ESERCIZI DI
ALGORITMI E STRUTTURE DATI 1
Parte 1

Livio Colussi
Dipartimento di Matematica Pura ed Applicata
Università di Padova

19 maggio 2006

Esercizio 1 Il problema dello zaino frazionario è il seguente:

Dati n tipi di merce $M_1, \ldots, M_n$ in quantità rispettive $q_1, \ldots, q_n$ e valori unitari $c_1, \ldots, c_n$ si vuole riempire uno zaino di capacità Q in modo che il contenuto abbia valore massimo.

Mostrare che il seguente algoritmo risolve il problema:

RIEMPIZAINO(q, c, n, Q)
/* Precondizione: $c_1 \geq c_2 \geq \ldots \geq c_n$ */
$Sp \leftarrow Q\ ,\ i \leftarrow 1$
while $i \leq n$ **do**
if $Sp \geq q_i$ **then**
$z_i \leftarrow q_i\ ,\ Sp \leftarrow Sp - q_i\ ,\ i \leftarrow i + 1$
else
$z_i \leftarrow Sp\ ,\ Sp \leftarrow 0\ ,\ i \leftarrow i + 1$
return z

Soluzione. L'algoritmo pone $z_i = q_i$ per ogni indice i fino ad un certo indice k per il quale pone $z_k = q_k$ e pone quindi $z_i = 0$ per tutti gli indici i successivi. Sia $y_1, y_2, \ldots, y_n$ una qualsiasi altra soluzione. Per ogni $i < k$ abbiamo $y_i \leq z_i = q_i$. La differenza tra il costo C della soluzione calcolata dall'algoritmo e il costo C' dell'altra soluzione è:

$$\begin{aligned} C - C' &= \sum_{i=1}^{k-1}(z_i - y_i)c_i + (z_k - y_k)c_k - \sum_{i=k+1}^{n} y_i c_i \\ &\geq c_k(\sum_{i=1}^{k-1}(z_i - y_i) + (z_k - y_k) - \sum_{i=k+1}^{n} y_i) \\ &= c_k(\sum_{i=1}^{k} z_i - \sum_{i=1}^{n} y_i) = c_k(Q - Q) = 0 \end{aligned}$$

Quindi la soluzione trovata dall'algoritmo è ottima.

Esercizio 2 Il problema dello zaino 0-1 il seguente:

Dati n tipi di oggetti $O_1, \ldots, O_n$ di volumi $v_1, \ldots, v_n$ e valori $c_1, \ldots, c_n$, si vuole riempire uno zaino di capacità V con tali oggetti in modo che il contenuto abbia valore massimo. Si suppone di avere a disposizione un numero illimitato di oggetti di ciascun tipo.

Mostrare che il seguente algoritmo **non** risolve il problema:

RIEMPIZAINO(v, c, n, V)
/* Precondizione: $c_1/v_1 \geq c_2/v_2 \geq \ldots \geq c_n/v_n$ */
$Sp \leftarrow V$, $i \leftarrow 1$
while $i \leq n$ **do**
$z_i \leftarrow \lfloor Sp/v_i \rfloor$, $Sp \leftarrow Sp - z_i * v_i$, $i \leftarrow i + 1$
return z

Soluzione. Basta trovare un controesempio. Supponiamo di avere tre tipi di oggetti di volumi $v_1 = 7$, $v_2 = 6$, $v_3 = 4$ e valori $c_1 = 35$, $c_2 = 24$, $c_3 = 12$ e zaino di capacità $V = 10$.

Siccome $c_1/v_1 = 5 \geq c_2/v_2 = 4 \geq c_3/v_3 = 3$, l'algoritmo prende un oggetto di valore 35. La soluzione ottima è invece prendere un oggetto di valore 24 ed uno di valore 12.

Esercizio 3 Nel problema della scelta delle attività abbiamo visto come si arrivi ad una soluzione globalmente ottima scegliendo ad ogni passo, tra le attività compatibili con quelle già scelte, quella con tempo di fine minimo. Mostrare che questo non è vero se scegliamo quella di durata minima. Mostrare che non è vero neppure scegliendo quella incompatibile con il minimo numero di attività rimanenti.

Soluzione. Consideriamo le tre attività $a_1 = (0, 3)$, $a_2 = (2, 4)$ e $a_3 = (3, 6)$. Se scegliamo quella di durata minima otteniamo una soluzione costituita soltanto dall'attività a_2 mentre una soluzione ottima è costituita dalle altre due attività.

Consideriamo le attività $a_1 = (0, 4)$, $a_2 = (3, 5)$, $a_3 = (1, 5)$, $a_4 = (1, 6)$, $a_5 = (4, 7)$, $a_6 = (6, 10)$, $a_7 = (8, 11)$, $a_8 = (10, 12)$, $a_9 = (10, 13)$, $a_{10} = (10, 14)$ e $a_{11} = (11, 14)$. Se scegliamo quella incompatibile con il minimo numero di attività rimanenti otteniamo una soluzione costituita dalle attività a_2, a_6, a_8 mentre una soluzione ottima è costituita dalle attività a_1, a_5, a_7, a_{11}.

Esercizio 4 Sia $a_1, \ldots, a_n$ un insieme di attività didattiche aventi tempi di inizio $s_1, \ldots, s_n$ e tempi di fine $f_1, \ldots, f_n$ e supponiamo di avere un insieme sufficientemente grande di aule $A_1, A_2, \ldots$ in cui svolgerle. Trovare un algoritmo goloso per programmare tutte le attività nel minimo numero possibile m di aule.

Soluzione. L'algoritmo goloso è il seguente:

PROGRAMMALEZIONI(s, f, n)
/* Precondizione: $s_1 \leq s_2 \leq \ldots \leq s_n$ */
$m \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
$h \leftarrow m + 1$
for $j \leftarrow 1$ **to** m **do**
if $s_i \geq t_j$ **then** $h \leftarrow j$
/* La lezione a_i si può eseguire nell'aula A_h. Se $h = m + 1$ in nessuna delle m aule usate finora si può effettuare a_i. */
$J[i] \leftarrow h$, $t_h \leftarrow f_i$
if $h = m + 1$ **then** $m \leftarrow m + 1$
/* Postcondizione: $J[1..n]$ è una programmazione ottima delle n attività didattiche che usa il minimo numero m di aule. $J[i] = j$ significa che la lezione a_i si terrà nell'aula A_j */
return J, m

Che la soluzione $J[1, n]$ usi il minimo numero possibile m di aule è evidente. Sia a_i la prima attività assegnata all'ultima aula A_m ed s_i il suo tempo di inizio. Al tempo s_i le precedenti $m-1$ aule erano tutte occupate e quindi tra le n attività ce ne sono almeno m che si sovrappongono nello stesso istante. Dunque sono necessarie almeno m aule.

Esercizio 5 Sia $a_1, \ldots, a_n$ un insieme di attività didattiche aventi tempi di inizio $s_1, \ldots, s_n$ e tempi di fine $f_1, \ldots, f_n$ e supponiamo di avere a disposizione m aule $A_1, \ldots, A_m$ in cui svolgerle. Trovare un algoritmo goloso per programmare il massimo numero possibile di attività nelle m aule disponibili.

Soluzione. L'algoritmo goloso è il seguente:

PROGRAMMALEZIONI(s, f, n, m)
/* Precondizione: $f_1 \leq f_2 \leq \ldots \leq f_n$ */
for $j \leftarrow 1$ **to** m **do** $t_j \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
$h \leftarrow 0$, $t_0 \leftarrow -1$
for $j \leftarrow 1$ **to** m **do**
if $s_i \geq t_j$ **and** $t_j > t_h$ **then** $h \leftarrow j$

/* A_h è l'aula che si libera per ultima tra quelle in cui è possibile effettuare la lezione a_i. Se $h = 0$ in nessuna delle m aule si può effettuare a_i. */
$J[i] \leftarrow h$, **if** $h \neq 0$ **then** $t_h \leftarrow f_i$

/* Postcondizione: $J[1..n]$ è una programmazione ottima del massimo numero di attività didattiche che si possono svolgere nelle m aule. $J[i] = j \neq 0$ significa che la lezione a_i si terrà nell'aula A_j. $J[i] = 0$ significa che la lezione a_i non si terrà in nessuna delle m aule. */
return J, k

Supponiamo che l'assegnazione $J[1, i-1]$ delle prime $i-1$ attività didattiche sia estensibile ad una programmazione ottima $B[1, n]$ di tutte le attività.

L'algoritmo, dopo aver assegnato le prime $i-1$ attività didattiche alle aule $J[1, i-1]$, con l'assegnazione $J[i] \leftarrow h$, assegna la i-esima attività a_i all'aula A_h che si libera per ultima tra tutte quelle che si liberano prima dell'inizio s_i di a_i. Se nessuna delle aule si libera prima di s_i l'attività a_i non viene assegnata e $J[i] = 0$.

Supponiamo che esista una programmazione ottima $B[1, n]$ che assegna le prime $i-1$ lezioni come l'algoritmo, ossia $B[1, i-1] = J[1, i-1]$, e dimostriamo che esiste una programmazione ottima $B'[1, n]$ tale che $B'[1, i] = J[1, i]$.

Se $B[i] = J[i]$ siamo a posto. Se $J[i] = 0$ l'attività i-esima è incompatibile con le attività $J[1, i-1]$ e quindi $B[i] = J[i] = 0$. Supponiamo quindi $J[i] \neq 0$ e $B[i] \neq J[i]$.

Se $B[i] = 0$, la programmazione ottima $B[1, n]$ non assegna nessuna aula all'attività i-esima. $B[1, n]$ deve assegnare all'aula A_h a cui l'algoritmo assegna la i-esima attività qualche altra attività successiva (altrimenti non sarebbe ottima). Consideriamo la prima attività di indice $j > i$ assegnata da $B[1, n]$ all'aula A_h. Sia $B'[1, n]$ la programmazione che assegna le attività come $B[1, n]$ tranne che sostituisce la j-esima con la i-esima (ossia $B'[i] = J[i]$ e $B'[j] = 0$). Siccome $f_j \geq f_i$ questo non comporta sovrapposizioni nell'aula A_h e quindi $B'[1, n]$ è una programmazione ottima tale che $B'[1, i] = J[1, i]$.

Se $B[i] \neq 0$, la programmazione ottima $B[1, n]$ assegna l'attività i-esima ad una diversa aula $A_{h'}$ con $h' = B[i]$ diverso da $h = J[i]$.

Quindi, dopo aver assegnato le prime $i-1$ attività entrambe le aule A_h ed $A_{h'}$ dovevano essere libere. In B possiamo quindi scambiare tra le due aule $A_{h'}$ ed A_h le attività di indice maggiore o uguale ad i ottenendo anche in questo caso una programmazione ottima tale che $B'[1, i] = J[1, i]$.

Quando l'algoritmo termina $J[1, n] = B[1, n]$ è una programmazione ottima.

Esercizio 6 Dimostrare che ogni algoritmo di compressione che accorcia qualche sequenza di bit deve necessariamente allungarne qualche altra. (Suggerimento: confrontare il numero di sequenze con il numero di codifiche.)

Soluzione.

Supponiamo, per assurdo, che esista un algoritmo di compressione che non allunga nessuna sequenza di bit ma ne accorcia qualcuna.

Sia f la più corta sequenza che viene accorciata dall'algoritmo e sia n la sua lunghezza.

Il numero di sequenze di lunghezza minore di n è $2^n - 1$.

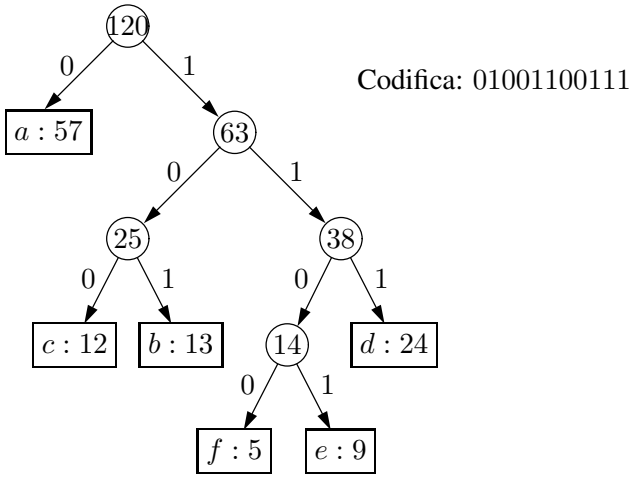
Le codifiche di sequenze di lunghezza minore di n sono pure $2^n - 1$ (per la corrispondenza uno-uno).

Siccome l'algoritmo non allunga nessuna sequenza ogni sequenza di bit di lunghezza minore di n è codifica di un file di lunghezza minore di n.

Pertanto la codifica di f non può avere lunghezza minore di n.

Esercizio 7 Sia $C = \{c_1, \ldots, c_n\}$ un insieme di caratteri ed $f_1, \ldots, f_n$ le loro frequenze in un file. Mostrare come si possa rappresentare ogni codice prefisso ottimo per C con una sequenza di $2n - 1 + n\lceil \lg n \rceil$ bits. (Suggerimento: usare $2n - 1$ bit per rappresentare la struttura dell'albero ed $n\lceil \lg n \rceil$ bits per elencare i caratteri nell'ordine in cui compaiono nelle foglie dell'albero del codice.)

Soluzione. Rappresentiamo la struttura dell'albero con la sequenza in preordine dei bit che etichettano gli archi dell'albero a cui aggiungiamo un bit 1 alla fine.



Per ricostruire la strutture dell'albero si parte dalla radice e quindi:

1. ogni volta che si incontra un bit 0 si aggiunge un figlio sinistro e si scende sul figlio appena aggiunto;

2. quando si incontra un bit 1 si risale fino a che si incontra un antenato privo di figlio destro. Se lo si trova si aggiunge un figlio destro a tale antenato e ci si sposta su tale figlio. Altrimenti abbiamo finito.

Rappresentiamo la sequenza dei caratteri associati alle foglie elencando i loro codici a lunghezza fissa (servono $\lceil \lg n \rceil$ bits per ogni carattere).

Per ricostruire l'associazione tra caratteri e foglie basta associare i caratteri alle foglie nell'ordine da sinistra a destra.

Esercizio 8 Dimostrare che le seguenti definizioni di matroide sono equivalenti:

Definizione 1

1. L'insieme vuoto è indipendente;
2. Se B è indipendente ed $A \subseteq B$ allora anche A è indipendente;
3. Se A e B sono indipendenti e A contiene più elementi di B allora esiste $x \in A \setminus B$ tale che $B \cup \{x\}$ sia indipendente.

Definizione 2

a. Esiste almeno un insieme indipendente;
b. Se B è indipendente ed $A \subseteq B$ allora anche A è indipendente;
c. Per ogni sottoinsieme R di S, se A e B sono due sottoinsiemi di R indipendenti e massimali in R allora $|A| = |B|$.

Soluzione. Assumiamo che la famiglia di insiemi $\mathcal{M}$ soddisfi la Definizione 1 e dimostriamo che soddisfa la Definizione 2.

a. l'insieme vuoto è indipendente e quindi esiste almeno un insieme indipendente;
b. è uguale al punto 2. della Definizione 1;
c. Siano A e B sottoinsiemi di $R \subseteq S$ indipendenti e massimali in R. Se $|A| < |B|$ allora, per il punto 3., esiste $x \in B \setminus A$ tale che $A \cup x$ sia indipendente e quindi A non sarebbe massimale. Analogamente se $|B| < |A|$ esiste $y \in A \setminus B$ tale che $B \cup y$ sia indipendente e quindi B non sarebbe massimale. Quindi $|A| = |B|$.

Nel verso opposto assumiamo che la famiglia di insiemi $\mathcal{M}$ soddisfi la Definizione 2 e dimostriamo che soddisfa la Definizione 1.

1. Per il punto a. esiste almeno un insieme indipendente A. Siccome $\emptyset \subseteq A$, per il punto b. l'insieme vuoto è indipendente.

2. è uguale al punto b. della Definizione 2;

3. Siano A e B indipendenti e $|A| < |B|$. Sia $R = A \cup B$ e sia C un indipendente massimale contenuto in R che estende A. Siccome $|C| \geq |B| > |A|$ esiste $y \in C \setminus A$. Per il punto b. $A \cup y \subseteq C$ è indipendente. Infine $y \in B \setminus A$ poiché $C \setminus A \subseteq R \setminus A \subseteq B \setminus A$.

Esercizio 9 Sia S un insieme di vettori in uno spazio lineare di dimensione d. Dimostrare che la famiglia $\mathcal{M}$ dei sottoinsiemi linearmente indipendenti di S è un matroide. Assumendo che a ciascun vettore di S sia associato un peso, scrivere un algoritmo che calcola un sottoinsieme di vettori linearmente indipendenti di peso massimo.

Soluzione. Ricordiamo che k vettori $v_1, \ldots, v_k$ si dicono linearmente dipendenti se esistono k coefficienti $a_1, \ldots, a_k$ dei quali almeno uno è diverso da 0 e tali che

$$a_1 v_1 + \ldots + a_k v_k = 0$$

Si dicono linearmente indipendenti se tale equazione è vera soltanto se $a_1, \ldots, a_k$ sono tutti nulli. In questo caso essi generano un sottospazio di dimensione k.

Per $k = 0$ l'equazione $a_1 v_1 + \ldots + a_k v_k = 0$ è banalmente vera e, non essendovi coefficienti, a maggior ragione non vi è alcun coefficiente diverso da 0. Dunque l'insieme vuoto è indipendente e la prima proprietà è verificata.

Dimostriamo la seconda proprietà.

Supponiamo che $v_1, \ldots, v_k$ siano linearmente indipendenti e sia $v_1, \ldots, v_h$ un sottoinsieme di tali vettori.

Mostriamo che anche i vettori $v_1, \ldots, v_h$ sono linearmente indipendenti.

Se, per assurdo, i vettori $v_1, \ldots, v_h$ fossero linearmente dipendenti esisterebbero coefficienti $a_1, \ldots, a_h$ non tutti nulli tali che $a_1 v_1 + \ldots + a_h v_h = 0$.

Estendendo la sequenza $a_1, \ldots, a_h$ con $k - h$ zeri otteniamo una sequenza $a_1, \ldots, a_k$ di coefficienti non tutti nulli tali che $a_1 v_1 + \ldots + a_k v_k = 0$. Assurdo perché $v_1, \ldots, v_k$ sono indipendenti.

Quindi tutti i sottoinsiemi di insiemi di vettori indipendenti sono indipendenti e la seconda proprietà è dimostrata.

Dimostriamo infine la terza proprietà.

Osserviamo intanto che se uno dei vettori $v_1, \ldots, v_k$ è 0 essi sono linearmente dipendenti (se $v_i = 0$ si prenda $a_i = 1$ e tutti gli altri coefficienti $a_j = 0$). Dunque gli insiemi indipendenti non contengono il vettore nullo.

Siano $v_1, \ldots, v_k$ e $w_1, \ldots, w_h$ due insiemi di vettori linearmente indipendenti (e quindi non nulli) e supponiamo $h < k$. Vogliamo dimostrare che esiste un v_i tale che i vettori $w_1, \ldots, w_h, v_i$ siano indipendenti.

Supponiamo, per assurdo, che per ogni vettore v_i i vettori $w_1, \ldots, w_h, v_i$ siano dipendenti.

Per ogni $i = 1, \ldots, k$, esistono allora $h + 1$ coefficienti $a_{i,1}, \ldots, a_{i,h}, b_i$ non tutti nulli tali che:

$$a_{i,1}w_1 + \ldots + a_{i,h}w_h + b_i v_i = 0$$

Siccome $w_1, \ldots, w_h$ sono indipendenti $b_i \neq 0$. Dunque si può esplicitare tali equazioni rispetto ai vettori v_i ottenendo le k equazioni

$$v_i = a'_{i,1}w_1 + \ldots + a'_{i,h}w_h$$

per $i = 1, \ldots, k$, dove $a'_{i,j} = -a_{i,j}/b_i$.

La matrice $A' = [a'_{i,j}]$ ha rango minore o uguale ad $h < k$ e dunque i vettori $v_1, \ldots, v_k$ non possono essere linearmente indipendenti. Assurdo.

Dunque la famiglia $\mathcal{M}$ dei sottoinsiemi linearmente indipendenti di S è un matroide.

L'algoritmo goloso è una istanza di quello generale:

SELEZIONA(v, p, n)

/* Precondizione: I vettori sono ordinati per peso $p_1 \geq p_2 \geq \ldots \geq p_n$. */

$A[1] \leftarrow v_1$, $k \leftarrow 1$

for $i \leftarrow 1$ **to** n **do**

  if "$A[1..k]$ unito v_i è indipendente" **then**

    $k \leftarrow k + 1$, $A[k] \leftarrow v_i$

/* Postcondizione: $A[1..k]$ è insieme indipendente di costo massimo. */

return A, k

Esercizio 10 Una ditta, al fine di ridurre le spese telefoniche, intende collegare con linea privata i centri calcolo delle sue molte sedi. A tal fine si è procurata i preventivi per un insieme S di connessioni dirette tra due sedi. Sia r_i il risparmio previsto con la connessione diretta tra le sedi x_i ed y_i. Una connessione diretta tra sedi già collegate da una catena di connessioni dirette è chiaramente inutile. Dimostrare che la famiglia $\mathcal{M}$ dei sottoinsiemi aciclici di S è un matroide. Sviluppare un algoritmo che calcola un sottoinsieme aciclico di connessioni che massimizza il risparmio.

Soluzione. Che l'insieme vuoto sia aciclico e che sottoinsiemi di insiemi aciclici siano aciclici è ovvio, e quindi la prima e seconda proprietà sono verificate.

Dimostriamo quindi la terza proprietà. Siano

$$A = \{(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\}$$

e

$$B = \{(z_1, w_1), (z_2, w_2), \ldots, (z_h, w_h)\}$$

insiemi aciclici di connessioni con $|A| = k < |B| = h$.

Dimostriamo per induzione su $t = |A \setminus B|$ che esiste una connessione (z_j, w_j) in $B \setminus A$ tale che sia aciclico l'insieme $A' = A \cup \{(z_j, w_j)\}$

Base: Se $t = 0$ allora $A \setminus B$ è vuoto e quindi A è sottoinsieme di B. Preso un qualsiasi elemento (z_j, w_j) in $B \setminus A$ l'insieme $A' = A \cup \{(z_j, w_j)\}$ è sottoinsieme di B ed è quindi aciclico.

Passo induttivo: Se $A \setminus B$ contiene $t > 0$ elementi prendiamone uno qualsiasi (x_i, y_i) ed aggiungiamolo a B ottenendo $B' = B \cup \{(x_i, y_i)\}$.

Caso 1: B' è aciclico. Siccome $|B'| = |B| + 1 > |A|$ l'insieme B' è un insieme aciclico più grande di A tale che $A \setminus B'$ ha solo $t - 1$ elementi.

Per ipotesi induttiva esiste allora (z_j, w_j) in $B' \setminus A$ (e dunque in $B \setminus A$) tale che $A' = A \cup \{(z_j, w_j)\}$ sia aciclico.

Caso 2: Aggiungendo (x_i, y_i) a B si forma un ciclo. In questo caso tale ciclo deve contenere qualche elemento (z_s, w_s) in $B \setminus A$ (perché A è aciclico).

Togliendo (z_s, w_s) da B' rimane l'insieme aciclico

$$B'' = B \cup \{(x_i, y_i)\} \setminus \{(z_s, w_s)\}$$

tale che $A \setminus B''$ abbia soltanto $t - 1$ elementi.

Siccome $|B''| = |B| > |A|$, per ipotesi induttiva esiste (z_j, w_j) in $B' \setminus A$ (e dunque in $B \setminus A$) tale che $A' = A \cup \{(z_j, w_j)\}$ sia aciclico.

Anche in questo caso l'algoritmo goloso è una istanza di quello generale:

SELEZIONA(c, r, n)
/* Precondizione: Le connessioni c_i sono ordinate per risparmio non crescente $r_1 \geq r_2 \geq \ldots \geq r_n$. */
$A[1] \leftarrow c_1$, $k \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
  if "$A[1..k]$ unito c_i è indipendente" **then**

$k \leftarrow k + 1$, $A[k] \leftarrow c_i$

/* Postcondizione: $A[1..k]$ è insieme indipendente con risparmio massimo. */

return A, k

Esercizio 11 Un cassiere vuole dare un resto di n centesimi di euro usando il minimo numero di monete.

a) Descrivere un algoritmo goloso per fare ciò con tagli da 1¢, 2¢, 5¢, 10¢, 20¢, 50¢, 1€ e 2€.

b) Dimostrare che l'algoritmo goloso funziona anche con monete di tagli

$$c^0 = 1, c^1 = c, \ldots, c^k$$

dove c è un intero maggiore di 1 e $k \geq 0$.

c) Trovare un insieme di tagli di monete per i quali l'algoritmo goloso non funziona.

Soluzione. La scelta golosa consiste nello scegliere il massimo numero possibile di monete di taglio massimo.

RESTO(R, t, n)

/* Precondizione: $t_1 > t_2 > \ldots > t_n = 1$ sono i tagli delle monete. R è il resto da dare. */

for $i \leftarrow 1$ **to** n **do**

$A[i] \leftarrow \lfloor R/t_i \rfloor$, $R \leftarrow R \bmod t_i$

/* Postcondizione: $A[1], A[2], \ldots, A[n]$ sono il numero di monete di tagli rispettivi $t_1, t_2, \ldots, t_n$. */

a) I tagli sono $t_1 = 2€$, $t_2 = 1€$, $t_3 = 50¢$, $t_4 = 20¢$, $t_5 = 10¢$, $t_6 = 5¢$, $t_7 = 2¢$ e $t_8 = 1¢$.

Sia $A_1, A_2, \ldots, A_8$ una qualsiasi soluzione ottima per un certo resto R.

Quindi $R = \sum_{i=1}^{8} A_i t_i$ con $n = \sum_{i=1}^{8} A_i$ minimo.

Sia $R_k = \sum_{i=k+1}^{8} A_i t_i$ la somma delle monete di taglio minore di t_k.

Se riusciamo a dimostrare che $R_k < t_k$ per ogni k allora la soluzione ottima $A_1, A_2, \ldots, A_8$ è proprio la soluzione che viene calcolata dall'algoritmo e dunque l'algoritmo calcola una soluzione ottima.

R_8**:** $R_8 = 0¢ < t_8$.

R_7**:** $R_7 = A_8¢$. Per l'ottimalità $A_8 < 2$ (altrimenti potrei sostituire due monete da $1¢$ con una da $2¢$) e quindi $R_7 \leq 1¢ < t_7$.

R_6**:** $R_6 = 2A_7 + A_8¢$. Per l'ottimalità $A7 < 3$ (altrimenti potrei sostituire tre monete da $2¢$ con una da $5¢$ ed una da $1¢$) ed inoltre se $A_7 = 2$ allora $A_8 = 0$ (altrimenti potrei sostituire due monete da $2¢$ ed una da $1¢$ con una da $5¢$). Quindi $R_6 \leq 4¢ < t_6$.

R_5**:** $R_5 = 5A_6 + R_6¢$. Per l'ottimalità $A_6 < 2$ (altrimenti potrei sostituire due monete da $5¢$ con una da $10¢$). Quindi $R_5 \leq 5¢ + R_6 < 5¢ + t_6 = t_5$.

R_4**:** $R_4 = 10A_5 + R_5¢$. Per l'ottimalità $A_5 < 2$ (altrimenti potrei sostituire due monete da $10¢$ con una da $20¢$). Quindi $R_4 \leq 10¢ + R_5 < 10¢ + t_5 = t_4$.

R_3**:** $R_3 = 20A_4 + 10A_5 + R_5¢$. Per l'ottimalità $A_4 < 3$ (altrimenti potrei sostituire tre monete da $20¢$ con una da $50¢$ ed una da $10¢$) ed inoltre se $A_4 = 2$ allora $A_5 = 0$ (altrimenti potrei sostituire due monete da $20¢$ ed una da $10¢$ con una da $50¢$). Quindi $R_3 \leq 40¢ + R5 < 40¢ + t5 < t3$.

R_2**:** $R_2 = 50A_3 + R_3¢$. Per l'ottimalità $A_3 < 2$ (altrimenti potrei sostituire due monete da $50¢$ con una da 1€). Quindi $R_2 \leq 50¢ + R_3 < 50¢ + t3 = t2$.

R_1**:** Infine $R_1 = 100A_2 + R2¢$. Per l'ottimalità $A_2 < 2$ (altrimenti potrei sostituire due monete da 1€ con una da 2€). Quindi $R_1 \leq 1e + R_2 < 1e + t_2 = t1$.

b) I tagli sono $t_1 = c^{k-1}, t_2 = c^{k-2}, \ldots, t_{k-1} = c^1 = c$ e $t_k = c^0 = 1$ con $c > 1$ e $k \geq 1$.

Sia $A_1, A_2, \ldots, A_k$ una soluzione ottima per un certo resto R.

Quindi $R = \sum_{i=1}^{k} A_i t_i$ con $n = \sum_{i=1}^{k} A_i$ minimo.

Sia $R_j = \sum_{i=j+1}^{8} A_i t_i$ la somma delle monete di taglio minore di t_j.

Se riusciamo a provare che $R_j < t_j$ per ogni j allora la soluzione ottima $A_1, A_2, \ldots, A_k$ è proprio la soluzione calcolata dall'algoritmo e dunque l'algoritmo calcola una soluzione ottima.

Intanto $R_k = 0 < t_k$.

Per ogni $j < k$ abbiamo $R_j = c^{k-j-1}A_{j+1} + R_{j+1}$. Per l'ottimalità $A_{j+1} < c$ (altrimenti potrei sostituire c monete di taglio $t_{j+1} = c^{k-j-1}$ con una di taglio $t_j = c^{k-j}$). Quindi $R_j \leq (c-1)t_{j+1} + R_{j+1} < ct_{j+1} = t_j$.

c) Scegliamo i tagli $t_1 = 7¢, t2 = 5¢$, e $t_3 = 1¢$. Con $R = 10¢$ l'algoritmo goloso sceglie una moneta da $7¢$ e tre da $1¢$. La soluzione ottima è invece costituita da due sole monete da $5¢$.

Esercizio 12 Mostrare che se al contatore binario A di k bit aggiungiamo anche una operazione DECREMENT(A) che decrementa di una unità il valore del contatore allora una sequenza di n operazioni può costare $O(nk)$.

Soluzione. Consideriamo una successione di $n = 2^k - 1$ operazioni (con $k > 2$) di cui le prime $2^{k-1} - 1$ sono INCREMENT(A) (dopo di che i bit del contatore sono tutti 1) mentre le altre 2^{k-1} sono una ripetizione di 2^{k-2} gruppi di due operazioni: una INCREMENT(A) seguita da una DECREMENT(A).

Il numero di bit che modificati dalle due operazioni di un gruppo è $k + k = 2k$ e il numero totale di bit modificati è maggiore di $2k2^{k-2} = O(kn)$.

Esercizio 13 Su di una certa struttura dati viene eseguita una sequenza di n operazioni. L'operazione i-esima costa i quando i è una potenza di 2 mentre ha costo 1 negli altri casi. Mostrare che tali operazioni hanno costo ammortizzato costante.

Soluzione. Il costo dell'i-esima operazione è:

$$c_i = \begin{cases} 1 & \text{se } i \text{ non è potenza di 2} \\ i & \text{se } i \text{ è potenza di 2} \end{cases}$$

Il costo totale della sequenza di n operazioni è:

$$\begin{aligned} C(n) &= \sum_{i=1}^{n} c_i = n - \lfloor \log_2 n \rfloor + \sum_{j=0}^{\lfloor \log_2 n \rfloor} 2^j \\ &= n - \lfloor \log_2 n \rfloor + 2^{\lfloor \log_2 n \rfloor + 1} - 1 \\ &\leq 3n \end{aligned}$$

Quindi il costo ammortizzato di una operazione è $O(3n)/n = O(1)$.

Esercizio 14 Realizzare un contatore binario di $k + 1$ bit $A[0..k]$ che prevede, oltre all'operazione INCREMENT(A), anche una operazione RESET(A) che azzera il contatore. Fare in modo che la complessità ammortizzata delle operazioni risulti costante. Suggerimento: memorizzare la posizione m del bit 0 successivo al bit 1 più significativo (ossia m è il minimo tale che tutti i bit in $A[m..k]$ sono 0).

Soluzione.

RESET(A)
    /* Tutti i bit in $A[m,k]$ sono 0 */
    for $i \leftarrow 0$ **to** $m-1$ **do** $A[i] \leftarrow 0$
    $m \leftarrow 0$

INCREMENT(A)
    $i \leftarrow 0$
    while $i \leq k$ **and** $A[i] = 1$ **do**
        $A[i] \leftarrow 0$, $i \leftarrow i+1$
    if $i = k+1$ **then**
        $m \leftarrow 0$
    else
        $A[i] \leftarrow 1$
        if $i = m$ **then** $m \leftarrow m+1$

Il costo effettivo di una operazione INCREMENT(A) è $t+1$ pari al numero di bit modificati. Tra questi vi è un certo numero $t \geq 0$ di 1 trasformati in 0 e al più un solo 0 trasformato in 1.

Il costo effettivo di una operazione RESET(A) è $m+1$.

Attribuiamo un costo ammortizzato 3 all'operazione INCREMENT(A) ed un costo ammortizzato 1 a RESET(A).

Quando eseguiamo una INCREMENT(A) usiamo una unità di costo per pagare l'eventuale bit 0 trasformato in 1, una unità di costo la attribuiamo come credito prepagato a tale bit 1 e l'altra unità di costo la attribuiamo alla variabile m se essa viene incrementata. Quindi ogni bit 1 ha sempre un suo credito prepagato e la variabile m ha sempre esattamente m crediti prepagati.

Quando eseguiamo una INCREMENT(A) paghiamo la trasformazione dei t bit 1 in 0 usando i crediti prepagati attribuiti a tali bit.

Quando eseguiamo una RESET(A) usiamo gli m crediti attribuiti alla variabile m per pagare l'azzeramento dei primi m bit del registro.

Alternativamente possiamo usare la funzione potenziale

$$\Phi = m + \sum_{i=0}^{k} A[i]$$

Esercizio 15 Realizzare una pila P con operazioni di costo ammortizzato costante avendo a disposizione memoria per al più M elementi. Se la memoria è piena quando si esegue una $Push$, prima di eseguire l'operazione vengono scaricati su disco alcuni elementi. Se una operazione Pop toglie l'ultimo elemento in memoria e ci sono degli elementi registrati su disco, dopo l'operazione se ne ricaricano alcuni in memoria.

Soluzione. Una scelta che funzione è scaricare e ricaricare dalla memoria secondaria blocchi di $M/2$ elementi.

Sia n il numero di elementi in memoria principale e sia b una variabile booleana avente valore $true$ se vi è qualche gruppo scaricato in memoria di massa e $false$ altrimenti. Prendiamo come funzione potenziale:

$$\Phi(n,b) = \begin{cases} n & \text{se } b = false \\ |n - M/2| + M/2 & \text{se } b = true \end{cases}$$

Se $b = false$ una $Push$ senza scaricamenti in memoria secondaria ha costo ammortizzato:

$$\begin{aligned} \hat{c} &= c + \Phi(n+1) - \Phi(n) \\ &= 1 + (n+1) - (n) = 2 \end{aligned}$$

mentre se vi scaricamento in memoria secondaria ($n = M$) allora

$$\begin{aligned} \hat{c} &= c + \Phi(M/2+1) - \Phi(M) \\ &= 1 + M/2 + (|M/2 + 1 - M/2| + M/2) - (M) = 2 \end{aligned}$$

Invece una Pop ha costo ammortizzato

$$\begin{aligned} \hat{c} &= c + \Phi(n-1) - \Phi(n) \\ &= 1 + (n-1) - (n) = 0 \end{aligned}$$

Se $b = true$ una $Push$ senza scaricamenti in memoria secondaria ha costo ammortizzato:

$$\begin{aligned} \hat{c} &= c + \Phi(n+1) - \Phi(n) \\ &= 1 + (|n + 1 - M/2| + M/2) - (|n - M/2| + M/2) \leq 2 \end{aligned}$$

mentre se vi scaricamento in memoria secondaria ($n = M$) allora

$$\begin{aligned} \hat{c} &= c + \Phi(M/2+1) - \Phi(M) \\ &= 1 + M/2 + (|M/2 + 1 - M/2| + M/2) - (|M - M/2| + M/2) = 2 \end{aligned}$$

Invece una Pop senza recupero da memoria secondaria ha costo ammortizzato

$$\begin{aligned}\hat{c} &= c + \Phi(n-1) - \Phi(n) \\ &= 1 + (|n-1-M/2| + M/2) - (|n-M/2| + M/2) \leq 2\end{aligned}$$

mentre se vi è recupero da memoria secondaria ($n = 0$) ma la memoria secondaria non si svuota (b rimane $true$) allora

$$\begin{aligned}\hat{c} &= c + \Phi(m/2-1) - \Phi(0) \\ &= 1 + M/2 + (|M/2-1-M/2| + M/2) - (|0-M/2| + M/2) = 2\end{aligned}$$

Infine se vi è recupero da memoria secondaria ($n = 0$) e la memoria secondaria si svuota (b diventa $false$) allora

$$\begin{aligned}\hat{c} &= c + \Phi(m/2-1) - \Phi(0) \\ &= 1 + M/2 + (M/2-1) - (|0-M/2| + M/2) = 0\end{aligned}$$

Esercizio 16 Realizzare una coda Q utilizzando due normali pile P_1 e P_2 in modo che le operazioni ENQUEUE(Q, x) e DEQUEUE(Q) richiedano tempo ammortizzato costante.

Soluzione.

ENQUEUE(Q, x)
    PUSH(P_1, x)

DEQUEUE(Q)
    if EMPTY(P_2) **then**
        while not EMPTY(P_1) **do**
            $x \leftarrow$ POP(P_1) , PUSH(P_2, x)
    $x \leftarrow$ POP(P_2)
    return x

Il costo effettivo di una ENQUEUE(Q, x) è 1, quello di una DEQUEUE(Q) senza travaso è 1 mentre quello di una DEQUEUE(Q) con travaso è $1 + |P_1|$.

Usiamo la funzione potenziale $\Phi = |P_1|$. ENQUEUE(Q, x) ha costo ammortizzato $\hat{c} = c + \Delta\Phi = 1 + 1 = 2$. Il costo ammortizzato di DEQUEUE(Q) senza travaso è $\hat{c} = c + \Delta\Phi = 1 + 0 = 1$. Infine il costo ammortizzato di DEQUEUE(Q) con travaso è $\hat{c} = c + \Delta\Phi = 1 + |P_1| - |P_1| = 1$.

Esercizio 17 Assumere che la contrazione della tavola dinamica venga effettuata quando $\alpha = \frac{1}{3}$ invece che quando $\alpha = \frac{1}{4}$ e che invece di ridurre la sua dimensione ad $\frac{1}{2}\,capacity$ essa venga ridotta a $\frac{2}{3}\,capacity$. Calcolare il costo ammortizzato delle operazioni usando la funzione potenziale:

$$\Phi = |2 \cdot size - capacity|$$

Soluzione. Il costo effettivo di una INSERT(T, x) senza espansione e di una DELETE(T, x) senza contrazione è 1, quello di una INSERT(T, x) con espansione e quello di una DELETE(T, x) con contrazione è $1 + size$.

Se $\alpha \geq 1/2$ il costo ammortizzato di una INSERT(T, x) senza espansione è

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot size_i - capacity_i) - (2 \cdot size_{i-1} - capacity_{i-1}) \\
&= 1 + 2(size_{i-1} + 1) - capacity_{i-1} - 2 \cdot size_{i-1} + capacity_{i-1}) \\
&= 3
\end{aligned}$$

mentre con espansione è:

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + size_{i-1} + (2 \cdot size_i - capacity_i) - (2 \cdot size_{i-1} - capacity_{i-1}) \\
&= 1 + size_{i-1} + 2(size_{i-1} + 1) - 2 \cdot size_{i-1} - 2 \cdot size_{i-1} + size_{i-1}) \\
&= 3
\end{aligned}$$

(Per $i = 1$ abbiamo $capacity_i = size_{i-1} + 1$ invece di $capacity_i = 2 \cdot size_{i-1}$. In questo caso $\hat{c}_i = 2$.)

Se $\alpha < 1/2$ non vi è sicuramente espansione e il costo ammortizzato di una INSERT(T, x) è:

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (capacity_i - 2 \cdot size_i) - (capacity_{i-1} - 2 \cdot size_{i-1}) \\
&= 1 + capacity_{i-1} - 2(size_{i-1} + 1) - capacity_{i-1} + 2 \cdot size_{i-1}) \\
&= -1
\end{aligned}$$

Se $\alpha \leq 1/2$ il costo ammortizzato di una DELETE(T, x) senza contrazione è:

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (capacity_i - 2 \cdot size_i) - (capacity_{i-1} - 2 \cdot size_{i-1}) \\
&= 1 + capacity_{i-1} - 2(size_{i-1} - 1) - capacity_{i-1} + 2 \cdot size_{i-1}) \\
&= 3
\end{aligned}$$

mentre con contrazione è:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + size_{i-1} + (capacity_i - 2 \cdot size_i) - (capacity_{i-1} - 2 \cdot size_{i-1}) \\ &= 1 + size_{i-1} + 2 \cdot size_{i-1} - 2(size_{i-1} - 1) - 3 \cdot size_{i-1} + 2 \cdot size_{i-1}) \\ &= 3\end{aligned}$$

Se $\alpha > 1/2$ non vi è sicuramente contrazione e il costo ammortizzato di una DELETE(T, x) è:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot size_i - capacity_i) - (2 \cdot size_{i-1} - capacity_{i-1}) \\ &= 1 + 2(size_{i-1} - 1) - capacity_{i-1} - 2 \cdot size_{i-1} + capacity_{i-1}) \\ &= -1\end{aligned}$$