

# Algoritmi e Strutture Dati

## 03/02/14

### Esercizio 1

Nei compiti passati, è possibile che abbiate notato che nelle equazioni di ricorrenza del tipo:

$$T(n) = \begin{cases} T(n/2) + T(n/4) + T(n/8) + 1 & n > 1 \\ 1 & n = 1 \end{cases}$$

dove la somma delle frazioni  $1/2 + 1/4 + 1/8$  è inferiore a 1, il limite superiore risultante è pari a  $O(n)$ . Questo può essere utilizzato come tentativo, visto che la sommatoria delle frazioni espresse nella sommatoria è inferiore a 1.

Proviamo quindi a dimostrare che  $\exists c > 0, \exists m \geq 0 : T(n) \leq cn, \forall n \geq m$ .

- Ipotesi induttiva:  $\forall n' < n, T(n') \leq cn'$ .
- Passo induttivo:

$$\begin{aligned} T(n) &= \left( \sum_{i=1}^{\lfloor \log n \rfloor} T(n/2^i) \right) + 1 \\ &\leq \left( \sum_{i=1}^{\lfloor \log n \rfloor} \frac{cn}{2^i} \right) + 1 && \text{Sostituzione dell'ipotesi induttiva} \\ &= cn \left( \sum_{i=1}^{\lfloor \log n \rfloor} (1/2)^i \right) + 1 && \text{Semplificazioni algebriche} \\ &\leq cn \left( \sum_{i=1}^{\infty} (1/2)^i \right) + 1 && \text{Estensione sommatoria} \\ &= cn + 1 && \text{Serie geometrica infinita decrescente} \\ &\leq cn \end{aligned}$$

Qui abbiamo un problema matematico;  $cn + 1 \not\leq cn$ , ma la diseuguaglianza non è verificata per un termine di ordine inferiore. Proviamo a dimostrare che  $\exists b > 0, \exists c > 0, \exists m \geq 0 : T(n) \leq cn - b, \forall n \geq m$ . Se riusciamo a dimostrare che  $T(n) \leq cn - b$ , abbiamo anche dimostrato che  $T(n) \leq cn$ .

- Ipotesi induttiva:  $\forall n' < n, T(n') \leq cn' - b$ .
- Passo induttivo:

$$\begin{aligned} T(n) &= \left( \sum_{i=1}^{\lfloor \log n \rfloor} T(n/2^i) \right) + 1 \\ &\leq \left( \sum_{i=1}^{\lfloor \log n \rfloor} \frac{cn}{2^i} - b \right) + 1 && \text{Sostituzione dell'ipotesi induttiva} \\ &= cn \left( \sum_{i=1}^{\lfloor \log n \rfloor} (1/2)^i \right) - b \lfloor \log n \rfloor + 1 && \text{Semplificazioni algebriche} \\ &\leq cn \left( \sum_{i=1}^{\infty} (1/2)^i \right) + 1 && \text{Estensione sommatoria} \\ &= cn - b \lfloor \log n \rfloor + 1 && \text{Serie geometrica infinita decrescente} \\ &\leq cn - b \end{aligned}$$

L'ultima disequazione è vera per  $b \geq \frac{1}{1+\lceil \log n \rceil}$  e per ogni  $c$ . Poichè  $\frac{1}{1+\lceil \log n \rceil} \geq 1$  per qualunque valore  $n \geq 1$ , possiamo scegliere  $b = 1$  e  $m = 1$  per questo caso.

- Passo base:

$$T(1) = 1 \leq c \cdot 1 - b = c - b \leq 1 - b$$

Per dimostrare l'ultima disequazione, è sufficiente che  $c \leq 1$ .

Possiamo quindi concludere che  $T(n) = O(n)$ , con parametri  $c = 1, m = 1, b = 1$ .

## Esercizio 2

Il minimo numero di turni necessari per informare tutti i nodi di un albero  $T$  dipende ricorsivamente dal numero di figli e da quanti turni sono necessari ad essi per informare tutti i nodi del loro sottoalbero. Possono darsi un certo numero di casi:

- Una foglia  $u$  richiede  $\text{turni}(u) = 0$  turni per informare il suo sottoalbero.
- Un nodo  $u$  con un figlio  $v$  richiede  $\text{turni}(u) = \text{turni}(v) + 1$  turni per consegnare il messaggio a tutto il suo sottoalbero
- Un nodo  $u$  con due figli  $v_1$  e  $v_2$ :
  - Se  $\text{turni}(v_1) > \text{turni}(v_2)$  (rispettivamente: se  $\text{turni}(v_2) > \text{turni}(v_1)$ ), sono necessari  $\text{turni}(v_1) + 1$  (rispettivamente:  $\text{turni}(v_2) + 1$ ) turni per consegnare il messaggio, in quanto il nodo  $u$  prima consegnerà il messaggio al nodo  $v_1$ , e poi parallelamente, mentre il nodo  $v_1$  informa il suo sottoalbero, potrà spedire il messaggio al nodo  $v_2$
  - Se  $\text{turni}(v_1) = \text{turni}(v_2)$ , sono necessari  $\text{turni}(v_1) + 2$  turni per avviare la spedizione nei sottoalberi di entrambi i figli.

Per comodità di scrittura del codice, concentriamo assieme molti di questi casi ritornando  $-1$  nel caso di un figlio **nil**.

---

```

int turni(TREE T)
  if T == nil then
    | return -1
  if T.left == T.right == nil then
    | return 0
  int tl = turni(T.left)
  int tr = turni(T.right)
  return iff(tl == tr, tl + 2, max(tl, tr) + 1)

```

---

Essendo una semplice post-visita dell'albero, il costo computazionale è pari a  $O(n)$ .

## Esercizio 3

La procedura `connesso(G, x)` visita il grafo  $G$  a partire da un nodo casuale utilizzando la DFS ricorsiva `ccdfs()`, evitando di passare attraverso il nodo  $x$ . Restituisce **true** se il grafo  $G$  privato del nodo  $x$  è connesso, **false** altrimenti.

La procedura `cercaNodo(G)` itera sui nodi di  $G$ , utilizzando la procedura `connesso()` per verificare se la rimozione di un nodo  $x$  disconnette il grafo. Restituisce il primo nodo che rimosso, lascia il grafo connesso.

Il costo computazionale della visita effettuato da `connesso()` è pari a  $O(m + n)$ . `cercaNodo()` chiama `connesso()` per  $n$  volte nel caso pessimo, per un costo computazionale pari a  $O(mn)$ .

---

**boolean** cercaNodo(GRAPH  $G$ )

---

```
foreach  $x \in G.V()$  do
  if connesso( $G, x$ ) then
    return  $x$ 
return nil
```

---

---

**boolean** connesso(GRAPH  $G$ , NODE  $x$ )

---

```
boolean[] visitato = new boolean[1... $G.n$ ]
foreach  $u \in G.V()$  do
  visitato[ $u$ ] = false
ccdfs( $G, x, \text{random}(G.V() - \{x\}), \text{visitato}$ )
foreach  $u \in G.V() - \{x\}$  do
  if not visitato[ $u$ ] then
    return false
return true
```

---

---

ccdfs(GRAPH  $G$ , NODE  $x$ , NODE  $u$ , int[] visitato)

---

```
visitato[ $u$ ] = true
foreach  $v \in G.\text{adj}(u) - \{x\}$  do
  if not visitato[ $v$ ] then
    ccdfs( $G, x, v, \text{visitato}$ )
```

---

Si può fare meglio di così?

#### Esercizio 4

Questo esercizio può essere risolto tramite la tecnica del backtrack, ed è stato valutato positivamente anche se la conseguente complessità è pari a  $O(2^n)$ . Ma è possibile risolvere il problema in tempo lineare tramite programmazione dinamica! Il trucco sta nel capire che è possibile identificare quali righe sono occupate utilizzando una maschera binaria con valori fra 0 e 15. Ad esempio, 0000 = 0 significa che tutte le righe sono libere; 1100 = 12 significa che le righe 2, 3 sono occupate, le righe 0, 1 no; 1111 = 15 significa che tutte le righe sono occupate.

Sia  $DP[k][m]$  il miglior guadagno che si può ottenere utilizzando le prime  $k$  colonne e avendo  $m$  come maschera che rappresenta le righe già occupate. Il valore che stiamo cercando sarà quindi contenuto in  $DP[n][0]$ , ad indicare che abbiamo disposizione  $n$  colonne e nessuna delle righe è già occupata.

La ricorrenza per il calcolo di  $DP[k][m]$  può essere espressa nel modo seguente:

$$DP[k][m] = \begin{cases} 0 & k = 0 \vee m = 15 \\ \min\{DP[k-1][m], \min_{i:0 \leq i \leq 3 \wedge (2^i \text{ and } m) = 0} \{DP[k-1][m \text{ or } 2^i] + S[i, k]\}\} & \text{altrimenti} \end{cases}$$

Se non ci sono più colonne disponibili ( $k = 0$ ), oppure se tutte le righe sono occupate ( $m = 15$ ), allora il guadagno massimo è 0, perchè non è possibile aggiungere alcuna torre. Se invece è ancora possibile aggiungere torri, ci sono due possibilità: si salta la colonna  $k$ , e si cerca in  $DP[k-1][m]$ ; altrimenti si prova ad aggiungere una torre nelle posizioni in cui questo è possibile. Per verificare se è possibile, si considerano tutte le righe  $i$  fra 0 e 3, e si verifica nella maschera che l' $i$ -esimo bit sia spento ( $(2^i \text{ and } m) = 0$ ). Se è spento, si considera il problema  $DP[k-1][m \text{ or } 2^i]$  dove l' $i$ -esimo bit è stato acceso. Fra tutti questi, va selezionato il minimo.

Si noti che il caso  $m = 15$  non è strettamente necessario, in quanto se tutte le righe sono occupate si considererà i casi  $DP[k - 1][m]$ ,  $DP[k - 2][m]$ ,  $DP[k - 3][m]$ , ... senza modificare la maschera, fino a quando non si avrà  $k = 0$ .

---

```

int maxGain(int[][] S, int n)
int[][] DP = new int[0...15][0...n]
for m = 0 to 15 do
   $DP[0][m] = 0$ 
for k = 0 to n do
   $DP[k][15] = 0$ 
for k = 1 to n do
  for m = 0 to 14 do
     $DP[k][m] = DP[m][k - 1]$ 
    for i = 0 to 3 do
      if (m and  $2^i$ ) = 0 then
        if  $DP[k - 1][m \text{ or } 2^i] + S[i, k] > DP[k][m]$  then
           $DP[k][m] = DP[k - 1][m \text{ or } 2^i] + S[i][k]$ 
  return  $DP[0][n]$ 

```

---

La complessità dell'algoritmo è pari a  $\Theta(n)$ , in quanto solo il ciclo su  $k$  dipende da  $n$ , mentre gli altri hanno valori costanti.