Progettazione di algoritmi

Discussione dell'esercizio [resto]

Disponendo di un numero illimitato di banconote o monete di un paese straniero di n tagli diversi $1 = v_1 < v_2 < \cdots < v_n$, bisogna determinare il minimo numero di banconote per dare un resto $R \geq 1$. Ci si potrebbe chiedere perchè mai questo problema non può essere risolto con il naturale algoritmo greedy che ad ogni passo sottrae dal resto la banconota più grande finché il resto non è esaurito. D'altronde con i tagli a cui siamo abituati, 1, 2, 5, 10, ecc., l'algoritmo greedy funziona (e si può dimostrare che è proprio così, cioè trova sempre il numero minimo di banconote). Però se, ad esempio, i tagli sono 1, 4, 5 e il resto è 8, l'algoritmo greedy sceglie le banconote 5,1,1,1 mentre bastano solamente 2 banconote da 4. Quindi in generale il problema non è così facile.

Come abbiamo fatto per il problema *file*, iniziamo con lo scrivere un algoritmo di ricerca esaustiva della soluzione ottima. Concentriamoci per il momento solamente sul calcolo del numero minimo di banconote. L'algoritmo esaustivo può essere convenientemente scritto in modo ricorsivo in cui ad ogni chiamata consideriamo le due possibilità di usare o non usare una banconota di un certo taglio:

```
RESTO_REC(V: array dei tagli delle banconote, k: numero tagli, r: resto)
    IF r = 0 THEN RETURN 0
    ELSE IF k = 1 THEN RETURN r   /* C'è solamente il taglio da 1 */
    ELSE
        min <- RESTO_REC(V, k - 1, r)   /* Non usiamo il k-esimo taglio */
        IF r >= V[k] THEN               /* Se possiamo usare il k-esimo taglio */
            m <- 1 + REST_REC(V, k, r - V[k])   /* Usiamo il k-esimo taglio */
            IF m < min THEN min <- m
        RETURN min
```

La prima chiamata è `RESTO_REC(V, n, R)`. Si osservi che `RESTO_REC(V, k, r)` ritorna il numero minimo di banconote per formare il resto `r` con tagli presi solamente tra i primi `k` (cioè `V[1]`, `V[2]`,..., `V[k]`). Possiamo allora sbarazzarci della ricorsione introducendo la seguente tabella M, per ogni $k = 1,\ldots,n$ e per ogni $r = 0,\ldots,R$

$$M[k,r] = \text{minimo numero di banconote, con tagli presi tra i primi } k\text{, per formare il resto } r$$

I casi base sono tradotti ponendo:

$$M[1,r] = r \qquad \text{e} \qquad M[k,0] = 0$$

E la regola per il calcolo della tabella è

$$M[k,r] = \begin{cases} M[k-1,r] & \text{se } V[k] > r \\ \min\{M[k-1,r], 1 + M[k,r-V[k]]\} & \text{altrimenti} \end{cases}$$

Adesso possiamo scrivere l'algoritmo di programmazione dinamica per il problema resto:

```
RESTO_PD(V: array dei tagli delle banconote, n: numero tagli, R: resto)
    M: tabella nx(R + 1)
    FOR r <- 0 TO R DO M[1, r] <- r
    FOR k <- 1 TO n DO M[k, 0] <- 0
    FOR k <- 2 TO n DO
        FOR r <- 1 TO R DO
            M[k, r] <- M[k - 1, r]
            IF r ≥ V[k] AND 1 + M[k, r - V[k]] < M[k, r] THEN
                M[k, r] <- 1 + M[k, r - V[k]]
    RETURN M[n, R]
```

Chiaramente la complessità dell'algoritmo è proporzionale alla dimensione della tabella, cioè O(nR). Una volta calcolata la tabella `M` dei valori delle soluzioni ottime di tutti i sotto-problemi, possiamo ottenere la soluzione ottima (cioè la lista delle

banconote) partendo dall'elemento `M[n, R]` e percorrendo la tabella seguendo a ritroso le decisioni che hanno portato alla soluzione ottima:

```
RESTO_SOL(V: array dei tagli, n: numero tagli, R: resto, M: tabella)
    SOL <- lista vuota
    k <- n
    r <- R
    WHILE r > 0 DO
        IF k = 1 OR M[k, r] < M[k - 1, r] THEN   /* Una banconota V[k] è usata */
            SOL.append(V[k])
            r <- r - V[k]
        ELSE                                     /* Le banconote V[k] non sono più usate */
            k <- k - 1
    RETURN SOL
```

È facile vedere che la complessità è O(*n* + *R*).

Il problema dello Zaino

Un problema di base che si presenta in situazioni in cui si ha un budget limitato e una collezione di possibili scelte ognuna con un costo e un valore può essere formulato come segue:

> Dati n oggetti 1,2,...,n, ognuno con un costo o peso p_i e un valore v_i, e un limite C, trovare un sottoinsieme degli oggetti la cui somma dei pesi non supera C e che massimizza il valore totale.

Per semplicità assumiamo che i pesi, i valori e *C* siano tutti interi positivi. In termini matematici il problema può essere formulato così:

$$\max \sum_{i=1}^{n} x_i v_i$$

$$\text{soggetto al vincolo: } \sum_{i=1}^{n} x_i p_i \leqslant C$$

$$x_i \in \{0,1\} \quad \text{per ogni } i = 1, \ldots, n$$

Questo problema è conosciuto con il nome di *Problema dello Zaino* (in inglese *Knapsack Problem*). Il nome deriva dall'interpretazione in cui si hanno *n* oggetti che si vorrebbero mettere in uno zaino ma lo zaino ha una capacità limitata *C* (in peso o in volume) perciò solamente un sottoinsieme degli oggetti potrà essere messo nello zaino, allora si vuole massimizzare il valore (veniale o affettivo) degli oggetti che riusciremo a portare con noi nello zaino. Il problema dello *Zaino* ha moltissime applicazioni in una grande varietà di campi: trasporti, logistica, taglio&imballaggio, telecomunicazioni, investimenti, allocazioni di risorse, ecc. Ad esempio, il problema *file* è un caso particolare del problema dello *Zaino* in cui gli oggetti sono i file e per ogni file *i*, $p_i = v_i = s_i$ e la capacità *C* è quella del disco. Quindi, siccome comprende come caso particolare un problema che sappiamo essere difficile (cioè, non si conoscono algoritmi efficienti che lo risolvono), il problema dello *Zaino* è anch'esso difficile.

Per risolvere il problema *file* abbiamo usato una tabella *T* con il seguente significato:

$$T[k, c] = \text{massimo spazio usabile dai primi } k \text{ file su un disco di capacità } c$$

Ovvero abbiamo considerato i sotto-problemi relativi ai primi *k* file e a una capacità $c \leq C$. Possiamo fare la stessa cosa per il problema dello *Zaino*:

$$Z[k, c] = \text{massimo valore ottenibile dai primi } k \text{ oggetti per uno zaino di capacità } c$$

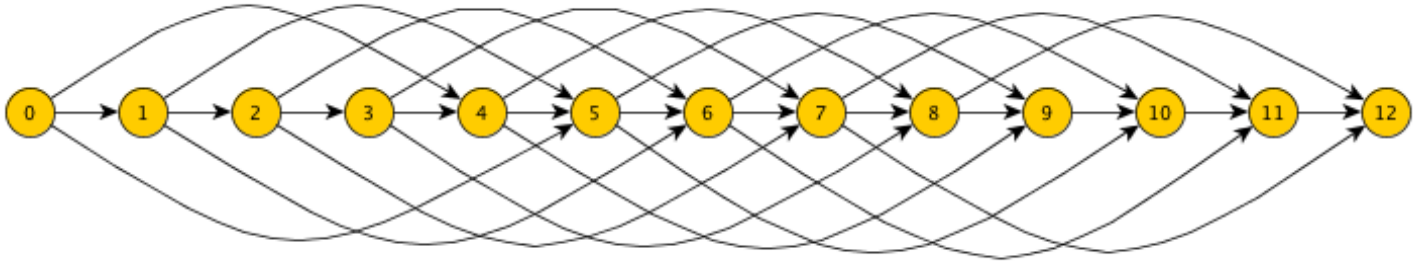
Per ogni *c*, *Z*[0, *c*] = 0. Inoltre, per ogni $k \geq 1$ e per ogni $c \geq 0$,

$$Z[k,c] = \begin{cases} Z[k-1,c] & \text{se } p_k > c \\ \min\{Z[k-1,c], v_k + Z[k-1,c-p_k]\} & \text{altrimenti} \end{cases}$$

Quindi è molto simile alla regola per il calcolo della tabella del problema *file*. Conseguentemente il programma di calcolo della tabella *Z* è del tutto simile a quello per la tabella *T* e anche il programma per trovare la soluzione ottima a partire dalla tabella dei valori ottimi è quasi identico a quello del problema *file* e non verranno riscritti. Le complessità sono ovviamente le stesse: O(*nC*) per calcolare la tabella *Z* e O(*n*) per ricavare una soluzione ottima dalla tabella *Z*.

Riduzione a problemi su grafi

Sia il problema del *resto* che il problema dello *Zaino* possono essere ridotti a problemi su grafi. Iniziamo dal problema del *resto*. Data una istanza $I = \langle v_1, v_2, \ldots, v_n; R \rangle$, consideriamo un grafo diretto G_I tale che i nodi corrispondono ai valori 0,1, …,*R* e per ogni coppia di nodi *r*, *r*' c'è un arco da *r* a *r*' se solo se esiste un taglio di banconota v_i tale che $r + v_i = r'$. Ecco un esempio relativo all'istanza <1, 4, 5; 12>.



Chiaramente G_I è un grafo diretto aciclico (in breve DAG che sta per Directed Acyclic Graph). Risolvere l'istanza *I* equivale a trovare un cammino minimo dal nodo 0 al nodo *R* nel grafo G_I. Infatti, ogni cammino dal nodo 0 al nodo *R* corrisponde a un modo di dare il resto *R* in cui ogni arco corrisponde a una banconota. Siccome G_I è un grafo senza pesi, è sufficiente una BFS dal nodo 0. Il grafo G_I ha *R* + 1 nodi ma quanti archi ha? Ogni nodo *r* con $R - r \geq v_n$ ha esattamente *n* archi (uscenti). Quindi se *R* è abbastanza più grande di v_n, la maggior parte dei nodi ha *n* archi. Perciò ci saranno $\Theta(Rn)$ archi e la BFS impiegherà O(*nR*), cioè lo stesso tempo (asintoticamente) dell'algoritmo di programmazione dinamica. Però non è necessario costruire il grafo prima di fare la BFS. Se il grafo è usato in modo implicito la memoria richiesta per la BFS (la coda dei nodi e l'albero di visita) è dell'ordine di O(*R*). Inoltre dall'albero di visita è possibile ricavare anche la soluzione ottima (non solo il valore ottimo). Quindi tramite questa riformulazione del problema *resto* e il relativo algoritmo, anche se non gudagnamo in tempo di calcolo risparmiamo memoria. Infatti, possiamo ottenere la soluzione ottima usando memoria O(*R*) mentre l'algoritmo basato sulla tabella *M* richiede memoria O(*nR*).

Possiamo fare una cosa simile anche per il problema dello *Zaino*. Data un'istanza $I = \langle p_1, \ldots, p_n; v_1, \ldots, v_n; C \rangle$, definiamo un grafo diretto e pesato G_I che ha *nC* + 1 nodi che corrispondono alle coppie <*k*,*c*> con *k* = 1,…,*n*, *c* = 0,1,…,*C* e un nodo speciale *z*. Per ogni coppia di nodi *u* = <*k*,*c*>, *v* = <*k*-1,*c*-p_k> c'è un arco da *u* a *v* di peso v_k (rappresenta la scelta dell'oggetto *k*). Inoltre, per ogni nodo <*k*,*c*> c'è un arco di peso 0 verso *z* (rappresenta la chiusura dello zaino) e se $k \geq 2$ c'è un arco di peso 0 al nodo <*k*-1,*c*> (rappresenta lo scarto dell'oggetto *k*). Chiaramente G_I è un DAG (pesato). Ogni nodo <*k*,*c*> rappresenta una situazione in cui dobbiamo ancora scegliere quali tra i primi *k* oggetti mettere nello zaino che ha capacità residua *c*. Ogni cammino dal nodo <*n*,*C*> (tutti gli *n* oggetti ancora da scegliere e lo zaino con capacità iniziale *C*) al nodo *z* corrisponde a un modo di riempire lo zaino in cui ogni arco di peso positivo corrisponde alla scelta di un oggetto. Quindi, risolvere l'istanza *I* equivale a trovare un cammino di peso massimo dal nodo <*n*,*C*> al nodo *z*. Il problema di trovare in un grafo pesato (o non pesato) un cammino semplice di massimo peso (o massima lunghezza) è chiamato *Longest Path Problem*. A differenza dello Shortest Path problem, è un problema difficile e quindi non si conoscono algoritmi efficienti. Se però il grafo è un DAG ci sono algoritmi efficienti basati proprio sulla tecnica della programmazione dinamica.

Esercizio [resto 2]

Dati *n* tagli di banconote $1 = v_1 < v_2 < \cdots < v_n$ e un valore *R*, vogliamo calcolare il numero di modi diversi in cui si può dare il resto *R*. Ad esempio, se i tagli sono 1,4,5 e *R* = 8, ci sono 4 modi:

```
1,1,1,1,1,1,1,1   1,1,1,1,4   1,1,1,5   4,4
```

Descrivere un algoritmo che risolve il problema.