori crescenti; il programma risponderà stampando a video i numeri in ordine crescente, su una sola riga, separati da spazi bianchi;
- `empty`: chiede al programma di indicare se l'insieme è vuoto oppure no; il programma risponderà stampando a video `T e' vuoto` o `T non e' vuoto` secondo il risultato, dove `T` sarà il nome convenzionale dell'insieme di numeri rappresentato;
- `min`: chiede al programma di indicare l'elemento minimo dell'insieme; il programma risponderà stampando `min(T) =` i, dove i è l'elemento di valore minimo richiesto;
- `max`: chiede al programma di indicare l'elemento massimo dell'insieme; il programma risponderà stampando `max(T) =` i, dove i è l'elemento di valore massimo richiesto;
- `exit`: chiede al programma di terminare, e il programma lo farà.

Dobbiamo poi decidere che cosa fare nel caso in cui l'utente digiti un comando sbagliato. Ancora una volta, si potrebbe terminare il programma, ma noi segnaleremo l'errore all'utente e attenderemo il comando successivo.

Se l'insieme non fosse dinamico, si tratterebbe banalmente di mettere i numeri in un vettore e applicare al vettore la ricerca sequenziale (o magari binaria, dopo averlo ordinato). La dinamicità complica tutto, perché a priori non conosciamo la dimensione necessaria (il che impedisce l'uso di un vettore e rende problematico quello di una tabella), perché vogliamo rispondere efficientemente al test di appartenenza (il che impedisce l'uso di una lista) e perché l'insieme dei numeri interi è infinito (o, quanto meno, enorme, se ci limitiamo agli interi rappresentabili in memoria), e questo impedisce l'uso di un vettore di incidenza.

Al principio dell'esercizio è disponibile un file `ricerca0.c`, che contiene le consuete direttive fondamentali, la solita costante simbolica per la lunghezza di una riga, il programma principale con la funzione `InterpretaLineaComando` che recupera il nome del file dei dati dalla linea di comando e lo copia in un'apposita variabile.

```
/* Programma principale */
int main (int argc, char *argv[])
{
  /* Parte dichiarativa */
  char file_dati[ROW_LENGTH];


  /* Parte esecutiva */
  InterpretaLineaComando(argc,argv,file_dati);

  /* Carica i dati contenuti in file_dati in un ABR */

  /* Esegue i comandi forniti da tastiera */

  /* Distrugge l'ABR */

  return EXIT_SUCCESS;
}
```

Abbiamo quindi una traccia da svolgere in modalità *top-down*. Daremo per scontato di poter disporre di una libreria per la gestione di alberi binari di ricerca, perché la fase iniziale di modellazione del problema ci fa intuire che per gestire l'insieme dinamico di numeri l'oggetto matematico migliore è un dizionario, e perchè ipotizziamo di aver deciso che realizzeremo questo dizionario attraverso un albero binario di ricerca. La libreria sarà in effetti costituita da un file di intestazione `abr.h` già completo che fornisce una dichiarazione praticamente identica a quella dell'albero binario. 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. C'è poi un file listato `abr0.c`, che fornisce un insieme di procedure con corpi vuoti che andremo a riempire subito dopo aver strutturato il nostro algoritmo al livello più alto. Queste funzioni sono quelle elencate nella sezione precedente quando abbiamo presentato la struttura dati astratta, più una semplice procedura di stampa ordinata.

```
#define NO_TREE  NULL

typedef struct _nodo nodo;
struct _nodo
{
  int val;
  nodo *padre;
  nodo *Ts;
  nodo *Td;
};

typedef nodo *ABR;


ABR creaABR ();

void distruggeABR (ABR *pT);

boolean ABRvuoto (ABR T);
```

```
boolean memberABR (int i, ABR T);

ABR insertABR (int i, ABR T);

ABR deleteABR (int i, ABR T);

int minABR (ABR T);

int maxABR (ABR T);

void stampaABR (ABR T);
```

11.3 Prima fase (ricerca.c)

Per prima cosa, decomponiamo il problema in due fasi, risolte da altrettante funzioni. La procedura `CaricaABR` carica i numeri interi dal file indicato nella linea di comando in un albero binario di ricerca.

```
/* Carica i dati contenuti in file_dati in un ABR */
ABR CaricaABR (char *file_dati);
```

È ovviamente necessario definire una variabile `T` di tipo `ABR`, il che richiede l'inclusione della libreria `abr.h`.

La procedura `EsegueComandi` riconosce ed esegue sull'albero binario di ricerca `T` i comandi forniti dall'utente da tastiera (dunque non ci sono dati espliciti). A rigore, al termine la procedura non dovrebbe fornire risultati, ma potremmo pensare a futuri sviluppi, nei quali la funzione dovesse restituire l'insieme finale, e potremmo quindi farle restituire un albero binario di ricerca (va detto che, come già per le funzioni che operano su liste, l'albero `T` viene modificato e non occorre restituirlo esplicitamente per conservare le modifiche, per cui lo facciamo solo per avvicinarci al concetto astratto e sottolineare che `T` fa sia da dato sia da risultato.

Al termine, dovremo deallocare l'albero, come tutte le strutture dati dinamiche. Useremo l'apposita funzione di libreria, cui passeremo l'albero per indirizzo, dato che anche l'indirizzo viene modificato, azzerandolo.

```
/* Programma principale */
int main (int argc, char *argv[])
{
  /* Parte dichiarativa */
  char file_dati[ROW_LENGTH];
  ABR T;


  /* Parte esecutiva */
  InterpretaLineaComando(argc,argv,file_dati);

  /* Carica i dati contenuti in file_dati in un ABR */
  T = CaricaABR(file_dati);

  /* Esegue i comandi forniti da tastiera */
  T = EsegueComandi(T);

  /* Distrugge l'ABR */
  distruggeABR(&T);
```

```
  return EXIT_SUCCESS;
}
```

Compilando, riceviamo due messaggi di avvertimento, i quali ci ricordano che le due funzioni dovrebbero restituire qualcosa, mentre al momento non restituiscono nulla. Trattandosi di un'osservazione ovvia, la risolveremo in seguito.

Procediamo implementando le due funzioni, come sempre stendendo prima una traccia attraverso i commenti e poi il codice in dettaglio, con eventuali funzioni secondarie per sostituire le operazioni più complicate, se necessario.

La procedura `CaricaABR` si limita ad aprire il file e creare un albero binario di ricerca vuoto. Poi, ci sarà un ciclo che scorre il file finché trova numeri interi, li legge e li inserisce via via nell'albero. Infine, chiude il file e restituisce l'albero.

```
/* Carica i dati contenuti in file_dati in un ABR */
ABR CaricaABR (char *file_dati)
{
  FILE *fp;
  ABR T;

  /* Apre il file file_dati */

  /* Crea un ABR vuoto */

  /* Finche' ci sono numeri interi nel file */
  /* li inserisce nell'ABR */
  while (fscanf(fp,"%d",&i) == 1)
    T = insertABR(i,T);

  /* Chiude il file file_dati */

  return T;
}
```

L'implementazione è estremamente semplice, approfittando delle funzioni di libreria per creare un albero vuoto (`creaABR`) e per inserirvi numeri interi (`insertABR`). Per leggere numeri interi, la candidata naturale è la funzione `fscanf` con la specifica `%d`, che restituisce il numero di valori letti ogni volta, dunque 1 nel caso abbia successo. Ovviamente, occorrerà una variabile intera `i` per conservare il numero letto e inserirlo nell'albero. Per inserire il numero, chiamiamo la funzione `insertABR`

```
/* Carica i dati contenuti in file_dati in un ABR */
ABR CaricaABR (char *file_dati)
{
  FILE *fp;
  ABR T;
  int i;


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

  /* Crea un ABR vuoto */
  T = creaABR();
```

```
  /* Finche' ci sono numeri interi nel file */
  /* li inserisce nell'ABR */
  while (fscanf(fp,"%d",&i) == 1)
    T = insertABR(i,T);

  /* Chiude il file file_dati */
  fclose(fp);

  return T;
}
```

La procedura `EsegueComandi` è più sofisticata e interessante, perché realizza, diversamente da tutti gli esercizi svolti sinora, un programma che interagisce con l'utente per un tempo indefinito, leggendo, interpretando ed eseguendo i suoi comandi, finché l'utente non indica di terminare. Il programma consiste quindi in un ciclo che:

- legge da tastiera l'operazione indicata dell'utente;
- riconosce l'operazione e il suo eventuale operando;
- esegue l'operazione.

Procediamo al solito a stendere una traccia per commenti.

```
/* Esegue sull'ABR T i comandi forniti da tastiera */
ABR EsegueComandi (ABR T)
{
  boolean fine;

  fine = FALSE;
  while (!fine)
  {
    /* Stampa un prompt per segnalare che si attende un comando */

    /* Legge il comando dell'utente */

    /* Identifica il comando dell'utente */
      /* Esegue il comando dell'utente */

  return T;
}
```

Il ciclo di lettura termina quando il comando dell'utente coincide con `exit`, e quindi implica di assegnare valore vero a una variabile logica `fine`, inizialmente falsa. Il ciclo potrebbe essere a condizione iniziale o finale: comunque, l'uno è meccanicamente trasformabile nell'altro.

Interagendo con un utente umano, è una buona norma comunicargli che il programma è pronto ad eseguire: non c'è niente di più frustrante che vedere il programma fermo e non capire se lo è perché è bloccato, perché sta eseguendo l'operazione precedente o perché è in attesa di un nuovo comando. La cosa più sensata è stampare un breve segnale che indichi quest'ultima situazione. In gergo, questo segnale viene detto *prompt* e molto spesso è il carattere `>` seguito da uno spazio bianco per evitare il fastidio di scrivere attaccato al carattere stesso. Fatto ciò, si legge il comando dell'utente. Un comando è tutto ciò che l'utente batte da tastiera terminando con l'apposito tasto di invio, dunque una "riga" terminata da un a capo.

Verrebbe spontaneo usare la funzione di libreria `gets`. Da più parti si consiglia di usare una variante più sicura, la funzione `fgets`, che aggiunge alla stringa `Comando` da leggere due parametri: il primo è un numero massimo di caratteri da leggere, il secondo è il file dal quale si intende leggere. Il numero di caratteri aumenta la sicurezza della lettura perché garantisce che un eventuale comando errato di grande lunghezza venga caricato comunque in una stringa in memoria dalla funzione `gets`, eccedendo la dimensione allocata per la stringa. Con l'uso di `fgets` possiamo garantire che, se anche l'utente scrive un poema, non vengono letti più caratteri di quelli che la stringa può ospitare. Per quanto riguarda il file, la funzione nasce per la lettura da file di testo, mentre noi leggiamo da tastiera, ma sappiamo che la tastiera è interpretata dal processore come il file `stdin`.

```
fgets(Comando,ROW_LENGTH,stdin);
```

A questo punto, dobbiamo riconoscere il comando caricato. Ve ne sono diversi possibili, e alcuni consistono in un sola operatore senza operandi, mentre altri consistono in un operatore seguito da un operando intero. Descriviamo una delle molte soluzioni possibili per questo problema. Se applichiamo la funzione di *parsing* `sscanf` alla stringa con il comando, cercando di riconoscere in essa una parola e un intero, otteniamo due possibili risultati corretti: che la funzione riconosca entrambi o che riconosca solo la parola. Infine, se la funzione non riconosce nessuna delle due, abbiamo un errore. Ci si potrebbe anche chiedere perché non applicare direttamente `scanf` alla lettura da tastiera. Un possibile motivo è che, così facendo, un eventuale comando errato costituito da più di due termini verrebbe interpretato solo in parte, lasciando nel *buffer* di lettura gli oggetti dopo il secondo, come compito da eseguire nelle chiamate successive. Se questo è accettabile (magari per concatenare più comandi), va bene. Abbiamo semplicemente deciso di non accettarlo e ignorare tutto ciò che eventualmente segue i primi due oggetti nel comando.

Nella nostra implementazione, `sscanf` distingue i due casi accettabili da quello chiaramente errato, salva la prima parola in una stringa `op` e l'eventuale numero intero in una variabile `i`

```
cont = sscanf(Comando,"%s %d",op,&i);
```

col consueto passaggio per indirizzo dei caratteri della stringa e del numero intero, dove la stringa non ha referenziazione perché è un vettore, e dunque già un puntatore.

Secondo il numero di oggetti riconosciuti passiamo a riconoscere il comando racchiuso nella stringa `op`. Per farlo, la confrontiamo con le varie possibilità attraverso la funzione `strcmp` e in ciascun caso corretto eseguiamo l'istruzione, eventualmente sull'argomento numerico. Sono tutte operazioni elementari per le quali esiste una funzione di libreria. In alcuni casi avremo da fare delle semplici stampe già descritte più sopra. Ovviamente, in caso di mancato riconoscimento segnaliamo la cosa all'utente. Il comando `exit` fa eccezione in quanto non comporta operazioni legate all'albero, ma l'assegnamento del valore `TRUE` alla variabile `fine`.

```
/* Esegue sull'ABR T i comandi forniti da tastiera */
ABR EsegueComandi (ABR T)
{
  boolean fine;
  char Comando[ROW_LENGTH];
  char op[ROW_LENGTH];
```

```
  int i;
  int cont;


  fine = FALSE;
  while (!fine)
  {
    /* Stampa un prompt per segnalare che si attende un comando */
    printf("> ");

    /* Legge il comando dell'utente */
    fgets(Comando,ROW_LENGTH,stdin);

    /* Identifica il comando dell'utente */
    cont = sscanf(Comando,"%s %d",op,&i);
    if (cont == 2)
    {
      /* Esegue il comando dell'utente */
      if (strcmp(op,"member") == 0)
      {
        if (memberABR(i,T))
          printf("%d in T\n",i);
        else
          printf("%d non in T\n",i);
      }
      else if (strcmp(op,"insert") == 0)
        T = insertABR(i,T);
      else if (strcmp(op,"delete") == 0)
        T = deleteABR(i,T);
      else
        printf("Comando non riconosciuto!\n");
    }
    else if (cont == 1)
    {
      /* Esegue il comando dell'utente */
      if (strcmp(op,"print") == 0)
      {
        stampaABR(T);
        printf("\n");
      }
      else if (strcmp(op,"empty") == 0)
      {
        if (ABRvuoto(T))
          printf("T e' vuoto\n");
        else
          printf("T non e' vuoto\n");
      }
      else if (strcmp(op,"min") == 0)
        printf("min(T) = %d\n",minABR(T));
      else if (strcmp(op,"max") == 0)
        printf("max(T) = %d\n",maxABR(T));
      else if (strcmp(op,"exit") == 0)
        fine = TRUE;
      else
        printf("Comando non riconosciuto!\n");
    }
    else
      printf("Comando non riconosciuto!\n");
  }

  return T;
}
```

Compilato il programma per valutarne la correttezza sintattica, otteniamo alcuni avvertimenti legati al fatto che le funzioni di libreria per la gestione degli alberi

di ricerca binaria sono tutte vuote, mentre alcune dovrebbero restituire dei valori. Lanciando il programma, non otteniamo risultati, ma possiamo verificare che i comandi errati (o i comandi corretti con operandi errati) vengono riconosciuti e segnalati, e che il comando di uscita fa effettivamente terminare il programma.

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 ...`).

11.3.1 Seconda fase (`abr1.c`)

A questo punto, come abbiamo sempre fatto, cominciamo a riempire le funzioni vuote della libreria di gestione della nuova struttura dati. In questo caso, fra l'altro, la cosa è coerente anche con l'approccio *top-down*, dato che il programma era molto semplice e le procedure ancora non realizzate sono effettivamente quelle di livello più basso.

Le prime tre funzioni non riguardano il contenuto informativo, ma solo la topologia dell'albero:

- `creaABR`: crea un albero vuoto;
- `distruggeABR`: distrugge un albero binario di ricerca;
- `ABRvuoto`: restituisce `TRUE` se l'albero è vuoto, `FALSE` altrimenti.

Siccome la topologia dell'albero binario di ricerca è identica a quella dell'albero binario, non stupisce che le tre funzioni siano identiche alle corrispondenti funzioni dell'albero binario. L'unica differenza riguarda la distruzione, che non può sfruttare una chiamata alla cancellazione del sottoalbero, perché tale funzione non esisterà. Tuttavia, il codice coincide con quello della chiamata stessa: è solo riportato esplicitamente.

```
ABR creaABR ()
{
  return NO_TREE;
}


void distruggeABR (ABR *pT)
{
  if (*pT != NO_TREE)
  {
    distruggeABR(&(*pT)->Ts);
    distruggeABR(&(*pT)->Td);
    free(*pT);
    *pT = NO_TREE;
  }
}


boolean ABRvuoto (ABR T)
{
  return (T == NO_TREE);
}
```

[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`.

Anche la loro complessità sarà la stessa: costante per la creazione e il test di vuotezza, lineare per la distruzione, che è una visita in postordine.

L'operazione di inserimento, invece, è particolarmente utile, perché consente di operare su alberi non vuoti. Quindi, ci consentirà anche di valutare se le altre operazioni sono state realizzate correttamente. Non ha molto senso passare oltre e implementare altre operazioni non verificabili. Affrontiamo quindi la procedura `insertABR` che restituisce l'albero ottenuto aggiungendo un elemento all'albero dato.

Per l'inserimento di un nuovo elemento in un albero binario di ricerca non possiamo ispirarci all'albero binario, perché l'operazione non esisteva o (se vogliamo) era possibile solo a patto di mettere il nuovo elemento come radice combinando due sottoalberi binari esistenti. Noi non abbiamo sottoalberi pronti e comunque non possiamo decidere di mettere il nuovo elemento nella radice, perché questo potrebbe violare la proprietà fondamentale di ordinamento dell'albero. La funzione `insertABR`, come tutte quelle che tengono conto del contenuto informativo dell'albero, sono abbastanza diverse da quelle dell'albero binario. In particolare, non indicano la posizione degli elementi in modo esplicito, come negli alberi binari, percé in un albero binario di ricerca la posizione di un elemento è assegnata automaticamente, non decisa dall'utente. Nonostante ciò, vedremo che la procedura di costruzione viene in realtà buona, ma in modo molto più sofisticato.

Torniamo a considerare l'esempio della Figura 11.5. Come inserire in questo albero binario di ricerca il valore 5, oppure 10? A meno che si voglia ristrutturare l'intero albero, questo valore non può andare in radice.

L'unico possibile inserimento in un albero così costituito è appendere il nuovo elemento a uno dei nodi che ha meno di due figli. In effetti, c'è un solo nodo a cui il nuovo elemento può essere appeso e un solo modo di farlo: come figlio di destra del nodo di valore 4. Questo perché tale albero, attualmente vuoto è l'unico che può contenere elementi di valore maggiore di 4 e minore di 6. Questa proprietà è generale, cioè qualsiasi elemento non ancora contenuto nell'albero ammette una sola posizione valida, se non si vogliono modificare le etichette e i collegamenti dei nodi attuali.

Trovare questa posizione è facile: si tratta di accedere alla radice dell'albero e confrontare il valore ad essa associato con quello del nuovo elemento. Se coincidono, non c'è nulla da fare, e si può terminare. Se il nuovo elemento precede quello in radice, il suo posto corretto (ammesso che esista) non può essere nel sottoalbero destro, per cui bisogna cercarlo in quello sinistro. Infine, se il nuovo elemento segue quello in radice, il suo posto corretto (ammesso che esista) deve stare nel sottoalbero destro. Questo meccanismo non vale solo per la radice, ma per tutti i nodi dell'albero. Quindi, il procedimento va ripetuto scendendo via via fino a che uno dei nodi visitati ha valore pari al nuovo elemento o fino a che il sottoalbero in cui si sta cercando è vuoto. In questo secondo caso, è chiaro che l'elemento è effettivamente nuovo e un nodo che lo contiene può diventare il sottoalbero mancante. Il procedimento combina chiaramente un meccanismo ricorsivo, che non visita l'intero albero, ma solo un cammino dalla radice verso il basso, con la creazione di un nuovo sottoalbero costituito da un solo nodo, senza sottoalberi.

Premessa Prima di procedere a implementare questa funzione, è opportuna una premessa. Nella stesura del codice per le librerie di gestione delle strutture dati di questo corso, si cerca sempre di adeguarsi il più possibile alla notazione delle dispense del modulo di teoria. Questo non è sempre possibile per motivi tecnici. Per esempio, abbiamo visto nel caso delle liste che siccome le dispense di teoria adottano un linguaggio di programmazione in cui i parametri vengono passati per

indirizzo, mentre in C il passaggio avviene per valore. Ne risulta che, quando un elemento viene inserito in una data posizione di una data lista, all'uscita dalla procedura il puntatore alla posizione potrebbe puntare l'elemento nuovo oppure quello vecchio, secondo come la procedura viene implementata. Lo stesso avviene nel caso della cancellazione (si veda la Sezione **??**. È estremamente difficile mantenere una perfetta coerenza, e quindi alcune implementazioni differiscono da quelle descritte nelle dispense di teoria. Questa succede in modo particolare per gli alberi binari di ricerca.

Infatti, alla questione del passaggio di parametri si aggiungono due questioni:

- le dispense di teoria ipotizzano quasi sempre di operare su alberi non vuoti, mentre qui consideriamo il caso generale;
- le dispense di teoria ipotizzano che gli elementi da aggiungere non siano già presenti nell'albero e che quelli da cancellare siano invece presenti, mentre qui consideriamo anche i casi rimanenti.

Queste due ipotesi semplificative sarebbero gestibili richiedendo all'utente di eseguire prima dell'operazione di inserimento o cancellazione una ricerca e di evitare l'operazione se l'esito della ricerca la rende superflua. Noi invece realizzeremo un codice leggermente più sofisticato, che affronta tutti i casi possibili e non richiede tale valutazione preliminare. Si tenga però presente la differenza nel preparare l'esame orale.

Inoltre,

- per semplificare alcuni passaggi, sostituiremo le funzioni `CREA_NODO_SIN` e `CREA_NODO_DES`, usate nelle dispense di teoria, con una funzione `costruisceABR`[3], che coincide con la funzione `costruiscealbero` descritta nella lezione precedente.

Realizziamo ora la funzione `insertABR`, cominciando con la consueta traccia. Ovviamente, si tratterà di una funzione ricorsiva. Partiamo con il caso base, nel quale l'albero esplorato è vuoto. Questo succederà ogni volta che l'elemento non fa già parte dell'albero, dato che in tal caso scorreremo l'albero fino a imbatterci in un sottoalbero vuoto che sta nella posizione corretta. Quando l'albero corrente è vuoto, dobbiamo costruire un nuovo nodo contenente il nuovo elemento e sostituirlo all'albero vuoto. Quindi, sostituiremo il valore corrente dell'albero `t` con il valore restituito da una funzione che, dato l'elemento nuovo `i`, costruisce un nodo che contiene `i` come informazione e ha due sottoalberi vuoti. Rimandiamo a dopo la scrittura della procedura `creanodoABR`. Nel caso ricorsivo, può succedere di dover esplorare il sottoalbero sinistro o destro, ma anche di scoprire che il nuovo elemento già è presente nella radice del sottoalbero corrente con il nome `T->val`. Quindi, esiste un secondo caso base, che però comporta di non fare nulla, per cui non viene esplicitamente riportato. Passando al caso ricorsivo, ce ne sono due: quando l'elemento nuovo è inferiore a quello in radice (`i < T->val`), andremo ad aggiungere il primo al sottoalbero sinistro, chiamando ricorsivamente `insertABR`; quando `i > T->val`, lo aggiungeremo al sottoalbero destro, sempre con una chiamata ricorsiva. Si noti che le due condizioni vanno riportate entrambe, perché limitarsi a un costrutto `else` includerebbe nel secondo caso ricorsivo il caso base in cui l'elemento nuovo coincide con quello in radice.

Apparentemente, tanto basta. In realtà, non basta, perché le operazioni descritte calano lungo l'albero nella direzione corretta, fino a raggiungere un albero vuoto.

[3] O `creanodoABR`? Controllare l'ultima versione.

A quel punto, creano un nuovo nodo contenente il nuovo elemento e assegnano tale nodo al sottoalbero sinistro `T->Ts` o destro `T->Td`, secondo i casi, dell'ultimo nodo visitato nell'albero dato. Manca un dettaglio: il nuovo nodo ha un puntatore al padre che va ancora indirizzato correttamente, cosa che la chiamata ricorsiva non fa. L'operazione va eseguita esplicitamente all'uscita dalla chiamata ricorsiva. Dunque il nuovo nodo deve avere come padre quello dato, cioè `T`. Perché non modificare il campo padre nella stessa procedura di creazione, anziché fuori? Per tenersi il più possibile vicino alle dispense di teoria.

Si osservi come questa operazione è necessaria solo nel nuovo nodo aggiunto, ma la struttura ricorsiva fa sì che venga eseguita in ciascun nodo visitato lungo il cammino che cala dalla radice al nodo aggiunto, confermando ogni volta il legame di ciascun nodo con il nodo padre.

La procedura `creanodoABR` prende in ingresso un numero intero e restituisce un albero binario di ricerca. Si è già osservato che è molto simile alla costruzione di un albero binario, salvo che parte da due sottoalberi vuoti, per cui non occorre riportarli come argomenti. Per il resto si tratta di allocare un nodo, riempirne opportunamente i campi informativi e i puntatori.

```
ABR costruisceABR (int i, ABR Ts, ABR Td)
{
  ABR T;

  T = (nodo *) malloc(sizeof(nodo));
  if (T == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione di un ABR!\n");
    exit(EXIT_FAILURE);
  }

  T->val = i;
  T->padre = NULL;
  T->Ts = Ts;
  if (Ts != NULL) Ts->padre = T;
  T->Td = Td;
  if (Td != NULL) Td->padre = T;

  return T;
}
```

È importante osservare che questa funzione va inserita nel file listato della libreria, `abr3.c`, ma non occorre che compaia nel file di intestazione, perché viene usata internamente e non è utile per un utente della struttura dati astratta (anzi, vorremmo evitare che un utente manipoli direttamente la struttura topologica dell'albero aggiungendo nodi: meglio che ne conservi un'idea astratta. L'unica accortezza che va mantenuta è che la procedura `creanodoABR` preceda `insertABR`, perché altrimenti il compilatore trova un nome di funzione sconosciuto e lo segnala come errore.

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'albero binario di ricerca, $\Theta(h)$. L'intera costruzione di un albero 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.

11.3.2 Terza fase (abr2.c): stampa, appartenenza, minimo e massimo

A questo punto, è urgente avere una procedura di stampa, per valutare se quanto abbiamo fatto ha senso. Per stampare gli elementi di un albero in modo ordinato, bisogna andare su e giù per l'albero apparentemente in modo confuso. In realtà, la strategia è tutt'altro che confusa, dato che si basa sulla proprietà fondamentale degli alberi binari di ricerca: che l'informazione associata ad ogni nodo sia strettamente successiva a quelle dei nodi del sottoalbero sinistro e strettamente precedente quelle dei nodi del sottoalbero destro. Corrispondentemente, per stampare l'intero albero bisognerà stampare prima il sottoalbero sinistro, poi il nodo radice e infine il sottoalbero destro. Questo vale ricorsivamente per ogni sottoalbero. Quindi, la stampa è una visita in in-ordine, un po' come la stampa delle espressioni aritmetiche in notazione infissa. Ovviamente, il caso base di albero vuoto corrisponde a non fare nulla (si potrebbe pensare di stampare un puntino o uno spazio, ma ce ne pentiremmo immediatamente dato che tutti i sottoalberi vuoti incontrati durante la visita di un albero non vuoto verrebbero trattati allo stesso modo, producendo un largo numero di stampe indesiderate). Nel complesso, la realizzazione è banale e la funzione è ovviamente lineare nel numero degli elementi ($\Theta(n)$), come nel caso degli alberi binari.

```
void stampaABR (ABR T)
{
  if (T != NO_TREE)
  {
    stampaABR(T->Ts);
    printf(" %d",T->val);
    stampaABR(T->Td);
  }
}
```

Realizzata la stampa, possiamo verificare il buon funzionamento anche dell'inserimento, dato che possiamo inserire valori nuovi e ristampare il tutto.

Procediamo con le funzioni di interrogazione dell'albero, partendo dal test di appartenenza `memberABR`, che ha dato origine all'intera impresa di definire un dizionario astratto. La procedura di ricerca sarà ancora una volta ricorsiva, al primo tentativo. Se l'albero è vuoto, la ricerca ovviamente fallisce e il risultato della procedura è `FALSE`. Altrimenti, l'idea è confrontare il numero `i` cercato con il valore conservato nella radice dell'albero. Se coincidono, abbiamo trovato il numero e possiamo restituire `TRUE`. Se invece `i` è strettamente minore, non può trovarsi nel sottoalbero destro, e dobbiamo quindi cercarlo in quello sinistro, con una chiamata ricorsiva alla stessa procedura. Infine, se `i` è strettamente maggiore del valore in radice, bisogna cercarlo nel sottoalbero destro. La struttura è semplice, con due casi base e due ricorsivi, come più volte è successo in precedenza.

```
boolean memberABR (int i, ABR T)
{
  if (T == NO_TREE)
    return FALSE;
  else if (i == T->val)
    return TRUE;
  else if (i < T->val)
    return memberABR(i,T->Ts);
  else /* if (i > T->val) */
    return memberABR(i,T->Td);
```

Notiamo che la ricorsione è terminale, perché compare addirittura nell'operazione di restituzione del risultato alla funzione chiamante. Quindi, la procedura si può trasformare meccanicamente in iterativa. Questa è tendenzialmente preferibile, perché evita tutto il lavoro ridondante di gestione dello *stack*: il valore di `i` viene copiato sempre uguale ad ogni chiamata, mentre quello di `T` è semplicemente un puntatore al nodo (radice del sottoalbero) visitato via via. Anche il risultato è sempre `FALSE` finché l'elemento non viene effettivamente trovato e allora (se succede) rimane sempre uguale a `TRUE`. La maggior parte di queste operazioni è inutile. Questa volta, la trasformazione è talmente diretta che merita descriverla. Un algoritmo iterativo non farebbe che confrontare il valore cercato con il nodo radice dell'albero corrente e poi spostarsi nel sottoalbero sinistro o destro, secondo il caso, finché il valore non viene trovato o finché non si arriva a un albero vuoto. Nel primo caso, restituirebbe `TRUE`, nel secondo `FALSE`.

```
boolean memberABR (int i, ABR T)
{
  while ( (T != NO_TREE) && (i != T->val) )
    if (i < T->val)
      T = T->Ts;
    else /* if (i > T->val) */
      T = T->Td;

  return (T != NO_TREE);
```

Notiamo che la condizione del ciclo, essendo di permanenza, è esattamente complementare ai due casi base che terminano la ricorsione. Il passaggio dei parametri viene sostituito invece da un aggiornamento della variabile cursore del ciclo. Usare direttamente il dato `T` come cursore può sembrare azzardato, ma dobbiamo ricordarci che i parametri sono passati per valore, per cui il puntatore `T` è solo una copia dell'albero passato dalla funzione chiamante. Il ciclo rimpiazza una sequenza di allocazioni di record di attivazione sullo *stack*, seguite da altrettante deallocazioni, con l'aggiornamento di un cursore. Al termine, in base al motivo per cui siamo usciti dal ciclo restituiamo l'esito della ricerca. Volendo compattare, possiamo anche restituire la condizione logica di albero non vuoto, che è equivalente (se non addirittura, restituire l'albero, che in base alle regole di interpretazione dei valori logici equivale a falso quando è vuoto e vero quando non lo è; sarebbe però una scelta poco leggibile).

Compilazione condizionale Il codice seguente combina le due soluzioni, consentendo all'utente di scegliere quella che preferisce al momento della compilazione (non dell'esecuzione!). Infatti, la direttiva di precompilazione `#ifdef`, seguita da una costante simbolica, da un qualsiasi primo brano di codice, dalla direttiva `#else`, da un secondo brano di codice e dalla direttiva `#endif`, indica al precompilatore di verificare se la costante simbolica è stata definita in precedenza da una direttiva `#define`. Si noti che la definizione non deve necessariamente assegnare un valore alla costante: conta solo il fatto che essa sia stata definita oppure no. In caso positivo, il primo brano di codice viene mantenuto e il secondo cancellato. Nell'altro caso, viene cancellato il primo brano e mantenuto il secondo. Per sincerarsene basta eseguire la precompilazione con l'opzione `-E` e osservare la differenza tra i risultati ottenuti aggiungendo e togliendo la definizione. Dunque, il codice viene compilato in maniera condizionata dalla presenza o meno di una definizione. La direttiva `#ifndef` funziona alla stessa maniera, ma attiva solo il primo brano di codice quando la costante non è definita e solo il secondo quando lo è.

```
boolean memberABR (int i, ABR T)
{
#ifdef RICORSIVA
  /* Implementazione ricorsiva */
  if (T == NO_TREE)
    return FALSE;
  else if (i == T->val)
    return TRUE;
  else if (i < T->val)
    return memberABR(i,T->Ts);
  else /* if (i > T->val) */
    return memberABR(i,T->Td);
#else
  /* Implementazione iterativa */
  while ( (T != NO_TREE) && (i != T->val) )
    if (i < T->val)
      T = T->Ts;
    else /* if (i > T->val) */
      T = T->Td;

  return (T != NO_TREE);
#endif
}
```

Questo consente di generare facilmente con una modifica in un solo punto codici anche molto diversi. La cosa viene usata in genere per scrivere codice che contiene istruzioni dipendenti dal sistema operativo: le istruzioni valide per ciascun sistema vengono scritte tutte, e la scelta di quale usare viene fatta definendo o no una o più costanti simboliche prima di compilare il codice.

Possiamo passare all'implementazione delle funzioni che determinano gli elementi di valore minimo e massimo. Si tratta di funzioni molto simili fra loro e abbastanza simili al test di appartenenza. Ancora una volta, sono ricorsive. Consideriamo per esempio `minABR`. Quando l'albero è vuoto, si è già osservato che occorre definire un valore fittizio, pari al massimo intero rappresentabile (`INT_MAX`), che la funzione restituirà in uscita. Quando non è vuoto, si tratta di capire se il valore in radice è minimo per l'intero albero oppure no. Il primo caso corrisponde alla non esistenza di valori più bassi, cioè al fatto che il sottoalbero sinistro sia vuoto. È un secondo caso base, come più volte accaduto in questo esercizio. Quando invece il sottoalbero sinistro non è vuoto, il minimo dell'intero albero coincide col minimo di tale sottoalbero, e quindi si trova con una chiamata ricorsiva su di esso.

```
int minABR (ABR T)
{
  if (T == NO_TREE)
    return INT_MAX;
  else if (T->Ts == NO_TREE)
    return T->val;
  else
    return minABR(T->Ts);
}
```

La chiamata ricorsiva è ancora una volta terminale, e si può facilmente trasformare la funzione in iterativa: per trovare il minimo di un albero, basta scendere il più possibile verso sinistra, fino a trovare un nodo che non ha sottoalbero di sinistra, e quindi non ha altri numeri strettamente minori. Questo è per definizione il minimo. Come si può facilmente immaginare, la versione iterativa comincia valutando

se l'albero sia vuoto e restituendo il valore fittizio nel caso lo sia. Altrimenti, si ha un ciclo che scende lungo l'albero sostituendo il valore corrente di `T` col sottoalbero sinistro, finché ne esiste uno non vuoto. Contrariamente al test di appartenenza, qui la condizione di permanenza considera solo uno dei due casi base (sottoalbero sinistro vuoto), perché l'altro (albero vuoto) non è mai verificato tranne prima che il ciclo cominci.

```
int minABR (ABR T)
{
  /* Implementazione iterativa */
  if (T == NO_TREE) return INT_MAX;
  while (T->Ts != NO_TREE)
    T = T->Ts;

  return T->val;
}
```

La funzione `maxABR` si ottiene banalmente sostituendo `INT_MAX` con `INT_MIN`, il sottoalbero sinistro col destro e `min` con `max`.

```
int maxABR (ABR T)
{
  /* Implementazione ricorsiva */
  if (T == NO_TREE)
    return INT_MIN;
  else if (T->Td == NO_TREE)
    return T->val;
  else
    return maxABR(T->Td);
}
```

E lo stesso vale per l'implementazione iterativa

```
int maxABR (ABR T)
{
  /* Implementazione iterativa */
  if (T == NO_TREE) return INT_MIN;
  while (T->Td != NO_TREE)
    T = T->Td;

  return T->val;
}
```

Per entrambe le funzioni, ed entrambe le implementazioni, la complessità è proporzionale al numero di chiamate ricorsive ovvero iterazioni, cioè alla profondità h dell'albero, dunque $\Theta(h)$.

11.3.3 Quarta fase (`abr3.c`): cancellazione di elementi

Rimane da realizzare la procedura che cancella un elemento dall'albero binario di ricerca. È la procedura più complicata: presenta diversi casi particolari e diversi passaggi. Anche nel trattare gli alberi binari, la cancellazione aveva posto il problema di come conservare la struttura gerarchica con due figli per nodo: cancellato un nodo, non è assolutamente ovvio come conservare la proprietà che ogni nodo

abbia due figli. Infatti, avevamo rinunciato a cancellare nodi singoli, limitandoci a cancellare interi sottoalberi. Qui non possiamo farlo, perché dobbiamo gestire un insieme di elementi ordinati e vogliamo effettivamente poter cancellare un elemento singolo. Inoltre, la posizione dell'elemento non è sotto il nostro controllo, perché la stabilisce l'ordine. Dovremo quindi realizzare una soluzione più sofisticata.

Cominciamo con la struttura generale, che sarà ricorsiva. L'idea fondamentale sarà confrontare l'elemento da cancellare con quello contenuto nella radice dell'albero corrente, ottenendo i soliti due casi ricorsivi. Se l'elemento da cancellare risulta più piccolo di quello in radice, allora l'operazione di cancellazione andrà eseguita nel sottoalbero sinistro; se risulta più grande, nel sottoalbero destro. Ovviamente, il caso base di albero vuoto è banale, e si risolve non facendo nulla. Più complicato è il caso base in cui l'elemento da cancellare coincide con la radice corrente. Infatti, questo caso si divide in tre possibili sottocasi.

Per prima cosa, osserviamo che alcuni nodi sono cancellabili senza problemi: sono le foglie. Basta deallocarle e azzerare il puntatore che le individua nel nodo padre. Poi ci sono nodi cancellabili con qualche difficoltà in più, cioè i nodi interni con un solo figlio. Tali nodi sono in una situazione simile a quella di una lista bidirezionale: il nodo ha un puntatore al figlio e uno al padre, e a loro volta il padre e il figlio hanno dei puntatori opportuni al nodo stesso. L'albero, infatti, è percorribile in entrambe le direzioni (verso l'alto e verso il basso). Per questi nodi, la soluzione è di collegare direttamente il nodo padre al singolo nodo figlio. La cosa fondamentale è che la proprietà di ordinamento non verrebbe violata da questo collegamento diretto, dato che l'intero sottoalbero radicato nel nodo figlio rimane dalla parte giusta del nodo padre. Il caso complicato è quello dei nodi con due figli. Anche questo caso, però, ha una analogia con una struttura che conosciamo, cioè la tabella. Per cancellare in tempo costante un elemento da una tabella, è sufficiente sovrascriverlo con l'ultimo elemento della tabella stessa e cancellando quest'ultimo, dato che è l'unico elemento che, scomparendo, non distrugge la struttura della tabella (si tratta solo di ridurne la dimensione). D'altra parte, nel caso dell'albero binario di ricerca non è chiaro a prima vista quale elemento si possa effettivamente usare allo scopo, e se ve ne sia sempre uno lecito. L'ordinamento, infatti, è un vincolo molto restrittivo. Fortunatamente, esistono sempre due nodi che possono essere usati per sovrascrivere quello da cancellare, ed essere poi cancellati a loro volta. Si tratta semplicemente del nodo precedente e del nodo successivo nell'ordinamento. Ogni altro nodo, se copiato al posto di quello da cancellare e poi cancellato, violerebbe l'ordinamento complessivo.

Come trovare l'elemento che precede immediatamente un elemento dato? La risposta è in realtà semplice: sta nel sottoalbero sinistro, ed è il massimo di tale sottoalbero. Per definizione, infatti, l'elemento precedente è il massimo fra quelli strettamente minori. Ma noi abbiamo appena realizzato una funzione per trovare il valore massimo in un albero, e non è difficile adattarla in modo che restituisca l'elemento massimo, anziché il valore. Potrebbe esserci una difficoltà: il nodo che vogliamo usare per sostituire quello cancellato potrebbe a sua volta essere difficile da cancellare. Ciò non avviene, però, perché, essendo l'elemento massimo di un sottoalbero, ha a sua volta un sottoalbero destro vuoto, e quindi ha al massimo un figlio, cioè ricade in uno dei due casi base semplici. Nell'esempio di Figura 11.5, per cancellare il valore 6 nella radice, basta sovrascriverlo con il 4 e cancellare quest'ultimo, che è un nodo interno con un solo figlio.

Analogamente, l'elemento che segue immediatamente quello da cancellare è il minimo del sottoalbero sinstro, e si trova in modo analogo. Nell'esempio, si può sovrascrivere il 6 con l'8, che è una foglia. La scelta fra i due è arbitraria: il nodo da cancellare ha entrambi i sottoalberi, altrimenti saremmo in uno dei casi base semplici. Arbitrariamente, discuteremo il primo approccio.

Gli elementi della procedura saranno quindi due semplici casi ricorsivi, basati sul confronto fra elemento da cancellare ed elemento nella radice dell'albero corrente, e quattro casi base, di cui uno (albero vuoto) banale, due semplici (foglia e nodo interno con un figlio) riducibili entrambi alla chiamata a una funzione ausiliaria con qualche tecnicalità, e uno (nodo interno con due figli) più complesso. Chiameremo `toglienodoABR` la funzione che elimina il nodo nel caso semplice, a cui passeremo l'indirizzo del nodo e l'albero, ricevendone l'albero modificato.

```
/* Appende direttamente l'unico sottoalbero del nodo n dell'ABR T
   al nodo padre e dealloca il nodo n */
ABR toglienodoABR (nodo *n, ABR T)
```

Il caso complesso richiederà una funzione `argmaxABR` che, dato un albero, determini non il valore massimo dell'albero, ma l'indirizzo del nodo in cui questo è conservato.

```
/* Determina il nodo che corrisponde al valore massimo dell'ABR T */
nodo *argmaxABR (ABR T)
```

Questa funzione, insieme alla sovrascrittura dell'elemento da cancellare con quello appena determinato e alla cancellazione del nodo determinato costituiranno la soluzione dell'ultimo caso base.

```
ABR deleteABR (int i, ABR T)
{
  nodo *n;

  /* Se l'albero e' vuoto, non fa nulla */
  if (T != NO_TREE)
  {
    if (i < T->val)
      T->Ts = deleteABR(i,T->Ts);
    else if (i > T->val)
      T->Td = deleteABR(i,T->Td);
    else if ( (T->Ts == NO_TREE) || (T->Td == NO_TREE) )
      /* Almeno un sottoalbero e' vuoto: cancella il nodo e appende
         il sottoalbero al padre */
      T = toglienodoABR(T,T);
    else
    {
      /* Ci sono due sottoalberi: sovrascrive il nodo con l'elemento
         massimo del sottoalbero sinistro
         e cancella quest'ultimo */
      n = argmaxABR(T->Ts);
      T->val = n->val;
      T = toglienodoABR(n,T);
    }
  }

  return T;
}
```

Riassumendo, se l'albero è vuoto non si fa nulla, per cui nel codice compare solo la condizione complementare. Contrariamente al solito scriviamo prima i due casi ricorsivi di quelli base, in omaggio a una regola di stile (arbitraria come tutte le regole di stile) per cui prima si riportano i casi facili e poi quelli difficili in modo da

aiutare la lettura. Inoltre, questa sequenza consente di semplificare la condizione dei casi base, dato che, esclusi i casi ricorsivi, è chiaro che l'elemento cercato sta nella radice dell'albero corrente. Secondo che l'elemento cercato sia minore o maggiore di quello in radice, si chiama ricorsivamente la funzione con lo stesso argomento numerico sul sottoalbero sinistro o destro, ricevendone il sottoalbero modificato. Risolti questi due casi, rimangono quelli base, in cui l'elemento cercato coincide con quello in radice. Il più semplice richiede che almeno uno dei sottoalberi sia vuoto. In tal caso, si chiama la funzione ausiliaria `toglienodoABR`, passandole il nodo radice e l'albero. Poiché con l'implementazione a puntatori la radice ha lo stesso indirizzo dell'albero, ne risulta una strana chiamata con due argomenti identici. Infine, il caso di elemento con due sottoalberi non vuoti richiede di chiamare la funzione ausiliaria `argmaxABR` per determinare un nodo `n`, l'assegnamento del valore conservato in `n` a quello conservato nella radice e la cancellazione del nodo `n` dall'albero.

Rimangono a questo punto da realizzare le due funzioni ausiliarie. La funzione `argmaxABR` è la più semplice, perché somiglia moltissimo alla funzione `maxABR`, di cui consideriamo direttamente la forma iterativa. È sufficiente copiare quest'ultima e modificare il valore restituito nei due punti di uscita: `NULL` anziché `INT_MAX` e `T` anziché `T->val`.

```
/* Determina il nodo che corrisponde al valore massimo dell'ABR T */
nodo *argmaxABR (ABR T)
{
  if (T == NO_TREE) return NO_TREE;
  while (T->Td != NO_TREE)
    T = T->Td;
  return T;
}
```

La funzione `toglienodoABR` è molto simile alla cancellazione di un elemento da una lista bidirezionale. Presenta però diverse complicazioni, legata al fatto che la catena di nodi può procedere in due direzioni (sinistra o destra) anziché una sola e al fatto che il nodo potrebbe non avere un nodo padre, cosa che le liste invece garantivano attraverso l'introduzione di una sentinella. L'assenza di una sentinella che faccia da successore complica anche il caso in cui il nodo da cancellare non ha nodi figli. Questo determina una serie di casi da trattare separatamente senza dimenticarne alcuno. Stenderemo quindi prima una traccia.

```
/* Appende direttamente l'unico sottoalbero del nodo n dell'ABR T
   al nodo padre e dealloca il nodo n */
ABR toglienodoABR (nodo *n, ABR T)
{
  ABR Tf;


  /* Determina l'unico sottoalbero non vuoto del nodo n
     (eventualmente NO_TREE) */

  /* Se esiste un sottoalbero, lo appende al padre (eventualmente
     NULL) */

  /* Se n e' la radice, sostituisce l'albero T con il sottoalbero */
  else /* Se n ha un padre, determina in quale sottoalbero del padre
     stava
         e sostituisce il sottoalbero a se stesso */

  /* Dealloca il nodo */

  return T;
```

```
}
```

Per prima cosa dobbiamo determinare il sottoalbero figlio `Tf` da sollevare di un livello, dando per scontato che la funzione chiamante abbia escluso che ve ne siano due. Tale sottoalbero può anche mancare, nel qual caso viene indicato con un albero vuoto. Quindi, dovremo appendere `Tf` al nodo padre, tenendo conto del fatto che ciascuno dei due oggetti può non esistere. Ciò fatto, dovremo garantire che il nodo padre punti correttamente il nuovo sottoalbero `Tf`, dunque aggiornare il sottoalbero sinistro o destro del nodo padre. Qualora però il nodo padre non esistesse, cioè il nodo corrente fosse la radice dell'albero, non dovremmo fare nulla del genere[4]. D'altra parte, in tale caso è l'albero nel suo complesso a cambiare, perché la sua radice deve diventare quella del sottoalbero figlio. Per ottenere questo risultato, o passiamo l'albero `T` per indirizzo oppure (come abbiamo scelto), restituiamo al termine della procedura un valore aggiornato di `T`. Tutto questo funziona anche qualora il sottoalbero `Tf` fosse vuoto. Prima di restituire l'albero, eventualmente modificato, deallochiamo il nodo `n`, che è finalmente del tutto scollegato dal resto dell'albero.

```
/* Appende direttamente l'unico sottoalbero del nodo n dell'ABR T
   al nodo padre e dealloca il nodo n */
ABR toglienodoABR (nodo *n, ABR T)
{
  ABR Tf;


  /* Determina l'unico sottoalbero non vuoto del nodo n
     (eventualmente NO_TREE) */
  Tf = n->Ts;
  if (Tf == NO_TREE) Tf = n->Td;

  /* Se esiste un sottoalbero, lo appende al padre (eventualmente
     NULL) */
  if (Tf != NO_TREE) Tf->padre = n->padre;

  /* Se n e' la radice, sostituisce l'albero T con il sottoalbero */
  if (n == T)
    T = Tf;
  else /* Se n ha un padre, determina in quale sottoalbero del padre
     stava
          e sostituisce il sottoalbero a se stesso */
  {
    if (n == n->padre->Ts)
      n->padre->Ts = Tf;
    else if (n == n->padre->Td)
      n->padre->Td = Tf;
  }

  /* Dealloca il nodo */
  free(n);

  return T;
```

[4]Per motivi che in questo momento non mi sono abbastanza chiari da commentarli in modo utile, il test deve valutare che se `n` sia o no la radice dell'albero corrente, che potrebbe non essere quella dell'albero complessivo su cui si sta ragionando, per cui non è corretto testare semplicemente se esista o no un nodo padre. Il punto però è che in `deleteABR` si aggiorna anche il puntatore del nodo padre. Ora ho il dubbio che non occorra farlo qui. Può darsi che non occorra perché chiameremo questa funzione sempre sulla radice dell'albero corrente? Può essere che `deleteABR` implicitamente presupponga l'implementazione a puntatori, perché lavora su sottoalberi e non ha riferimenti all'albero complessivo? Domande basilari, ma superflue finché si resta nei limiti del corso.

```
}
```

Entrambe le funzioni non compaiono nel file di intestazione `abr.h` perché non è verosimile che un utente esterno le usi per operare sull'albero e perché presuppongono esplicitamente un'implementazione a puntatori, dato che hanno un argomento o un risultato di tipo `nodo *`. Affinché possano essere chiamata dalla funzione `deleteABR` senza apparire nell'intestazione, devono entrambe essere definite prima di quella, in modo che il compilatore le abbia già lette quando arriva a occuparsi della funzione chiamante.

Complessità Dato che entrambe le funzioni richiedono un numero costante di operazioni, oltre alla chiamata ricorsiva, anche nel caso pessimo, la complessità è proporzionale al numero di chiamate ricorsive, cioè alla profondità h dell'albero: $\Theta(h)$.

11.4 Esercizi

11.4.1 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 e si valuti la relativa complessità tenendo conto della lunghezza delle stringhe che rappresentano le parole.

11.4.2 Esercizio

Dato un albero binario di ricerca che rappresenta un insieme di numeri interi, si scriva una funzione che determini l'elemento immediatamente precedente un valore intero dato (eventualmente non appartenente all'insieme) e una funzione che determini l'elemento immediatamente successivo. Si consideri anche il caso in cui non esista un tale elemento.

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

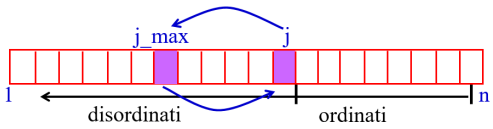


Figura 12.2: SelectionSort

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--)                   sum_{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++)                  sum_{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.

12.2 Heap

Se riuscissimo a trovare l'elemento massimo in tempo costante, potremmo ottenere una complessità lineare, perché eseguiremmo n iterazioni di complessità costan-

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*

te. 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 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

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

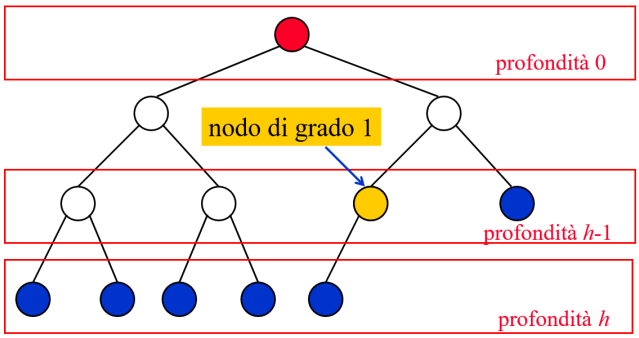


Figura 12.5: La struttura *max-heap*

"in fondo a sinistra" e c'è al massimo un nodo di grado 1, che ha soltanto il figlio di sinistra (vedi Figura 12.2). 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.

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 12.2, 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.

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

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

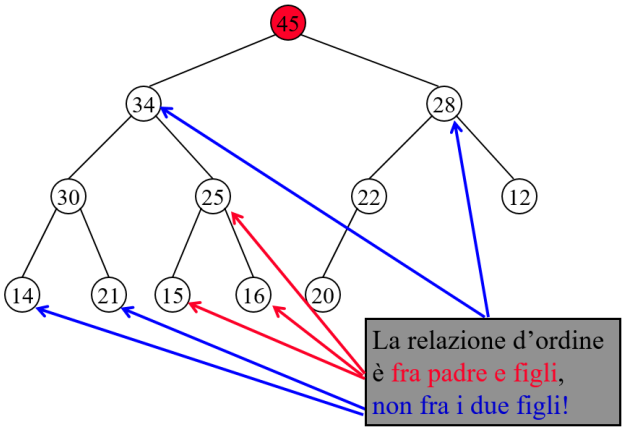


Figura 12.6: La struttura *max-heap*

Un *heap* rappresenta un ordinamento parziale, cioè una relazione binaria

- riflessiva
- transitiva
- antisimmetrica

Questo suggerisce che possa essere utile per il problema dell'ordinamento

Manca solo la completezza!

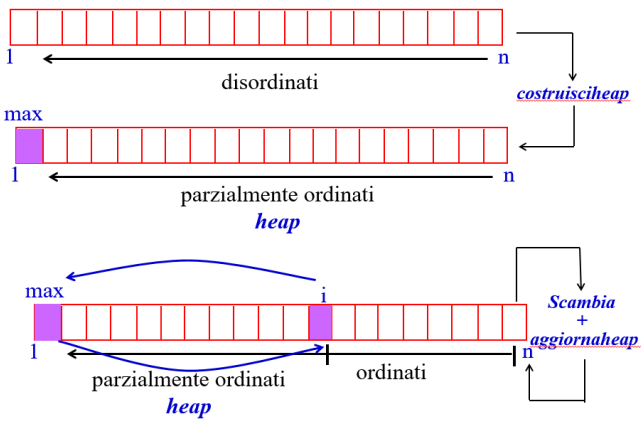


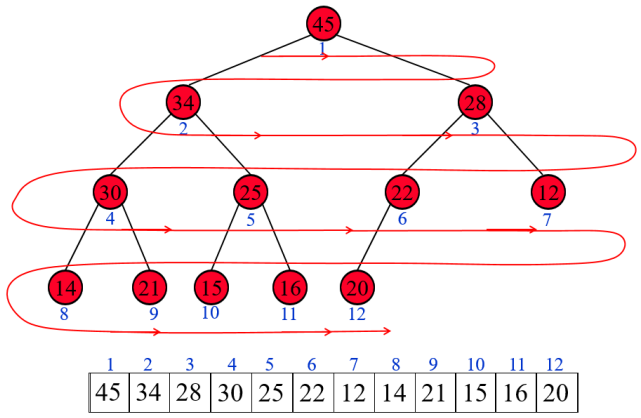
Figura 12.7: *Heap* e ordinamenti parziali

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.

Un *heap* può facilmente essere rappresentato come una tabella



La quasi completezza consente di sostituire puntatori espliciti con operazioni aritmetiche sugli indici:

- il figlio sinistro di $V[i]$ sta in $V[2i]$
- il figlio destro di $V[i]$ sta in $V[2i+1]$
- il padre di $V[i]$ sta in $V[i/2]$

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 12.2, 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* (vedi Figura 12.2) sono:

- la *creazione*, che avviene trasformando un vettore generico non ordinato di n elementi dell'insieme universo U in un heap di elementi sull'universo stesso;
- l'*aggiornamento*: qui la notazione usata in Figura 12.2 è impropria, perché l'operazione a rigore non parte da un *heap* corretto, ma da un oggetto in cui al massimo una singola posizione viola la proprietà fondamentale sulle etichette, e corregge l'errore restituendo un *heap*

Nel descrivere come realizzare queste operazioni partiamo dall'aggiornamento, perché la creazione può essere ridotta ad esso (vedi Figura 12.2). Abbiamo quindi un oggetto in cui una posizione può avere etichetta minore di quella di uno o di entrambi i figli. La prima cosa da fare sarà determinare quale dei tre nodi coinvolti (quello potenzialmente errato e i due figli) ha l'etichetta massima, dato che sarà quello a dover occupare la posizione del padre. Questo comporta prima ancora di determinare le posizioni dei due figli, con le regole introdotte precedentemente (l'indice doppio di quello del padre per il figlio di sinistra e l'indice successivo per quello di destra). Dopo di che, si ipotizza che il valore massimo sia associato al

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 12.10: Aggiornamento di un *heap*

padre e si procede a correggere l'assunzione confrontando il valore con quello di ciascuno dei due figli.

Il confronto è complicato da un test ulteriore, il quale verifica che le posizioni dei due figli non superino n, perché può darsi che il nodo considerato non abbia figli o abbia solo quello di sinistra. In tal caso, la proprietà *heap* andrebbe considerata implicitamente verificata.

Trovato il nodo di etichetta massima, occorre capire se l'etichetta è già nella posizione corretta, cioè quella del nodo padre, oppure no. Se lo è, non si fa nulla. Se invece non lo è, bisogna scambiare l'etichetta massima con quella del nodo padre. Questo implica che uno dei due figli sia intatto, mentre l'altro riceve un'etichetta inferiore. A valle di questa operazione, il nodo padre e il figlio intatto sono corretti, mentre quello modificato potrebbe ora violare la proprietà *heap*. Che fare allora?

Abbiamo già una procedura `AggiornaHeap` che serve a riparare un *heap* con un solo errore: basta chiamarla (ricorsivamente) sulla nuova posizione `iMax`. Notiamo anzi tutto che la ricorsione termina sicuramente perché la nuova posizione sta più avanti della vecchia (ha indice almeno doppio), e quindi raggiunge prima o poi la fine del vettore. Inoltre, è una ricorsione terminale, quindi abbastanza facile da sostituire con una versione iterativa del codice, che risparmierebbe spazio (per lo *stack*) e tempo (per il passaggio dei parametri).

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 12.11: Creazione di un *heap*

A questo punto, possiamo anche costruire un *heap* a partire da un generico vettore (vedi Figura 12.2). La memoria è chiaramente già disponibile, se si sceglie di distruggere il vettore originale nel processo. Il vettore è completamente disordinato: equivale sempre a un albero binario quasi completo (imponendo la regola di visita per livelli introdotta sopra), ma la proprietà *heap* manca in generale in tutte le posizioni.

L'idea è sfruttare la procedura di aggiornamento sopra descritta. Ma tale proprietà richiede oggetti in cui la proprietà è violata in una sola posizione. Il punto è dimenticare temporaneamente l'intero vettore, ovvero albero binario, e concentrarsi su un sottoalbero abbastanza piccolo da avere al massimo una posizione scorretta. Questi sottoalberi sono le foglie dell'albero complessivo. Nelle foglie, in effetti, la

proprietà *heap* è implicitamente soddisfatta, perché le foglie per definizione non hanno figli. Di conseguenza, si può partire dai nodi del livello immediatamente superiore e applicare la procedura `AggiornaHeap` procedendo all'indietro. Sostanzialmente, si ipotizza che la proprietà sia rispettata in tutti i livelli inferiori e si usa questa ipotesi per imporla ai livelli superiori, consentendo di eseguire la procedura anche su di loro nelle iterazioni successive. Siccome i livelli inferiori vengono trattati per primi, l'ipotesi è corretta per costruzione. Di conseguenza, la funzione `CreaHeap` non fa altro che scorrere tutti gli elementi del vettore, dall'ultimo al primo, e applicare la funzione `AggiornaHeap`. Siccome abbiamo osservato che è inutile applicare la procedura alle foglie, dato che i relativi sottoalberi sono già corretti, e quindi si può partire dall'ultimo nodo dotato di figli, che in un *heap* di n elementi occupa la posizione $\lfloor n/2 \rfloor$ (padre della posizione n). Nel complesso, eseguiamo $n/2$ volte la funzione `AggiornaHeap`.

Qual è la complessità di queste funzioni? La funzione `AggiornaHeap` applica un insieme finito di istruzioni prima di chiamare ricorsivamente se stessa (se necessario). Siccome ad ogni passo viene chiamata su una posizione di indice almeno doppio rispetto alla precedente, il caso pessimo avviene quando viene chiamata a partire dalla posizione di indice 1 e ogni livello procede a una chiamata sul livello seguente dell'albero, fino ad arrivare ad una foglia. Il numero di livelli di un albero binario completo di n nodi è $\Theta(\log n)$, e quindi la complessità di `AggiornaHeap` è la stessa. Per quanto riguarda `CreaHeap`, si rimanda alla lezione di teoria, dove si giunge alla non ovvia conclusione che, pur eseguendo $n/2$ volte una funzione di complessità $\Theta(\log n)$, il risultato è semplicemente lineare, $\Theta(n)$. In breve, questo deriva dal fatto che la stragrande maggioranza delle chiamate avvengono dai livelli inferiori dell'albero, con complessità praticamente costante, mentre solo un numero molto ridotto ha davvero complessità logaritmica.

12.3 HeapSort

A questo punto, possiamo sfruttare la capacità di creare e aggiornare un *heap* per ovviare alle inefficienze dell'algoritmo *SelectionSort*. L'algoritmo risultante, detto *HeapSort* (vedo Figura 12.3) applica la stessa idea, cioè di gestire sul vettore dei dati due sottovettori, associati agli elementi più piccoli, non ancora ordinati, e ai più grandi, già totalmente ordinati. Questo comporta di creare inizialmente un *heap* sul vettore dato, e poi ripetutamente determinarne l'elemento massimo (che sta per definizione nella posizione di indice 1), scambiarlo con quello nell'ultima posizione (definita da un indice `j`) e cedere questa posizione a una tabella che si allarga via via decrementando l'indice `j`. Lo scambio di `V[1]` con `V[j]` porta in prima posizione un elemento non massimo (anzi, tipicamente piuttosto piccolo) e quindi introduce un errore nell'*heap*. Di conseguenza, occorrerà aggiornare la posizione 1 dell'*heap*, che occupa le posizioni da 1 a `j-1` (avendo espulso la posizione finale e non ancora aggiornato la lunghezza).

La Figura 12.3 mostra chiaramente come `SelectionSort` e HeapSort siano praticamente lo stesso algoritmo con una struttura dati diversa: il secondo compie le stesse operazioni concettuali del primo lavorando su un *heap* una tabella ordinata conservate all'interno di un vettore, anziché su una tabella disordinata e una totalmente ordinata. Questo comporta un aggravio segnato in rosso, costituito dalla costruzione dell'*heap* in tempo lineare e dall'aggiornamento in tempo logaritmico per ognuna delle n iterazioni. D'altra parte, si ha il grosso vantaggio di ridurre la funzione che determinava l'elemento massimo del vettore in tempo lineare alla semplice determinazione del primo elemento, in tempo costante (nel codice non separeremmo neppure la determinazione dell'indice `i` dal suo uso nell'operazione di

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 12.12: *HeapSort* e *SelectionSort*

scambio, che qui sono distinte solo per maggiore chiarezza). Nel complesso, questo abbassa la complessità temporale da $\Theta\left(n^2\right)$ a $\Theta\left(n \log n\right)$.

0:27:30 Siamo quindi ora perfettamente in grado di implementare questo sotto forma di codice. Consideriamo lo stesso file contenente 100 numeri interi (non tutti distinti) che abbiamo usato nelle Sezioni 6.3 e 6.4 per implementare gli algoritmi di *InsertionSort* e *SelectionSort* e partiamo dallo stesso codice iniziale `sort0.c`, che contiene l'interpretazione della linea di comando (per ottenere dall'utente il nome del file dei dati), il caricamento del solito vettore dinamico di interi `V` di tipo `vint` (compresa la sua allocazione), la stampa del risultato (per verificare che l'ordinamento sia corretto) e la deallocazione finale del vettore. Manca solo la procedura di ordinamento (un opportuno commento ne marca il posto). Andremo quindi ad aggiungere la chiamata dell'algoritmo con i suoi argomenti: il vettore `V` e la sua lunghezza `n`.

```
/* Programma principale */
int main (int argc, char *argv[])
{
  char filedati[ROW_LENGTH];
  vint V;
  int n;


  /* Legge da linea di comando il file che contiene i dati */
  InterpretaLineaComando(argc,argv,filedati);

  /* Legge la lunghezze e gli elementi del vettore */
  CaricaVettoreInteri(filedati,&V,&n);

  /* Ordina il vettore */
  HeapSort(V,n);

  /* Stampa il vettore ordinato */
  StampaVettoreInteri(V,n);

  /* Dealloca il vettore (anche per controllare eventuali errori di
     accesso) */
  free(V);

  return EXIT_SUCCESS;
```

```
}
```

Al solito aggiungeremo anche fra i prototipi la dichiarazione della nuova procedura, che avrà come parametri un vettore di interi e un intero, oltre a un commento appropriato.

```
/* Ordina il vettore V di lunghezza n con l'algoritmo HeapSort */
void HeapSort (vint V, int n);
```

Infine, avremo in basso la solita dichiarazione con un corpo vuoto, pronta per le fasi di implementazione successive. Compilando il file, dovremmo ottenere un programma che si limita a caricare il vettore da file e a stamparlo, dato che la funzione di ordinamento non fa nulla. Quindi, possiamo valutare quanto meno di aver letto correttamente i dati confrontando la stampa a video con il file originale.

A questo punto, possiamo pedissequamente seguire lo pseudocodice: creare un *heap*, ridurre gradualmente la prima tabella da n a un solo elemento e allargare la seconda scambiando ad ogni passo l'elemento massimo, che è il primo dello *heap* con l'ultimo elemento (in posizione `j`), diminuire la dimensione dello *heap* e aggiornare la posizione di indice 1, che ora è scorretta. La funzione `creaheap` opera sull'intero vettore `V` di lunghezza `n`. Il ciclo seguente fa scorrere l'intero `j` (che va precedentemente dichiarato) da `n` a `1` escluso (esattamente come per `SelectionSort`, dato che l'ultimo elemento rimasto non ordinato è automaticamente il minimo e sta nella posizione corretta). Ad ogni iterazione, la posizione dell'elemento massimo è quella di indice `1`, per cui applichiamo direttamente la funzione `Scambia` a `V[1]` e all'ultima posizione `V[j]`. Passiamo i due numeri interi per indirizzo, in modo che la funzione li modifichi fisicamente. Infine, aggiorniamo con una procedura `aggiornaheap` l'*heap* corrente, cioè quella parte del vettore `V` compresa fra la posizione `1` e la posizione `j-1`, dato che la posizione `j` è stata spostata nella tabella ordinata, ma la lunghezza `j` non è ancora stata ridotta; l'aggiornamento si riferisce in particolare alla posizione `1`, che è stata modificata e molto probabilmente violerà la proprietà *heap*.

```
/* Ordina il vettore V di lunghezza n con l'algoritmo HeapSort */
void HeapSort (vint V, int n)
{
  int j;

  creaheap(V,n);
  for (j = n; j > 1; j--)
  {
    Scambia(&V[1],&V[j]);
    aggiornaheap(V,j-1,1);
  }
}
```

Per ottenere la correttezza sintattica, dobbiamo aggiungere i prototipi e i corpi della tre funzioni `creaheap`, `aggiornaheap` e `Scambia`.

Qui si apre una piccola discussione stilistica. Abbiamo quasi sempre adottato la strategia di prendere la chiamata, riportarla all'inizio nell'elenco dei prototipi e modificarla per costruire la sua dichiarazione, e riportare questa in basso per ottenere la sua definizione. Lavorando su alcune librerie, abbiamo fatto una cosa leggermente diversa: poiché la funzione `creaheap` viene chiamata solo dalla funzione `HeapSort`, riportarla fra i prototipi appare superfluo. Può essere sufficiente limitarsi

a definire la funzione `creaheap` subito sopra la definizione di `HeapSort`. Infatti, quando il compilatore scorre il codice trova la definizione di `creaheap` prima della sua chiamata ed è perfettamente in grado di gestirla. Aggiungere il prototipo di una funzione al principio di un file consente a tutte le funzioni del file stesso di usarla. Non farlo limita la possibilità alle funzioni definite dopo di essa, ed equivale quindi un po' a nasconderla. Nel caso presente, non c'è una grossa differenza. Se però si volesse utilizzare la funzione `HeapSort` in altri programmi (un'eventualità non improbabile, vista la sua generalità), probabilmente converrebbe costruire una libreria a parte, e allora bisognerebbe capire se l'intestazione di questa libreria dovrà limitarsi a riportare il prototipo di `HeapSort` e continuare a tenere nascosta `creaheap` oppure fornirà al mondo esterno anche una struttura *heap* con diverse funzionalità. In tal caso, un'operazione di creazione dovrebbe comparire anche nell'interfaccia con un prototipo esplicito. Se vogliamo preparare la realizzazione di questa libreria, senza farlo ora perché eccede i limiti di tempo della lezione, possiamo aggiungere tutte le dichiarazioni in alto e tutte le definizioni in basso. Questo consentirà di costruire la libreria semplicemente spostando le dichiarazioni nel file *header* e le definizioni nel file listato.

```
/* Crea un heap sul vettore V di lunghezza n */
void creaheap (vint V, int n);

/* Scambia i numeri interi a e b */
void Scambia (int *pa, int *pb);

/* Aggiorna l'heap V di lunghezza n a partire dalla posizione i */
void aggiornaheap (vint V, int n, int i);
```

dove la funzione `creaheap` riceve un vettore dinamico di interi e la sua lunghezza intera, la funzione `aggiornaheap` riceve un vettore dinamico di interi e la sua lunghezza intera e la posizione intera da modificare, e la funzione `Scambia` riceve gli indirizzi di due numeri interi[1].

Un'altra considerazione stilistica è che realizzare una libreria consentirebbe anche di rendere astratta la struttura dati *heap* e nascondere all'utente la sua implementazione, che invece ora è chiaramente basata su un vettore. Questo sarebbe coerente con le lezioni precedenti. D'altra parte, qui l'accento è sugli algoritmi di ordinamento e non si è voluto sovraccaricare la lezione con aspetti diversi ormai già ampiamente trattati.

Come si è già detto, la creazione dell'*heap* consiste nel chiamare la funzione di aggiornamento sulla prima metà del vettore, procedendo dal punto di mezzo e andando all'indietro fino alla seconda.

```
/* Crea un heap sul vettore V di lunghezza n */
void creaheap (vint V, int n)
{
  int i;

  for (i = n/2; i >= 1; i--)
    aggiornaheap(V,n,i);
}
```

[1]La funzione `Scambia` ha l'iniziale maiuscola, mentre le altre due l'hanno minuscola perché tendenzialmente nel corso teniamo minuscoli i nomi delle funzioni che lavorano su strutture dati di base. La distinzione è assolutamente convenzionale. Nota: nel capitolo dove compare la prima volta è minuscola: conviene uniformare tutto.

Lo scambio fra due numeri interi è la solita funzione già descritta nella Sezione 5.4 e nell'algoritmo *SelectionSort*, che si serve di una variabile ausiliaria per conservare il numero sovrascritto dall'altro.

```
/* Scambia i numeri interi a e b */
void Scambia (int *pa, int *pb)
{
  int temp;

  temp = *pa;
  *pa = *pb;
  *pb = temp;
}
```

L'aggiornamento dell'*heap* è la procedura più significativa e complessa. Si determinano gli indici `s` e `d`, rispettivamente del figlio sinistro e destro (ovviamente potremmo usare direttamente le espressioni algebriche senza dichiarare variabili: è solo per chiarezza). Poi si determina quale nodo fra i due figli e il padre ha il valore massimo. L'ipotesi iniziale è che il corrispondente indice intero `iMax` sia quello del padre, nel qual caso non bisognerebbe fare assolutamente nulla. Poi si verifica l'ipotesi per confronto con il figlio di sinistra, eventualmente sostituendo `iMax` con `s`. Questo comporta preliminarmente di valutare se l'indice `s` corrisponda davvero a un nodo figlio, cioè non ecceda la lunghezza dell'*heap*. Fatto lo stesso con il figlio destro, l'indice `iMax` determina effettivamente l'elemento di valore massimo fra i tre. Qualora coincida col padre, non si fa nulla. Qualora coincida con uno dei due figli, si scambiano le etichette del padre e del figlio così determinato, con la solita funzione `scambia`. Ora la posizione del padre e del figlio non modificato sono corrette, mentre quella del figlio modificato potrebbe essere scorretta. Quindi, bisogna aggiornare l'*heap* anche nella nuova posizione `iMax`.

```
/* Aggiorna l'heap V di lunghezza n a partire dalla posizione i */
void aggiornaheap (vint V, int n, int i)
{
  int s, d;
  int iMax;


  s = 2*i;
  d = 2*i+1;

  iMax = i;
  if ( (s <= n) && (V[s] > V[iMax]) ) iMax = s;
  if ( (d <= n) && (V[d] > V[iMax]) ) iMax = d;

  if (iMax != i)
  {
    Scambia(&V[iMax],&V[i]);
    aggiornaheap(V,n,iMax);
  }
}
```

Ora si può verificare compilando ed eseguendo l'algoritmo che abbiamo effettivamente realizzato *HeapSort*.

12.4 MergeSort

Passiamo ora a un secondo algoritmo di ordinamento efficiente, detto algoritmo di ordinamento per fusione, cioè *MergeSort*. Questo algoritmo applica il fondamentale principio del *divide et impera*, esattamente come il successivo algoritmo *QuickSort* (vedi Figura 12.4).

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 12.13: Ordinamento per fusione (*MergeSort*)

Entrambi sono algoritmi ricorsivi e consistono nel dividere il vettore `V` in due sottovettori, individuati dai due indici estremi (sinistro e destro). In teoria, si potrebbe dividere il vettore in più di due parti, ma questo non porta alcun vantaggio. Nel caso di *MergeSort*, i due sottovettori hanno esattamente la stessa lunghezza e si ottengono determinando l'elemento mediano del vettore originale. Questo è il modo più semplice in assoluto di dividere un vettore. La fase denominata *impera* consiste nel risolvere i due sottoproblemi così ottenuti. Questo si ottiene ricorsivamente, chiamando l'algoritmo su ciascuno dei due.

Supponendo che l'algoritmo fornisca la soluzione corretta, cioè permuti gli elementi di ciascuno dei due sottovettori in maniera da renderli internamente ordinati, rimane la fase finale, spesso denominata *combina*, nella quale le soluzioni ottenute, cioè i due sottovettori ordinati, vanno ricombinate in modo da ottenere un vettore complessivo totalmente ordinato. Nel caso di *MergeSort*, gli elementi sono ordinati all'interno di ciascuno dei due sottovettori, ma non sono ordinati nel complesso. Quindi, occorre una procedura *Merge* che proceda a fonderli.

La Figura 12.4 riporta la struttura dell'algoritmo. Come in ogni algoritmo ricorsivo, c'è un caso base, che però qui è talmente semplice da non comparire. Il caso base si presenta quando il vettore è talmente corto da essere già ordinato, cioè contiene un solo elemento (e dunque gli indici sinistro e destro coincidono) o nessuno (in tal caso, per convenzione l'indice sinistro eccede il destro). Di conseguenza, per `s >= d` non si fa nulla.

Rimane il caso ricorsivo (`s < d`). In tale caso, prima di tutto si divide il problema in sottoproblemi determinando l'indice mediano `m`. Poi ci sono le chiamate ricorsive sui due sottoproblemi. Infine, c'è la ricombinazione delle due soluzioni in una sola con la funzione `Merge`.

A dispetto dell'apparene banalità, l'algoritmo nasconde delle insidie. L'indice mediano viene calcolato come la semisomma dei due estremi, ma questa può avere un valore frazionario e dover quindi essere arrotondata, per difetto o per eccesso. Inoltre, bisogna decidere quale dei due sottovettori includerà l'elemento di indice mediano. Sembrano questioni secondarie, ma non lo sono affatto perché due dei quat-

```
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 12.14: *MergeSort*: pseudocodice

tro casi a cui danno luogo sono errati. Per esempio, se calcoliamo l'indice mediano arrotondandolo per difetto, come nella Figura 12.4, ma assegniamo l'elemento corrispondente al secondo sottovettore (dunque con le istruzioni `MergeSort(V,s,m-1);` e `MergeSort(V,m,d);`), cadrebbe una proprietà fondamentale per la correttezza della ricorsione: i due sottoproblemi non sarebbero sempre strettamente più piccoli del problema originale.

Consideriamo, ad esempio, `s = 5` e `d = 6`, cioè un vettore di due elementi. La semisomma vale 5.5 e il suo arrotondamento per difetto vale 5. Il sottovettore sinistro andrebbe da `s = 5` a `m-1 = 4`, cioè sarebbe vuoto, mentre il sottovettore destro andrebbe da `m = 5` a `d = 6`, cioè sarebbe identico al vettore di partenza. Siccome la chiamata corrente contiene un'altra chiamata ricorsiva perfettamente identica, si ottiene un'infinita sequenza di chiamate ricorsive, che in pratica viene interrotta dall'esplosione dello *stack*, cioè dall'esaurimento della memoria ad esso dedicata. Abbiamo sbagliato la decomposizione del problema.

Se si arrotonda per difetto, l'indice mediano deve essere assegnato al primo sottovettore. Si potrebbe arrotondare per eccesso assegnando l'indice mediano al secondo sottovettore (è una convenzione meno comune, per motivi arbitrari). Le altre due combinazioni sono invece errate.

12.4.1 La procedura *Merge*

Rimane solo da discutere in dettaglio la procedura che fonde due vettori ordinati in un vettore complessivo ordinato (vedi Figura 12.4.1). La procedura nella sua versione più semplice usa un vettore ausiliario di appoggio. Esiste una versione molto più sofisticata che non ha bisogno, ma va al di là degli scopi del corso. In questo vettore B vengono copiati via via gli elementi dei due sottovettori nell'ordine corretto per ottenere la soluzione finale. L'idea banale è partire da un vettore B

vuoto e aumentarlo passo per passo accodandovi ogni volta l'elemento minimo non ancora inserito. Tale elemento può stare solo in due posizioni: o è l'elemento minimo del primo sottovettore (quindi sta alla sua testa) o è l'elemento minimo del secondo sottovettore (quindi sta alla testa di questo). Basta confrontare gli elementi dei due sottovettori e mettere il minore in fondo al vettore B. È chiaro che in questo modo il vettore B è ordinato e contiene solo elementi non più grandi di quelli dei due sottovettori. Tutto procede finché uno dei due sottovettori diventa vuoto. Allora, non ha più senso cercare il minimo, ma basta copiare l'intero sottovettore residuo in fondo al vettore B. Riempito il vettore B basta copiarlo nel vettore originale V. In realtà vedremo che si fa una cosa leggermente più efficiente.

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 12.15: La procedura *Merge*

12.4.2 Correttezza

L'algoritmo *MergeSort* funziona grazie al principio di induzione matematica forte (vedi Figura 12.4.2). Si dimostra facilmente che funziona per vettori molto piccoli (vuoti o con un solo elemento): l'algoritmo non fa nulla, ma il vettore è già ordinato. D'altra parte, se si suppone per induzione forte che l'algoritmo sia valido su tutti i vettori fino a dimensione $n-1$, possiamo osservare che le due chiamate a *MergeSort* operano su vettori di dimensione $n/2$ (opportunamente arrotondata), e quindi sono corretti: dopo averle eseguite, i due sottovettori sono ordinati. Per ottenere l'ordinamento di tutto il vettore occorre semplicemente dimostrare che la funzione *Merge* è corretta, cioè trasforma due sottovettori ordinati in un vettore complessivo ordinato. Ne deriva che un algoritmo valido fino a dimensione $n-1$ vale anche per la dimensione n. Il principio di induzione forte permette di concludere che l'algoritmo vale per tutte le dimensioni, cioè per tutte le possibili istanze.

12.4.3 Complessità

Procediamo ora a valutare la complessità dell'algoritmo *MergeSort* (vedi Figura 12.4.3. Abbiamo un algoritmo ricorsivo, in cui il caso base richiede tempo costante, perché consiste nel valutare se `s < d` oppure no. Il caso ricorsivo è costituito da

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: *MergeSort*: correttezza

una operazione in tempo costante (il calcolo dell'indice mediano), da due chiamate ricorsive eseguite su vettori di dimensione $n/2$ (arrotondata, rispettivamente, per eccesso e per difetto) e dalla procedura *Merge*. Questa richiede tempo lineare nella lunghezza n del vettore, perché ciascuno degli elementi viene copiato una prima volta dal vettore V al vettore B e una seconda volta dal vettore B al vettore V.

```
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 12.17: *MergeSort*: complessità

Applicando il teorema fondamentale delle equazioni ricorrenti o gli altri metodi disponibili per risolvere queste equazioni (vedi Figura 12.4.3, otteniamo una complessità pari a $\Theta(n \log n)$ (per questo si rimanda alle dispense del modulo di teoria). Tale complessità è ottimale per il problema dell'ordinamento, a meno di costanti moltiplicative[2].

[2]Questa osservazione va anticipata nella discussione su *MergeSort*, se non al principio

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 12.18: *MergeSort*: complessità

12.4.4 Implementazione

Torniamo al solito punto di partenza per gli algoritmi di ordinamento, cioè al file `sort0.c`, e completiamolo con una chiamata a `MergeSort`. Rispetto a `HeapSort` abbiamo una fondamentale differenza, benché il problema sia lo stesso: siccome la procedura sarà ricorsiva, verrà più comodo indicare come parametri non il vettore `V` e la lunghezza `n`, ma il vettore `V` e i due indici estremi `s` e `d`. A rigore, questo non è necessario, perché sappiamo che potremmo definire sottovettori su un vettore applicando l'aritmetica dei puntatori, come discusso nella Sezione **??**. Per maggiore chiarezza, non lo facciamo. Abbiamo quindi un prototipo leggermente diverso.

```
/* Ordina il sottovettore del vettore V compreso fra s e d con
    l'algoritmo MergeSort */
void MergeSort (vint V, int s, int d);
```

Il corpo della funzione comincia col caso base, che viene semplicemente negato, dato che implica di non compiere alcuna operazione, seguito dal blocco ricorsivo in cui si calcola la posizione mediana con la semisomma troncata attraverso la divisione intera. Quindi, si chiama ricorsivamente la funzione `MergeSort` fra gli indici `s` e `m` e fra gli indici `m+1` e `d`. Infine, si fondono le due parti del vettore: evidentemente è inutile riportare tutti e quattro gli indici, dato che quello sinistro del secondo sottovettore segue immediatamente quello destro del primo.

```
/* Ordina il sottovettore del vettore V compreso fra s e d con
    l'algoritmo MergeSort */
void MergeSort (vint V, int s, int d)
{
  int m;
```

dell'intero capitolo.

```
  if (s < d)
  {
    m = (s+d)/2;
    MergeSort(V,s,m);
    MergeSort(V,m+1,d);
    Merge(V,s,m,d);
  }
}
```

Affinché tutto funzioni basta dichiarare la funzione `Merge`. Come nel caso di *HeapSort*, si può discutere se mettere tutto in una libreria a parte o tenere queste procedure nel file corrente e se la funzioni secondarie come `Merge` vanno semplicemente riportate prima di quella principale oppure devono anche avere un prototipo al principio del file. Dipende da quanto possa essere utile fondere due sottovettori ordinati in uno solo complessivo. In questa forma, probabilmente non molto (diverso sarebbe forse fondere due vettori ordinati completamente separati). Seguiamo quindi la via più semplice, anche per mostrare concretamente entrambe le possibilità.

La funzione `Merge` scorre le due metà del vettore `V` con i cursori `is` e `id`, ma richiede anche un vettore ausiliario di interi `B` e un cursore `i` per scrivere in esso. Dichiarate tutte le variabili, dobbiamo allocare il vettore ausiliario, e deallocarlo alla fine. La sua lunghezza dipende dai valori specifici di `s` e `d`. Quindi, dovremo scorrere i sottovettori `V[s,m]` e `V[m+1,d]` finché uno dei due non rimane vuoto, cioè il relativo cursore non supera il relativo indice destro. Ad ogni passo, si sposta il minore dei due elementi iniziali nel vettore ausiliario `B`. Al termine, dovremo gestire il residuo, cioè la parte rimanente del primo o del secondo sottovettore che non è ancora stata copiata in `B`. Siccome nel complesso non sono operazioni banali (e l'ultima in particolare verrà realizzata in maniera abbastanza astuta, per ora ci limitiamo a riportare dei commenti. Al termine copieremo il vettore ausiliario in quello originale.

```
/* Fonde il sottovettore ordinato V[s,m] con il sottovettore ordinato
    V[m+1,d] */
void Merge (vint V, int s, int m, int d)
{
  int is, id;
  vint B;
  int i;


  /* Alloca il vettore ausiliario B */

  /* Scorre i sottovettori V[s,m] e V[m+1,d] finche' uno dei due non
      e' vuoto
     spostando ogni volta il minimo dei due nel vettore ausiliario B
         */
  /* Gestisce il residuo del primo sottovettore, spostando gli
      elementi in fondo al vettore;
     (lo fa a ritroso per non sovrascrivere gli elementi ancora da
         spostare);
     se e' il secondo sottovettore ad avere un residuo, gli elementi
         sono gia' nella posizione corretta. */

  /* Copia il vettore ausiliario B in V */

  /* Dealloca il vettore ausiliario B */
```

Procediamo quindi a risolvere i singoli punti. L'allocazione è molto semplice: non occorre azzerare il vettore con `calloc`, dato che verrà riempito esplicitamente. La lunghezza deve consentire di ospitare tutti gli elementi, dall'indice `s` all'indice `d`, quindi `d-s+1`. Volendo partire dall'indice 1, occorre un elemento in più.

```
/* Alloca il vettore ausiliario B */
B = (vint) calloc(d-s+2,sizeof(int));
if (B == NULL)
{
  fprintf(stderr,"Errore nell'allocazione del vettore ausiliario
      B!\n");
  exit(EXIT_FAILURE);
}
```

Per scorrere i due sottovettori, inizializziamo i due cursori `is` e `id` alle posizioni iniziali `s` e `m+1`. Il cursore `i` relativo al vettore `B` potrebbe avere inizialmente valore `1`, inteso come l'indice della prima posizione libera in `B`, oppure valore `0`, inteso come l'indice dell'ultima posizione occupata in `B`. Le due scelte sono equivalenti, ma la seconda risulterà un po' più spontanea quando la implementeremo. Il ciclo di scorrimento procede finché entrambi i sottovettori sono non vuoti, cioè `is <= m` e `id <= d`. A questo punto gli elementi `V[is]` e `V[id]` sono gli elementi minimi dei due sottovettori, e il minore dei due può essere estratto dal proprio sottovettore e copiato nella prima posizione libera di `B`: se si incrementa `i`, questa posizione diventa `B[i]`. Può ovviamente capitare che i due elementi minimi siano uguali. In tal caso, la scelta è arbitraria: copieremo il primo per motivi che risulteranno più chiari fra poco. Copiato in `B[i]` l'elemento prescelto, bisogna incrementare il cursore del vettore corrispondente, in modo da eliminare tale elemento dal vettore stesso.

```
/* Scorre i sottovettori V[s,m] e V[m+1,d] finche' uno dei due non
    e' vuoto
   spostando ogni volta il minimo dei due nel vettore ausiliario B
       */
is = s;
id = m+1;
i = 0;
while ( (is <= m) && (id <= d) )
{
  i++;
  if (V[is] <= V[id])
  {
    B[i] = V[is];
    is++;
  }
  else
  {
    B[i] = V[id];
    id++;
  }
}
```

Terminato il ciclo, uno dei due sottovettori è vuoto, mentre l'altro non lo è. I casi possibilisono ovviamente due: può rimanere un pezzo del primo sottovettore, oppure un pezzo del secondo. Il secondo caso è interessante, perché implica che la parte finale del secondo sottovettore è costituita da elementi più grandi di quelli copiati nel vettore ausiliario `B`, cioè di tutti gli altri elementi. Inoltre, sono ordinati sin dal principio. Quindi, tali elementi sono già nella posizione corretta, e non è necessario copiarli in `B` per poi ricopiarli indietro nel vettore di partenza. Solo quando rimane

un residuo del primo sottovettore occorre fare qualcosa: si potrebbero copiare gli elementi in fondo a `B` e poi riportarli in fondo a `V`, ma tanto vale fare un passaggio solo scalando tali elementi in avanti nel vettore `V`. C'è un'avvertenza fondamentale: gli elementi vanno scalati in avanti partendo dall'ultimo e retrocedendo, perché in caso contrario si rischia che i primi elementi riscalati in avanti vadano a sovrascrivere gli ultimi elementi non ancora riscalati. Quindi si partirà dal fondo del primo sottovettore (indice `m`) e si procederà all'indietro fino al principio del residuo del primo sottovettore (indice `is`). Per gestire correttamente la riscalatura, basta tenere a mente che tutti gli elementi vanno spostati avanti di un valore uguale e che in particolare l'elemento di indice `m` deve finire in posizione `d`. Quindi, ogni elemento va spostato avanti di un valore `d-m`. Per inciso, non è necessario testare se esiste o no un residuo del primo sottovettore perché quando non esiste (`is > m`) il ciclo viene semplicemente saltato (infatti `j = m` e quindi `j < is`).

```
/* Gestisce il residuo del primo sottovettore, spostando gli
    elementi in fondo al vettore;
   (lo fa a ritroso per non sovrascrivere gli elementi ancora da
       spostare);
   se e' il secondo sottovettore ad avere un residuo, gli elementi
       sono gia' nella posizione corretta. */
for (j = m; j >= is; j--)
  V[j-m+d] = V[j];
```

Conclusa questa parte, rimane da riportare nel vettore originale gli elementi copiati nel vettore ausiliario. Siccome le posizioni occupate di questo vettore vanno da `1` a `i`, potremmo eseguire un ciclo a conteggio con un ulteriore cursore oppure limitarci a decrementare via via `i` copiando via via gli elementi da `B` a `V`, ancora una volta all'indietro, anche se in questo caso non è strettamente necessario. Anche qui occorre stabilire una riscalatura di indici, perché `V` parte dall'indice `s` anziché dall'indice `1`. Per farlo basta osservare che la differenza fra gli indici è fissa anche in questo caso, e precisamente è tale da convertire `1` in `s`. Dunque, è pari a `s-1`.

```
/* Copia il vettore ausiliario B in V */
while (i > 0)
{
  V[i-1+s] = B[i];
  i--;
}
```

A questo punto in teoria siamo in grado di compilare e verificare che l'algoritmo sia implementato correttamente.

12.5 QuickSort

1:14:00 Il nome dell'algoritmo *QuickSort*, cioè "ordinamento veloce" è vagamente paradossale, perché l'algoritmo nella sua forma base in realtà ha complessità quadratica nel caso pessimo. D'altra parte, a meno di non compiere sistematicamente scelte particolarmente sfortunate nel corso dell'esecuzione, l'algoritmo richiede tempo $\Theta(n \log n)$, con costanti particolarmente vantaggiose. Il caso pessimo si verifica solo in caso di scelte particolarmente cattive nella gran maggioranza dei passi.

L'algoritmo applica una strategia di tipo *divide et impera* simile a quelle di *MergeSort* (vedi Figura 12.5): divide il vettore in due sottovettori, li ordina e li ricombina. La divisione però non è più bilanciata, in due sottovettori della stessa

lunghezza, ma risulta da una operazione più costosa, ma più raffinata, che ottiene già in partenza da una parte gli elementi più piccoli e dall'altra quelli più grandi del vettore originale. Con “più piccoli” e “più grandi” si intende inferiori e superiori a un valore di soglia, che è il valore di un opportuno elemento (detto *pivot*) del vettore originale. Il vantaggio è che, una volta ordinati i due sottovettori, per ricombinarli non occorre fare altro che accodare il secondo al primo, dato che sono già vicendevolmente ordinati, oltre che internamente ordinati: gli elementi piccoli stanno a sinistra, quelli grandi a destra e l'elemento *pivot* in mezzo.

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 12.19: Ordinamento “veloce” (*QuickSort*)

Quindi l'idea è di non eseguire una divisione banale in due parti uguali col calcolo di un elemento mediano in tempo $\Theta(1)$, ma una divisione raffinata, con una procedura che definiremo *Partition*, due chiamate ricorsive sui sottoproblemi e una ricombinazione del tutto banale, dato che i due sottovettori sono già parte del vettore originale e sono già nella posizione corretta. Il caso base sarà il solito: un vettore vuoto o con un solo elemento non richiede di fare nulla per essere ordinato. Inoltre, il requisito che i due sottoproblemi siano entrambi rigorosamente più piccoli di quello di partenza è garantito dalla presenza del *pivot*, che non fa parte di nessuno dei due: anche nel caso pessimo in cui uno dei due sottovettori è vuoto, l'altro è comunque più piccolo del vettore iniziale.

Lo pseudocodice di Figura 12.5 conferma quanto descritto. Il caso base (`s <= d`) è invisibile, dato che comporta di non fare nulla. Il caso ricorsivo costruisce i due sottoproblemi con la procedura `Partition`, che restituisce l'indice `q` della posizione dove si trova l'elemento *pivot*. Questa posizione non è nota a priori, dato che non si sa quanti elementi del vettore originale sono minori o maggiori di esso. La procedura `Partition` permuta gli elementi del vettore in maniera tale che dalla posizione `s` alla posizione `q-1` ci siano gli elementi “piccoli”, cioè minori o uguali al *pivot*, e dalla posizione `q+1` alla posiione `d` ci siano gli elementi maggiori del *pivot*. I due sottovettori non sono in genere ordinati. Vengono quindi ordinati dalle due chiamate ricorsive a `QuickSort` sui due pezzi del vettore `V`. Poi ci dovrebbe

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 12.20: *QuickSort*: pseudocodice

essere una procedura di ricombinazione, che prenda il primo sottovettore ordinato, l'elemento *pivot* e il secondo sottovettore ordinato e costruisca il vettore complessivo ordinato. Evidentemente, non c'è nulla da fare, perché tutto questo è avvenuto in loco all'interno del vettore `V`. Quindi, questo passo non appare.

Il caso più dannoso è che l'indice `q` coincida con `s` o con `d`, cioè che l'elemento *pivot* scelto dalla procedura `Partition` sia quello minimo o quello massimo dell'intero vettore. In questo caso, una delle due chiamate è inutile (il sottovettore è vuoto), ma l'altra opera su un vettore di dimensione `n-1`. In questo caso, è facile dimostrare che l'intero algoritmo ha una complessità in $\Theta\left(n^2\right)$.

12.5.1 Correttezza

La correttezza dell'algoritmo *QuickSort* sia corretto deriva, al solito, dal principio di induzione forte (vedi Figura 12.5.1). Per vettori contenenti zero o un elemento, l'algoritmo funziona perché non fa nulla e il vettore è già ordinato. Se si suppone che l'algoritmo funzioni correttamente su problemi di dimensione minore di n, le due chiamate ricorsive sono certamente corrette. Il passo rimanente è garantire che la funzione *Partition* effettivamente metta nelle prime posizioni del vettore gli elementi minori o uguali al *pivot* e nelle ultime quelli maggiori. Garantito questo, l'algoritmo funziona anche per vettori di dimensione n, e quindi per vettori di qualsiasi dimensione.

12.5.2 Complessità

Per quanto riguarda la complessità, dobbiamo distinguere il caso pessimo nel quale uno dei due sottovettori risulta vuoto in una larga maggioranza di chiamate ricorsive da quello medio in cui i due sottovettori risultano abbastanza bilanciati. Con op-

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 $\ell < n \ldots$
  4. il concatenamento crea banalmente un vettore ordinato
     $\ldots$ l'algoritmo ordina anche qualsiasi vettore di lunghezza n

L'algoritmo ordina qualsiasi vettore

Figura 12.21: *QuickSort*: correttezza

portune ipotesi sulla distribuzione di probabilità dei numeri all'interno del vettore, si riesce a dimostrare che mediamente la complessità è in $\Theta(n \log n)$.

Va anche detto che ci sono metodi per determinare un elemento *pivot* in modo tale che il caso pessimo venga evitato. Il metodo più semplice è stocastico: scegliere l'elemento in modo casuale con probabilità uniforme, in modo che sia estremamente improbabile che sia sempre quello minimo o quello massimo. Ci sono anche metodi deterministici, che investono un piccolo sforzo computazionale nella scelta di un elemento *pivot* in modo che i due vettori abbiano una dimensione minima garantita sufficiente a garantire la complessità $\Theta(n \log n)$. Si può dunque garantire tale complessità anche nel caso pessimo. Qui ci limiteremo a una versione assolutamente basica per non chiamare in causa generatori di numeri pseudocasuali o procedure di scelta raffinate: sarà arbitrariamente scelto come elemento *pivot* il primo elemento del sottovettore, quello in posizione `s`. La scelta si rivela deleteria quando il vettore di partenza è ordinato in modo crescente (l'elemento *pivot* è allora il minimo) oppure in modo decrescente (l'elemento *pivot* è allora il massimo). In altre situazioni, invece, probabilmente la scelta è valida quanto qualsiasi altra.

12.5.3 Implementazione

Partendo dal solito file `sort0.c`, vi aggiungiamo la solita chiamata alla procedura di ordinamento, il prototipo e il corpo vuoto. Salvo il nome e il commento esplicativo, non cambia nulla rispetto a `MergeSort`.

```
/* Ordina il sottovettore del vettore V compreso fra s e d con
    l'algoritmo QuickSort */
void QuickSort (vint V, int s, int d);
```

Passando alla procedura vera e propria, la struttura ricorsiva sarà ancora la solita. Il caso base è vuoto. Quello ricorsivo chiama la procedura `Partition` per produrre i due sottoproblemi. Contrariamente a `MergeSort`, ne riceve un'informazione: l'indice intero `q` che l'elemento *pivot* va a occupare, separando i due sottovettori generati. Fatta la decomposizione del problema, si risolvono questi ricorsivamente, applicando la procedura al vettore `V`, rispettivamente da `s` a `q-1` e da `q+1` a `d`. Infine, la ricombinazione non richiede alcuna operazione.

```
/* Ordina il sottovettore del vettore V compreso fra s e d con
    l'algoritmo QuickSort */
void QuickSort (vint V, int s, int d)
{
  int q;

  if (s < d)
  {
    q = Partition(V,s,d);
    QuickSort(V,s,q-1);
    QuickSort(V,q+1,d);
  }
}
```

Passando alla procedura `Partition`, facciamo ancora la scelta di non costruire un prototipo, con l'idea che la procedura servirà solo qui. La procedura assume convenzionalmente come elemento *pivot* `V[s]`: lo lasceremo intatto fino all'ultimo e poi lo scambieremo con quello nella posizione `q` opportuna. Il resto del vettore andrà permutato in modo da portare al principio gli elementi minori o uguali e al termine quelli maggiori. In letteratura ci sono diverse versioni di `Partition`. L'idea comune a tutte è di gestire sul vettore `V` due tabelle e un residuo contenente gli elementi del vettore non ancora considerati. Le due tabelle, inizialmente vuote, crescono progressivamente considerando un elemento del residuo e spostandolo nella tabella appropriata in base al suo valore. Il residuo disordinato inizialmente contiene tutti gli elementi tranne il *pivot*. Alcune versioni mettono le due tabelle al principio del vettore, altre alla fine, altre ne mettono uno al principio e uno alla fine. Dal punto di vista asintotico, tutte le versioni hanno la stessa complessità. Le costanti moltiplicative dipendono molto dalla particolare permutazione dei numeri nel singolo vettore. La versione che descriviamo qui gestisce due tabelle nella parte iniziale del vettore, subito dopo l'elemento *pivot* `V[s]`. Queste tabelle, quindi, procedono da sinistra verso destra inglobando gli elementi non ancora considerati, che stanno a destra di entrambe le tabelle. Siccome alla fine le due tabelle diventeranno le soluzioni dei sottoproblemi, è naturale che la tabella degli elementi più piccoli preceda fin da subito quella degli elementi più grandi. È anche naturale scorrere il residuo da sinistra a destra, perché in questo modo l'elemento considerato lascia il posto per allargare le due tabelle. Meglio ancora, se ad allargarsi è la seconda tabella l'elemento può essere incluso in essa semplicemente allargandola, senza doverlo nemmeno spostare. Quando invece l'elemento considerato va messo nella prima tabella abbiamo il problema di dover scalare in avanti l'intera seconda tabella per fare spazio al nuovo elemento al termine della prima tabella. In realtà, non è necessario scalare davvero l'intera tabella: dato che l'ordine in ciascuna tabella è arbitrario, basta scambiare il primo elemento della seconda tabella con l'elemento considerato, allargare la prima tabella a includere l'elemento considerato e ridefinire il principio e il termine della seconda tabella un passo avanti. Quindi, in pratica, la prima tabella (quella degli elementi piccoli) continua ad allargarsi verso destra, mentre la seconda (quella degli elementi grandi) a volte si allarga e a volte invece scala spostando in avanti le posizioni del primo e dell'ultimo elemento.

Per essere chiari, dobbiamo fissare delle convenzioni per indicare principio e termine di ognuna delle due tabelle. La prima tabella comincia ovviamente in posizione `s+1`, quindi non ha bisogno di un indice esplicito. Finisce in una posizione che indicheremo con `d1` (si tratta quindi dell'ultimo elemento appartenente alla prima tabella). Al principio, la prima tabella è vuota, e quindi `d1` indicherà la posizione che precede la prima, cioè `s`. La seconda tabella comincia nella posizione immediatamente successiva (`d1+1`), per cui non occorre un indice esplicito. Termina in una posizione che precede immediatamente quella dell'elemento via via considerato. Quindi, ancora una volta, un indice esplicito non occorre davvero. Diciamo infine `i` l'elemento considerato ad ogni iterazione. Lo scorrimento parte dal secondo elemento dell'intero vettore (`s+1`) e arriva all'ultimo (`d`). Per ogni elemento, si tratta solo di capire se inserirlo nella prima o nella seconda tabella. Quando l'elemento è strettamente maggiore del *pivot* (`V[i] > V[s]`), va nella seconda tabella: questo comporta semplicemente di incrementare `i`, cosa che avviene comunque alla fine di ogni iterazione, per cui non si fa nulla. Quando invece l'elemento considerato non supera il *pivot* (`V[i] >= V[s]`), va nella prima tabella. Limitarsi ad allargare semplicemente la prima tabella incrementando `d1` la renderebbe scorretta, includendo un elemento maggiore del *pivot*, ma è sufficiente scambiare questo elemento `V[d1]` con l'elemento considerato `V[i]` per risolvere il problema. Questo scambio e l'incremento dei due cursori scala in avanti di un passo la seconda tabella. Quindi, secondo il caso, o la seconda tabella si allarga o la prima tabella si allarga e la seconda scorre in avanti. Comunque, la cosa richiede un numero di operazioni costante.

Non abbiamo ancora finito perché l'elemento *pivot* è ancora al principio del vettore, mentre dovrebbe trovarsi a cavallo fra le due tabelle. Per ottenerlo, basta scambiare l'elemento `V[s]` con l'elemento `V[d1]`, in modo che il *pivot* diventi l'ultimo elemento della prima tabella, a cui appartiene per definizione. Per concludere, possiamo restituire alla funzione chiamante l'indice occupato dall'elemento *pivot*, cioè `d1`. Rimarrebbe da scrivere la procedura `Scambia`, che però abbiamo già discusso più volte e ci limitiamo a copiare.

```
/* Partiziona il vettore V fra s e d in un sottovettore V[s,q] di
    elementi <= pivot
   e un sottovettore V[q+1,d] di elementi > pivot, pone il pivot fra
       i due sottovettori
   e restituisce l'indice finale del pivot */
int Partition (vint V, int s, int d)
{
  int d1; /* ultimo elemento della prima tabella (elementi <= pivot)
      */
  int d2; /* primo elemento a destra della seconda tabella  (elementi
      > pivot) */


  d1 = s;
  for (d2 = s+1; d2 <= d; d2++)
    if (V[d2] <= V[s])
    {
      d1++;
      Scambia(&V[d1],&V[d2]);
    }

  Scambia(&V[d1],&V[s]);

  return d1;
}
```

12.6 Ordinamento di vettori con elementi complessi

L'intero capitolo è stato dedicato, come si suole all'ordinamento di un vettore di numeri interi. Da un punto di vista pratico, però, può capitare di dover ordinare strutture più sofisticate. Per esempio, si può avere un elenco di persone da ordinare per cognome, per numero di matricola, per età, ecc...

La questione è quanto sia corretto trattare questi oggetti come se fossero dei semplici numeri interi. In particolare, abbiamo sempre ipotizzato che le operazioni di confronto e di scambio fra elementi richiedessero tempo costante. Ma in generale confrontare due oggetti complessi potrebbe richiedere un tempo superiore. Ancor più probabile è che il tempo richiesto a scambiarli sia superiore. Questo significa che la complessità degli algoritmi che abbiamo trattato non è sempre banalmente $\Theta(n \log n)$. Tanto per fare un esempio, ordinando vettori di stringhe il confronto potrebbe richiedere la procedura `strcmp`, la cui complessità è proporzionale alla lunghezza delle stringhe confrontate, dato che richiede di scorrerle. Se lo scambio avviene per copia esplicita, per esempio con la funzione `strcpy`, il tempo richiesto è, per lo stesso motivo, proporzionale alla lunghezza della stringa.

Anche nel caso in cui gli oggetti hanno una dimensione costante (per esempio, stringhe di lunghezza limitata), ignorare completamente la loro dimensione ha senso solo dal punto di vista dell'analisi teorica, ma non dell'implementazione, perché una costante moltiplicativa di valore grande ha comunque un impatto sul tempo complessivo di esecuzione da non trascurare.

Un altro aspetto degno di nota è il fatto che gli algoritmi di ordinamento considerati sono tutti *in place*, cioè sfruttano la memoria occupata dai dati, e quindi distruggono il vettore originale. Qualora l'ordinamento iniziale sia utile, o qualora gli oggetti del vettore vadano ordinati più volte in modo diverso, distruggere tutti gli ordinamenti tranne l'ultimo ha poco senso. Una possibile soluzione è creare copie multiple del vettore, una per ogni ordinamento utile. Questa soluzione può funzionare per vettori di oggetti la cui dimensione è paragonabile a quella di un numero intero. Altrimenti, spreca spazio (per conservare informazioni spesso ridondanti, dato che non concernono lo specifico ordinamento) e tempo (per le operazioni di copia).

La vera soluzione è non ordinare direttamente gli oggetti, ma farlo indirettamente. Si tratta di costruire dei vettori di "posizioni" e ordinare queste ultime. Vedremo un esempio concreto nel Capitolo **??**. Qui ne parliamo in termini generali. Abbiamo già declinato il concetto astratto di posizione in due modi nel Capitolo 7: come puntatore, cioè indirizzo in memoria, e come cursore, cioè indice intero. In entrambi i casi, si tratta di costruire vettori di oggetti che hanno dimensioni limitate (paragonabili o identiche a un numero intero) e usarli per accedere, quando necessario, ai veri e propri oggetti, nel vettore originario. Quindi, gli oggetti originali vengono letti per eseguire i confronti senza moltiplicare informazioni ridondanti, e non vengono mai modificati, perché gli scambi vengono eseguiti sui vettori ausiliari di posizioni.

Il vantaggio in termini di spazio e di tempo viene parzialmente pagato in termini di tempo per l'accesso indiretto ai dati durante i confronti. Supponendo di avere un vettore ausiliario `A` di indici interi e indicando per semplicità il confronto con l'operatore `<=` anche se in pratica dovrebbe essere qualcosa di più complicato, il test `V[i] <= V[s]` dovremo essere sostituito da `V[A[i]] <= V[A[s]]`. Si tratta di un incremento di complessità praticamente trascurabile, se il risparmio ottenuto altrove è interessante.

Analogamente, gli scambi dovrebbero operare non sugli elementi del vettore `V`,

ma su quelli del vettore `A`. In questo caso, non vi sono aggravi.

Tutto questo riguarda tutti gli algoritmi descritti in questo capitolo, ma anche quelli quadratici discussi nel Capitolo 6. Può meritare una menzione il fatto che la presenza di un vettore ausiliario si riflette anche nei prototipi delle varie procedure ausiliarie introdotte in ciascun caso, ma questo avviene in maniera abbastanza ovvia da non doverne parlare in dettaglio.

12.7 Ordinamento su strutture diverse da vettori

Si è sempre assunto che i dati da ordinare fossero in un vettore. Che cosa succede se invece sono in una lista o albero o qualsiasi altro oggetto?

Tutti gli algoritmi descritti non sono direttamente applicabili, o meglio lo sono, ma le singole operazioni non hanno la complessità che abbiamo ipotizzato. È un esercizio utile su quali operazioni cambierebbero e come. D'altra parte, l'idea di costruire un vettore di puntatori e operare su di esso rimane del tutto valida. Non vale quella di usare un vettore di indici interi, dato che non c'è un vettore base a cui riferirsi.

sè

12.8 Uno schema riassuntivo

Tutti gli algoritmi di ordinamento descritti sinora condividono, nelle loro differenze, uno schema fondamentale che non è ovvio a prima vista (vedi Figura 12.8). Gli algoritmi quadratici *InsertionSort* e *SelectionSort*, infatti, si basano sull'idea di allargare progressivamente un sottovettore ordinato aggiungendovi un singolo elemento. È in effetti possibile pensare a questi due oggetti come a sottovettori ottenuti da una divisione molto sbilanciata del vettore originale. Al contrario, gli algoritmi efficienti *MergeSort* e *QuickSort* utilizzano una divisione bilanciata (perfettamente il primo, meno il secondo).

Detto ciò, *InsertionSort* e *MergeSort* condividono il fatto che la divisione del problema è banale: nel primo algoritmo, l'elemento tenuto da parte e aggiunto per ultimo agli altri è l'ultimo del vettore di partenza; nel secondo, i due sottovettori sono le due metà del vettore originale. Questo viene pagato con una procedura complessa di ricostruzione della soluzione.

D'altra parte, *SelectionSort* e *QuickSort* condividono il fatto che la divisione del problema è invece sofisticata: nel primo algoritmo, l'elemento tenuto da parte è per aggiungerlo alla fine è il minimo; nel secondo algoritmo, i due sottovettori sono separati in modo che siano giè ordinati l'uno rispetto all'altro. Il vantaggio si ha in fase di combinazione, dato che le due parti del problema vengono banalmente accodate l'una all'altra.

Per quanto riguarda *HeapSort*, già sappiamo che si tratta semplicemente di una variante di *SelectionSort* resa efficiente da un'opportuna struttura dati ausiliaria.

12.9 Esercizi

12.9.1 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/tabel-

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 12.22: Uno schema mnemonico

le) in uno solo senza elementi ripetuti.

12.9.2 Esercizio

Ispirandosi alla funzione *Merge*, si realizzi una funzione *Intersection*, che determina l'intersezione di due insiemi ordinati (rappresentati da vettori/tabelle).

12.9.3 Esercizio

Ispirandosi alla funzione *Merge*, si realizzi una funzione *SetDifference*, che determina l'insieme differenza di due insiemi ordinati (rappresentati da vettori/tabelle).

12.9.4 Esercizio

Si ponga in forma iterativa la funzione di aggiornamento di un *heap*.

12.9.5 Esercizio

Si definisca una funzione per aggiungere un nuovo elemento a un *heap*, eventualmente riallocando la struttura dati, qualora quella corrente avesse esaurito la memoria.

12.9.6 Esercizio

Si definisca una funzione per cancellare da un *heap* un elemento la cui posizione sia nota.

12.9.7 Esercizio

Si definisca una funzione per modificare in un *heap* il valore di un elemento la cui posizione sia nota.

12.9.8 Esercizio

Si realizzi una libreria per la gestione di *heap* in forma completamente astratta.

12.9.9 Esercizio

Ispirandosi alla funzione *QuickSort*, si realizzi una funzione *QuickSelect* che, dato un vettore e un indice i, determina l'i-esimo elemento del vettore in ordine non decrescente. (Suggerimento: dato un elemento *pivot*, il vettore viene partizionato in due parti, almeno una delle quali è certamente non contiene l'elemento cercato)

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)

13.1 Il problema dello zaino

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*. Partiamo da un'ipotetica situazione pratica. Vogliamo aiutare un ladro entrato in una gioielleria ad uscirne col massimo bottino possibile. La gioielleria contiene un insieme di oggetti preziosi, per ognuno dei quali è noto il valore e il volume. Il ladro ha uno zaino di capacità nota, 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). Dobbiamo suggerire quali oggetti prendere per rispettare il vincolo massimizzando il valore totale del sottoinsieme di oggetti scelti.

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

[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.

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.

Dal punto di vista formale, abbiamo un insieme base B di oggetti, una funzione volume v, che associa ad ogni oggetto un numero naturale, un numero naturale V, che è la capacità dello zaino, e una funzione ϕ che associa ad ogni oggetto il suo valore, ancora una volta un numero naturale. La regione ammissibile è costituita da tutti i sottoinsiemi di oggetti il cui volume totale, cioè la somma dei volumi dei singoli oggetti, non supera il valore della capacità. L'obiettivo che ci proponiamo di massimizzare è una funzione $f(x)$ additiva, cioè pari alla somma dei valori di una funzione ausiliaria (il valore ϕ) sui singoli oggetti della soluzione x, cioè del sottoinsieme che stiamo considerando: $f(x) = \sum_{j \in x} \phi_j$. Siccome le soluzioni sono insiemi di oggetti contenuti in B, possiamo concludere che il problema è di Ottimizzazione Combinatoria. Come risolverlo?

In questo capitolo, lo risolveremo in maniera esatta, con vari algoritmi che hanno molto in comune fra loro, pur essendo diversi. Il capitolo seguente tratterà due modi euristici di risolverlo. Gli algoritmi sviluppati qui saranno non polinomiali, gli altri invece saranno polinomiali.

Consideriamo un esempio per familiarizzare col problema (vedi Figura13.4). Abbiamo 6 oggetti, indicati con le lettere da a a f. Ciascuno ha un valore e un volume, riportati in una tabella, e la capacità è fissata al valore 8. Il sottoinsieme $x' = \{c, d, e\}$ è una soluzione ammissibile, perché la somma dei volumi dei suoi elementi è pari a $v(x') = 6 \leq V$. Il suo valore è la somma dei valori dei singoli elementi, $f(x') = 13$. Vedremo che la soluzione ottima vale 15. Il sottoinsieme $x'' = \{a, c, d\}$ vale $f(x) = 16$, ma non è una soluzione ammissibile, perché la somma dei volumi dei suoi elementi è pari a $v(x'') = 10 > V$. A volte, si usa impropriamente chiamare "soluzione" anche i sottoinsiemi inammissibili. In tal caso, bisogna allora sempre precisare se si tratta di soluzione ammissibile o di soluzione inammissibile.

Torniamo alla questione di come risolvere il problema. Esiste un algoritmo ricorsivo abbastanza semplice, descritto nella Figura 13.5. La sua idea fondamentale è che ogni istanza del problema dello zaino corrisponde a una quaterna di oggetti matematici: l'insieme di oggetti B, le funzioni ϕ e v definite su B e il numero intero non negativo V. Ora, questa istanza si può in effetti ricondurre a due altre istanze più "piccole", nel senso che sono sottoproblemi risolti i quali siamo in grado di risolvere l'istanza di partenza.

La riduzione consiste nello scegliere un elemento qualunque dell'insieme B (per esempio, l'ultimo, quello di indice n). Tale elemento o fa parte della soluzione ottima o non ne fa parte. Se consideriamo il caso in cui l'elemento non fa parte della soluzione, cancellarlo dall'insieme B non ha nessun effetto sulla soluzione ottima, che conserva il suo valore e non può essere sostituita da soluzioni migliori, che sarebbero state migliori in partenza, negando l'ottimalità. Cancellare l'elemento n significa considerare l'istanza con insieme base $B_{n-1} = B \setminus \{n\}$, cioè gli oggetti con indice compreso fra 1 e $n-1$. Le funzioni valore ϕ e volume v vanno sostituite dalle loro restrizioni al nuovo insieme, che per semplicità continuiamo a chiamare con gli stessi nomi (basta B_{n-1} per identificarle univocamente). Infine, la capacità dello zaino rimane uguale. Supponiamo di saper risolvere all'ottimo questo problema: la sua soluzione ottima è anche la migliore fra quelle del problema di partenza che non contengono l'oggetto n. In poche parole, se si suppone di rifiutare l'oggetto n,

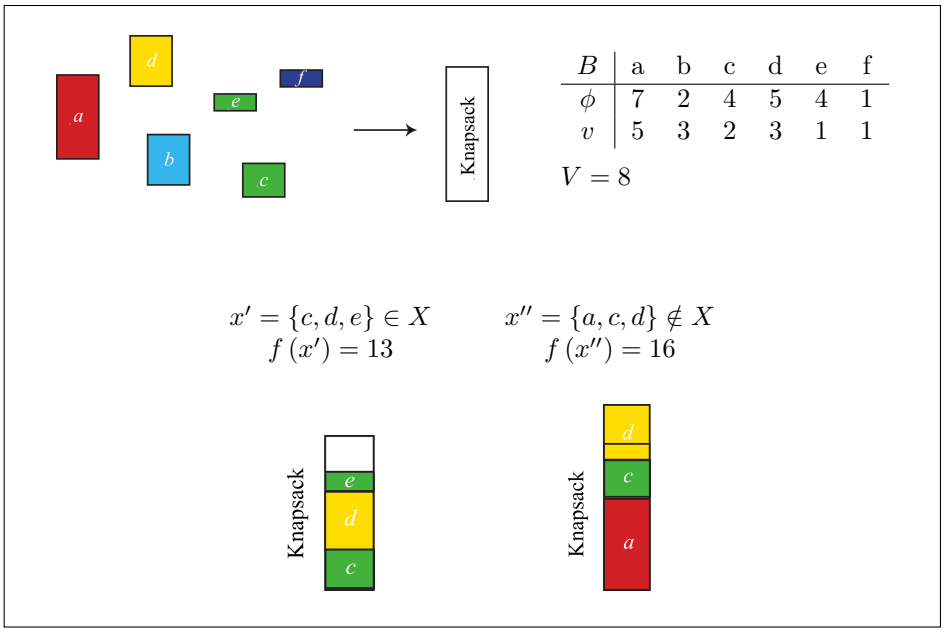


Figura 13.4: Esempio

non occorre più risolvere il problema con n elementi, ma basta risolvere quello con lo stesso zaino e $n-1$ elementi. Il nuovo problema è evidentemente più piccolo. Questo è banale e chiaramente insufficiente, perché ogni tanto capiterà di prendere l'oggetto n, altrimenti si continuerebbe ricorsivamente a cancellare elementi fino a ridursi ad una soluzione completamente vuota.

Consideriamo quindi il caso in cui l'elemento n fa parte della soluzione ottima del problema. In tal caso, l'oggetto viene messo nello zaino e ci troviamo di fronte a un problema di zaino con $n-1$ elementi, con lo stesso insieme base B_{n-1} e le stesse restrizioni del primo caso, ma con uno zaino parzialmente occupato. Quest'ultimo, però equivale a uno zaino di capacità più piccola, ridotta esattamente del volume $v(n)$ occupato dall'oggetto preso. Trovata la soluzione ottima di questo problema ridotto, possiamo recuperare la soluzione ottima del problema originale: basta aggiungerle l'oggetto che è già nello zaino iniziale.

Note le due soluzioni dei sottoproblemi, la migliore delle due è la soluzione ottima del problema. Dunque, per ogni istanza del problema dello zaino, possiamo costruire due istanze più piccole, ancora del problema dello zaino, tali che le loro soluzioni consentono di ottenere la soluzione ottima dell'istanza data: è la migliore fra la soluzione in cui si rifiuta l'oggetto n e e quella in cui l'oggetto n viene accantonato, si riduce la dimensione dello zaino e si riaggiunge nuovamente l'oggetto stesso.

Questo ragionamento è riassumibile nell'equazione ricorrente riportata nella Figura 13.7. Contrariamente alle equazioni discusse nel Capitolo 12, non riguardano il tempo di calcolo di un algoritmo ricorsivo, ma la soluzione ottima di un problema. Tuttavia, hanno la stessa struttura e si affrontano allo stesso modo. L'equazione

$$f^*(n,V) = \begin{cases} 0 & \text{se } n = 0 \\ f^*(n-1,V) & \text{se } n > 0 \text{ e } v_n > V \\ \max\left[f^*(n-1,V), \phi_n + f^*(n-1,V-v_n)\right] & \text{se } n > 0 \text{ e } v_n \leq V \end{cases}$$

ha ha un caso base e due casi ricorsivi.

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 13.5: Un algoritmo ricorsivo per il problema dello zaino

Il caso base è quello in cui l'insieme di oggetti B è vuoto, quindi $n = 0$. In questo caso, la soluzione ottima è anche l'unica, ed è quella vuota, che ha valore nullo. I casi ricorsivi sono due perché nell'analisi precedente abbiamo ipotizzato di cancellare l'oggetto n o di metterlo nello zaino, ma non è detto che il secondo caso sia sempre possibile. Quando il volume dell'oggetto eccede la capacità dello zaino, c'è un solo caso: siamo costretti a cancellare l'oggetto. Quando non la eccede, rimangono i due casi già discussi e confrontiamo le soluzioni ottime dei relativi sottoproblemi per determinare la migliore. In una manteniamo l'intera capacità, ma rinunciamo al valore dell'oggetti; nell'altra, guadagnamo il valore dell'oggetto, ma rinunciamo alla capacità corrispondente al suo volume.

L'equazione ricorrente in effetti si riferisce soltanto a due valori interi, mentre le istanze sono quaterne di oggetti che contengono un insieme, due funzioni e un intero. Il motivo è che l'insieme è individuato univocamente dalla sua cardinalità (esattamente come il numero di vertici o nodi di un grafo individuava l'intero insieme), una volta fissata una qualsiasi permutazione, e che le funzioni sono univocamente individuate dalle funzioni iniziali, definite su B, e dall'insieme ristretto. Dal punto di vista informatico, le funzioni saranno rappresentate come vettori, e quindi restringerle significa semplicemente aggiornare via via il sottovettore utilizzato, dunque ancora una volta la cardinalità.

Tutto questo suggerisce il semplice algoritmo risolutivo descritto nella Figura 13.7. Se $n = 0$, il valore ottimo è nullo. Se l'oggetto n ha volume superiore alla capacità V, applichiamo ricorsivamente l'algoritmo all'istanza ridotta con $n-1$ oggetti, gli stessi vettori ϕ e v (automaticamente ridotti dalla riduzione di n) e la stessa capacità. Negli altri casi, risolveremo separatamente il problema con $n-1$ oggetti, le stesse funzioni e la stessa capacità e il problema con $n-1$ oggetti, le stesse funzioni e una capacità ridotta di un fattore v_n dove poi poi sommeremo al risultato il premio ϕ_n. La migliore di queste due soluzioni sarà la soluzione complessiva.

Notiamo che questo algoritmo restituisce il valore della soluzione, non la so-

$$f^*(n,V) = \begin{cases} 0 & \text{se } n = 0 \\ f^*(n-1,V) & \text{se } n > 0 \text{ e } v_n > V \\ \max\left[f^*(n-1,V), \phi_n + f^*(n-1,V-v_n)\right] & \text{se } n > 0 \text{ e } v_n \leq V \end{cases}$$

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 13.6: Un algoritmo ricorsivo per il problema dello zaino

luzione stessa, il che è un difetto. È possibile risolvere il problema, ma poiché miglioreremo immediatamente l'algoritmo, che è molto inefficiente, non è il caso di affrontare subito l'argomento.

13.2 Punto di partenza

Andiamo quindi a prendere i materiali di partenza, cioè il file listato `knapsack0.c`. Il codice è un po' più ricco che nei capitoli precedenti. Abbiamo la classica funzione che interpreta la linea di comando caricando in una stringa il nome del file dei dati.

Abbiamo le direttive che includono le librerie per gestire input/output e stringhe, la definizione di una lunghezza massima per le stringhe, il tipo logico, i vettori di numeri interi e di valori logici

```
/* knapsack.c */

/* Direttive */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>


#define ROW_LENGTH 256

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

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

```
typedef int * vint;
typedef vint * mint;
```

Questa volta abbiamo anche la definizione del tipo per i dati e per le soluzioni, cioè le strutture scelte per rappresentare istanze e soluzioni del problema dello zaino. Le istanze sono definite da un record `datiKP`, contenente una quaterna di oggetti.

```
typedef struct _datiKP datiKP;
struct _datiKP
{
  int n;
  int V;
  vint phi;
  vint v;
};
```

La soluzione del problema è descritta da un record di tipo `soluzioneKP`, che include il valore intero `f` dell'obiettivo e (dato che si tratta di un sottoinsieme) un vettore di incidenza logico `x`, definito sugli oggetti, che indica per ciascuno se appartiene o no alla soluzione.

```
typedef struct _soluzioneKP soluzioneKP;
struct _soluzioneKP
{
  int f;
  vboolean x;
};
```

A priori, avremmo potuto anche usare una tabella: la scelta è in realtà motivata dagli algoritmi che dovremo implementare. Rispetto alla tabella, il vettore di incidenza occupa più spazio e rende lo scorrimento della soluzione più lento. In entrambi i casi, si tratta di una complessità lineare in n, non terribile. Inoltre, il vettore di incidenza è molto più veloce della tabella nelle aggiunte, cancellazioni e nei test di appartenenza.

Sono anche già presenti una funzione che carica i dati da file, una che li distrugge liberando la memoria, una funzione che crea una soluzione vuota e una che la distrugge. Infine, è data una funzione che stampa una soluzione. Sono tutte funzioni molto simili a quelle che abbiamo via via introdotto per gestire le varie strutture dati astratte. Ci si potrebbe chiedere allora perché non farne subito una libreria per gestire problemi di zaino. La risposta è che questo è lasciato come esercizio alla fine del capitolo.

```
/* Carica da filedati l'istanza I del problema dello zaino */
void CaricaDati (char *filedati, datiKP *pI);

/* Distrugge l'istanza I */
void DistruggeDati (datiKP *pI);

/* Crea una soluzione vuota */
void CreaSoluzione (int n, soluzioneKP *pS);

/* Distrugge la soluzione S */
void DistruggeSoluzione(soluzioneKP *pS);

/* Stampa la soluzione */
void StampaSoluzione (int n, soluzioneKP *pS);
```

Formati di ingresso e uscita Abbiamo quattro file di esempio a disposizione, che prendono il nome dal numero di elementi. Il primo contiene 6 elementi, ed è l'esempio rappresentato in figura. Poi vi sono file da $n = 5$, 10, 24 e 50 oggetti. Il file da 24 elementi ha la caratteristica che i suoi valori numerici per valori e volumi sono piuttosto grandi, mentre per gli altri file i valori non superano le tre cifre.

Il formato 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 =` . Per esempio, il primo file è il seguente.

```
n = 6
phi = 7 2 4 5 4 1
v = 5 3 2 3 1 1
V = 8
```

In effetti, al termine i file contengono altre due righe, che non rappresentano i dati, ma la soluzione ottima del problema, che è stata precalcolata per poterla confrontare con le soluzioni ottenute via via dai vari algoritmi che implementeremo.

La funzione di caricamento dei dati legge solo le prime quattro righe, per cui nelle righe successive si può scrivere quello che si vuole senza fare danni. Ovviamente, si potevano conservare le soluzioni in altri file appositi, e in situazioni più complicate sarebbe probabilmente stato meglio.

La procedura che stampa le soluzioni segue il formato utilizzato in queste ultime due righe. Per prima cosa, stampa a video il valore dell'ottimo del problema, preceduto dalle parole chiave `f* =`. Quindi, stampa in una seconda riga la soluzione ottima, rappresentata come elenco degli indici degli oggetti che ne fanno parte, ordinati per valori crescenti, separati da spazi, racchiusi fra parentesi tonde e preceduti dalle parole chiave `x* =`. Il risultato nel caso del primo esempio è:

```
f* = 15
x* = ( 1 3 5 )
```

13.3 Un algoritmo ricorsivo elementare

A questo punto, possiamo scrivere la procedura ricorsiva di risoluzione basata sull'equazione ricorrente, in modo da determinare il valore ottimo delle quattro istanze disponibili. Non ci interessiamo per ora di determinare anche la soluzione, che quindi rimarrà vuota, dando luogo a un'incoerenza tra valore dell'obiettivo e vettore di incidenza.

Contrariamente a quanto faremo per gli algoritmi seguenti, l'algoritmo non riceve il record che racchiude i dati e non restituisce il record che contiene la soluzione, che sarebbe la forma più chiara, ma riceve separatamente i singoli dati e restituisce solo il valore della soluzione. Questo perché l'algoritmo, oltre a non restituire la soluzione completa, è ricorsivo, cioè ad ogni chiamata richiede di ricevere dati leggermente modificati. Se ad ogni chiamata (e nel nostro caso ce ne possono essere anche due ad ogni singola esecuzione dell'algoritmo ricorsivo), dovessimo costruire il record dei dati del problema, avremmo bisogno di allocare la memoria relativa a due interi e due vettori, copiarvi i dati dell'istanza padre, modificarli opportunamente e poi passarli alla chiamata ricorsiva. Al termine, dovremmo deallocare tutti questi dati. È chiaro che il procedimento, per quanto formalmente corretto, sarebbe estremamente inefficiente. Se possibile, è meglio evitare tutto questo. Come fare?

In genere, i problemi figli sono molto simili al problema padre. Prendiamo quindi i dati del problema padre, modifichiamo quel poco che va cambiato e passiamoli alla chiamata ricorsiva. Al termine, ripreso il controllo, ricostruiremo i dati originali. Nel caso specifico, questo significa che la cardinalità n va decrementata e a volte la capacità V va ridotta, mentre valori ϕ e volumi v rimangono identici, perché decrementare n automaticamente trasforma le funzioni nella corretta restrizione. All'uscita dovremo recuperare i dati originali del problema padre, cioè reincrementare n ed eventualmente ripristinare la capacità V.

Nel caso specifico, però, si può fare ancora di meglio: si può sfruttare il fatto che i due dati che cambiano sono interi passati per valore, e quindi automaticamente copiati dal meccanismo di chiamata in altre celle. Quindi, possiamo direttamente passare come argomenti i nuovi valori di n e V senza modificare quelli originali, e quindi senza doverli ricostruire al termine. Per questo, anziché passare come argomento all'algoritmo l'intera istanza `I`, passiamo separatamente i quattro campi, in modo da poter modificare i due che è necessario adattare senza avere effetti sui dati originali. Ne deriva una dichiarazione in cui i dati rimangono disaggregati.

```
/* Algoritmo ricorsivo per il problema dello zaino */
int AlgoritmoRicorsivoKP (int n, vint phi, vint v, int V);
```

E ne deriva una definizione che ripercorre i passi dello pseudocodice già visto, costruendo i due sottoproblemi in modo quasi invisibile, dato che la costruzione avviene nel corpo stesso della chiamata.

```
/* Algoritmo ricorsivo per il problema dello zaino */
int AlgoritmoRicorsivoKP (int n, vint phi, vint v, int V)
{
  int phi0, phi1;

  if (n == 0)
    return 0L;
  else if (v[n] > V)
    return AlgoritmoRicorsivoKP(n-1,phi,v,V);
  else /* (v[n] <= V) */
  {
    phi0 = AlgoritmoRicorsivoKP(n-1,phi,v,V);
    phi1 = phi[n] + AlgoritmoRicorsivoKP(n-1,phi,v,V-v[n]);
    return ( (phi0 > phi1) ? phi0 : phi1 );
  }
}
```

13.4 Misurazione del tempo di calcolo

Per la prima volta abbiamo introdotto anche variabili e operazioni che servono a registrare il tempo di calcolo. Fra le direttive, infatti, compare l'inclusione della libreria `time.h`, fra le variabili del `main` compaiono le variabili `inizio` e `fine`, entrambe di tipo `clock_t` e la variabile reale `tempo`. Infine, prima e dopo la chiamata alla funzione che risolverà il problema compaiono delle chiamate alla funzione `clock()`. Tale funzione determina il numero di unità elementari di tempo (di cui diremo altro in seguito) trascorse dall'inizio dell'esecuzione del programma. Si tratta in sostanza di un numero intero positivo, cui viene dato però il tipo `clock_t` per distinguerlo dai semplici numeri interi. Questo numero è nullo al principio del programma cresce via via durante l'esecuzione. Registrando questo numero prima e dopo l'esecuzione del vero e proprio algoritmo risolutivo, possiamo valutare quanto tempo quest'ultimo abbia richiesto valutando la differenza tra i due valori. Siccome, però, il tempo è misurato in unità diverse dai secondi, per tradurlo in secondi occorre dividere la differenza per il numero di unità elementari contenute in un secondo. Questo numero è fornito dalla costante simbolica `CLOCKS_PER_SEC`, che è fornita dalla libreria `time.h`. Dato che i numeri iniziali sono valori interi, vengono convertiti in numeri reali prima della divisione, in modo da eseguire la divisione reale, e non quella intera, che troncherebbe il resto. Il valore di `CLOCKS_PER_SEC` dipende dalla macchina utilizzate, e i valori più tipici sono 1 000 e 1 000 000. Dunque, sostanzialmente l'unità elementare di tempo è in genere un millisecondo o un microsecondo, ma non occorre preoccuparsene esplicitamente. Determinato il numero di secondi richiesti dall'algoritmo, possiamo stamparlo a video come informazione utile sulla prestazione dell'algoritmo.

Applichiamo tutto questo alle quattro istanze disponibili. Ovviamente, le soluzioni, che non abbiamo calcolato, risultano vuote, ma i valori sono corretti. È ora interessante osservare il valore del tempo di calcolo. Le prime due istanze vengono risolte in meno di un microsecondo. La terza si risolve in pochi centesimi di secondo, che è pochissimo, ma molto più delle precedenti. L'ultimo problema, che ha poco più del doppio degli elementi rispetto al precedente, impiega un tempo nettamente superiore: parecchi minuti. Qual è la complessità di questo algoritmo?

L'equazione ricorrente non è ovvia, ma è abbastanza evidente che ad ogni decremento unitario del numero di oggetti il problema corrente ne genera uno o due. Se il caso pessimo è frequente, cioè finché la capacità è abbastanza grande da contenere gli oggetti, ogni livello di ricorsione raddoppia il numero dei problemi. Per scendere da n oggetti a nessuno occorrono n livelli. Anche se in ogni chiamata le operazioni elementari sono in numero costante (confronti, somme, differenze), i sottoproblemi $O(2^n)$. L'algoritmo è esponenziale.

Proviamo a fare una stima molto approssimata. Con istanze di dimensione $n = 6$ o $n = 10$, i problemi potrebbero essere $\approx 2^6 = 64$ o $\approx 2^{10} = 1024$ e le operazioni elementare un numero proporzionale a questi. Se un calcolatore della frequenza di alcuni GigaHertz esegue miliardi di operazioni al secondo, alcune migliaia di operazioni richiedono al massimo microsecondi. Un problema di dimensione $n = 24$ richiederebbe al più $2^{24} = 16\,777\,216$ problemi, dunque centesimi di secondo. Un problema di dimensione $n = 50$ richiederebbe al più $2^{50} \approx 10^{15}$ problemi, dunque milioni di secondi, cioè parecchi giorni. In realtà, il tempo richiesto dalla nostra istanza è "solo" di vari minuti, perché molti problemi hanno una capacità abbastanza piccola da generare un solo problema piccolo, anziché due. Infatti, i volumi sono espressi da numeri di una o due cifre e la capacità ne ha tre, per cui in genere le soluzioni raramente arrivano a contenere 50 elementi. L'aspetto fondamentale, comunque, è che raddoppiare la dimensione dell'istanza non porta a un raddoppio del tempo di calcolo, ma una crescita molto più veloce. È chiaro che questo tipo di algoritmo non è valido.

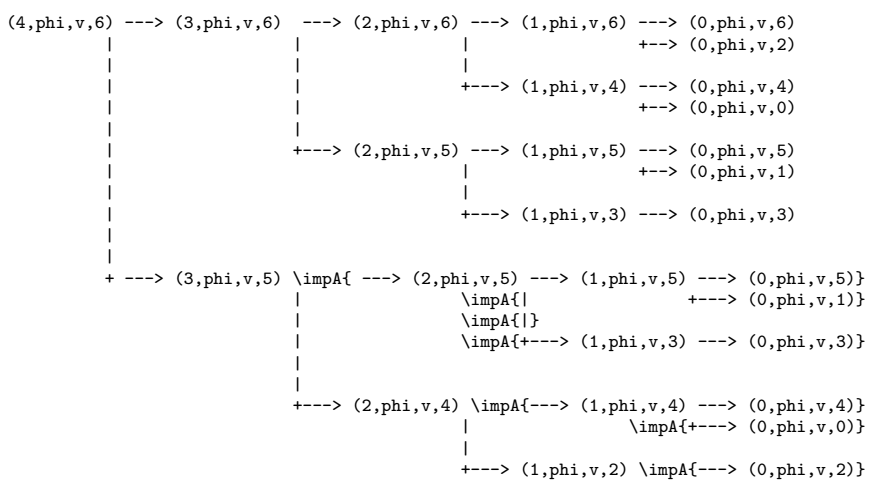
13.5 Seconda fase (`knapsack2.c`): programmazione dinamica "top-down"

Per migliorare l'algoritmo descritto in precedenza, bisogna capire dov'è il suo difetto principale. Per vederlo, consideriamo un'istanza ulteriormente ridotta a 4 oggetti (vedi Figura 13.8). L'algoritmo ricorsivo genera un albero di ricorsione nel quale ogni nodo è un sottoproblema affrontato dall'algoritmo e ridotto a uno o due sottoproblemi figli, a meno che non si raggiunga il livello base in cui il problema è vuoto, perché allora il problema è banalmente risolto. Ogni nodo dell'albero riporta il sottoproblema corrispondente, cioè la quaterna (n, ϕ, v, V). Per esempio, il problema alla radice $(4, \phi, v, 6)$ genera il sottoproblema $(3, \phi, v, 6)$ nell'ipotesi di rifiutare l'elemento 4 e il sottoproblema $(3, \phi, v, 5)$ nell'ipotesi di accettarlo. È evidente che l'algoritmo continua a generare e risolvere ripetutamente gli stessi sottoproblemi, il che è ovviamente uno spreco di tempo. Per esempio, il problema $(2, \phi, v, 5)$ compare due volte, generando due volte tutti i suoi sottoproblemi. Questo produce interi sottoalberi completamente inutili, ognuno dei quali tende a esplodere combinatoricamente. Nel caso specifico l'aggravio è limitato (10 problemi su 28), ma al crescere delle dimensioni la frazione occupata dai doppioni tende a diventare sempre più grande. Come si può evitare tutto questo?

L'idea di fondo della programmazione dinamica è abbastanza semplice (vedi Figura 13.9): ogni volta che si risolve un problema, si salva la soluzione in un'opportuna struttura, nella speranza che in un momento futuro il problema si ripresenti e non sia necessario risolverlo di nuovo, ma basti consultare la struttura e ricavarne la soluzione. Di conseguenza, ogni volta che si genera un nuovo problema, prima di provare a risolverlo si consulta la struttura, per vedere se già contiene la soluzione. Il modo in cui questa idea viene realizzata dà luogo due grandi categorie di algoritmi, o meglio di implementazioni degli stessi algoritmi, definite *top-down* e *bottom-up*.

È un algoritmo molto inefficiente: risolve ripetutamente problemi uguali

Per esempio, $\phi = [\ 6\ \ 8\ \ 3\ \ 2\ ]$, $v = [\ 4\ \ 2\ \ 1\ \ 1\ ]$ e $V = 6$



10 *problemi uguali su* 28

Figura 13.8: Cause dell'inefficienza dell'algoritmo ricorsivo

La modalità *top-down* corrisponde all'implementazione di un comune algoritmo ricorsivo, accompagnato dalla gestione di una struttura che conservi le soluzioni dei sottoproblemi. Ogni volta che si considera un sottoproblema, per prima cosa si interroga la struttura ausiliaria per determinare se il sottoproblema è già stato risolto. In caso positivo, si recupera e si usa la soluzione. In caso negativo, il sottoproblema viene risolto compiutamente e la soluzione viene aggiunta alla struttura ausiliaria. Un altro modo di vedere il concetto è che, invece di avere dei casi base elementari e dei casi ricorsivi, abbiamo anche l'ulteriore caso base nel quale il problema è non elementare, ma già risolto, per cui la ricorsione si arresta. Tutto questo può far risparmiare tempo (se davvero i sottoproblemi si ripetono), ma ovviamente costa spazio aggiuntivo: lo spazio richiesto dalla struttura ausiliaria.

Che genere di struttura potrebbe essere? La risposta dipende dal problema, ma in generale deve consentire di riconoscere un sottoproblema e accedere alla soluzione velocemente. Le strutture ideali da questo punto di vista sono vettori e matrici, dove l'accesso in lettura e scrittura è in tempo costante. Nel caso dello zaino, un sottoproblema è da due numeri interi, il primo dei quali è il numero n di oggetti (compreso fra 0 e $|B|$), mentre il secondo è la capacità V (compresa fra 0 e la capacità dello zaino iniziale). Quindi, ogni sottoproblema teoricamente possibile è facilmente rappresentabile in una matrice con $V+1$ righe e $n+1$ colonne (o viceversa). Nell'esempio, quindi, la matrice avrebbe 7 righe e 5 colonne. I sottoproblemi sono in realtà meno di $(V+1)(n+1)$: sono 28, anziché 35, perché alcuni sottoproblemi non possono essere ottenuti in alcun modo usando gli oggetti che compongono il problema complessivo. Per esempio, se vi sono tutti e 4 gli oggetti, la capacità deve essere pari a V.

Una volta realizzata la matrice, l'albero di ricorsione viene ridotto accorpando i nodi che corrispondono allo stesso sottoproblema, ovvero alla stessa cella della

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 13.9: Programmazione dinamica

matrice. Non è più un albero, ma un grafo orientato aciclico, dato che i nodi possono essere raggiunti dalla radice seguendo più cammini diversi, ma gli archi vanno comunque sempre da un livello al successivo. Quindi l'esplosione combinatorica che potenzialmente raddoppiava il numero dei nodi ad ogni livello viene compressa dal fatto che in ogni livello non possono esserci più di $V + 1 = 7$ nodi.

13.5.1 Implementazione

Per realizzare l'algoritmo di programmazione dinamica *top-down*, partiamo dall'algoritmo ricorsivo realizzato in precedenza. Lo modifichiamo per tener conto del fatto che deve sfruttare un'informazione ausiliaria racchiusa in una matrice. Le modifiche non sono tanto concettuali quanto di programmazione e struttura del codice. L'algoritmo ricorsivo riceverà i dati del sottoproblema corrente, ma anche la matrice ausiliaria. Cominciamo a conservare nella matrice solo il valore dell'obiettivo, ignorando la soluzione, come abbiamo fatto prima. La matrice dovrà quindi solo salvare un valore intero per ogni sottoproblema, e sarà una semplice matrice dinamica di interi.

```
/* Algoritmo ricorsivo per il problema dello zaino */
```

La matrice andrà ovviamente allocata all'inizio (prima della chiamata sul problema originale) e deallocata alla fine (dopo aver risolto il problema originale). Cominciamo a incapsulare tutto questo in una procedura `PDricorsivaKP`, che riceverà i dati e restituirà la soluzione. Riceve i dati per indirizzo, non perché vadano

modificati, ma per efficienza, cioè per ridurre il passaggio dei dati alla copia di un puntatore. Riceve la soluzione per indirizzo, perché è un finto dato, in realtà un risultato composito da modificare. Questa è, del resto, la struttura che ci eravamo proposti di ottenere sin dal principio per l'algoritmo risolutivo. Avevamo rinunciato a farlo solo per semplificare la forma della procedura ricorsiva, distinguendo i dati da modificare da quelli che rimangono uguali nella costruzione dei sottoproblemi.

```
/* Programmazione dinamica ricorsiva per il problema dello zaino */
void PDricorsivaKP (datiKP *pI, soluzioneKP *pS);
```

Una cosa che merita discussione è se l'algoritmo debba non solo riempire la soluzione ma anche eventualmente allocare le sue strutture dati. Que sto potrebbe avere senso, ma in genere non si fa: si preferisce distinguere la creazione di una soluzione vuota dalla costruzione di una vera soluzione a partire da quella vuota, ovvero la gestione della struttura dati dalla vera e propria risoluzione del problema. I motivi sono vari. Uno è che distinguere le due cose potrebbe essere utile in situazioni nelle quali l'algoritmo viene eseguito molte volte e occorre riazzerare la soluzione, ma non occorre liberare la relativa memoria. Un altro motivo verrà discusso al principio del capitolo seguente.

La procedura alloca la matrice ausiliaria con le dimensioni corrette, chiama l'algoritmo ricorsivo di programmazione dinamica, ricostruisce la soluzione (cosa che per il momento rimandiamo, limitandoci a ottenere il valore della soluzione) e infine dealloca la matrice ausiliaria.

```
/* Programmazione dinamica ricorsiva per il problema dello zaino */
void PDricorsivaKP (datiKP *pI, soluzioneKP *pS)
{

  /* Crea la matrice ausiliaria che conserva i sottoproblemi */

  /* Programmazione dinamica ricorsiva per il problema dello zaino */

  /* Dealloca la matrice ausiliaria */
}
```

Procediamo quindi all'implementazione delle singole componenti di `PDricorsivaKP`.

```
/* Programmazione dinamica ricorsiva per il problema dello zaino */
void PDricorsivaKP (datiKP *pI, soluzioneKP *pS)
{
  mint Phi;
  int i, v;


  /* Crea la matrice ausiliaria che conserva i sottoproblemi */
  Phi = int2alloc(pI->n+1,pI->V+1);
  for (i = 0; i <= pI->n; i++)
    for (v = 0; v <= pI->V; v++)
      Phi[i][v] = -1;

  /* Programmazione dinamica ricorsiva per il problema dello zaino */
  pS->f = AlgoritmoRicorsivoKP(pI->n,pI->phi,pI->v,pI->V,Phi);

  /* Dealloca la matrice ausiliaria */
  for (i = 0; i <= pI->n; i++)
    free(Phi[i]);
```

```
  free(Phi);
}
```

Come abbiamo dichiarato un tipo `vint` per chiarire quando un puntatore a intero indica in effetti un vettore dinamico di interi, così dichiariamo un tipo `mint` per indicare che un doppio puntatore a intero ha il ruolo di una matrice dinamica di interi, e non di un doppio puntatore a interi, o di un vettore di puntatori a intero, o di un puntatore a un vettore di interi (tutte interpretazioni perfettamente lecite della scrittura `int **`). Per allocarla, ci serviremo di una procedura `mintalloc` a cui passeremo il numero di righe e di colonne della matrice, cioè $n+1$ righe e $V+1$ colonne (dove n e V si ottengono dai relativi campi del record `I` che racchiude l'istanza[6].

```
/* Alloca una matrice di int con nr righe e nc colonne */
mint int2alloc (int nr, int nc)
{
  mint M;
  int i;


  M = (mint) calloc(nr,sizeof(vint));
  if (M == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione di una matrice di
        interi!\n");
    exit(EXIT_FAILURE);
  }
  for (i = 0; i < nr; i++)
  {
    M[i] = (vint) calloc(nc,sizeof(int));
    if (M[i] == NULL)
    {
      fprintf(stderr,"Errore nell'allocazione del vettore
          M[%d]!\n",i);
      exit(EXIT_FAILURE);
    }
  }

  return M;
}
```

Allocata la matrice `Phi`, la inizializzeremo con un valore che distingua chiaramente per ogni sottoproblema se conosciamo o non conosciamo la soluzione. Per i problemi non risolti useremo il valore negativo -1, che è impossibile da ottenere come soluzione di qualsiasi sottoproblema, dato che la funzione valore è non negativa. Abbiamo in effetti scambiato righe e colonne rispetto alla figura dell'esempio, ma questo non cambia nulla concettualmente.

Poi chiameremo l'algoritmo ricorsivo, in modo identico all'algoritmo già discusso, salvo per l'uso della matrice ausiliaria. Per il momento, saltiamo la ricostruzione della soluzione, che discuteremo in seguito, e quindi ci limitiamo ad assegnare il risultato al campo opportuno della soluzione.

Infine, deallocheremo la matrice ausiliaria, prima riga per riga, poi la "costola" che contiene i puntatori alle singole righe.

[6]Se non ricordo male le ultime versioni del codice, ora c'è un'intera libreria con funzioni di allocazione di vettori e matrici di vario tipo.

La correttezza sintattica non è ancora raggiunta, dato che abbiamo modificato la chiamata dell'algoritmo ricorsivo sul problema originale, ma non ancora le chiamate ricorsive interne all'algoritmo stesso. Fatto ciò (e implementata l'allocazione della matrice di interi), l'algoritmo funziona, ed è anche corretto, ma finge solamente di utilizzare la matrice ausiliaria, per cui coincide sostanzialmente con la versione originale inefficiente, e per di più esegue operazioni aggiuntive che rallentano ulteriormente il tutto.

Adesso, però, possiamo aggiornare la funzione ricorsiva in modo che sfrutti la conoscenza delle soluzioni dei sottoproblemi. In che modo? Abbiamo detto che lo faremo introducendo nell'algoritmo già descritto un ulteriore caso base, corrispondente alla situazione in cui la soluzione del problema corrente è nota. Questo corrisponde ad avere un valore non negativo nella matrice `Phi`. Quindi, quando `Phi[i][v]` è non negativo, il problema con i oggetti e capacità v è risolto, e basta restituire `Phi[i][v]`. Altrimenti, possiamo proseguire con gli altri casi già discussi. Affinché ciò funzioni, però, quando il problema viene risolto, bisogna salvare il risultato nella matrice, altrimenti il nuovo caso base non si verificherà mai e non ne trarremo alcun beneficio. Quindi, sia nel caso base di insieme vuoto sia nei casi ricorsivi, prima di restituire il valore calcolato e terminare la funzione, dovremo copiare il valore stesso in `Phi[i][v]`. Quindi, potremmo anche approfittarne per raccogliere le istruzioni `return` in una sola, terminale, che restituisca all'esterno proprio il valore `Phi[i][v]`. E questo permette addirittura di eliminare apparentemente il nuovo caso base, dato che in tal caso la matrice ha già il valore corretto. È un po' quello che succedeva con il caso base degli algoritmi di ordinamento, che spariva perché un vettore abbastanza piccolo è già automaticamente ordinato.

```
/* Algoritmo ricorsivo per il problema dello zaino */
int AlgoritmoRicorsivoKP (int n, vint phi, vint v, int V, int **Phi)
{
  int phi0, phi1;

  if (Phi[n][V] < 0)
  {
    if (n == 0)
      Phi[n][V] = 0L;
    else if (v[n] > V)
      Phi[n][V] = AlgoritmoRicorsivoKP(n-1,phi,v,V,Phi);
    else /* (v[n] <= V) */
    {
      phi0 = AlgoritmoRicorsivoKP(n-1,phi,v,V,Phi);
      phi1 = phi[n] + AlgoritmoRicorsivoKP(n-1,phi,v,V-v[n],Phi);
      Phi[n][V] = ( (phi0 > phi1) ? phi0 : phi1 );
    }
  }

  return Phi[n][V];
}
```

13.5.2 Tempi di calcolo e analisi di complessità

Proviamo a compilare, eseguire e rivalutare i tempi di calcolo. I valori delle soluzioni rimangono ovviamente gli stessi. Per le istanze da 6 e 10 oggetti, anche i tempi di calcolo non cambiano, rimanendo inferiori a un microsecondo. L'istanza da 24 oggetti subisce, inaspettatamente, un forte rallentamento: si impiega più di un secondo, anziché pochi centesimi di secondo. Che cosa è successo? Per l'istanza da 50 oggetti, invece, cè un calo brusco da 3 minuti a meno di un microsecondo. Perché?

Per capirne qualcosa, dobbiamo studiare la complessità asintotica nel caso pessimo. L'analisi non è elementare, ma neppure ostica (vedi Figura 13.10). Il nuovo caso base non è facile da trattare con le tecniche standard, perché non corrisponde a una dimensione chiaramente determinata: i problemi già risolti possono essere piccoli o grandi, secondo il caso. Possiamo però osservare che la procedura ricorsiva consiste in un costrutto di selezione nel quale si entra una volta sola: la prima in cui il sottoproblema viene considerato, e quindi non è ancora risolto. Al termine di questa esecuzione, la cella della matrice riceve un valore non negativo e non si entrerà mai più nel corpo del costrutto. Le volte successive si entra nella funzione, ma subito si esce, restituendo il valore contenuto nella cella corrispondente. Se trascuriamo questa operazione (che a rigore non è trascurabile, per cui l'analisi è un po' semplificata), rimangono solo le operazioni in tempo costante eseguite su ogni sottoproblema. Ma i sottoproblemi non sono $O(2^n)$, come i nodi dell'albero calcolati per l'algoritmo ricorsivo semplice. Sono al più $(n+1)(V+1)$, dunque $O(nV)$, perché l'albero è stato sostituito da un grafo aciclico e i nodi corrispondono ad alcune delle celle della matrice. In ognuna di queste celle si eseguono calcoli in tempo costante,. Tutto questo spinge nella direzione di ipotizzare che l'algoritmo abbia complessità $O(nV)$.

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 13.10: Valutazione di complessità

Consideriamo le nostre quattro istanze. Per l'istanza con $n = 6$ e $V = 8$, otteniamo $nV = 48$ contro $2^n = 64$, che è più o meno lo stesso. Per l'istanza con $n = 10$ e $V = 1\,765$, otteniamo $nV = 17\,650$ contro $2^{10} = 1\,024$, che ancora non cambia molto. Per l'istanza con $n = 50$ e $V = 420$, otteniamo $nV = 20\,100$ contro $2^{50} \approx 10^{15}$, il che spiega l'enorme calo nel tempo di calcolo, fino a valori paragonabili a quelli delle istanze più piccole. Per l'istanza con $n = 24$ e $V = 6\,404\,180$, otteniamo $nV \approx 1.5 \cdot 10^8$ contro $2^{24} \approx 1.6 \cdot 10^7$, il che spiega l'aumento di un ordine di grandezza nel tempo di calcolo. Quindi, l'analisi teorica, pur essendo condotta con stime per eccesso e ignorando costanti moltiplicative incognite, è in buon accordo con le osservazioni sperimentali.

Ma allora è meglio l'algoritmo di programmazione dinamica o l'algoritmo ricorsivo semplice? Dipende dall'istanza: l'algoritmo ricorsivo semplice conviene per

istanze con valori di capacità V molto. Teniamo però conto che stiamo confrontando 2^n con nV, dunque l'algoritmo ricorsivo è più efficiente solo quando $V \gg 2^n/n$, che tende ad essere un numero veramente grande. Sono poche le istanze in cui questo avviene.

Un'altra domanda spontanea è: che genere di algoritmo è, questo algoritmo di programmazione dinamica? È esponenziale o polinomiale? Essendo lineare in n e lineare in V, viene il sospetto che sia decisamente polinomiale. Invece, strettamente parlando, non lo è.

Per capire il motivo, bisogna tornare alla definizione di "dimensione di un'istanza". La definizione informale dice che, se ci sono n oggetti, la dimensione è n. Si tratta, però, di una definizione informale e approssimativa. La vera definizione è il numero di simboli (diciamo, caratteri ASCII) necessari per rappresentare l'istanza. Quanti simboli occorrono per rappresentare un'istanza del problema dello zaino? È ovvio che al crescere del numero di oggetti n crescerà anche il numero di caratteri necessari a descrivere i loro volumi e valori. Tipicamente, la crescita è proporzionale a n, per cui n diventa un buon indice della dimensione stessa, dato che le costanti moltiplicative non contano. La presenza della capacità V, però, cambia le cose, perché V può essere un numero molto piccolo o molto grande, e la dimensione complessiva dell'istanza dipende poco da V. Per esempio, se V passa da 8 a 8 000 000, il numero di caratteri dell'istanza cresce di 6 caratteri, mentre il tempo di calcolo dell'algoritmo viene moltiplicato per un milione. Abbiamo un algoritmo che diventa enormemente più lento per una crescita minima nelle dimensioni dell'istanza. Questo comportamento è tipico degli algoritmi esponenziali. Infatti la complessità $O(nV)$ è esponenziale rispetto alla dimensione $O(n \log V)$ (dove si è assunto per semplicità che tutti i valori dei dati abbiano un numero di cifre analogo a quello della capacità). Rispetto all'algoritmo ricorsivo semplice, che è esponenziale rispetto a n, questo è esponenziale rispetto a V, pur essendo solo lineare rispetto a n.

È che il parametro V potrebbe essere piccolo anche per istanze con un numero enorme di oggetti, e in tal caso l'algoritmo sarebbe molto efficiente. Quando la non polinomialità non dipende da n, si può sperare che il comportamento rimanga efficiente anche per istanze molto grandi, grazie al fatto che i parametri veramente influenti non hanno valori eccessivi. Questi algoritmi vengono detti *pseudopolinomiali*.

13.6 Terza fase (`knapsack3.c`): programmazione dinamica "bottom-up"

La realizzazione *bottom-up*, che vedremo in seguito, sfrutta l'idea di realizzare una costruzione iterativa della struttura ausiliaria, risolvendo i problemi via via, a partire da quelli più semplici, per arrivare ai più complessi, e finalmente a quello originale (vedi Figura 13.9). Nel caso dello zaino, l'idea è di creare la matrice di 7 righe e 5 colonne che contiene tutti i sottoproblemi possibili e osservare che alcuni di questi sottoproblemi sono banali. Per esempio, i sottoproblemi con $n = 0$ hanno soluzione vuota, dato che corrispondono a cercare il sottoinsieme di valore massimo di un insieme vuoto. In effetti, sono i casi base della ricorsione. Quindi, parte della matrice si può riempire fin da subito. Fatto ciò, possiamo combinare le soluzioni dei problemi più semplici per risolvere alcuni dei problemi più complicati. In particolare, possiamo partire da un livello e costruire le soluzioni del livello successivo, con un elemento in più, e date queste possiamo procedere con il livello ancora seguente, fino ad arrivare all'ultimo livello e al problema originale. Questo è un algoritmo tipicamente iterativo, basato sul fatto che si ricombinano le soluzioni. Viene a

mancare la decomposizione del problema in sottoproblemi, tipica degli algoritmi ricorsivi, perché fin dal principio, costruendo la struttura dati ausiliaria, abbiamo elencato esaustivamente tutti i sottoproblemi possibili. Rimane invece la parte di ricombinazione delle soluzioni dei sottoproblemi nelle soluzioni dei problemi di livello superiore.

L'opposizione tra *top-down* e *bottom-up* descrive proprio la direzione del movimento con il quale si visita la matrice dei sottoproblemi: dall'alto in basso, generandoli per decomposizione prima di risolverli, oppure dal basso in alto, generando le soluzioni di quelli più grandi a partire da quelle dei problemi più piccoli. Per esempio, riferendosi alla Figura 13.8, i problemi di livello 0 hanno tutti soluzione nulla. Poi, il problema $(1, \phi, v, 6)$ si risolve a partire dai problemi $(0, \phi, v, 6)$ e $(0, \phi, v, 2)$; il problema $(1, \phi, v, 4)$ si risolve a partire dai problemi $(0, \phi, v, 4)$ e $(0, \phi, v, 0)$; e si può procedere fino al livello più alto.

Qual è la complessità di questo algoritmo? Si tratta di scorrere tutti i problemi e per ognuno ricostruire la soluzione. Per ogni problema, questo richiederà un tempo costante, e quindi la complessità sarà ancora una volta proporzionale al numero di sottoproblemi, ovvero $O(nV)$.

A questo punto ci si potrebbe chiedere se preferire la modalità *top-down* o quella *bottom-up* o considerarle del tutto equivalenti. Il punto è che la ricorsione intrinsecamente richiede un *overhead* di allocazione dei record di attivazione sullo *stack*, di copia degli argomenti, di deallocazione e restituzione del risultato, che in genere vale la pena di evitare. Questo favorisce la versione iterativa. D'altra parte, vedremo che nel caso dell'algoritmo per il problema dello zaino le due modalità non sono perfettamente equivalenti.

Esempio Prima di realizzare il codice, però, discutiamo in maggior dettaglio l'applicazione dell'algoritmo sull'istanza da 6 oggetti con $V = 8$, in modo da chiarirne meglio il funzionamento (vedi Figura 13.11). Abbiamo una matrice con $6 + 1$ righe, associate a sottoinsiemi di elementi via via crescenti, e $8 + 1$ colonne, associate ai possibili valori di volume occupato. Inizialmente, si può pensare che tutti i valori siano pari al valore fittizio -1, anche se non è necessario, perché li riempiremo comunque metodicamente riga per riga. Sappiamo che con 0 elementi la soluzione ottima è necessariamente vuota e di valore nullo. Quindi possiamo risolvere i casi base azzerando tutta la riga 0. A questo punto, siamo in grado di risolvere i problemi con il solo primo elemento, cioè quelli della riga 1, semplicemente combinando i due sottoproblemi di livello 0 che gli corrispondono. A volte, sarà un singolo sottoproblema. Per esempio, siccome l'elemento 1 ha volume 5, è chiaro che zaini di capacità 0, 1, 2, 3 e 4 non possono contenerlo, per cui la soluzione ottima rimane vuota. Invece, zaini di capacità 5, 6, 7 e 8 hanno due soluzioni possibili: scartare o prendere l'unico elemento. La prima eredita il valore dal sottoproblema della riga precedente con uguale capacità; la seconda, invece, deriva dal sottoproblema della riga precedente con capacità ridotta del volume del nuovo oggetto (dunque, $v_1 = 5$) per aggiunta del valore $\phi_1 = 7$. Quindi, per risolvere il sottoproblema $(1, 5)$ bisogna confrontare il valore della cella $(0, 5)$ (un oggetto in meno e ugual capacità) con il valore della cella $(0, 0)$ (un oggetto in meno e capacità ridotta di v_1) incrementato di ϕ_1. Il secondo valore $(0 + 7)$ domina il primo (0), e quindi viene scelto.

La riga 2 è un po' più interessante: siccome $v_2 = 3$, le celle $(2, 0)$, $(2, 1)$ e $(2, 2)$ possono solo rifiutare l'elemento e quindi ereditare il valore delle celle sovrastanti, che è nullo. La cella $(2, 3)$, invece, può confrontare il rifiuto (cella $(1, 3)$, di valore 0) con l'accettazione (cella $(2, 0)$, di valore 0, più $\phi_2 = 2$), e quest'ultima prevale. Quando arriviamo alla cella $(2, 5)$ confrontiamo $(1, 5)$, che vale 7, con $(1, 2)$ più ϕ_2, che vale 2, e prevale 7. Infine, quando arriviamo alla cella $(2, 8)$ confrontiamo

$(1, 8)$, che vale 7, con $(1, 5)$ più ϕ_2, che vale 9, e questa vince. Nella riga 3 partiamo con un po' di celle vuote, finché la capacità è insufficiente a contenere v_3, e poi incrementiamo via via la soluzione, sempre facendo il confronto tra le due possibilità alternative. Ogni volta che una delle due soluzioni di riferimento alla riga precedente migliora, può migliorare la soluzione nella riga corrente.

L'andamento è necessariamente non decrescente perché via via aumenta la capacità e aumenta l'insieme degli oggetti disponibili, dunque quello delle possibili soluzioni.

Nella cella $(5, 5)$ avviene un caso interessante: scartare l'elemento 5 significa ereditare la soluzione della cella $(4, 5)$, che vale 9; prendere l'elemento 5 significa ereditare la soluzione della cella $(4, 4)$, che vale 4, e sommarle $\phi_5 = 4$. Le due alternative sono perfettamente equivalenti. In effetti, un problema può tranquillamente avere molte soluzioni ottime, tutte equivalenti fra loro. Arrivati alla cella $(6, 8)$, cioè (n, V), otteniamo la soluzione ottima del problema originale, che vale 15.

i\v	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	7	7	7	7
2	0	0	0	2	2	7	7	7	9
3	0	0	4	4	4	7	7	11	11
4	0	0	4	5	5	9	9	11	12
5	0	4	4	8	9	9	13	13	15
6	0	4	5	8	9	10	13	14	15

Figura 13.11: Esempio di esecuzione dell'algoritmo di programmazione dinamica *bottom-up*

Implementazione Passiamo al codice. Partiamo con il solito programma principale e una funzione `PDiterativaKP`, che riceve i dati e restituisce la soluzione.

```
/* Risolve il problema dello zaino con la programmazione dinamica
   bottom-up (iterativa) */
void PDiterativaKP (datiKP *pI, soluzioneKP *pS);
```

Nel corpo di questa funzione, useremo ancora la matrice `F`, e quindi allocazione iniziale e deallocazione finale rimangono. Non occorrerà inizializzarla, perché calcoleremo i valori dal basso, partendo da problemi già risolti e non occorrerà mai consultare la matrice per vedere se conosciamo la soluzione del problema corrente: ogni problema verrà affrontato una volta, senza conoscere la soluzione, ma conoscendo certamente le soluzioni dei suoi sottoproblemi. Non avremo invece la procedura ricorsiva, ma un ciclo che procede di riga in riga (da 0 a n) e di colonna in colonna (da 0 a V) riempiendo le varie celle. Per ogni cella, dobbiamo determinare se siamo nel caso base, cioè nella riga 0. Questo potrebbe essere un ciclo di inizializzazione che precede quello principale (sarebbe più elegante ed efficiente, risparmiando il test di verifica, per lo più inutile), ma per ora lo teniamo insieme per rendere più chiaro il legame con la versione ricorsiva. In tal caso, la cella va azzerata. Nelle righe

seguenti, distinguiamo i due casi ricorsivi. Se il volume dell'oggetto corrente supera la capacità, basta copiare nella nuova cella quella della riga precedente e della stessa colonna. Infine, è possibile che vi siano due possibilità e che si debba confrontare il valore ottenuto da una e dall'altra per assegnare il migliore alla nuova cella.

Confrontando la versione iterativa con quella ricorsiva è molto evidente la loro forte somiglianza formale, dove si distinguono tre casi, e in ognuno si compie sulla matrice lo stesso assegnamento, facendo uso delle stesse variabili ausiliarie. È ovvio che, in realtà, la riga 0 può essere trattata a parte, prima del ciclo principale, senza eseguire su ogni cella il test del caso base, e che le due variabili ausiliarie `phi0` e `phi1` possono essere direttamente sostituite con i relativi valori. Il valore finale della soluzione ottima è il contenuto della cella della matrice che descrive il problema originale, cioè la cella (n, V), che va quindi copiato nel campo `f` della soluzione `S`

```
/* Risolve il Knapsack Problem con la programmazione dinamica
    bottom-up (iterativa) */
void PDiterativaKP (datiKP *pI, soluzioneKP *pS)
{
  int **Phi;
  int i, v;
  int phi0, phi1;


  /* Crea la matrice ausiliaria che conserva i sottoproblemi */
  Phi = int2alloc(pI->n+1,pI->V+1);

  /* Programmazione dinamica iterativa per il problema dello zaino */
  for (i = 0; i <= pI->n; i++)
  {
    for (v = 0; v <= pI->V; v++)
    {
      /* Risolve il sottoproblema (i,v) */
      if (i == 0)
        Phi[i][v] = 0;                                  /* return 0;
                                                        */
      else if (pI->v[i] > v)
        Phi[i][v] = Phi[i-1][v];                        /* return
            PDricorsivaKP(n-1,phi,v,V);                 */
      else /* (pI->v[i] <= v) */
      {
        phi0 = Phi[i-1][v];                             /* return
            PDricorsivaKP(n-1,phi,v,V);                 */
        phi1 = pI->phi[i] + Phi[i-1][v-pI->v[i]]; /* return
            PDricorsivaKP(n-1,phi,v,V-pI->v[n]); */
        Phi[i][v] = ( (phi0 > phi1) ? phi0 : phi1 );
      }
    }
  }

  /* Il problema originale corrisponde alla cella (n,V): il suo
      valore ottimo e' il contenuto della cella */
  pS->f = Phi[pI->n][pI->V];

  /* Dealloca la matrice ausiliaria */
  for (i = 0; i <= pI->n; i++)
    free(Phi[i]);
  free(Phi);
}
```

Compilando ed eseguendo il codice, possiamo verificare che l'algoritmo calcola correttamente tutte le soluzioni. Osserviamo che sull'istanza da 24 oggetti c'è un ulteriore piccolo rallentamento. La causa di questo rallentamento è che in realtà

l'algoritmo iterativo *bottom-up* non compie rigorosamente le stesse operazioni di quello ricorsivo *top-down*, nonostante la fortissima somiglianza formale. Il fatto è che, nel passare in rassegna sistematicamente tutte le righe e tutte le colonne della matrice, si risolvono anche sottoproblemi che l'albero ricorsivo non contiene, per esempio quelli con tutti gli oggetti e capacità strettamente minore di quella totale (l'ultima riga, salvo l'ultima cella). Quindi, la versione iterativa è apparentemente più pesante ancora di quella ricorsiva. D'altra parte, com'è noto, la versione iterativa risparmia la gestione dello *stack*, con le sue ridondanze temporali e spaziali. Inoltre, esistono versioni più raffinate dell'algoritmo iterativo che saltano di cella in cella, evitando di risolvere quelle inutili. Sono raffinamenti ovviamente molto utili, ma che eccedono i limiti del corso.

13.6.1 Determinazione della soluzione ottima

In tutta la trattazione abbiamo semplificato drasticamente il problema riducendolo al calcolo del valore ottimo. In realtà, vogliamo conoscere anche la soluzione ottima che corrisponda a tale valore. In molti problemi, questo richiede strutture dati aggiuntive, che consentano di ricostruire in ogni cella la soluzione ottima che corrisponde al valore in essa conservato. Apparentemente, ciò richiederebbe di conservare una soluzione in ogni cella, con un aggravio enorme dello spazio in memoria (un vettore di incidenza di n elementi per $(n+1)(V+1)$ celle, dunque $O\left(n^2V\right)$) e del tempo necessario a gestirlo (in ogni cella dovremmo copiare la soluzione ottenuta, dunque consumando tempo $O\left(n^2V\right)$). Questo è chiaramente inaccettabile[7]

Il punto è che non occorre conservare un'intera soluzione in ogni cella della struttura dati ausiliaria. Basta conservare l'informazione minima sufficiente a ricostruire al termine l'unica soluzione interessante. Questa informazione è: attraverso quale scelta singola si è arrivati nella cella corrente? Nel caso dello zaino, la scelta è: scartare o prendere l'elemento corrispondente alla riga.

Si potrebbe quindi concludere che basti una seconda matrice, che contenga valori binari, che indicano in ogni cella se la soluzione ottima del problema corrispondente è ottenuta scartando o prendendo l'elemento corrispondente alla riga. In alcuni problemi, strutture dati di questo genere sono necessarie. Esempi sono l'algoritmo di Dijkstra e quello di Floyd-Warshall per il calcolo dei cammini minimi (rispettivamente, da un nodo a tutti gli altri e da tutti i nodi a tutti i nodi di un grafo dato).

Nel caso dello zaino, si può fare di meglio, evitando completamente ogni altra struttura. Infatti, sappiamo che ogni cella viene raggiunta da due possibili celle che appartengono alla riga precedente: quella con uguale capacità e quella con capacità ridotta del volume dell'oggetto corrente. Inoltre, possiamo velocemente ricostruire le due soluzioni che otteniamo partendo da tali celle, e possiamo immediatamente confrontarle per determinare la migliore. In pratica, possiamo nuovamente calcolare e confrontare le due alternative. L'aggravio computazionale costa molto meno di prima, perché non lo facciamo sull'intera matrice, ma solo risalendo all'indietro di cella in cella lungo il cammino che ha portato alla soluzione ottima, che ora conosciamo e invece durante il calcolo non era noto. Il tutto costa tempo $O(n)$, trascurabile rispetto al calcolo del valore, con un notevole risparmio in spazio e un tempo paragonabile a quello necessario a ricostruire la soluzione sfruttando una seconda matrice ausiliaria di valori binari, e un tempo e uno spazio molto migliori di quelli necessari a gestire una soluzione ottima in ogni cella.

Torniamo all'esempio della Figura 13.11. Partiamo dalla cella $(6, 8)$: il valore 15 potrebbe derivare o da una copia della cella $(5, 8)$ o dalla somma della cella $(5, 7)$

[7] E tuttavia abbastanza frequente nello svolgimento dei progetti d'esame.

con il valore ϕ_5. Essendo la prima soluzione migliore, concludiamo che la soluzione ottima non contiene l'oggetto 6. Risaliti alla cella $(5, 8)$, iteriamo il procedimento, ottenendo la soluzione $x = \{1, 3, 5\}$.

In alcuni passaggi, le due soluzioni possono essere equivalenti. Questo consente di scegliere liberamente, e quindi anche di ottenere più soluzioni ottime, volendo. In pratica, ne sceglieremo una arbitrariamente (quella che scarta l'oggetto).

Implementazione Torniamo al codice e realizziamo una procedura che riceve la matrice ausiliaria e ne ricava il vettore di incidenza[8].

```
/* Ricostruisce la soluzione, determinando all'indietro quale
    sottoproblema
   ha determinato la soluzione di ogni problema */
v = pI->V;
for (i = pI->n; i >= 1; i--)
  if (v < pI->v[i])
    pS->x[i] = FALSE;
  else if (Phi[i][v] == Phi[i-1][v])
    pS->x[i] = FALSE;
  else
  {
    pS->x[i] = TRUE;
    v -= pI->v[i];
  }
```

La procedura parte avendo a disposizione l'intera capacità e scorre all'indietro gli oggetti, dall'ultimo al primo, valutando per ognuno se convenga scartarlo o prenderlo, in base al valore ottimo contenuto nella matrice ausiliaria. Quando l'oggetto ha volume superiore alla capacità, è ovvio che bisogna scartarlo (inoltre, valutare il caso alternativo comporterebbe di accedere a una cella con capacità negativa, con i problemi che si possono facilmente immaginare). Altrimenti, se la soluzione del sottoproblema di pari capacità alla riga precedente ha lo stesso valore significa che scartare l'oggetto è almeno altrettanto buono che prenderlo, e quindi lo scartiamo. In ogni altro caso, accettiamo l'oggetto e aggiorniamo la capacità, riducendola del volume dell'oggetto stesso.

13.7 Il problema del cammino minimo

Questo problema da diversi anni non trova tempo sufficiente per essere svolto in laboratorio, per cui non è disponibile una trattazione dettagliata. Tuttavia, metto a disposizione in queste note i lucidi originali, come supporto alla realizzazione di codice che può essere necessario a risolvere il progetto d'esame. I relativi argomenti, infatti, sono svolti nelle lezioni del modulo di teoria.

13.8 Esercizi

13.8.1 Esercizio 1

Si realizzi una libreria per la gestione dei dati e delle soluzioni per il problema dello zaino.

[8] Al momento, non è ancora una funzione isolata, ma lo diventerà nel prossimo aggiornamento del codice.

Figura 13.12: 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 $c_X = \sum_{(i,j) \in X} c_{ij}$

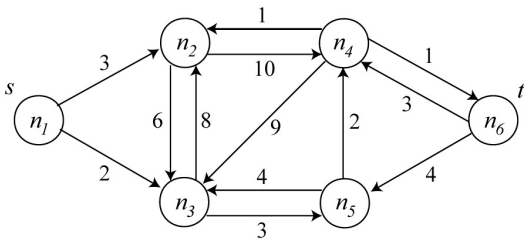


Figura 13.13: Varianti

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*)

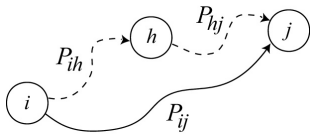
Consideriamo la ricerca dei cammini minimi da ogni nodo a ogni nodo

Figura 13.14: 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 13.15: 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* $\ell_{h+1,j}^{(h)}$

L'algoritmo rappresenta tutti i cammini minimi attraverso due matrici

- ℓ_{ij} è il costo di un cammino minimo da i a j
- π_{ij} è il nodo predecessore lungo il cammino minimo da i a j

La complessità temporale è $\Theta\left(n^3\right)$

Figura 13.16: Algoritmo di Floyd-Warshall (1962)

Figura 13.17: Applicazione dell'algoritmo di Floyd-Warshall (1)

Figura 13.18: Applicazione dell'algoritmo di Floyd-Warshall (2)

Figura 13.19: Applicazione dell'algoritmo di Floyd-Warshall (3)

Figura 13.20: 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

13.8.2 Esercizio 2

Si realizzi una versione dell'algoritmo ricorsivo per il problema dello zaino che restituisca anche la soluzione del problema, in forma di record.

13.8.3 Esercizio 3

Si realizzi una versione dell'algoritmo iterativo per il problema dello zaino con le semplificazioni descritte nei commenti, cioè trattando a parte la riga 0 e non usando variabili ausiliarie, salvo la matrice e i cursori che la scorrono.

13.8.4 Esercizio 4

Si realizzi l'algoritmo di Dijkstra nella versione di complessità $O\left(n^2\right)$ che conserva per ogni nodo non ancora marcato il nodo più vicino dal quale esso è raggiungibile, terminando quando il nodo di destinazione è stato definitivamente raggiunto. Si stampino i vertici incontrati lungo il percorso, dall'origine alla destinazione.

13.8.5 Esercizio 5

Si realizzi l'algoritmo di Dijkstra nella versione di complessità $O\left(m \log n\right)$ che conserva un *min-heap* per gestire la coda con priorità che fornisce l'elemento minimo da marcare ad ogni iterazione, terminando quando il nodo di destinazione è stato definitivamente raggiunto. Si stampino i vertici incontrati lungo il percorso, dall'origine alla destinazione.

13.8.6 Esercizio 6

Si realizzi l'algoritmo di Floyd-Warshall e si stampino i vertici incontrati lungo il percorso da ciascuna origine a ciascuna destinazione.

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.

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.

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 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

[1] Controllare che la cosa fosse esplicita nel capitolo precedente

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 sottoinsieme 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.

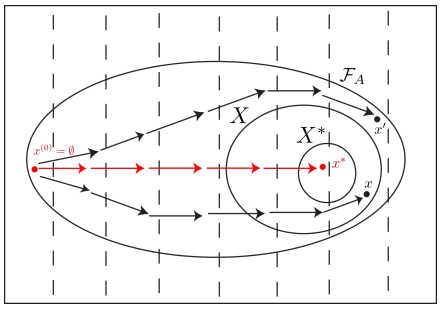
In generale, ci sarà più di un elemento appartenente a $B \setminus X$ che aggiunto a x consenta di rimanere nello spazio di ricerca. Fra questi, l'algoritmo deve scegliere in base a qualche criterio. Esisterà quindi una funzione $\phi_A(i, x)$, che è propria dell'algoritmo A considerato e dipende dall'elemento 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*

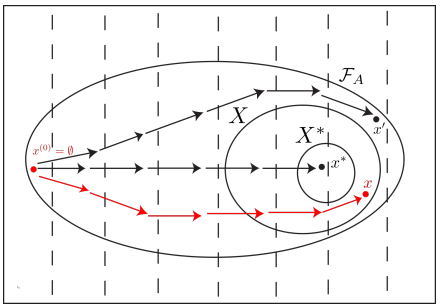
Figura 14.3: Un grafico

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 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



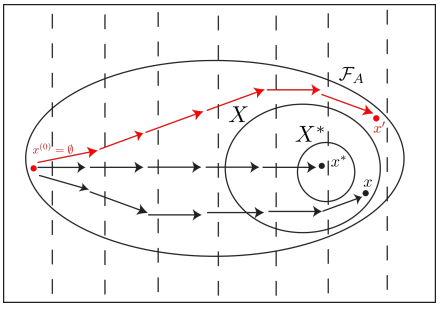
L'algoritmo visita una catena di sottoinsiemi $\emptyset = x^{(0)} \subset \ldots \subset x^{(k)}$

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



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

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.

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.

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 $i \in B \setminus x$

Figura 14.7: Il problema dello zaino unitario

Consideriamo per un momento un problema dello zaino con ipotesi restrittive, cioè ipotizziamo che tutti gli oggetti abbiano lo stesso volume v (vedi Figura 14.7). 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.

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

B	a	b	c	d	e	f
ϕ	7	2	4	5	4	1

$$v_i = 1 \text{ per ogni } i \in B$$

$$V = 4$$

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?

Figura 14.8: Esempio: il problema dello zaino unitario

nella Figura 14.8. È ragionevole prendere prima l'oggetto che vale di più, poi il secondo, poi il terzo e infine il quarto, fermandosi perché la capacità è esaurita. Chiunque considererebbe ovvio che questa sia la soluzione ottima. In matematica però bisogna in qualche modo dimostrarlo. La cosa interessante è che il teorema di Rado, che viene enunciato, discusso e dimostrato nelle dispense di teoria, dà una motivazione solida del fatto che questo meccanismo non funziona solo per questo banale problema, ma per tutta una serie di problemi che hanno caratteristiche comuni con esso.

A questo punto, potremmo affrontare speranzosamente il problema dello zaino generico, con oggetti di volume vario (vedi Figura14.9). Un insieme x di oggetti è estendibile con un nuovo elemento i quando il volume totale della soluzione x più il volume del nuovo oggetto i non supera la capacità V. L'algoritmo *greedy*, adattato al caso specifico, parte dall'insieme vuoto e, finché esistono oggetti esterni il cui volume non supera la capacità residua dello zaino, cioè la capacità iniziale ridotta del volume corrente della soluzione, si sceglie l'elemento di valore massimo fra quelli che godono di questa proprietà e si aggiunge alla soluzione.

Se lo applichiamo alla già nota istanza riportata nella Figura 14.10, però, prendiamo l'elemento a e l'elemento d e ci fermiamo, perché la capacità è completamente saturata. D'altra parte, la soluzione $\{a, d\}$ vale 12, mentre abbiamo calcolato nel capitolo precedente che la soluzione ottima è $\{a, c, e\}$ e vale 15. Perché l'algoritmo *greedy* fallisce in questo caso e funziona in quello unitario? Perché il caso unitario rispetta le ipotesi del teorema di Rado e quello generale ne viola una (vedi Figura 14.11).

La chiave sta nell'uniformità del volume. L'algoritmo *greedy* tiene poco conto del volume, salvo vietare violazioni della capacità, cioè interviene solo all'ultimo momento, anziché per tempo.

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;

Figura 14.9: 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 14.10: Esempio: il problema dello zaino

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 14.11: Correttezza dell'algoritmo greedy

14.1 L'algoritmo *greedy* banale per il problema dello zaino

Cominciamo a implementare l'algoritmo *greedy* base, che chiameremo banale. Sia chiaro che questo algoritmo spesso funziona male, e non viene praticamente mai adottato. Partiamo come nel capitolo scorso dal codice `knapsack0.c` che fornisce le strutture dati e le funzioni fondamentali per gestire il problema dello zaino, ma nulla ancora per risolverlo. Vogliamo costruire la procedura `GreedyBanale`, che risolverà il problema dello zaino scegliendo sistematicamente l'oggetto più prezioso che possa essere ancora raccolto nello zaino corrente. Come nel capitolo precedente, la funzione riceve l'istanza `I` (per indirizzo, solo per efficienza) e restituisce la soluzione `S` (per indirizzo, dato che la deve modificare).

```
/* Risolve l'istanza I del problema dello zaino con l'euristica
    greedy base.
   Nota: assume che la soluzione S sia inizialmente vuota */
void GreedyBanaleKP (datiKP *pI, soluzioneKP *pS);
```

Abbiamo già discusso possibili motivi per tenere la creazione di soluzioni vuote distinta dalla vera e propria risoluzione di un problema di zaino. Un altro motivo è che, mentre un algoritmo *greedy* parte di solito da soluzioni vuote, si può anche farlo partire da una soluzione parzialmente riempita. Ad esempio, possono esservi proprietà matematiche le quali dimostrano che certi elementi certamente appartengono a una soluzione ottima (mentre altri potrebbero essere dimostrabilmente da scartare). In questo caso, diventa utile avere un algoritmo capace di tener conto di questa informazione e quindi di partire da essa. È chiaro che, se facciamo questa

scelta, dobbiamo poi essere sicuri che la soluzione sia in partenza vuota quando ipotizziamo che lo sia. Nel caso presente, lo siamo perché l'abbiamo appena creata.

Dovremo ora implementare l'algoritmo *greedy*, a partire dallo pseudocodice della Figura 14.9. L'azzeramento della soluzione x si suppone già eseguito in precedenza. Determinare il sottoinsieme di oggetti esterni alla soluzione attuale e con volume non eccedente la capacità residua, e in particolare l'elemento di valore massimo di tale sottoinsieme è un compito tutt'altro che banale da eseguire ad ogni iterazione.

Comporta ad ogni iterazione (quindi, al massimo n volte) di scorrere tutti gli oggetti e valutare per ciascuno ($O(n)$ volte) se è esterno alla soluzione (un semplice test in $O(1)$ sul vettore di adiacenza) e se ha volume accettabile, per poi determinare fra questi quale ha valore massimo. Se si commette l'errore madornale di valutare l'accettabilità del volume ricalcolando da zero il volume corrente della soluzione, tutto questo costa $O(n^3)$. Ovviamente, conviene definire una variabile `v`, inizializzarla con valore nullo e aggiornarla sommandole il volume di ogni oggetto via via aggiunto. Evitato in questo modo il ricalcolo ripetuto, il tutto ha apparentemente costo $O(n^2)$.

Se potessimo limitarci a cercare l'elemento di valore massimo esterno alla soluzione, senza tener conto del volume, potremmo sfruttare un *max-heap* per rappresentare gli elementi esterni (vedi Sezione 12.2). Determineremmo allora l'elemento massimo in tempo costante e, una volta aggiunto alla soluzione, lo toglieremmo dall'*heap* in tempo logaritmico, in modo da garantire che gli elementi estratti siano sempre esterni. È praticamente la stessa cosa che avviene nell'algoritmo di ordinamento *HeapSort*. Rimane il problema di determinare un elemento che non sia solo esterno e abbia valore massimo, ma abbia anche volume accettabile. La soluzione di questo problema, però, è molto semplice. Ignoriamo la condizione sul volume e determiniamo l'elemento esterno di valore massimo: se il suo volume è accettabile (cosa determinabile in tempo costante, noto il volume attuale della soluzione), possiamo aggiungerlo e passare all'iterazione seguente. Se invece non lo è, sappiamo che tale elemento, inaccettabile ora, non sarà mai accettabile, perché il volume totale della soluzione cresce continuamente e la capacità residua nei passi seguenti sarà via via più bassa. L'idea, quindi, è estrarre l'elemento massimo dell'*heap*, aggiungerlo alla soluzione solo quando è ammissibile, ma comunque toglierlo dall'*heap*. Di conseguenza, l'*heap* non è più $B \setminus x$, ma quella parte di $B \setminus x$ che non è ancora stata scartata. Modifichiamo di conseguenza lo pseudocodice, che rivelerà un secondo problema da risolvere.

Riassumendo, invece di:

1. partire da un sottoinsieme vuoto
2. valutare se esistono elementi aggiungibili al sottoinsieme corrente nel rispetto del vincolo di capacità;
3. determinare l'oggetto di valore massimo fra quelli aggiungibili;
4. aggiungerlo alla soluzione corrente e tornare al punto 2

preferiremo:

1. partire da un sottoinsieme vuoto
2. determinare l'oggetto di valore massimo fra quelli esterni;
3. se tale elemento è aggiungibile, procedere ad aggiungerlo alla soluzione corrente; altrimenti, eliminarlo; in entrambi i casi, tornare al punto 2.

Partiamo creando un *heap* H che contiene tutti gli elementi dell'insieme base B, e inizializzando il volume corrente della soluzione a zero. Finché l'insieme H non è vuoto, determiniamo il suo elemento di valore massimo. Quindi, valutiamo se questo elemento si possa aggiungere alla soluzione x, in base al volume corrente di x, al volume dell'elemento e alla capacità. In caso positivo, lo aggiungiamo a x. In ogni caso, togliamo l'elemento da H. L'aspetto problematico a cui si è accennato è che un *heap* fornisce il valore massimo di un insieme, ma non l'elemento che ha tale valore: sappiamo che il valore massimo nell'istanza che stiamo usando vale 7, ma non che è l'elemento a e non che il suo volume è 5. Gli indici sono necessari perché, una volta scambiati gli oggetti rispetto all'ordinamento iniziale, non si ha più modo di indicarli per nome.

14.1.1 Heap indiretto

Una prima soluzione ingenua è costruire un vettore di strutture che contenengano tutta l'informazione necessaria (indici, valori e volumi), e ordinarla per valori crescenti oppure gestirla come un *heap*. Equivalentemente, anziché un vettore di terne, si possono usare tre vettori:

```
ind = [ 1 2 3 4 5 6 ]
phi = [ 7 2 4 5 4 1 ]
v   = [ 5 3 2 3 1 1 ]
```

La costruzione di un *max-heap* sui valori con la ben nota procedura `creaheap` comporta di scorrere all'indietro le posizioni della prima metà del vettore (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 ]
```

In questo modo, le tre informazioni (indice, valore e volume) vanno di pari passo: ogni volta che si modifica in qualche modo un valore, si modificano in parallelo anche il volume e l'indice per non sfasarli.

Scambiare gli elementi di tutti i vettori (o gli elementi dell'unico vettore di strutture) è chiaramente pesante (l'abbiamo già osservato parlando del problema dell'ordinamento (vedi Sezione **??**). Inoltre, se i dati iniziali avevano un ordine significativo (cosa che non avviene nel nostro esempio, ma potrebbe succedere in altri), l'ordinamento iniziale dei dati viene distrutto. Questo vale sia per la creazione iniziale dell'*heap* sia per il suo uso durante l'algoritmo *greedy*, nel quale addirittura i dati potrebbero essere cancellati definitivamente.

La soluzione è quella già discussa presentando l'ordinamento indiretto, cioè di eseguire gli scambi non su tutti e tre i vettori, ma solo sul vettore degli indici. Ovviamente questo comporta un leggero aggravio nella lettura dei dati, perché il valore e il volume conservano la loro posizione iniziale, e quindi per accedervi bisogna prima recuperare quest'ultima dal vettore degli indici, eseguendo un indirizzamento indiretto. Per esempio:

```
ind = [ 1 4 3 2 5 6 ]
phi = [ 7 2 4 5 4 1 ]
v   = [ 5 3 2 3 1 1 ]
```

dove i vettori `phi` e `v` sono intatti e la proprietà *max-heap* vale *indirettamente* sul vettore `phi` attraverso `ind`: per identificare i nodi figli si usano, al posto degli indici diretti, quelli 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 radice non corrisponde più all'indice 1, così che `phi[1]` è il valore massimo, ma all'indice `ind[1]` (qui casualmente ancora pari ad 1) e quindi `phi[ind[1]]` è il valore massimo.

La grande forza del meccanismo di indirizzamento indiretto è di non richiedere scambi sui dati e di consentire molti ordinamenti diversi sugli stessi dati: basta un vettore di indici per ogni ordinamento. Chiaramente, questo costa spazio (un vettore aggiuntivo di interi per ogni ordinamento) e tempo per l'accesso indiretto. Il vantaggio temporale in termini di scambi prevale nettamente, via via che i dati diventano più corposi.

È ovvio che sarebbe possibile anche usare un vettore di puntatori, anziché di indici, ma questo permetterebbe l'accesso solo a uno dei due vettori con i dati (valori e volumi); sarebbe allora necessario raccogliere i dati in un vettore di strutture, come già proposto più sopra.

14.1.2 Implementazione dell'algoritmo *greedy* banale

Procediamo quindi ad implementare l'algoritmo *greedy* banale, cominciando con il solito schema basato su commenti. Prima allochiamo un vettore dinamico di indici interi e lo inizializziamo con gli indici correnti, da 1 a n. Il vettore va deallocato ovviamente al termine della procedura. Poi costruiamo un *heap* indiretto sugli oggetti (tutti, per l'ipotesi che la soluzione iniziale sia vuota, che l'utente ha la responsabilità di garantire), riferito ai loro valori: dovremo passare alla procedura non solo il vettore dei valori e la sua lunghezza, ma anche quello degli indici, che è l'unico effettivamente modificato. Entrambi hanno la stessa lunghezza. Poi applicheremo l'algoritmo *greedy* già descritto, estraendo dall'*heap* l'indice `iMax` dell'elemento di valore massimo (con il quale potremo accedere a tutti i dati), inserendolo in soluzione quando ha volume non superiore alla capacità residua e in ogni caso eliminando dall'*heap* il primo elemento con la classica sovrascrittura da parte dell'ultimo elemento, la riduzione della dimensione dell'*heap* e l'aggiornamento dell'heap indiretto a partire dalla posizione iniziale. Ovviamente, ad ogni inserimento dovremo anche aggiornare il vettore di incidenza e il valore dell'obiettivo nella soluzione, ma anche incrementare il volume totale[2]. Il ciclo termina quando l'*heap* è vuoto. Sovrascrittura e test di vuotezza richiedono di definire una variabile `nEst` che conta gli elementi dell'*heap*, cioè gli elementi ancora candidati a entrare in soluzione.

```
/* Risolve l'istanza I del problema dello zaino con l'euristica
    greedy base.
   Nota: assume che la soluzione S sia inizialmente vuota */
void GreedyBanaleKP (datiKP *pI, soluzioneKP *pS)
```

[2] Se si ritiene che possa essere utile, potrebbe essere un'idea di includerlo nella struttura dati `soluzioneKP`, e potrebbe anche essere sensato definire una procedura per inserire elementi in una soluzione, invece di accedere direttamente ai campi della struttura.

```
{
  vint Indice;
  int nEst;
  int i, iMax;
  int v;


  Indice = (vint) calloc(pI->n+1,sizeof(int));
  if (Indice == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione di Indice!\n");
    exit(EXIT_FAILURE);
  }

  for (i = 1; i <= pI->n; i++)
    Indice[i] = i;

  nEst = pI->n;
  creaheapindiretto(Indice,pI->phi,nEst);

  v = 0;
  while ( (nEst > 0) && (v < pI->V) )
  {
    /* Trova l'elemento di valore massimo */
    iMax = Indice[1];

    /* Se tale elemento sta nello zaino, lo aggiunge alla soluzione */
    if (v + pI->v[iMax] <= pI->V)
    {
      pS->x[iMax] = TRUE;
      pS->f = pS->f + pI->phi[iMax];
      v = v + pI->v[iMax];
    }

    /* Cancella l'elemento dall'heap */
    Indice[1] = Indice[nEst];
    nEst--;
    aggiornaheapindiretto(Indice,pI->phi,nEst,1);
  }

  free(Indice);
}
```

Merita osservare che, se per qualche motivo sapessimo in partenza che alcuni elementi certamente non appartengono a soluzioni ottime, potremmo semplicemente evitare di inserire i loro indici nel vettore. Questo ridurrebbe automaticamente la dimensione del problema (accelerando anche l'algoritmo seguente), un altro piccolo vantaggio dell'indirizzamento indiretto.

14.1.3 Implementazione dell'*heap* indiretto

Avendo proceduto in modalità *top-down*, il codice compila, ma non fa nulla, perché non abbiamo riempito, ma solo dichiarato e definito con corpi vuoti le procedure di gestione dell'*heap* indiretto.

```
void creaheapindiretto (vint Indice, vint V, int n);

void aggiornaheapindiretto (vint Indice, vint V, int n, int i);
```

Riempirle è facile ispirandosi alle analoghe procedure relative all'*heap* diretto, riportate nella Sezione 12.2. La procedura `creaheapindiretto` riceve un vettore di indici interi, uno di valori e una lunghezza, La modifica è quasi banale: si tratta di scorrere dalla posizione mediana fino alla prima, applicando la procedura `aggiornaheapindiretto`, anziché `aggiornaheap`, con il parametro aggiuntivo costituito dal vettore degli indici.

```
void creaheapindiretto (vint Indice, vint V, int n)
{
  int i;

  for (i = n/2; i >= 1; i--)
    aggiornaheapindiretto(Indice,V,n,i);
}
```

La procedura `aggiornaheapindiretto` riceve un vettore di indici interi, uno di valori, una lunghezza e una posizione di riferimento. Determina le posizioni dei due figli come al solito. Poi determina il valore massimo fra padre e figli, come sempre verificando prima se i figli esistono o no. Questa volta, però, i valori da confrontare non si determinano accedendo direttamente al vettore dei dati `V[.]` negli indici `iMax`, `s` e `d`, ma indirettamente passando attraverso il vettore `Indice[.]`. Se l'indice massimo non è quello del padre `i`, si scambiano non i valori contenuti nel vettore dei dati, ma gli indici. Concludiamo chiamando ricorsivamente `aggiornaheapindiretto`, con i relativi quattro argomenti, anziché tre.

```
void aggiornaheapindiretto (vint Indice, vint V, int n, int i)
{
  int s, d;
  int iMax;

  s = 2*i;
  d = 2*i+1;

  iMax = i;
  if ( (s <= n) && (V[Indice[s]] > V[Indice[iMax]]) ) iMax = s;
  if ( (d <= n) && (V[Indice[d]] > V[Indice[iMax]]) ) iMax = d;

  if (iMax != i)
  {
    Scambia(&Indice[iMax],&Indice[i]);
    aggiornaheapindiretto(Indice,V,n,iMax);
  }
}
```

La funzione `Scambia` è del tutto invariata.

Se consideriamo le soluzioni generate, troviamo che per l'istanza da 6 elementi si ottiene la soluzione $\{1, 4\}$, che vale 12, molto meno del valore ottimo pari a 15. Per l'istanza da 10 elementi, il valore ottenuto è 247, anziché 309, cioè ancora peggiore. Per l'istanza da 24 elementi, il valore è abbastanza buono: 13 141 140 anziché 13 549 094 Infine, la soluzione ottenuta sull'istanza da 50 elementi è 981, anziché 1 166. Tutte le soluzioni vengono calcolate in tempi trascurabili. Infatti, la complessità dell'algoritmo è data dal numero di iterazioni del ciclo principale, cioè n, moltiplicato per il tempo necessario ad aggiornare l'*heap* (le altre operazioni richiedono tempo costante), per un totale di $O(n \log n)$, che è chiaramente molto meglio di quella dei due algoritmi esatti discussi nello scorso capitolo.

Si potrebbe anche ragionare su ulteriori miglioramenti. In particolare, l'idea di dover eseguire sempre n iterazioni non sembra giustificata: se si potesse determinare che gli elementi rimasti eccedono la capacità residua, si potrebbe direttamente terminare l'algoritmo. Questo non cambierebbe la complessità nel caso pessimo, ma migliorerebbe quella nel caso medio. D'altra parte, per poterlo fare dovremmo conoscere il volume minimo degli elementi nell'heap, o almeno una sua stima per difetto. Non sviluppiamo ulteriormente questa linea di pensiero, ma è il genere di problemi con i quali ci si scontra regolarmente nella realizzazione pratica di algoritmi euristici per problemi di ottimizzazione. Ovviamente, la cosa ha senso su istanze abbastanza grandi da rendere i tempi di calcolo degni di nota (qui siamo sotto il microsecondo).

14.2 L'algoritmo *greedy* per il problema dello zaino

L'altro lato della questione, rispetto al miglioramento dell'efficienza, è valutare se non si possa migliorare l'efficacia dell'algoritmo, cioè la qualità dei risultati, che al momento è piuttosto scadente. Il fatto che per lo zaino unitario l'algoritmo funzioni bene suggerisce che forse qualcosa si può fare.

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 $\varphi_A(i, x) = \frac{\phi_i}{v_i}$

L'algoritmo risultante tipicamente funziona molto meglio

Figura 14.12: Algoritmi greedy euristici: il KP

Se lo spazio di ricerca non ha le proprietà richieste dal teorema di Rado, si può cercare di mettere una pezza rendendo più sofisticato il criterio di scelta (vedi Figura 14.12). Invece di massimizzare soltanto la funzione obiettivo f, si cerca di tenere conto non soltanto del contributo che ogni elemento dà alla funzione obiettivo, ma anche di quanto può impattare sui vincoli del problema. Il limite fondamentale dell'algoritmo *greedy* banale è che il problema dello zaino ha un vincolo di capacità e che, se riempiamo troppo lo zaino con oggetti di valore alto, ma anche di volume alto, rischiamo di guadagnare meno di quanto si potrebbe. Vorremmo scegliere

oggetti di valore alto, ma al tempo stesso anche di volume basso. Un modo grezzo, ma non stupido, di farlo è massimizzare il rapporto fra il valore ϕ_i dell'elemento scelto e il suo volume v_i. Potremmo chiamare questo rapporto *valore unitario*. Questo algoritmo funziona molto meglio, come possiamo vedere dall'esempio della Figura 14.13 che riporta la solita istanza da 6 oggetti.

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 14.13: Esempio: il *KP*

La tabella riporta i rapporti ϕ_i/v_i e mostra che, massimizzando il valore unitario anziché quello complessivo, non partireo scegliendo l'oggetto a, che ha valore massimo, ma anche volume grande, ma l'oggetto e, che ha un buon valore (4) e un volume molto piccolo (1). Il secondo elemento scelto sarà c, poi d e infine f. Ne deriva una soluzione di valore 14, decisamente migliore di quella trovata in precedenza, ma non ancora ottima.

Perché tutto questo ha senso? Informalmente, lo ha perché decidere in base al valore unitario è un po' come se noi prendessimo ciascuno di questi oggetti e lo dividessimo in pezzi di volume unitario, distribuendo equamente il loro valore fra i vari pezzi. Avremmo quindi 5 pezzi di valore 7/5 che derivano dall'oggetto a, 3 pezzi di valore 2/3 che derivano dall'oggetto b, ecc... In uno zaino unitario, l'algoritmo funzionerebbe correttamente e ci consentirebbe di risolvere esattamente il problema prendendo i pezzi in ordine decrescente di valore. Quindi, accumuleremmo in sequenza i pezzi appartenenti a ciascun oggetto in ordine di valore unitario decrescente, ricomponendoli come se non li avessimo divisi, con una sola (possibile) eccezione: l'oggetto che viola la capacità. Nell'esempio, dopo aver preso e, c e d rimangono due unit'à di capacità residua. Si vorrebbero prendere due pezzi dell'oggetto a, guadagnando un premio pari a $2 \cdot 7/5 = 2.8$, con un totale di 15.8, superiore all'ottimo. Siamo invece costretti a ripiegare sull'oggetto f, guadagnando solo 1, con un totale di 14, e lasciando ancora un'unità di volume inutilizzata. In un certo senso, abbiamo applicato il problema dello zaino unitario fin dove possibile, poi ci siamo fermati e abbiamo ottenuto un ulteriore piccolo miglioramento con l'oggetto

finale. Siccome il problema dello zaino unitario si può risolvere all'ottimo, la sua soluzione è migliore di quella dello zaino (dato che ignora un vincolo fondamentale, e quindi è la migliore in un insieme più ampio) e le abbiamo apportato una modifica ridotta, presumibilmente la soluzione che otterremo non sarà di qualità pessima.

14.2.1 Implementazione dell'algoritmo *greedy*

Proviamo quindi ad implementare questo algoritmo, al posto di quello banale. Si tratta di ragionare non in termini di ϕ massimo, ma di ϕ/v massimo. Non cambia quasi nulla: bisogna sempre lavorare su un *heap* indiretto, ma il vettore dei suoi dati non fa parte direttamente dell'istanza. Possiamo modificare le funzioni di gestione dell'*heap* in modo che ricevano i due vettori di dati, `pI->phi` e `pI->v`, ed eseguano il rapporto `pI->phi/pI->v` ogni volta, ma questo è abbastanza confuso, inelegante e funziona solo in questo caso specifico. Meglio invece costruire un vettore di dati contenente i rapporti e applicare ad esso le funzioni di gestione dell'*heap*, senza cambiare nulla. L'unico problema è che i dati per ò non sono pi ù numeri interi, ma razionali, dunque reali. Quindi dobbiamo comunque mettere mano alle dichiarazioni e definizioni delle funzioni di gestione degli *heap*, ma solo per generalizzare il tipo di dato su cui operano: `vdouble` anziché `vint` (dichiarando il nuovo tipo al principio del file).

```
void creaheapindiretto (vint Indice, vdouble V, int n)
{
  int i;

  for (i = n/2; i >= 1; i--)
    aggiornaheapindiretto(Indice,V,n,i);
}


void aggiornaheapindiretto (vint Indice, vdouble V, int n, int i)
{
  int s, d;
  int iMax;

  s = 2*i;
  d = 2*i+1;

  iMax = i;
  if ( (s <= n) && (V[Indice[s]] > V[Indice[iMax]]) ) iMax = s;
  if ( (d <= n) && (V[Indice[d]] > V[Indice[iMax]]) ) iMax = d;

  if (iMax != i)
  {
    Scambia(&Indice[iMax],&Indice[i]);
    aggiornaheapindiretto(Indice,V,n,iMax);
  }
}
```

A questo punto, possiamo costruire la nuova procedura `GreedyKP`, che riceve un'istanza e restituisce una soluzione, ma non applica l'algoritmo *greedy* banale, bensì quello basato sul rapporto ϕ/v.

```
void creaheapindiretto (vint Indice, vdouble V, int n)
{
  int i;

  for (i = n/2; i >= 1; i--)
    aggiornaheapindiretto(Indice,V,n,i);
```

```
}


void aggiornaheapindiretto (vint Indice, vdouble V, int n, int i)
{
  int s, d;
  int iMax;

  s = 2*i;
  d = 2*i+1;

  iMax = i;
  if ( (s <= n) && (V[Indice[s]] > V[Indice[iMax]]) ) iMax = s;
  if ( (d <= n) && (V[Indice[d]] > V[Indice[iMax]]) ) iMax = d;

  if (iMax != i)
  {
    Scambia(&Indice[iMax],&Indice[i]);
    aggiornaheapindiretto(Indice,V,n,iMax);
  }
}
```

La nuova procedura è praticamente identica alla precedente `GreedyBanaleKP`, ma lavora su un vettore `phi_v` di numeri reali, che va ovviamente allocato, non essendo più parte dei dati. Inoltre, va calcolato inizialmente, ricordando che valore e volume sono numeri interi, e quindi bisogna convertirne almeno uno in un numero reale prima di dividerli, per non eseguire la divisione intera senza resto. Infine, va deallocato al termine. Il resto della procedura è invariato, salvo che l'*heap* indiretto opera sul vettore di valori reali `phi_v`, anziché sul vettore intero `pI->phi`, ma questa modifica è già stata preparata. Ovviamente, questa modifica rende non più utilizzabile la vecchia procedura `GreedyBanaleKP`.

```
/* Risolve l'istanza I del problema dello zaino con l'euristica
    greedy base.
   Nota: assume che la soluzione S sia inizialmente vuota */
void GreedyKP (datiKP *pI, soluzioneKP *pS)
{
  vint Indice;
  vdouble phi_v;
  int nEst;
  int i, iMax;
  int v;


  Indice = (vint) calloc(pI->n+1,sizeof(int));
  if (Indice == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione di Indice!\n");
    exit(EXIT_FAILURE);
  }

  for (i = 1; i <= pI->n; i++)
    Indice[i] = i;

  phi_v = (vdouble) calloc(pI->n+1,sizeof(double));
  if (phi_v == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione di phi_v!\n");
    exit(EXIT_FAILURE);
  }

  for (i = 1; i <= pI->n; i++)
    phi_v[i] = (double) pI->phi[i]/pI->v[i];
```

```
  nEst = pI->n;
  creaheapindiretto(Indice, phi_v, nEst);

  v = 0;
  while ( (nEst > 0) && (v < pI->V) )
  {
    /* Trova l'elemento di valore massimo */
    iMax = Indice[1];

    /* Se tale elemento sta nello zaino, lo aggiunge alla soluzione */
    if (v + pI->v[iMax] <= pI->V)
    {
      pS->x[iMax] = TRUE;
      pS->f = pS->f + pI->phi[iMax];
      v = v + pI->v[iMax];
    }

    /* Cancella l'elemento dall'heap */
    Indice[1] = Indice[nEst];
    nEst--;
    aggiornaheapindiretto(Indice, phi_v, nEst, 1);
  }

  free(Indice);
  free(phi_v);
}
```

Eseguendo il nuovo algoritmo sull'istanza da 6 oggetti, sappiamo che otterremo una soluzione di valore 14 anziché 12 (contro un ottimo di 15). Sull'istanza da 10 oggetti, raggiungiamo l'ottimo. Sull'istanza da 24 oggetti, otteniamo 13 415 886 anziché 13 141 140 (contro un ottimo pari a 13 549 094). Infine, sull'istanza da 50 oggetti, otteniamo 1153 anziché 981 (contro un ottimo pari a 1166). Arriviamo regolarmente molto più vicino alla soluzione ottima. I tempi sono praticamente identici a quelli dell'euristica banale: la gestione del nuovo vettore di numeri reali non aggiunge quasi nulla al tempo di calcolo.

Un controesempio Pur nella sua buona efficacia media, l'algoritmo ha però prestazioni pessime in alcune situazioni. Nella Figura 14.14, consideriamo un problema di zaino con due oggetti a e b, rispettivamente di valore 10 e 90 e di volume 1 e 10. Supponiamo che lo zaino abbia capacità 10. È chiaro che la soluzione ottima consiste nel prendere l'oggetto b, che vale 90. In questo caso, l'algoritmo *greedy* banale trova l'ottimo, mentre quello più intelligente, basato su valore unitario, preferirebbe scegliere l'oggetto a, che ha rapporto $10/1 = 10$, strettamente superiore a quello di b ($90/10 = 9$), per poi trovarsi impossibilitato a prendere alcun altro oggetto e costretto a lasciare lo zaino quasi vuoto. Esistono istanze come queste in cui l'euristica più raffinata può esibire uno scarto grande a piacere tra l'ottimo e il valore trovato.

Tuttavia, si può fare una piccola modifica che rende l'algoritmo dimostrabilmente 2-approssimato. Che cosa si intende con questa espressione? Che il rapporto tra la soluzione ottima e quella determinata dall'algoritmo modificato non può essere maggiore di 2, cioè che l'algoritmo trova almeno metà del valore ottimo. Metà non è un gran che, ma è molto meglio che avere un risultato illimitatamente cattivo.

L'algoritmo 2-approssimato (si veda la Figura 14.15) consiste nell'affiancare all'algoritmo basato sul valore unitario la ricerca dell'elemento di valore massimo che non eccede la capacità dello zaino. Tale elemento è in effetti una soluzione, anche se molto particolare. Abbiamo visto che il difetto dell'algoritmo riguarda il caso in

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 14.14: Istanze critiche del KP per l'algoritmo greedy

cui un elemento di valore massimo ha un volume molto grande, che rende impossibile sceglierlo dopo aver preso altri elementi molto più piccoli di valore unitario leggermente superiore.

Si può dimostrare con un paio di semplici passaggi algebrici che la migliore fra le due soluzioni (quella dell'algoritmo raffinato e quella fornita dall'elemento massimo) vale almeno metà dell'ottimo Il problema dello zaino ha ulteriori proprietà, in base alle quali non solo esiste un algoritmo polinomiale 2-approssimato e un algoritmo esponenziale esatto, ma esiste addirittura tutta una famiglia parametrica di algoritmi polinomiali che migliorano via via la garanzia di approssimazione avvicinandosi sempre più all'ottimo al prezzo di aggravare passo per passo la complessità computazionale, con polinomi di grado crescente.

14.2.2 Proprietà algebriche del problema dello zaino

Nella sezione precedente abbiamo applicato l'algoritmo *greedy* al problema dello zaino, scoprendo che in generale non è corretto, ma che lo è nel caso particolare dello zaino unitario. Abbiamo anche visto che sostituendo il valore degli oggetti con il loro valore unitario le prestazioni migliorano. A questo punto, è interessante discutere le ragioni per le quali questo algoritmo funzioni su un problema e non sull'altro. Il motivo è la struttura algebrica dei due problemi.

Come si è visto (Figura 14.11), un problema di Ottimizzazione Combinatoria richiede per definizione un *insieme base* B, di cui le soluzioni devono essere sottoinsiemi. Nel caso del problema dello zaino, tale insieme è quello degli oggetti. Per poter applicare un algoritmo *greedy*, inoltre, occorre definire uno *spazio di ricerca* $\mathcal{F}$, più correttamente *collezione di indipendenti*, tale che la coppia $(B, \mathcal{F})$ costituisca un sistema di indipendenza dotato di opportune proprietà algebriche. Nel caso

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': $f_A = \max\left[f\left(x\right), f\left(x'\right)\right]$

È facile dimostrare che

- la somma delle due soluzioni è una stima per eccesso dell'ottimo

$$f\left(x\right) + f\left(x'\right) = \sum_{\tau=1}^{t} \phi_{i_\tau} \geq f^*$$

- la migliore delle due soluzioni è almeno metà della somma

$$f_A = \max\left[f\left(x\right), f\left(x'\right)\right] \geq \frac{f\left(x\right) + f\left(x'\right)}{2} \geq \frac{1}{2} f^*$$

Figura 14.15: Un algoritmo 2-approssimato per il *KP*

del problema dello zaino, abbiamo proposto come spazio di ricerca la collezione di tutti i sottoinsiemi di volume totale non superiore alla capacità dello zaino. Tali sottoinsiemi, infatti, sono promettenti come parti di soluzioni (in effetti, nel caso dello zaino sono soluzioni ammissibili).

Il teorema di Rado dimostra che l'algoritmo *greedy* risolve qualunque problema di Ottimizzazione Combinatoria con funzione obiettivo additiva e soluzioni date dai sottoinsiemi massimali se e solo se valgono alcune proprietà di base. Nel caso dello zaino, la funzione obiettivo è additiva. Le soluzioni non si riducono ai sottoinsiemi massimali, ma è chiaro che questi sono migliori di quelli non massimali, per cui possiamo restringere l'attenzione ai sottoinsiemi massimali senza pregiudicare la ricerca dell'ottimo. Possiamo quindi passare in rassegna le tre proprietà per verificarne la validità.

La prima proprietà richiede che il sottoinsieme vuoto sia un indipendente, cioè faccia parte della collezione $\mathcal{F}$. Questa proprietà serve perché l'algoritmo *greedy* parte dal sottoinsieme vuoto e deve rimanere sempre all'interno dell'insieme degli indipendenti. Quindi è una condizione ovviamente necessaria. Nel caso dello zaino, l'insieme vuoto ha volume nullo, e quindi rispetta la capacità, e la prima proprietà è soddisfatta.

La seconda proprietà richiede che, se un sottoinsieme è un indipendente, allora tutti i suoi sottoinsiemi propri siano anch'essi indipendenti. La ragione è che vogliamo arrivare a una soluzione aggiungendo un elemento alla volta. Con un'ipotesi di chiusura rispetto all'inclusione, imponiamo in modo molto forte che si possa costruire un sottoinsieme in qualunque modo, attraverso qualsiasi sequenza di inserimenti di singoli elementi in qualunque ordine. La condizione non appare strettamente necessaria, ma è chiaramente utile. Nel caso dello zaino, la proprietà vale ovviamente, perché qualunque sottoinsieme di oggetti di volume totale non superiore alla capacità ha parti proprie di volume inferiore, e quindi ancora rispettoso della capacità.

La terza proprietà è più sofisticata. Dati due indipendenti, uno dei quali abbia cardinalità leggermente maggiore (un elemento in più), si richiede che sia sempre possibile prendere almeno uno degli elementi dal sottoinsieme più grande, che non faccia già parte del più piccolo (altrimenti, sarebbe inutile) e aggiungerlo al sottoinsieme più piccolo (ingrandendolo), in modo che il risultato rimanga nella collezione degli indipendenti.

Nel problema dello zaino, possiamo pensare, per esempio, di avere un insieme di tre oggetti e uno di due, entrambi ammissibili rispetto alla capacità. La proprietà richiede che si possa sempre prendere uno dei tre oggetti e aggiungerlo all'insieme da due. È chiaro che in generale questo non funziona, perché se la capacità vale 6, un insieme di tre oggetti di volume 2 e uno insieme di due oggetti di volume 3 sono entrambi ammissibili. D'altra parte, nessuno dei tre oggetti può essere aggiunto al sottoinsieme da due mantenendo l'ammissibilità, perché l'insieme è saturo e qualunque oggetto si aggiunga viola la capacità. Quindi, la terza proprietà nel caso del problema dello zaino non vale, ma per il più specifico problema dello zaino unitario invece vale, perché il vincolo di capacità si riduce a un semplice vincolo di cardinalità: se il sottoinsieme da tre elementi è ammissibile, qualunque suo elemento aggiunto a quello da due elementi lo allarga in maniera ammissibile. Il teorema di Rado garantisce quindi che, per qualunque funzione obiettivo additiva, l'algoritmo *greedy* ottiene una soluzione ottima per il problema dello zaino unitario, mentre per alcune funzioni additive non ottiene soluzione ottima per il problema dello zaino generico.

Questo è interessante perché è estendibile a diversi altri problemi. La sezione seguente riguarda uno di questi problemi, molto più interessante e sofisticato del

prendere i k oggetti più preziosi di un insieme dato.

14.3 Il problema dell'albero ricoprente minimo

Introducendo i grafi nel Capitolo 8, abbiamo definito un grafo non orientato come un insieme di vertici e un insieme di lati (cioè coppie non ordinate di vertici). Vogliamo un grafo connesso perché ne cercheremo un sottografo connesso. Il grafo avrà anche una funzione di costo che associa ad ogni lato del grafo un numero, per semplicità un numero naturale (vedi Figura 14.16).

Di questo grafo cerchiamo un sottografo che abbia costo totale minimo fra quelli che godono di tre proprietà fondamentali. La prima è di essere ricoprente, cioè che l'insieme dei vertici del fotografo coincide con quello del grafo. La seconda è che il sottografo sia connesso, cioè che qualunque coppia di vertici si prenda in esso (e quindi nell'intero grafo) deve avere un cammino che li collega fatto di lati appartenenti al sottografo. La terza proprietà è che il sottografo sia aciclico, cioè non contenga cammini chiusi che tornano sullo stesso vertice. Poiché un sottografo un grafo connesso e aciclico è definito albero, ci interessano gli alberi ricoprenti. Fra questi ne cerchiamo uno di costo totale minimo, dove il costo è inteso come la somma dei costi dei lati che fanno parte dell'albero.

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 $c_X = \sum_{e \in X} c_e$

Figura 14.16: Il problema dell'albero ricoprente minimo

Un esempio di grafo non orientato del quale determineremo l'albero ricoprente minimo è riportato nella Figura 14.17.

A che cosa serve questo problema? Il problema ha molte applicazioni. Per esempio, serve a costruire reti elettriche, di comunicazione, ecc... ma ci sono anche applicazioni più astratte.

Inoltre, è chiaramente un problema di Ottimizzazione Combinatoria, perché il numero di soluzioni è finito. Le soluzioni sono sottografi, il che non significa direttamente sottoinsiemi (un sottografo è una coppia di oggetti, non un oggetto singolo), ma essendo ricoprenti l'insieme dei vertici è automaticamente determinato:

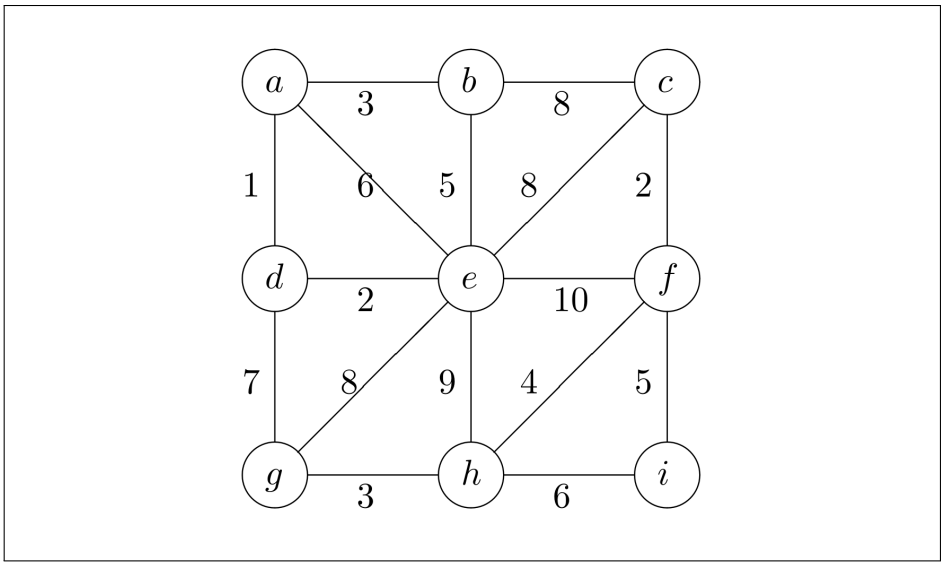


Figura 14.17: Esempio di problema dell'albero ricoprente minimo

rimane l'insieme dei lati dell'albero, che è un sottoinsieme dell'insieme dei lati del grafo. Quindi, l'insieme base è l'insieme dei lati del grafo (vedi Figura 14.18).

Ora ci serve una collezione di indipendenti, ovvero uno spazio di ricerca, costituito da sottoinsiemi di lati che sono promettenti in quanto parti potenziali di una soluzione. Vi sono diverse risposte possibili, ma la più semplice è costituita dai sottoinsiemi ricoprenti aciclici, cioè dalle *foreste ricoprenti*. Il motivo è che, dovendo necessariamente considerare insiemi di pochi lati, non possiamo pretendere di mantenere la connessione totale, mentre non è difficile conservare un sottografo ricoprente (basta tenere tutti i vertici) e aciclico. D'altra parte, un sottoinsieme contenente cicli è chiaramente dannoso, dato che nessuna soluzione ammissibile potrebbe contenerlo come parte: conterrebbe anche il ciclo.

L'algoritmo che ne deriva, noto come *algoritmo di Kruskal* dal nome dello scopritore, è molto semplice. Parte con un sottoinsieme di lati vuoto. Poi, di passo in passo, trova il lato e^* di costo minimo fa quelli che non appartengono già alla soluzione e, aggiunti ad essa, non chiudono cicli. Poiché non sono condizioni elementari da valutare ogni volta, e poicé un lato che ne viola una continuerà a farlo in tutti i passi seguenti, via via che la soluzione si allarga, faremo come nel caso dello zaino, cioè ogni volta considereremo semplicemente il lato di costo minimo e poi valuteremo se il lato scelto fa uscire dallo spazio di ricerca, ovvero viola i vincoli del sistema di indipendenza (l'aciclità). In caso negativo, aggiungeremo il nuovo lato alla soluzione e in ogni caso lo cancelleremo da ogni futura considerazione, esattamente come nel caso dello zaino cercavamo l' elemento di valore massimo e poi verificavamo se rispettasse la capacità oppure no, inserendolo solo nel primo caso ed eliminandolo comunque da ogni futura indagine. Nel caso dell'albero ricoprente minimo, determiniamo il lato di costo minimo e rimuoviamo quelli che chiudono cicli perché continueranno a farlo in ogni nessun passo successivo, mentre aggiungiamo alla soluzione quelli che non chiudono cicli e li rimuoviamo perché non sono più esterni alla soluzione. Di conseguenza, l'algoritmo gestisce la soluzione come un insieme di lati X e l'insieme di lati potenzialmente interessanti E' (inizialmente pari all'intero insieme E del grafo) come un *min-heap*, dato che ad ogni passo bisognerà determinare quello di costo minimo. Rispetto allo zaino, cambia il verso

Applichiamo l'algoritmo *greedy* usando come

- insieme base l'insieme dei lati del grafo ($B = E$)
- indipendenti le foreste ricoprenti il grafo

Quindi, l'algoritmo

1. parte con $X = \emptyset$
2. trova il lato di costo minimo $e^* = (u^*, v^*)$ non in X e non scartato
   - se $X \cup \{e^*\}$ non contiene cicli (cioè u^* e v^* non sono connessi in X)
     aggiunge e^* a X
   - se $X \cup \{e^*\}$ contiene cicli, scarta e^* permanentemente
     (ogni sottoinsieme più ampio conterrebbe cicli)

Figura 14.18: Algoritmo di Kruskal

dell'ottimizzazione e si sostituisce il test sulla capacità con un test di aciclicità.

L'inversione del verso dell'ottimizzazione non cambia le condizioni algebriche di ottimalità. Come discusso nelle dispense di teoria, a cui rimandiamo, l'algoritmo *greedy* non determina solo il sottoinsieme di valore massimo, ma anche quello di costo minimo fra i sottoinsiemi massimali, cioè non estendibili aggiungendo elementi.

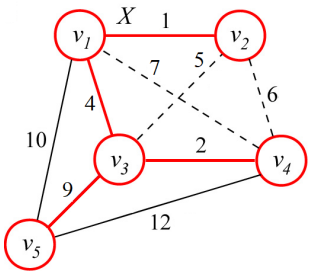


Figura 14.19: Esempio

La Figura 14.19 riporta un esempio di applicazione dell'algoritmo di Kruskal. Dato un grafo da 5 vertici con costi sui lati, partiamo con un insieme di lati X vuoto. Il lato di costo minimo dell'intero grafo è (v_1, v_2) che costa 2. Siccome non chiude cicli, viene aggiunto alla soluzione X e cancellato da E'. Il lato successivo è (v_3, v_4), che con (v_1, v_2) non chiude cicli, e quindi viene aggiunto anch'esso. Segue (v_1, v_3), che connette i due lati precedenti, ma non chiude cicli, e quindi viene aggiunto. Adesso si ottiene (v_2, v_3), che costa 5, ma questo lato chiude il ciclo (v_2, v_3, v_1), e quindi viene scartato. Analogamente, vengono scartati (v_2, v_4) e (v_1, v_4). Il lato (v_3, v_5) non chiude alcun ciclo e viene aggiunto. Teoricamente, dovremmo procedere con i lati (v_1, v_5) e (v_4, v_5), ma possiamo terminare anticipatamente l'esecuzione sfruttando una proprietà degli alberi che abbiamo già discusso: gli alberi hanno

un numero di lati pari al numero dei vertici meno 1. Gli alberi ricoprenti, quindi, hanno esattamente $|V| - 1$ lati, cioè in questo caso 4. Siccome abbiamo già 4 lati, è impossibile che i lati seguenti entrino nella soluzione. Questa condizione non è necessaria alla correttezza dell'algoritmo, ma lo rende più efficiente, esattamente come nel problema dello zaino si poteva terminare l'algoritmo quando la capacità diventava insufficiente a ospitare gli elementi non ancora considerati.

14.3.1 Implementazione efficiente dell'algoritmo di Kruskal

A questo punto, affrontiamo il problema di implementare tutto questo con l'efficienza migliore possibile, dunque di quali strutture utilizzare per rappresentare la soluzione X e l'insieme E' dei lati non ancora considerati. Lo pseudocodice in Figura 14.18 riporta aggiunte e cancellazioni di lati, che sappiamo fare in tempo costante in molte strutture dati diverse, ma anche due operazioni più complicate: la determinazione ed estrazione dell'arco di costo minimo da E' e la valutazione di aciclicità dell'insieme $X \cup e^*$.

Gestire E' come *min-heap*

- costruirlo: $\Theta(m)$
- estrarre il minimo: $\Theta(1)$
- aggiornarlo: $\Theta(\log m)$

Gestire X come *foresta con bilanciamento*

- costruirla: $\Theta(n)$
- trovare le componenti di u^* e v^*: $\approx \Theta(1)$
- unire le componenti: $\Theta(1)$

Figura 14.20: Complessità temporale

La Figura 14.20 confronta lo pseudocodice più astratto visto in precedenza con una versione più dettagliata che affronta i sottoproblemi in modo più efficiente, nei limiti delle strutture discusse in questo corso. La determinazione ed estrazione del lato di costo minimo ad ogni passo si risolve ovviamente gestendo E' come un *min-heap*, che è una struttura del tutto analoga al *max-heap*, salvo mantenere una proprietà complementare sulle etichette dei nodi, in base alla quale ogni nodo ha etichette non superiori a quelle degli eventuali nodi figli. Ovviamente, questo comporta di costruire un *heap* al principio (in tempo lineare rispetto al numero di lati, $O(m)$), estrarre il minimo in tempo costante ($O(1)$), eliminare e aggiornare il primo elemento ad ogni iterazione in tempo logaritmico, $O(\log m)$[3]. Si tratterà anche di gestire l'*heap* in modo indiretto, perché oltre al costo del lato è necessario conoscere il lato stesso, per poterlo aggiungere alla soluzione.

Si apre a questo punto il problema del test di aciclicità, che richiede una struttura dati ancora non discussa. Prima di introdurla, mostriamo che il test di aciclicità può facilmente essere ricondotto a un altro problema che abbiamo già affrontato. Consideriamo il sottografo costituito da tutti i vertici e dai lati della soluzione X corrente. Questo grafo ha delle componenti connesse indotte dai lati di X. Ora consideriamo il lato e^*, di estremi u^* e v^*. Se essi appartengono alla stessa componente connessa indotta da X, esiste un cammino composto da lati in X che li collega. Quindi, aggiungere e^* chiuderebbe un ciclo. Viceversa, se sono in componenti distinte, aggiungere e^* fonde le due componenti in una sola, lasciando invariate tutte le altre componenti. Ne deriva che un test di aciclicità equivale a determinare

[3] E qui merita osservare che $m \leq n^2$, per cui $\log m \leq 2 \log n$, e quindi $O(\log m) = O(\log n)$, ignorando le costanti moltiplicative.

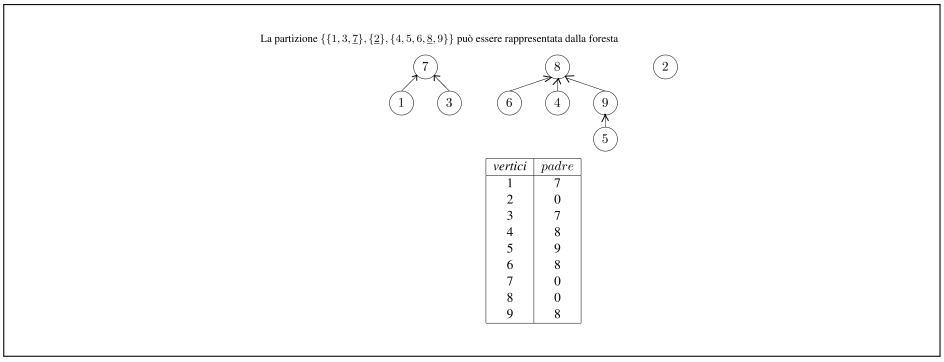
se due vertici siano o no nella stessa componente connessa, in una situazione in cui le componenti vanno modificandosi iterativamente per fusioni successive.

14.3.2 Foreste con bilanciamento (strutture "union-find")

Capita spesso di non dover rappresentare un semplice insieme di oggetti, ma la partizione di un insieme in sottoinsiemi. Un esempio che abbiamo già discusso è la divisione dei vertici di un grafo non orientato in componenti connesse.

Nel Capitolo 9), avevamo rappresentato le componenti connesse con un *vettore di marcatura*, cioè un vettore che associa ad ogni elemento l'indice progressivo della componente connessa di cui fa parte o l'indice di un vertice di questa assunto come capocomponente. È un'ottima struttura per determinare a quale componente appartenga un vertice e per aggiungere vertici a una componente, dato che entrambe le operazioni richiedono tempo costante. A noi però serve fondere due componenti in una sola. Con il vettore di marcatura, questo richiede lo scorrimento di tutto il vettore in tempo lineare ($O(n)$).

Usando liste (una per ogni componente), la fusione richiederebbe tempo costante: basterebbe appendere una delle due liste all'altra, manipolando opportunamente le posizioni (puntatori o cursori che siano). Ma determinare a quale componente appartenga un vertice richiederebbe lo scorrimento di tutte le liste in tempo $O(n)$. Mantenere entrambe le strutture e usare l'una o l'altra secondo il caso funzionerebbe se le due operazioni avessero una frequenza molto diversa tra loro, ma nell'algoritmo di Kruskal la fusione di due componenti avviene ogni volta che un lato ha i due vertici in componenti diverse, dunque potenzialmente metà delle volte.



vertici	*padre*
1	7
2	0
3	7
4	8
5	9
6	8
7	0
8	0
9	8

Figura 14.21: Foreste con bilanciamento

Adotteremo quindi una struttura più sofisticata, detta *foresta con bilanciamento*, o strutture "union-find" o "merge-find set". Passiamo rapidamente in rassegna le idee fondamentali su cui si basano. La Figura 14.21 riporta un insieme di 9 oggetti divisi in tre sottoinsiemi: $\{1,3,7\}$, $\{4,5,6,8,9\}$ e $\{2\}$. Ognuno è rappresentato da un albero, che non ha nulla a che vedere con la topologia degli oggetti. Anche se nel caso dell'albero ricoprente minimo gli oggetti sono incidentalmente vertici, le frecce nella figura, che sono chiaramente orientate, non corrispondono ai lati del grafo, ma servono a indicare che gli elementi 1 e 3 fanno parte della stessa componente dell'elemento 7, ecc... La radice di ogni albero di questa foresta è un capocomponente scelto convenzionalmente. La struttura è facilmente descrivibile con il vettore "padre", di cui si è già discusso nella Sezione 9.4.5 parlando della visita di grafi come strumento per rappresentare le componenti, anche se in quel caso gli archi dell'albero corrispondevano effettivamente ad alcuni dei lati del grafo. Il

vettore padre indica per ogni elemento l'elemento padre nella foresta che rappresenta l'intera partizione. Gli oggetti radice sono identificati da un valore convenzionale, che può essere 0 (come nella figura) oppure l'indice stesso di ciascun elemento, come se ci fosse nell'albero un autoanello.

Ora si tratta di capire come eseguire su questa struttura le operazioni che ci interessano. Al principio, tutti i vertici del grafo sono isolati e formano una componente connessa a sé. Il corrispondente vettore contiene in ogni posizione il valore dell'indice corrispondente. Per determinare a quale componente appartenga un elemento, bisogna risalire l'albero di figlio in padre, fino alla radice. Questo richiede tempo proporzionale alla profondità dell'albero, dunque certamente superiore al tempo costante richiesto dal vettore di marcatura. La funzione *Find* dello pseudocodice corrisponde appunto a questa operazione. Per capire la convenienza, bisogna considerare la fusione di due alberi, che corrisponde alla funzione *Union*. Affinché due alberi diventino uno solo, basta appendere la radice dell'uno a un nodo qualsiasi dell'altro, per esempio alla radice. Per esempio, volendo fondere i sottoinsiemi $\{1, 3, 7\}$ e $\{4, 5, 6, 8, 9\}$, per esempio perché abbiamo aggiunto alla soluzione il lato $(3, 6)$, non faremo altro che appendere l'elemento 7 all'elemento 8 cambiando `padre[7]` da 7 a 8. Il tutto richiede tempo costante, se si conoscono le due radici. L'albero, infatti, non ha nessun vincolo sul grado; per esempio, non c'è nessun motivo perché sia binario. Inoltre, nell'algoritmo di Kruskal fondiamo due alberi quando abbiamo determinato che due vertici sono in componenti diverse, perché i loro capocomponente sono diversi, il che significa che conosciamo le radici dei due alberi. Appendere una radice direttamente all'altra ha il vantaggio che l'albero risultante ha un'altezza più limitata che appendendo la radice di un albero a un nodo generico dell'altro. Questo avrà una ripercussione positiva sulle successive operazioni di ricerca, che (per quanto non costanti) rimarranno abbastanza efficienti. Infatti, un teorema garantisce che, appendendo la radice dell'albero con meno elementi a quella dell'albero con più elementi la profondità massima dell'albero non supererà $\log_2 n$, e questo limite si riflette immediatamente sulla complessità della ricerca[4]. È questo il "bilanciamento" a cui fa riferimento il nome della struttura dati. Il risultato è ulteriormente migliorabile con un interessante raffinamento che vedremo in seguito. Ovviamente, per poter applicare quest'idea, bisogna conoscere la cardinalità di ciascuna componente (vedremo come).

14.3.3 Complessità

Discussi i singoli sottoproblemi, possiamo caratterizzare la complessità dell'algoritmo di Kruskal. Partiamo preparando le strutture dati ausiliarie, dunque creando un *heap* indiretto sui lati, in tempo $O(m)$, e una foresta bilanciata sui vertici, in tempo $O(n)$ (per ogni vertice il vettore padre riporta l'indice del vertic stesso). Il ciclo principale viene eseguito un numero di volte indeterminato, ma certamente compreso fra $n - 1$ (nel caso fortunato in cui nessun lato chiude ciclo) e m (nel caso sfortunato in cui il lato di costo massimo fa parte della soluzione ottima). Nel ciclo, determiniamo il lato di costo minimo con un'operazione in tempo costante, aggiorniamo l'*heap* con un'operazione in tempo $O(\log n)$, determiniamo i due capocomponenti dei vertici estremi del lato corrente, ancora in tempo $O(\log n)$, eventualmente fondiamo i due alberi in tempo costante e aggiungiamo un lato alla soluzione corrente, ancora in tempo costante. Come vedremo in seguito, in effetti, la ricerca dei capocomponenti si ridurrà a un tempo che è crescente con la dimensio-

[4] Un'idea analoga, e forse ancora più naturale è appendere l'albero di altezza minore a quello di altezza maggiore, dato che il risultato avrà la stessa altezza dell'albero maggiore (a meno che non siano inizialmente identiche, nel qual caso il risultato aumenta l'altezza di un'unità.

ne dell'istanza, ma in maniera talmente lenta che si può praticamente considerare costante. In totale, la complessità sarà data da $O(m \log n)$.

14.3.4 Avvio dell'implementazione dell'algoritmo di Kruskal

Mettiamo ora in pratica le idee discusse sopra. Si tratta di realizzare un codice molto simile a quello dell'algoritmo *greedy* per il problema dello zaino. Il file `kruskal0.c` contiene già la solita funzione `InterpretaLineaComando` per ricevere dall'utente il nome del file che contiene i dati. Questo avrà una struttura leggermente più complicata di quella usata nel Capitolo 8. Anziché una semplice lista di lati fra parentesi tonde, abbiamo l'esplicita indicazione del numero dei vertici e dei lati (precedute da `n =` e `m =`, e una lista di lati, ciascuno seguito dal valore del proprio costo. Per esempio, il file `grafo1.txt` riporta il grafo rappresentato nella Figura 14.17 nel formato appena introdotto:

```
   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
```

Per descrivere i dati usiamo un grafo dichiarato come lista di lati, dunque con la libreria `grafo-la.h` già discussa nel Capitolo 8, che contiene anche una procedura `CaricaDati` per caricare il grafo dal file nel formato sopra descritto. A sua volta, questa richiede l'uso della libreria `listaarchi.h`.

Perché fra le tre strutture abbiamo scelto la lista di lati? Perché l'algoritmo di Kruskal ha solo bisogno di scorrere i lati: la minimizzazione dei costi è già affidata a un *heap*, non occorre accedere a un lato dai suoi estremi e non occorre conoscere la topologia del grafo, dato che le componenti connesse saranno gestite da una foresta con bilanciamento, non eseguendo visite del grafo. Di conseguenza, scegliamo la rappresentazione più compatta e semplice. A rigore, siccome il grafo è caricato una volta per tutte al principio dell'esecuzione, nulla vieterebbe di usare addirittura un vettore di lati, risparmiando tutti i puntatori. Non ci spingiamo oltre per riutilizzare la libreria già disponibile. In realtà, però, tale libreria rappresenta un grafo non pesato. Nel nostro caso, il grafo è invece pesato. Due strade si aprono: tenere l'informazione sul costo in una struttura *ad hoc* oppure conservarla nella struttura grafo. In questo caso, poi, l'informazione potrebbe essere conservata nel grafo o in ciascun arco. Abbiamo scelto la prima strada[5] Un arco, quindi, avrà una coppia origine-destinazione e due puntatori per muoversi lungo la lista, ma anche un costo intero.

```
}


void distruggelistaarchi (listaarchi *pL)
{
  posarco p;
```

[5]E questa è una scelta che amerei cambiare, dato che non aggiunge molto in chiarezza e complica inutilmente le cose. Però separare archi e costi richiederebbe di definire il concetto di indice di un arco, per tenere "sincronizzata" la lista nel grafo e il vettore dei costi e comporterebbe che l'*heap* indiretto indirizzi in qualche modo non solo gli archi ma anche i costi. Problemi di lana caprina, ma una soluzione formalmente valida e semplice sarebbe auspicabile.

Questo ha portato un po' di modifiche nella lettura e scrittura degli archi appartenenti alla lista, nell'inserimento di archi e nella stampa del grafo, nonché l'aggiunta di una funzione `leggecosto` per leggere il costo di un arco data la sua posizione.

È poi già dichiarata una struttura per descrivere le soluzioni. Una soluzione è un sottoinsieme di lati. Potrebbe essere una lista, ma la dinamicità richiesta è veramente minima: bisogna aggiungere via via lati fino ad ottenerne $n-1$. Va benissimo una tabella, che potrebbe contenere posizioni di lati oppure direttamente l'informazione che ci interessa, cioè i due vertici estremi di ogni lato. Per quanto l'indirizzamento indiretto eviterebbe ogni ridondanza, abbiamo preferito riportare direttamente i due vertici (anche se sono copie di informazioni contenute in altre celle), perché lo spreco è minimo (due interi per lato anzichè un puntatore). Inoltre, anziché definire una tabella di coppie di vertici si è preferito usare due vettori di interi, per pura semplicità. Alla dimensione allocata `dim`, alla dimensione effettiva `nl` e ai due vettori `n1` e `n2` abbiamo aggiunto il valore della funzione obiettivo. Notiamo che questa struttura è in grado di rappresentare non solo soluzioni ammissibili, ma anche soluzioni parziali, il che è fondamentale in un algoritmo *greedy*, che procede aumentando via via una soluzione parziale (in generale non ammissibile) fino a terminare.

```
typedef struct _soluzioneMSTP soluzioneMSTP;
struct _soluzioneMSTP
{
  int f;
  int dim;
  int nl;
  vnodo v1;
  vnodo v2;
};
```

La struttura `soluzioneMSTP` è accompagnata dalle procedure per creare, distruggere e stampare soluzioni (su due righe, la prima delle quali riporta il costo totale della soluzione, preceduto dalle parole chiave `f* =`, mentre la seconda riporta l'elenco dei lati nello stesso formato usato per il file dei dati, ma preceduto dalle parole chiave `x* =`).

```
/* Crea una soluzione vuota */
void CreaSoluzione (int n, soluzioneMSTP *pS)
{
  pS->f = 0;
  pS->dim = n;
  pS->nl = 0;
  pS->v1 = (vint) calloc(pS->dim+1,sizeof(int));
  if (pS->v1 == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione del vettore dei primi
        estremi!\n");
    exit(EXIT_FAILURE);
  }
  pS->v2 = (vint) calloc(pS->dim+1,sizeof(int));
  if (pS->v2 == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione del vettore dei secondi
        estremi!\n");
    exit(EXIT_FAILURE);
  }
}


/* Distrugge la soluzione S */
```

```
void DistruggeSoluzione (soluzioneMSTP *pS)
{
  pS->f = 0;
  pS->dim = pS->nl = 0;
  free(pS->v1);
  pS->v1 = NULL;
  free(pS->v2);
  pS->v2 = NULL;
}


/* Stampa la soluzione */
void StampaSoluzione (soluzioneMSTP *pS)
{
  int i;

  printf("f* = %d\n",pS->f);
  printf("x* = ");
  for (i = 1; i <= pS->nl; i++)
    printf(" (%d,%d)",pS->v1[i],pS->v2[i]);
  printf("\n");
}
```

14.3.5 Implementazione dell'algoritmo di Kruskal

La procedura `Kruskal`, che risolverà il problema, riceve il grafo e restituisce la soluzione.

```
/* Risolve l'istanza G del MSTP con l'algoritmo di Kruskal
    restituendo la soluzione S */
void Kruskal (grafo *pG, soluzioneMSTP *pS);
```

Facciamo il solito schema attraverso commenti. L'algoritmo seguirà la struttura base di tutti gli algoritmi *greedy*. Parte con una soluzione vuota, che ancora una volta ipotizzeremo di ricevere dal programma principale, subito dopo averla creata.

Dobbiamo costruire un *min-heap* che contenga tutti gli archi del grafo, per poterne estrarre via via quello di costo minimo. Ovviamente, si pone il problema del tipo di *heap*. Se contenesse solo i valori dei costi, non potremmo determinare il lato da aggiungere via via alla soluzione. Se contenesse i lati, ad ogni scambio dovremmo muovere diverse informazioni. Abbiamo già concluso che l'*heap* indiretto è la soluzione più efficiente. Nel caso dello zaino, essendo gli oggetti associati a numeri interi, abbiamo implementato l'*heap* indiretto come vettore di indici interi, che consentivano di accedere ai vettori dei dati. Qui la cosa è più complicata, perché i dati non sono in un vettore, ma in una lista. Ovviamente, potremmo copiare i dati in un vettore, o implementare direttamente il grafo con una tabella di lati senza duplicarli. Vediamo però una soluzione alternativa, che consente di realizzare un *heap* indiretto su dati contenuti in qualsiasi tipo di struttura. L'idea è costruire l'*heap* indiretto come vettore di puntatori ai dati, cioè ai lati del grafo. In sostanza, prima allocheremo un vettore di puntatori a lati e vi inseriremo gli indirizzi dei lati del grafo. Poi costruiremo un *min-heap* indiretto su questo vettore.

Quindi, costruiremo una foresta con bilanciamento sui vertici del grafo, dove inizialmente ogni vertice è una componente a sé.

Avviamo l'algoritmo vero e proprio, che procede finché la soluzione non è composta da $n-1$ lati, oppure finché l'*heap* non è vuoto. Se il grafo è connesso, troveremo

un albero prima di terminare i lati, altrimenti, svuoteremo il grafo prima di riuscirvi. Ad ogni iterazione, troveremo il lato di costo minimo nell'*heap* e lo cancelleremo da lì. Dovremo quindi determinare le componenti a cui i due estremi del lato appartengono nella foresta con bilanciamento. Se stanno nella stessa componente, i due vertici sono già connessi, il lato chiude un ciclo e dobbiamo semplicemente ignorarlo. Se stanno in componenti diverse, dobbiamo aggiungere il lato alla soluzione e unire le due componenti.

L'algoritmo è terminato: si tratta solo ancora di deallocare le strutture dinamiche ausiliarie (*heap* indiretto e foresta con bilanciamento).

```
/* Risolve l'istanza G del MSTP con l'algoritmo di Kruskal
    restituendo la soluzione S */
void Kruskal (grafo *pG, soluzioneMSTP *pS)
{


  /* Costruisce un vettore di posarco che puntino ai lati del grafo G
      */

  /* Struttura il vettore come un min-heap indiretto */

  /* Costruisce la foresta con bilanciamento delle componenti
      connesse della soluzione vuota */

  /* Finche' |x| < n-1 e E' != \emptyset */
    /* Trova il lato di costo minimo in E' */

    /* Se il lato non chiude ciclo (u e v stanno in componenti
        connesse diverse),
       aggiunge il lato alla soluzione e unisce le due componenti */

  /* Dealloca le strutture dinamiche ausiliarie */
}
```

Al solito, prima di proseguire conviene fare un tentativo individuale di svolgere la traccia.

Implementazione dell'algoritmo di Kruskal L'*heap* indiretto è costituito da un vettore `Pos` di `posarco` (puntatori a lato, ma noi potremmo non saperlo, e in effetti l'implementazione della lista potrebbe essere con un vettore e indici interi, e allora si tratterebbe di interi). Il vettore è indicizzato da 1 a m e definendo il tipo `vposarco` si chiarisce meglio che si tratta di un vettore di puntatori, non di un doppio puntatore ad arco o di un puntatore a un vettore di archi. Viene inizializzato scorrendo la lista dei lati e via via assegnando le loro posizioni agli elementi del vettore, per cui servirà sia una posizione che scorra la lista sia un indice intero che scorra in parallelo il vettore. Costruire un *heap* indiretto sul vettore non è banale, per cui chiameremo un'opportuna funzione `creaheapindiretto`, che implementeremo in seguito con la solita modalità *top-down*.

```
  /* Costruisce un heap indiretto (vettore di posizioni dei lati del
      grafo) */
  Pos = (vposarco) calloc(pG->m+1,sizeof(posarco));
  if (Pos == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione dell'heap indiretto sui
        lati!\n");
    exit(EXIT_FAILURE);
  }
```

```
for (i = 1, pa = primoarco(pG); !finearchi(pG, pa); i++, pa =
    succarco(pG, pa))
  Pos[i] = pa;
```

Abbiamo già risolto nel caso dello zaino un problema quasi identico: la funzione riceveva un vettore di indici interi, uno di dati interi e la lunghezza dei due. Qui abbiamo un vettore di posizioni, un grafo (oppure, volendo, la sua lista di archi, ma potremmo voler nascondere l'implementazione del grafo) e il numero dei lati.

```
void creaheapindiretto (vposarco Pos, grafo *pG, int n);
```

Creato l'*heap* indiretto, possiamo costruire la foresta con bilanciamento, che consiste banalmente in un vettore `padre` di indici interi, che all'indice di ogni vertice associ l'indice stesso. Per poter unire componenti in modo bilanciato, sappiamo che dobbiamo appendere il meno numeroso al più numeroso. Questo comporta di conoscere la cardinalità di ogni albero. Il modo più semplice di farlo è costruire un secondo vettore `num` di valori interi, ancora definito sui vertici del grafo, che ad ogni radice associa la cardinalità dell'albero in essa radicato, mentre nei vertici che non sono radice, il contenuto del vettore è insignificante. Servono quindi due vettori di interi. Ovviamente, potremmo definire una struttura dati astratta, costituita da un *record* che racchiuda i vettori e gestisca tutto solo attraverso funzioni. Li inizializziamo assegnando ad ogni vertice il proprio stesso indice e alla cardinalità il valore 1.

```
/* Costruisce una foresta con bilanciamento sui vertici del grafo */
padre = (vint) calloc(pG->n+1,sizeof(int));
if (padre == NULL)
{
  fprintf(stderr,"Errore nell'allocazione del vettore padre!\n");
  exit(EXIT_FAILURE);
}
num = (vint) calloc(pG->n+1,sizeof(int));
if (num == NULL)
{
  fprintf(stderr,"Errore nell'allocazione del vettore num!\n");
  exit(EXIT_FAILURE);
}
for (i = 1; i <= pG->n; i++)
{
  padre[i] = i;
  num[i] = 1;
}
```

Il ciclo che realizza l'algoritmo vero e proprio ha due condizioni di permanenza. La prima è che ci siano ancora lati da considerare, cioè la dimensione `mEst`, inizialmente pari a m, deve essere positiva. La seconda condizione è la cardinalità della soluzione (che è un campo della relativa struttura dati) non sia ancora arrivata a $n-1$.

```
/* Finche' ci sono lati da esplorare in E' e la soluzione e'
    incompleta */
mEst = pG->m;
while ( (mEst > 0) && (pS->nl < pG->n-1) )
{
}
```

Fatto questo, troviamo la posizione del lato di costo minimo, `paMin`, che corrisponde alla posizione contenuta nel primo elemento dell'*heap*. Avendo salvato la posizione, possiamo subito cancellarla dall'*heap*, sovrascrivendola con l'elemento finale e riducendo la dimensione `mEst`. Siccome questo distrugge la proprietà *heap*, bisogna restaurarla chiamando una funzione `aggiornaheapindiretto`, da implementare opportunamente.

```
/* Trova il lato di costo minimo in E' */
paMin = Pos[1];

/* Cancella il lato da E' */
Pos[1] = Pos[mEst];
mEst--;
aggiornaheapindiretto(Pos,mEst,pG,1);
```

Per il momento, prepariamo una chiamata, una dichiarazione e una definizione vuota, cambiando il necessario rispetto al caso dello zaino, cioè ricevendo un vettore di posizioni con la sua dimensione intera, un grafo e l'indice dell'elemento modificato (ho cambiato l'ordine per tenere la dimensione col vettore e non col grafo, dato che in questo caso il legame è meno chiaro).

```
void aggiornaheapindiretto (vposarco Pos, grafo *pG, int n, int i);
```

Nota la posizione e il grafo, accediamo agli estremi dell'arco con la funzione `leggeestremiarco`. Dobbiamo poi determinare la componente in cui sta ciascuno dei due estremi. Trattandosi di una funzione non banale, che implementa la gestione della foresta con bilanciamento, chiameremo un'opportuna funzione da definire meglio in seguito.

```
while ( (mEst > 0) && (pS->nl < pG->n-1) )
{
  /* Trova il lato di costo minimo in E' */
  paMin = Pos[1];

  /* Cancella il lato da E' */
  Pos[1] = Pos[mEst];
  mEst--;
  aggiornaheapindiretto(Pos,mEst,pG,1);

  /* Se i due estremi del lato stanno in componenti distinte della
     foresta con bilanciamento */
  leggeestremiarco(pG,paMin,&u,&v);
  cu = Find(u,padre,num);
  cv = Find(v,padre,num);
```

La nuova funzione riceve il nodo di cui si cerca la componente e i due vettori che realizzano la foresta, e restituisce l'informazione sulla componente (che sarà l'indice del suo vertice radice)

```
/* Trova la componente cui appartiene l'elemento i nella foresta con
   bilanciamento descritta da padre e num */
int Find (int i, vint padre, vint num);
```

Ora, se le due componenti sono diverse, aggiungiamo il lato alla soluzione coerente. Essendo questa affine a una tabella, basta aumentare il numero dei lati (non occorre verificare la dimensione allocata, perché il ciclo impedisce di eccederla), salvare i due nodi estremi nella nuova posizione finale dei due appositi vettori e aggiungere al costo corrente della soluzione il costo del nuovo lato. Questo va recuperato dal grafo attraverso la funzione `leggecosto`.

```
/* Aggiunge il lato alla soluzione corrente */
pS->nl++;
pS->v1[pS->nl] = u;
pS->v2[pS->nl] = v;
pS->f += leggecosto(pG,paMin);
```

Ciò fatto, occorre fondere le due componenti nella foresta con bilanciamento, e questo richiede una funzione dedicata.

```
/* Unisce le due componenti della foresta */
Union(cu,cv,padre,num);
```

La nuova funzione riceve gli indici delle due componenti e i vettori che rappresentano la foresta.

```
/* Unisce le componenti radicate in c1 e c2 nella foresta con
    bilanciamento descritta da padre e num
   (non controlla che siano diverse!) */
void Union (int c1, int c2, vint padre, vint num);
```

Al termine deallocheremo i tre vettori che descrivono *heap* e foresta con bilanciamento. Riassumendo:

```
/* Risolve l'istanza G del MSTP con l'algoritmo di Kruskal
    restituendo la soluzione S */
void Kruskal (grafo *pG, soluzioneMSTP *pS)
{
  vposarco Pos;
  posarco pa, paMin;
  vint padre, num;
  int i;
  int mEst;
  nodo u, v;
  int cu, cv;


  /* Costruisce un heap indiretto (vettore di posizioni dei lati del
      grafo) */
  Pos = (vposarco) calloc(pG->m+1,sizeof(posarco));
  if (Pos == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione dell'heap indiretto sui
        lati!\n");
    exit(EXIT_FAILURE);
  }
  for (i = 1, pa = primoarco(pG); !finearchi(pG, pa); i++, pa =
      succarco(pG, pa))
    Pos[i] = pa;

  creaheapindiretto(Pos,mEst,pG);
```

```
  /* Costruisce una foresta con bilanciamento sui vertici del grafo */
  padre = (vint) calloc(pG->n+1,sizeof(int));
  if (padre == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione del vettore padre!\n");
    exit(EXIT_FAILURE);
  }
  num = (vint) calloc(pG->n+1,sizeof(int));
  if (num == NULL)
  {
    fprintf(stderr,"Errore nell'allocazione del vettore num!\n");
    exit(EXIT_FAILURE);
  }
  for (i = 1; i <= pG->n; i++)
  {
    padre[i] = i;
    num[i] = 1;
  }


  /* Finche' ci sono lati da esplorare in E' e la soluzione e'
      incompleta */
  mEst = pG->m;
  while ( (mEst > 0) && (pS->nl < pG->n-1) )
  {
    /* Trova il lato di costo minimo in E' */
    paMin = Pos[1];

    /* Cancella il lato da E' */
    Pos[1] = Pos[mEst];
    mEst--;
    aggiornaheapindiretto(Pos,mEst,pG,1);

    /* Se i due estremi del lato stanno in componenti distinte della
       foresta con bilanciamento */
    leggeestremiarco(pG,paMin,&u,&v);
    cu = Find(u,padre,num);
    cv = Find(v,padre,num);
    if (cu != cv)
    {
      /* Aggiunge il lato alla soluzione corrente */
      pS->nl++;
      pS->v1[pS->nl] = u;
      pS->v2[pS->nl] = v;
      pS->f += leggecosto(pG,paMin);

      /* Unisce le due componenti della foresta */
      Union(cu,cv,padre,num);
    }
  }

  /* Dealloca l'heap indiretto e la foresta con bilanciamento */
  free(Pos);
  free(padre);
  free(num);
}
```

Possiamo ricompilare il tutto per valutare la correttezza sintattica, ricordando che il progetto comprende non solo il programma principale `kruskal.c`, ma anche la libreria `grafo-la.c` per descrivere il grafo, e di conseguenza la libreria `listaarchi.c` per gestire la lista degli archi. Se avessimo applicato coerentemente l'approccio delle strutture dati astratte, probabilmente dovremmo includere diverse altre librerie, per le strutture ausiliarie di cui abbiamo fatto uso. Ovviamente, il

codice non fa ancora nulla, salvo stampare la soluzione vuota creata all'inizio e poi non modificata, via via che i lati del grafo vengono cancellati dal ciclo.

Bisogna dedicarsi ancora all'implementazione delle procedure di livello più basso.

Implementazione dell'*heap* indiretto dei lati Per la gestione dell'*heap* indiretto, ci ispiriamo pesantemente a quella analoga realizzata per il problema dello zaino. La creazione si limita a chiamare l'aggiornamento retrocedendo dalla metà del vettore al primo elemento. L'unica modifica riguarda il tipo e i nomi degli argomenti passati dall'una all'altra funzione.

```
/* Crea un heap indiretto di lunghezza n attraverso il vettore di
    posizioni Pos sui lati del grafo G */
void creaheapindiretto (vposarco Pos, int n, grafo *pG)
{
  int i;

  for (i = n/2; i >= 1; i--)
    aggiornaheapindiretto(Pos,n,pG,i);
}
```

L'aggiornamento è decisamente più complicato, anche se segue la stessa traccia. Determina i figli del nodo corrente con le solite regole. Ipotizzare che l'indice migliore sia quello corrente, ma questa volta si tratta dell'indice minimo `iMin`, anziché massimo. Inoltre, non essendo il valore accessibile direttamente, ma attraverso una funzione (per quanto semplice) è buona norma non recuperarlo chiamando la funzione ogni volta che serve, ma farlo una volta sola e salvarlo in una variabile *cMin*. Per ognuno dei due figli, si valuta se il valore del costo è minore di quello corrente, e nel caso aggiornare l'indice minimo. Anche in questo caso, è meglio evitare il ripetuto ricorso a chiamate di funzione per ottenere sempre lo stesso risultato. Per essere del tutto chiari, sono da evitare scritture dove chiamate a funzione perfettamente identiche si ripetono in modo ossessivo. Per esempio:

```
   if ( (s <= n) && (leggecosto(pG,Pos[s]) < leggecosto(pG,Pos[iMin])
)
{
iMin = s;
cMin = leggecosto(pG,Pos[s]);
}
if ( (d <= n) && (leggecosto(pG,Pos[d]) < leggecosto(pG,Pos[iMin]) )
{
iMin = d;
cMin = leggecosto(pG,Pos[d]);
}
```

Trovato il minimo dei tre elementi dell'*heap*, procediamo come al solito, vale a dire che, se il minimo è il padre terminiamo la procedura, mentre se è uno dei due figli, ne scambiamo il puntatore con quello che indica il padre e chiamiamo ricorsivamente la funzione sull'indice del figlio. La funzione `Scambia` è la solita, ma opera su `posarco` passati per indirizzo anziché su interi.

```
/* Scambia i valori di due variabili di tipo posarco a e b */
void Scambia (posarco *pa, posarco *pb)
{
  posarco temp;
```

```
  temp = *pa;
  *pa = *pb;
  *pb = temp;
}
```

Tutto questo dà luogo alla soluzione che segue.

```
/* Aggiorna l'heap indiretto di lunghezza n descritto dal vettore di
    posizioni Pos
   sui lati del grafo G a partire dalla posizione i */
void aggiornaheapindiretto (vposarco Pos, int n, grafo *pG, int i)
{
  int s, d;
  int iMin;
  int cs, cd, cMin;


  s = 2*i;
  d = 2*i+1;


  iMin = i;
  cMin = leggecosto(pG,Pos[i]);
  if (s <= n)
  {
    cs = leggecosto(pG,Pos[s]);
    if (cs < cMin)
    {
      iMin = s;
      cMin = cs;
    }
  }
  if (d <= n)
  {
    cd = leggecosto(pG,Pos[d]);
    if (cd < cMin)
    {
      iMin = d;
      cMin = cd;
    }
  }

  if (iMin != i)
  {
    Scambia(&Pos[iMin],&Pos[i]);
    aggiornaheapindiretto(Pos,n,pG,iMin);
  }
}
```

Compilando possiamo valutare la correttezza sintattica. L'algoritmo ancora non funziona, perché non è attiva la foresta con bilanciamento, ma quanto meno l'*heap* induce lo scorrimento dei lati in ordine di costo crescente, per cui potremmo aggiungere una stampa temporanea, per verificare che questo stia effettivamente avvenendo, e intervenire in caso contrario, invece di aspettare la conclusione di tutti i passaggi, e scoprire eventualmente una quantità di errore intrecciati fra loro.

Implementazione dell foresta con bilanciamento Terminiamo con l'implementazione delle procedure di gestione della foresta con bilanciamento. Cominciamo con la funzione `Union`, che è più semplice. La procedura consiste sostanzialmente

nello scegliere una delle due radici e assegnarle come padre l'altra radice, ma anche nel modificare la cardinalità dei due alberi.

Notiamo che stiamo ipotizzando che i due indici ricevuti dalla funzione chiamante siano effettivamente due radici e che siano diversi tra loro, altrimenti le operazioni seguenti sarebbero devastanti. Come al solito, un controllo è possibile, ma affliggerebbe la complessità dell'operazione, che è critica per gli algoritmi che la utilizzano. Quindi, ci affidiamo all'attenzione dell'utente, che sarà avvertito nella documentazione delle ipotesi che ha la responsabilità personale di garantire.

Si è detto che un teorema garantisce complessità logaritmica alla procedura `Find` se la procedura `Union` appende la componente meno numerosa alla più numerosa (lasciando libertà in caso di parità). Essendo la cardinalità semplicemente conservata nel vettore `num`, basta un test sul suo contenuto nelle due celle a risolvere la questione. Per convenzione arbitraria, quando le due componenti hanno lo stesso numero di elementi, appendiamo il secondo al primo[6]. In ciascuno dei due casi, si modifica il vettore `padre` nella posizione della radice che va a scomparire e si aumenta la cardinalità dell'altra radice del valore prima associato alla radice che scompare. A rigore, non è necessario azzerare la cardinalità nella radice che scompare, dato che nessuno leggerà i valori contenuti nelle posizioni che non sono radici[7].

```
/* Fonde le componenti c1 e c2 nella foresta con bilanciamento
     descritta da padre e num
   (non controlla che siano diverse!) */
void Union(int c1, int c2, vint padre, vint num)
{
  if (num[c1] < num[c2])
  {
    padre[c1] = c2;
    num[c2] += num[c1];
    num[c1] = 0;
  }
  else
  {
    padre[c2] = c1;
    num[c1] += num[c2];
    num[c2] = 0;
  }
}
```

La funzione `Find` nella forma elementare è semplice: si tratta banalmente di risalire di padre in padre, fino a raggiungere un nodo che ha come padre sé stesso.

```
/* Trova la componente cui appartiene l'elemento i nella foresta con
     bilanciamento descritta da padre e num */
int Find (int i, vint padre, vint num)
{
  int j, r, pj;

  j = i;
  while (padre[j] != j)
    j = padre[j];
```

[6] Ragionamenti statistici suggeriscono che nel caso medio sia meglio scegliere casualmente con probabilità uguali, per evitare polarizzazioni, ma questo eccede i limiti del corso.

[7] Può sembrare incoerente l'attenzione portata a singole operazioni in alcune situazioni, mentre in altre si passa sopra a interi blocchi di operazioni, perché "di tempo costante". Oltre alla consueta questione del compromesso fra astrazione ed efficienza, il punto è una singola operazione in una procedura eseguita una volta è trascurabile, in una procedura eseguita milioni di volte è fondamentale: il contesto domina.

```
  return j;
}
```

A rigore, si potrebbe persino usare direttamente l'argomento `i` come cursore, dato che è una copia di quello passato dalla funzione chiamante (le dispense di teoria usano un linguaggio che passa i parametri per indirizzo, e quindi usano questa forma).

In realtà, è possibile appesantire leggermente questa operazione in modo che modifichi la foresta con bilanciamento, rendendo in prospettiva più efficienti le chiamate successive alla funzione di ricerca. L'idea è di ridurre l'altezza dell'albero che viene visitato, usando un meccanismo detto di *compressione dei cammini*. Il meccanismo osserva che, nel risalire la catena di nodi per trovare la radice della componente cui appartiene un elemento `i`, determiniamo una serie di elementi che stanno tutti nella stessa componente. Ora, se questi elementi fossero direttamente appesi a tale radice, anziché formare una catena, ogni futura ricerca avrebbe un costo costante, dato che basterebbe salire di un livello per trovare la radice (vedi Figura 14.22).

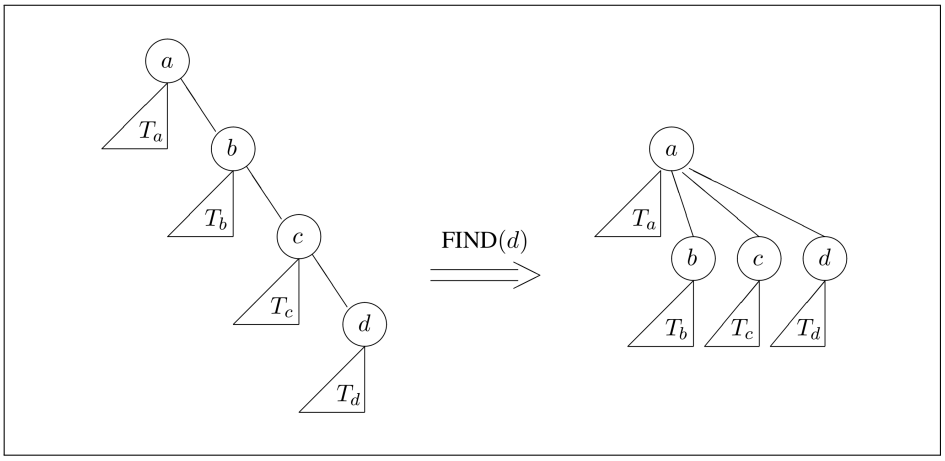


Figura 14.22: Il meccanismo di compressione dei cammini

Di conseguenza, compiuta la risalita che determina la radice, si esegue una seconda risalita nella quale ogni nodo visitato viene direttamente appeso alla radice. Questo richiede qualche precauzione nell'aggiornare il vettore `padre`: prima di cancellare l'informazione necessaria a proseguire la risalita, bisogna salvarla in una variabile ausiliaria `pj`. Inoltre, in questo caso dobbiamo conservare l'elemento di partenza per poter ripetere la risalita, per cui non possiamo utilizzare l'argomento della chiamata come cursore.

```
/* Trova la componente cui appartiene l'elemento i nella foresta con
    bilanciamento descritta da padre e num */
int Find (int i, vint padre, vint num)
{
  int j, r, pj;

  j = i;
  while (padre[j] != j)
    j = padre[j];
  r = j;

  j = i;
```

```
  while (j != r)
  {
    pj = padre[j];
    padre[j] = r;
    j = pj;
  }

  return j;
}
```

È chiaro che in questo modo perdiamo del tempo, perché ripetiamo due volte il cammino, però è anche chiaro che investiamo sul futuro: se in seguito ci toccherà di nuovo di accedere a uno qualsiasi dei nodi visitati (e a uno qualsiasi dei loro discendenti!), impiegheremo molto meno per raggiungere la radice.

A questo punto, finalmente, l'algoritmo determina l'albero ricoprente minimo. Sull'esempio della Figura 14.17, l'algoritmo determina:

```
   CPU = 0.000000
f* = 27
x* = (1,4) (4,5) (3,6) (1,2) (7,8) (6,8) (6,9) (4,7)
```

che è effettivamente l'albero di costo minimo che ricopre il grafo dato.

14.4 Esercizi

14.4.1 Esercizio 1

,

Nell'implementazione dell'algoritmo *greedy* banale per il problema dello zaino si introduca un *max-heap* che contiene gli oggetti ordinati per volume. Questo consente di determinare e cancellare ad ogni passo l'oggetto più voluminoso, nel caso in cui ecceda la capacità residua. Nasce però il problema di cancellare gli elementi da entrambi gli *heap* senza aumentare la complessità dell'algoritmo risultante, cosa possibile con un'opportuna ulteriore struttura ausiliaria. È un approccio promettente?

14.4.2 Esercizio 2

Una volta risolto l'Esercizio 1 del capitolo precedente (cioè definita una libreria per la gestione di dati e soluzioni del problema dello zaino), si aggiunga alla soluzione un campo per il volume totale della soluzione, si modifichino tutte le procedure per tenerne conto e si aggiungano procedure per l'inserimento in soluzione di un elemento nuovo e la cancellazione di un elemento esistente. Si valuti l'opportunità di aggiungere a tali procedura un test per verificare che il nuovo elemento da aggiungere non sia già incluso e che l'elemento da togliere appartenga effettivamente alla soluzione. Si modifichino gli algoritmi realizzati in questo capitolo usando la procedura di inserimento.

14.4.3 Esercizio 3

Si realizzi la versione iterativa di `aggiornaheapindiretto`, adattando quella di `aggiornaheap` realizzata nel corrispondente esercizio del Capitolo **??**.

14.4.4 Esercizio 4

Si realizzi l'algoritmo di Kruskal implementando il grafo come un vettore di lati.

14.4.5 Esercizio 5

Si realizzi una libreria per gestire l'*heap* indiretto dei lati, con le consuete funzioni di creazione e distruzione e le specifiche funzioni necessarie al funzionamento della struttura, e si modifichi il codice dell'algoritmo di Kruskal in modo che la sfrutti.

14.4.6 Esercizio 6

Si sostituisca l'implementazione ricorsiva di *aggiornaheapindiretto* con una iterativa, sfruttando il fatto che si tratta di ricorsione terminale.

14.4.7 Esercizio 7

Si realizzi una libreria per gestire la foresta con bilanciamento, con le consuete funzioni di creazione e distruzione e le specifiche funzioni necessarie al funzionamento della struttura, e si modifichi il codice dell'algoritmo di Kruskal in modo che la sfrutti.

14.4.8 Esercizio 8

Si realizzi una libreria per gestire dati e soluzioni di problemi di albero ricoprente minimo, con le consuete funzioni di creazione e distruzione, e funzioni per caricare i dati, aggiungere elementi alle soluzioni, stampare dati e soluzioni, svuotare soluzioni senza deallocarle.

14.4.9 Esercizio 9

Si realizzi una libreria per gestire dati e soluzioni di problemi di albero ricoprente minimo, con le consuete funzioni di creazione e distruzione, e funzioni per caricare i dati, aggiungere elementi alle soluzioni, stampare dati e soluzioni, svuotare soluzioni senza deallocarle.

14.4.10 Esercizio 10

Dato un grafo non orientato, si realizzi un algoritmo che ne calcola le componenti connesse usando non una visita, ma uno scorrimento dei lati e una foresta con bilanciamento inizialmente formata da vertici isolati, progressivamente uniti in componenti via via che si considerano i lati del grafo. Si valuti anche la complessità temporale e spaziale di questo algoritmo con quella degli algoritmi basati su visite, tenendo conto delle strutture ausiliarie, ma anche dei dati.

14.4.11 Esercizio 10

Si implementi l'algoritmo di Prim per il problema dell'albero ricoprente minimo.

14.4.12 Esercizio 11

Si implementi l'algoritmo di Dijkstra per il problema del cammino minimo (in entrambe le varianti presentate nelle lezioni di teoria).