Algoritmi e Strutture Dati - 07/02/2017

Esercizio 1

Utilizzando il master theorem, è facile vedere che $T(n) = \Theta(\sqrt[3]{n}\log n)$.
Dimostriamolo per sostituzione, partendo da $T(n) = O(\sqrt[3]{n}\log n)$.
Coinvolgendo il logaritmo, il caso base è fra quelli problematici:

$$T(1) = 1 \not\leq c\sqrt[3]{1}\log 1 = 0$$

Per questo motivo, consideriamo i valori i compresi fra 2 e 15, estremi inclusi; $i/8$ in questo caso è minore di 2; scriviamo quindi

$$T(i) = 2T(i/8) + \sqrt[3]{i} = 2 + \sqrt[3]{i} \leq c\sqrt[3]{i}\log i \qquad \forall i : 2 \leq i \leq 15$$

da cui si ottiene:

$$c \geq \frac{2 + \sqrt[3]{i}}{\sqrt[3]{i}\log i} \qquad \forall i : 2 \leq i \leq 15$$

Per $i = 16$, $i/8$ è pari a 2 e rientra nei casi base già risolti. Possiamo quindi fermarci a 15.
Nel passo induttivo, dobbiamo dimostrare che $T(n) \leq c\sqrt[3]{n}\log n$ e supponiamo che la relazione $T(n') \leq c\sqrt[3]{n'}\log n'$ sia già stata dimostrata per $2 \leq n' < n$.

$$\begin{aligned} T(n) &\leq 2c\sqrt[3]{n/8}\log n/8 + \sqrt[3]{n} \\ &= c\sqrt[3]{n}\log n/8 + \sqrt[3]{n} \\ &= c\sqrt[3]{n}(\log n - \log 8) + \sqrt[3]{n} \\ &= c\sqrt[3]{n}\log n - 3c\sqrt[3]{n} + \sqrt[3]{n} \leq c\sqrt[3]{n}\log n \end{aligned}$$

L'ultima disequazione è soddisfatta se $c \geq 1/3$. Poiché questa disequazione per c e tutte quelle derivanti dal caso base sono di tipo $\geq$, è sufficiente prendere il valore più alto fra questi valori come valore per c.
Consideriamo ora $T(n) = \Omega(\sqrt[3]{n}\log n)$. Il caso base è più facile del precedente:

$$T(1) = 1 \geq c\sqrt[3]{1}\log 1 = 0$$

che è vero per tutti i valori di c. Nel passo induttivo, dobbiamo dimostrare che $T(n) \geq c\sqrt[3]{n}\log n$ e supponiamo che la relazione $T(n') \geq c\sqrt[3]{n'}\log n'$ sia già stata dimostrata per $2 \leq n' < n$.

$$\begin{aligned} T(n) &\geq 2c\sqrt[3]{n/8}\log n/8 + \sqrt[3]{n} \\ &= c\sqrt[3]{n}\log n/8 + \sqrt[3]{n} \\ &= c\sqrt[3]{n}(\log n - \log 8) + \sqrt[3]{n} \\ &= \sqrt[3]{n}(c\log n - 3c + 1) \geq c\sqrt[3]{n}\log n \end{aligned}$$

L'ultima disequazione è vera per $c \leq 1/3$.
Abbiamo quindi che $T(n) = \Theta(\sqrt[3]{n}\log n)$.

Esercizio 2

L'esercizio può essere risolto in modo ricorsivo. La funzione countTree() prende in input l'albero e un contatore di ascendendenti (*ancestors*), e restituisce un contatore di discendenti e un contatore di nodi che hanno il numero di discendenti uguali al numero di ascendenti.

```
(int, int) countTreeRec(TREE t, int ancestors)
    if t == nil then
        return (0,0)
    pred_L, count_L = countTreeRec(t.left, ancestors + 1)
    pred_R, count_R = countTreeRec(t.right, ancestors + 1)
    return (pred_L + pred_R + 1, count_L + count_R + iif(ancestors = pred_L + pred_R, 1, 0))
```

L'algoritmo viene invocato dalla seguente funzione wrapper:

```
int countTree(TREE t)
    int, int pred, count = countTreeRec(t, 0)
    return count
```

Esercizio 3

Questo esercizio è più semplice di quanto si possa pensare. L'idea è che tutte le volte si incontra una parentesi chiusa, la si associa alla più vicina parentesi aperta precedente.

```
longestBalanced(ITEM[] S, int n)
int count = 0                          % Total number of balanced parenthesis found so far
int open = 0                           % Total number of open parenthesis not yet balanced found so far
for i = 1 to n do
    if S[i] = "(" then
        open = open + 1
    else if S[i] = ")" and open > 0 then
        count = count + 2
        open = open − 1
return count
```

La complessità di questo algoritmo è $\Theta(n)$.
Per dimostrare la correttezza dell'algoritmo, sia j la posizione della prima parentesi chiusa che sia preceduta da almeno una parentesi aperta. Sia i la posizione della più vicina parentesi aperta che precede j ($i < j$). I caratteri compresi fra i e j non possono essere parentesi aperte, in quanto i è la parentesi aperta più vicina a j; non possono essere parentesi chiuse, perchè j è la prima parentesi chiusa.
Si consideri quindi la stringa derivata da S cui tutti i carattere fra i e j sono stati eliminati, estremi inclusi, e sommiamo 2 al numero di parentesi bilanciate. La stringa così ottenuta sarà potenzialmente composta da parentesi aperte (quelle prima di i) e da parentesi chiuse (quelle dopo di j). Il fatto di aver associato i e j non influisce sulle parentesi chiuse successive (che possono essere associate a quelle precedenti) nè su quelle precedenti (che possono essere associate a quelle successive).
Esistono anche tanti modi per risolvere il problema con programmazione dinamica. Ed esistono anche tanti modi per sbagliare a risolvere il problema con programmazione dinamica. Ne faccio vedere alcuni (senza codice, solo formula ricorsiva).
Sia $DP[i][j]$ la lunghezza della più lunga sottosequenza contenuta in $S[i \ldots j]$ che sia una stringa bilanciata di parentesi. $D[i][j]$ può essere calcolata in questo modo:

$$DP[i][j] = \begin{cases} 0 & i \geq j \\ DP[i+1][j] & S[i] \neq \text{"("} \\ DP[i][j-1] & S[j] \neq \text{")"} \\ \max\{DP[i+1][j-1]+2, \max_{i \leq k \leq j}\{DP[i][k] + DP[k+1][j]\}\} & S[i] = \text{"("} \wedge S[j] = \text{")"} \end{cases}$$

L'idea è la seguente:

- Il caso base è costituito da stringhe di 0 o 1 caratteri, ed ovviamente la più lunga sottostringa bilanciata è lunga 0
- Se ci sono caratteri diversi da parentesi aperte e chiuse all'inizio e alla fine della stringa, rispettivamente, la accorciamo facendo scorrere gli indici i e j
- Se il primo e ultimo carattere sono parentesi tonde, allora possono darsi due casi:
  - la stringa corrisponde al secondo caso della definizione di stringhe di parentesi bilanciate, ovvero $w = (x)$; togliamo il primo e l'ultimo carattere e sommiamo due alla più lunga sottosequenza di parentesi bilanciate contenuta all'interno
  - la stringa corrisponde al terzo caso della definizione di stringhe di parentesi bilanciate, ovvero $w = xy$; nel qual caso proviamo a spezzare la stringa in tutte le posizioni possibili e restituiamo il massimo fra esse

  Fra questi due casi, dovremo prendere il massimo.

Trasformando questa formula ricorsiva tramite programmazione dinamica o memoization, si ottiene un costo pari a $O(n^3)$. Non utilizzando programmazione dinamica o memoization, il costo è esponenziale.
Un errore comune è stato dimenticarsi del terzo caso, e scrivere una formula tipo questa:

$$DP[i][j] = \begin{cases} 0 & i \geq j \\ DP[i+1][j] & S[i] \neq \text{"("} \\ DP[i][j-1] & S[j] \neq \text{")"} \\ DP[i+1][j-1]+2 & S[i] = \text{"("} \wedge S[j] = \text{")"} \end{cases}$$

Se l'input è "()()", questa formula restituisce 2 invece che 4.

Esercizio 4

Una formula ricorsiva per calcolare il numero di alberi k-limitati strutturalmente diversi può essere derivata dalla formula ricorsiva per calcolare il numero di alberi binari strutturalmente diversi vista a lezione:

$$DP[n][k] = \begin{cases} 1 & n = 1 \land k = 0 \\ 0 & k \geq n \\ 2DP[n-1][k-1] + \sum_{i=1}^{n-2} DP[i][k] \cdot DP[n-1-i][k] & n > 1 \land k < n \end{cases}$$

Spiegazione:

- Se $n = 1$, esiste un solo albero binario 0-limitato: l'albero formato da un singolo nodo.
- Se $k = n$, non è possibile costruire un albero binario k-limitato: tutti i nodi dovrebbero avere esattamente un figlio, ma almeno uno di essi è una foglia; ancora più banalmente, non è possibile avere k nodi con un figlio se $k > n$.
- Altrimenti, ci sono due possibilità: questo nodo ha esattamente un figlio, oppure ne ha due. Se ha un figlio, questo può essere destro o sinistro; i restanti $n-1$ nodi devono contenere $k-1$ nodi con max 1 figlio; questo spiega la componente $2DP[n-1][k-1]$. Se ha due figli, siano i i nodi che vanno a sinistra; e siano $n-1-i$ i nodi che vanno a destra. Il numero di combinazioni possibili è quindi pari a $DP[i][k] \cdot DP[n-1-i][k]$, in quanto il numero di nodi con un figlio non va diminuito. Dobbiamo sommare tutti questi casi, con i che va da 1 a $n-2$.

L'algoritmo per calcolare $DP[n][k]$, basato su programmazione dinamica è il seguente:

```
k-limitato(int n, int k)
    int[][] DP = new int[0...n][0...k]          % Initialized with ⊥
    k-limitato-rec(DP, n, k)
    return DP[n][k]
```

```
k-limitato-rec(int [ ][ ] DP, int n, int k)
    if n == 1 and k == 0 then
        return 1
    if k ≥ n then
        return 0
    if DP[n][k] = ⊥ then
        DP[n][k] = 2 · k-limitato-rec(DP, n − 1, k − 1)
        for i = 2 to n − 2 do
            DP[n][k] = DP[n][k] + k-limitato-rec(DP, i, k) · k-limitato-rec(DP, n − i − 1, k)
    return DP[n][k]
```

Il costo della procedura è pari a $O(kn^2)$