Lezione 15

La lezione è centrata su un esercizio per impratichirsi sui seguenti algoritmi per il problema dello zaino:

- algoritmo ricorsivo banale;
- algoritmo ricorsivo di programmazione dinamica *top-down*;
- algoritmo iterativo di programmazione dinamica *bottom-up*.

Chi notasse eventuali incoerenze o errori, oppure avesse dubbi sul contenuto di queste pagine e dei codici, è pregato di segnalarmeli per contribuire a migliorare la qualità dei materiali del corso.

Problema

Si vuole scrivere un programma `knapsack.c` che aiuti un ladro entrato in una gioielleria ad uscirne col massimo bottino possibile. La gioielleria contiene un insieme $B = \{1, \ldots, n\}$ di n oggetti preziosi, per ognuno dei quali è noto il valore ϕ_i e il volume v_i. Il ladro ha uno zaino di capacità V, e per semplicità si suppone che il volume totale dei preziosi rubati sia sufficiente a indicare se possono essere contenuti nello zaino oppure no (trascurando il problema della loro forma, che evidentemente potrebbe creare zone inutilizzate nello zaino e ridurne la capacità). Alternativamente si può pensare a un escursionista che debba decidere quali oggetti portare con sé in una gita, avendo fissato un peso massimo per il proprio bagaglio (ignorando questa volta forme e volumi degli oggetti). Il programma deve indicare quali oggetti prendere per rispettare il vincolo massimizzando il valore totale del sottoinsieme di oggetti scelti.

Formati di ingresso e uscita Il file di ingresso che contiene l'istanza del problema consiste di quattro righe. La prima fornisce il numero di oggetti n, preceduto dalle parole chiave `n =`. La seconda fornisce i valori ϕ_i degli oggetti, indicati da numeri interi separati da spazi, preceduti dalle parole chiave `phi =`. La terza riga riporta i volumi v_i degli oggetti, indicati da numeri interi separati da spazi, preceduti dalle parole chiave `v =`. Infine, l'ultima riga contiene la capacità V dello zaino, preceduta dalla parola chiave `V =` .

Sono disponibili quattro esempi, rispettivamente con $n = 5$, 10, 24 e 50 oggetti. Il primo corrisponde all'esempio dei lucidi, ed è:

```
n = 6
phi = 7 2 4 5 4 1
v = 5 3 2 3 1 1
V = 8
```

Il programma deve stampare a video il valore dell'ottimo del problema, preceduto dalle parole chiave `f* =`. Dovrebbe anche (ma per semplicità lo faremo solo nell'ultima fase della risoluzione) stampare in una seconda riga la soluzione ottima, rappresentata come elenco degli indici degli oggetti che ne fanno parte, separati da spazi, racchiusi fra parentesi tonde e preceduti dalle parole chiave `x* =`. Il risultato nel caso dell'esempio sopra riportato è:

```
f* = 15
x* = ( 1 3 5 )
```

I quattro file di esempio forniti riportano le due righe con la soluzione dopo le quattro con i dati, sfruttando il fatto che questo non disturba la lettura dei dati stessi.

Traccia della risoluzione

Si può cominciare scorrendo i lucidi che introducono l'Ottimizzazione Combinatoria, l'algoritmo esaustivo, la distinzione tra algoritmi esatti ed euristici e il problema dello zaino, che è ovviamente un buon modello per il problema da risolvere.

L'esercizio parte dal file `knapsack0.c`. Questo file contiene molte cose, per potersi dedicare immediatamente all'aspetto algoritmico del problema. Ovviamente, contiene la consueta funzione `InterpretaLineaComando` per recuperare il nome del file dalla linea di comando e copiarlo nella corrispondente variabile. Contiene direttive di inclusione, costanti simboliche e dichiarazioni di tipo già usate più volte nelle lezioni precedenti. Contiene però anche le strutture e le funzioni necessarie a gestire istanze e soluzioni per un problema di zaino:

- `datiKP` descrive un'istanza attraverso due numeri interi (il numero degli oggetti `n` e la capacità `V`) e due vettori dinamici di numeri interi (il valore `phi` e il volume `v`); sono già disponibili funzioni per caricare un'istanza da file di testo (nel formato sopra descritto) e per distruggerla;
- `soluzioneKP` descrive una soluzione attraverso un numero intero (il valore totale della soluzione `f`) e un vettore di incidenza, cioè un vettore dinamico di valori logici `x`; sono anche disponibili funzioni per creare, distruggere e stampare (nel formato sopra descritto) una soluzione.

Il codice contiene anche variabili e istruzioni per misurare il tempo di esecuzione degli algoritmi che andremo realizzando, dato che faremo alcune osservazioni interessanti su questo tempo. In particolare, la funzione `clock()` assegna ad una variabile di tipo `clock_t` il numero di unità elementari di tempo (tipicamente, millisecondi o microsecondi) trascorse dall'inizio dell'esecuzione. La differenza dei valori registrati al principio e al termine dell'esecuzione fornisce la durata. Per esprimerla in secondi, occorre dividere il numero di unità elementari trascorse per quello delle unità contenute in un secondo, che è fornito dalla costante simbolica `CLOCKS_PER_SEC`. Di conseguenza, si include anche la libreria `time.h`.

Prima fase (`knapsack1.c`) I lucidi propongono un algoritmo ricorsivo che risolve il problema dello zaino in maniera abbastanza simile a una versione ricorsiva della *InsertionSort*[1], vale a dire

1. riducendo il problema con l'eliminazione di un elemento;
2. risolvendo il problema ridotto;
3. riaggiungendo l'elemento eliminato alla soluzione del problema originale.

L'analogia non funziona completamente, perché in generale non è detto che l'elemento eliminato sia reinseribile nella soluzione del problema originale (lo zaino potrebbe essere ormai troppo pieno), e d'altra parte non è neanche detto che l'elemento eliminato si possa ignorare (altrimenti li butteremmo tutti e otterremmo una soluzione vuota, che è la peggiore in assoluto). Dobbiamo quindi complicare la

[1] Che non è in programma, ma viene descritta nei materiali di approfondimento della lezione 14.

ricorsione ammettendo due possibili casi ricorsivi: quello in cui l'ultimo elemento non trova posto nella soluzione ottimale del sottoproblema e quello in cui vi trova posto. Il caso in cui l'ultimo elemento trova posto nello zaino richiede che la capacità sia manipolata (ridotta) in modo da lasciare il posto necessario. A volte, il primo caso è banalmente l'unico possibile, perché lo zaino è di per sé troppo piccolo per contenere l'ultimo elemento, e quindi non è necessario calcolare la soluzione ottima del sottoproblema per sapere che l'elemento va buttato. Riassumendo, abbiamo:

- un caso base, nel quale l'insieme degli oggetti potenziali è vuoto;
- due casi ricorsivi, nei quali l'insieme degli oggetti potenziali non è vuoto e:
  1. lo zaino è troppo piccolo per contenere l'ultimo oggetto dell'insieme, così che questo va scartato, producendo un insieme di oggetti potenziali più piccolo;
  2. lo zaino è abbastanza grande da contenere l'ultimo oggetto dell'insieme, così che questo si può scartare, producendo un insieme di oggetti potenziali più piccolo da inserire nello stesso zaino, oppure scegliere, producendo un insieme di oggetti potenziali più piccolo da inserire in uno zaino parzialmente occupato (che equivale a uno zaino più piccolo).

Nel primo caso ricorsivo, c'è una sola chiamata ricorsiva (come nell'*InsertionSort*); nel secondo, invece, ce ne sono due, e la soluzione ottima dell'intero problema è la migliore fra le soluzioni ottime dei due sottoproblemi.

Fortunatamente, i due sottoproblemi sono entrambi rappresentabili come problemi di zaino:

1. il sottoproblema in cui si rifiuta l'ultimo elemento, è semplicemente uno zaino sui primi $n-1$ elementi con la stessa capacità V: l'insieme B viene ristretto al sottoinsieme $\{1,\ldots,i\}$ ponendo $i = n-1$, e le funzioni ϕ e v vengono banalmente ristrette al nuovo sottoinsieme;
2. il sottoproblema in cui si accetta l'ultimo elemento, è ancora uno zaino sui primi $n-1$ elementi, con la stessa restrizione di B, ϕ e v, ma la capacità dello zaino diventa $V - v[n]$, dato che l'oggetto fissato va ad occupare spazio nello zaino originale (cioè il nuovo zaino è la parte rimasta libera dello zaino originale).

Quando $v_n > V$, il secondo sottoproblema è impossibile, e quindi c'è solo il primo.

Nella prima fase dell'esercizio, realizziamo l'algoritmo ricorsivo descritto più sopra, che è molto semplice ed elegante. Va osservato che, siccome l'algoritmo nelle chiamate ricorsive opera su sottoproblemi del problema originale, bisogna trovare il modo per passare tali sottoproblemi alla procedura stessa. Ci sono diversi modi per farlo:

1. ricostruire i dati di ogni sottoproblema copiando, ed eventualmente adattando, quelli del problema originale (questo è molto inefficiente sia rispetto allo spazio sia rispetto al tempo);
2. modificare i dati del sottoproblema corrente e passarli alla procedura ricorsiva; nel caso specifico, si tratta di decrementare `n` ed eventualmente ridurre `V` (i vettori possono rimanere invariati, dato che il decremento di `n` nasconde automaticamente l'elemento scartato); se i dati sono passati per indirizzo, però, bisogna ricordarsi di ripristinarli subito dopo la chiamata, dato che le chiamate annidate ai livelli inferiori introdurranno ulteriori modifiche (per esempio, decrementando `n` fino a zero);

3. passare alla procedura ricorsiva copie modificate dei dati, in modo da non alterare gli originali; per far questo, basta sfruttare il passaggio dei parametri per copia, cioè non fornire alla procedura l'intera istanza, ma solo i campi `n`, `phi`, `v` e `V`, modificando `n` e `V` così che ogni singola chiamata abbia i propri valori indipendenti[2].

Scegliamo l'ultima strada, che produce il codice più semplice e vicino all'equazione ricorrente, anche se forse la seconda usa qualche operazione in meno. Per semplicità, chiediamo che l'algoritmo restituisca solo il valore della soluzione, e non l'intero risultato[3].

Seconda fase (`knapsack2.c`) L'algoritmo ricorsivo risolve molto in fretta i primi tre problemi di esempio, mentre il quarto non accenna a terminare nemmeno dopo svariati minuti. Il motivo è che si tratta di un algoritmo molto inefficiente: nell'albero delle chiamate ricorsive, infatti, ogni nodo corrisponde a un sottoproblema in cui gli oggetti sono accettati, scartati oppure ancora non presi in considerazione. Le foglie, in particolare, corrispondono ad accettare o scartare tutti gli oggetti, e quindi a sottoinsiemi completamente determinati. Nel caso pessimo (quando il test sulla capacità non interviene mai), le foglie sono 2^n e i nodi complessivi circa il doppio. I primi tre problemi di esempio hanno, rispettivamente, $n = 6$, $n = 10$ e $n = 24$ elementi, e quindi potenzialmente $2^6 = 64$, $2^{10} = 1024$ e $2^{24} = 16\,777\,216$ foglie, che si possono tranquillamente scorrere in frazioni di secondo. Il quarto esempio, invece, ha $n = 50$ oggetti, e quindi $2^{50} \approx 1.126 \cdot 10^{15}$ foglie: anche ipotizzando di esplorare un miliardo di foglie al secondo, occorrerebbero circa 13 giorni per visitarle tutte.

Ora osserviamo che i sottoproblemi sono in effetti individuati semplicemente dai due numeri naturali `n` e `V`, per cui ci sono al massimo $(n+1)(V+1)$ sottoproblemi, e non 2^n. In pratica, molti dei sottoproblemi generati e risolti dall'algoritmo ricorsivo si ripetono, e quindi vengono risolti ripetutamente sprecando tempo. A seconda dei valori di `n` e `V`, può prevalere una o l'altra stima, ma la differenza può essere enorme. La soluzione è, ogni volta che si risolve un sottoproblema, salvarne il risultato in un'apposita struttura. Questa struttura consenta, di fronte a un altro sottoproblema, di verificare se sia nuovo oppure già affrontato, e nel secondo caso di recuperare la soluzione senza doverla ricalcolare. Dove conservare le soluzioni dei sottoproblemi? Essendo questi indicizzati su due numeri naturali, viene spontaneo conservarli in una matrice bidimensionale, con indici `i` e `v`, rispettivamente estesi da `0` a `n` e da `0` a `V`. Ogni sottoproblema, con la relativa soluzione, sarà associato ad una cella. Dobbiamo quindi definire una matrice, allocarla e inizializzarla con un valore che indichi il fatto che il sottoproblema non è ancora stato risolto (un valore intero negativo può andare bene, dato che ogni valore non negativo è potenzialmente una soluzione sensata). Quindi incapsuleremo la procedura ricorsiva realizzata nella prima fase in una procedura `PDricorsivaKP` che alloca e inizializza la matrice, chiama la procedura ricorsiva e infine dealloca la matrice. Inoltre, modificheremo la procedura ricorsiva affinché sfrutti la matrice e la aggiorni, cioè:

- aggiungeremo un altro caso base, corrispondente alle situazioni in cui il sottoproblema corrente risulta già risolto, cioè la matrice contiene un valore non negativo: in tale caso, restituiremo il valore contenuto nella matrice (in tempo costante);

[2] I due vettori non subiscono modifiche, e quindi si possono passare impunemente senza farne copia.

[3] Ottenere il risultato comporta di passare alla procedura ricorsiva le strutture per contenerlo, e aggiornarle nel modo corretto. È un buon esercizio, ma inutilmente complicato per un algoritmo che poi abbandoneremo in quanto inefficiente.

- modificheremo i casi ricorsivi in modo che, una volta calcolato il risultato, prima di restituirlo lo salvino anche nella cella appropriata della matrice.

Applicando questo algoritmo agli esempi, osserviamo che l'istanza con $n = 50$ oggetti viene ora risolta molto in fretta. Il motivo è semplice: la nuova procedura risolve al massimo `(n+1)(V+1)` sottoproblemi, cioè nel caso specifico $51 \cot 421 = 21\,471$, che è nettamente meno di $2^n \approx 10^{15}$. D'altra parte, il terzo esempio risulta invece rallentato. Infatti, in esso $n = 24$ e $V = 6\,404\,180$, per cui $2^n = 2^{24} = 16$ milioni, mentre $(n + 1)(V + 1) \approx 160$ milioni.

Terza fase (`knapsack3.c`) Il modo più comune di realizzare l'algoritmo di programmazione dinamica per il problema dello zaino non è tuttavia quello appena descritto (che viene definito *top-down*), ma un algoritmo iterativo, definito di programmazione dinamica *bottom-up*. L'idea di fondo è che generare i sottoproblemi in modo ricorsivo sia una spesa inutile, dato che i sottoproblemi, in fin dei conti, consistono semplicemente in coppie di numeri non negativi (i, v), e mantenere un'architettura di allocazioni e deallocazioni sullo *stack* di sistema solo per scorrere queste coppie appare sproporzionato. L'idea è di procedere invece sistematicamente, risolvendo tutti i sottoproblemi possibili per $i \leq n$ e $v \leq V$. Ogni sottoproblema (i, v) si può risolvere in tempo costante a patto di disporre delle soluzioni ottime dei sottoproblemi con $i - 1$ oggetti: basta applicare l'equazione ricorrente. Dunque, basta un tempo costante per estendere la soluzione dai problemi più piccoli a quelli immediatamente più grandi. L'idea è "risolvere" tutti i problemi di dimensione $i = 0$, salvare i risultati (tutti nulli) nella matrice che abbiamo introdotto più sopra, e procedere a riempirla salendo progressivamente da $i = 0$ a $i = n$, sfruttando in ogni riga i il contenuto delle celle della riga precedente $i - 1$. L'estensione avviene con un banale costrutto di selezione che ripete (*mutatis mutandis*) i casi dell'equazione ricorrente e dell'algoritmo ricorsivo. Rispetto alla programmazione dinamica *top-down*, questo algoritmo risolve un numero maggiore di sottoproblemi (li risolve tutti, anziché solo quelli davvero ottenibili sviluppando l'albero delle chiamate ricorsive), ma evita tutta la fase di costruzione dei sottoproblemi. Potrebbe quindi essere più o meno efficiente, secondo che il numero dei sottoproblemi effettivi sia più o meno vicino a $(n + 1)(V + 1)$[4]

In questa variante, è abbastanza semplice ricostruire a posteriori la soluzione ottima, oltre a determinarne il valore. Anzi tutto, l'ottimo del problema corrisponde al contenuto della cella (n, V) della matrice, dato che il problema complessivo ha n oggetti e capacità V. Partendo da quella cella, è possibile ricostruire a ritroso il percorso seguito per raggiungerla fino ad arrivare alla cella origine $(0, 0)$. Infatti, ogni cella (i, v) può essere raggiunta solo da due possibili celle, entrambe al livello immediatamente precedente: $(i - 1, v)$ e $(i - 1, v - v_i)$. Se si è arrivati dalla prima, l'elemento i non è stato aggiunto alla soluzione parziale, e quindi il valore della soluzione deve essere lo stesso; se si è arrivati dalla seconda, invece, l'elemento i è stato aggiunto, e quindi il valore della soluzione deve essere cresciuto. L'idea è confrontare i valori conservati nelle due potenziali celle di partenza con quello della cella di arrivo per dedurre da dove si sia effettivamente partiti. Questo determina il valore della variabile x_i. Può capitare che le due potenziali celle di partenza siano entrambe compatibili con quella di arrivo. Questo succede quando il problema ammette diverse soluzioni ottime equivalenti. In tal caso, sceglieremo la soluzione con $x_i =$ `false`, ma questo è del tutto arbitrario (corrisponde a preferire le soluzioni che non usano gli elementi di indice maggiore).

[4] In realtà è anche possibile costruire solo i sottoproblemi utili, ma questo comporta una struttura dati leggermente più complicata, che consenta di saltare le celle inutili della matrice.