Algoritmi e Strutture Dati - 05/11/15

Esercizio 1

1. C'erano due versioni simili della stessa ricorrenza. In entrambi i casi, era possibile utilizzare il teorema delle ricorrenze lineari con partizione bilanciata:
   - $T(n) = T(\lfloor \beta n \rfloor) + n^\beta$: abbiamo che $a = 1$ e $b = 1/\beta > 1$. Ignorando l'operatore $\lfloor\rfloor$, si ha che $\alpha = \log_b a = 0$. Siamo nel caso (3) ($\alpha < \beta$) e quindi la complessità è pari a $\Theta(n^\beta)$.
   - $T(n) = \left\lfloor \frac{1}{\beta} \right\rfloor T(\lfloor \beta n \rfloor) + n^\beta$. Ignorando l'operatore $\lfloor\rfloor$, si ha che $a = b = 1/\beta > 1$. Quindi si ottiene che $\alpha = \log_b a = 1$. Siamo nel caso (1) ($\alpha > \beta$) e quindi la complessità è pari a $\Theta(n^1)$.

2. C'erano due versioni simili della stessa ricorrenza. Riportiamo le due soluzioni:
   - $T(n) = T\left(\frac{1}{2}n\right) + T\left(\frac{4}{5}n\right) + T\left(\frac{3}{10}n\right) + n^2$: ipotizziamo che $T(n) = O(n^2)$. Dobbiamo quindi dimostrare che:

     $$\exists c > 0, \exists m \geq 0 : T(n) \leq cn^2, \forall n \geq m$$

     - Passo base: $T(1) = 1 \leq c \cdot 1^2 = c$, che implica $c \geq 1$.
     - Ipotesi induttiva: $T(n') \leq c(n')^2$, per tutti i valori $n' < n$.
     - Passo induttivo:

     $$\begin{aligned} T(n) &= T\left(\tfrac{1}{2}n\right) + T\left(\tfrac{4}{5}n\right) + T\left(\tfrac{3}{10}n\right) + n^2 \\ &\leq \tfrac{1}{4}cn^2 + \tfrac{16}{25}cn^2 + \tfrac{9}{100}cn^2 + n^2 \\ &= \tfrac{25+64+9}{100}cn^2 + n^2 \\ &= \tfrac{98}{100}cn^2 + n^2 \\ &\stackrel{?}{\leq} cn^2 \end{aligned}$$

     L'ultima disequazione è vera per $c \geq 50$.

     Possiamo quindi concludere che $T(n) = O(n^2)$, con $m = 1$ e $c = 50$. È facile poi vedere che $T(n) = \Omega(n^2)$ (per via del termine non ricorsivo), e quindi $T(n) = O(n^2)$.
   - $T(n) = T\left(\frac{1}{2}n\right) + T\left(\frac{4}{5}n\right) + T\left(\frac{3}{10}n\right) + n^2$: ipotizziamo che $T(n) = O(n^2)$. Dobbiamo quindi dimostrare che:

     $$\exists c > 0, \exists m \geq 0 : T(n) \leq cn^2, \forall n \geq m$$

     - Passo base: $T(1) = 1 \leq c \cdot 1^2 = c$, che implica $c \geq 1$.
     - Ipotesi induttiva: $T(n') \leq c(n')^2$, per tutti i valori $n' < n$.
     - Passo induttivo:

     $$\begin{aligned} T(n) &= T\left(\tfrac{1}{2}n\right) + T\left(\tfrac{2}{5}n\right) + T\left(\tfrac{7}{10}n\right) + n^2 \\ &\leq \tfrac{1}{4}cn^2 + \tfrac{4}{25}cn^2 + \tfrac{49}{100}cn^2 + n^2 \\ &= \tfrac{25+16+49}{100}cn^2 + n^2 \\ &= \tfrac{90}{100}cn^2 + n^2 \\ &\stackrel{?}{\leq} cn^2 \end{aligned}$$

     L'ultima disequazione è vera per $c \geq 10$.

     Possiamo quindi concludere che $T(n) = O(n^2)$, con $m = 1$ e $c = 10$. È facile poi vedere che $T(n) = \Omega(n^2)$ (per via del termine non ricorsivo), e quindi $T(n) = O(n^2)$.

Esercizio 2

Anche questo problema era presente in due versioni. In una si chiedeva di valutare la media dei valori delle chiavi presenti nel sottoalbero, nell'altro la differenza fra il valore massimo e il valore minimo delle chiavi presenti nel sottoalbero.
Riportiamo sotto le due soluzioni, che hanno identica struttura basata su una post-visita. Entrambe le soluzioni hanno una chiamata esterna, che restituisce il valore desiderato, e una chiamata ricorsiva, che calcola i valori necessari. Nel primo caso, il valore restituito dalla chiamata ricorsiva è una coppia di interi che rappresentano la somma dei valori e il numero totale di nodi contenuti nel sottoalbero radicato. Nel secondo caso, il valore restituito dalla chiamata ricorsiva è una coppia di interi che rappresentano il minimo e il massimo dei valori contenuti nel sottoalbero radicato.
Essendo post-visite, la complessità di entrambi gli algoritmi è $O(n)$.

```
real computeAverage(TREE t)
    tot, count = computeAverageRec(t)
    return tot/count
```

```
(int, int) computeAverageRec(TREE t)
    if T == nil then
        return (0, 0)
    tot_L, count_L = computeAverageRec(t.left)
    tot_R, count_R = computeAverageRec(t.right)
    return (T.key + tot_L + tot_R, 1 + count_L + count_R)
```

```
int computeDiff(TREE t)
    min, max = computeDiffRec(t)
    return max − min
```

```
(int, int) computeDiffRec(TREE t)
    if T == nil then
        return (+∞, −∞)
    min_L, max_L = computeDiffRec(t.left)
    min_R, max_R = computeDiffRec(t.right)
    return (min(T.key, min_L, max_R), max(t.key, max_L, max_R))
```

Esercizio 3

1. Per la prima parte, è sufficiente modificare una visita in profondità, in modo tale che non visiti mai il nodo u. Facendo partire una visita modificata da s, se il nodo t viene raggiunto, allora esiste un cammino da s a t che non passa per u e quindi si ritorna **false**. Se invece tutti i cammini da s a t passano necessariamente per u, la visita modificata non può raggiungere t e viene restituito **true**. **true** viene restituito anche nel caso non esistano cammini da s a t, come richiesto nel compito. La complessità è quella di una visita, ovvero $O(m + n)$.

```
boolean checkAll(GRAPH G, NODE s, NODE t, NODE u
    boolean[] visited = new boolean[1 ... G.n]
    foreach u ∈ G.V() do
        visited[u] = 0
    checkAll-DFS(G, s, u, visited)
    return notvisited[t]
```

```
checkAll-DFS(GRAPH G, NODE x, NODE u, boolean[] visited)
    visited[x] = true
    foreach v ∈ G.adj(x) do
        if not visited[v] and v ≠ u then
            checkAll-DFS(G, v, uvisited)
```

2. Per la seconda parte, è sufficiente fare una visita da s, e verificare se u è raggiungibile; e poi una visita a partire da u, e verificare se t è raggiungibile. Per minimizzare la quantità di scrittura e massimizzare il riuso del codice, utilizziamo la funzione erdos(). La complessità è pari a $O(m + n)$.

```
boolean checkExists(GRAPH G, NODE s, NODE t, NODE u
    int[] erdős = new int[1 ... G.n]
    erdos(G, s, erdős)
    if erdős[u] == ∞ then
        return false
    erdos(G, u, erdős)
    if erdős[t] == ∞ then
        return false
    return true
```

Esercizio 4

Anche questo problema era presente in due versioni, che differivano di ben poco: una chiedeva la somma, l'altra la frequenza del valore più frequente. Presentiamo qui la somma, il prodotto richiede di passare dalla moltiplicazione alla potenza.
Conoscendo il numero n_a, n_b dei valori a, b presenti nel vettore, è possibile calcolare il risultato finale in tempo costante: $n_a \cdot a + n_b \cdot b$.
Per conoscere il numero di valori presenti, è sufficiente identificare l'indice k dell'ultima occorrenza del valore a. In questo modo,

- il numero di occorrenze di a è pari a $n_a = k$;
- il numero di occorrenze di b è pari a $n_b = n - k$.

Sebbene il vettore sia ordinato, non possiamo possiamo utilizzare la ricerca binaria pubblicata nel libro per trovare l'ultima occorrenza di a, in quanto restituisce un'occorrenza qualunque.
Per risolvere questo problema, si lavora su sottovettori $A[i \ldots j]$ in cui $A[i] = a$ e $A[j] = b$. Questa condizione è vera anche nella chiamata iniziale su $A[1 \ldots n]$, in quanto abbiamo assunto che vi sia almeno un valore a e almeno un valore b.
Si prende quindi l'elemento centrale $A[m]$ del vettore; se questo è pari a $A[i]$, allora tutti gli elementi in $A[i \ldots m-1]$ sono pari ad a e ci si può concentrare sul sottovettore $A[m \ldots j]$; altrimenti, tutti gli elementi in $A[m+1 \ldots j]$ sono pari a b e ci si può concentrare sul sottovettore $A[i \ldots m]$.
Il caso base è rappresentato da un vettore di 2 elementi, in cui il primo è pari ad a ed è l'ultimo ad avere tale valore. Possiamo quindi restituire il suo indice i.
Tale procedura ha complessità $T(n) = T(\lceil n/2 \rceil) + 1$; utilizzando il Master Theorem, otteniamo che $T(n) = \Theta(\log n)$.
La versione mostrata qui sotto si basa sulle assunzioni elencate nel problema e non necessita di avere in input i valori di a, b, che si trovano in $A[1], A[n]$.

```
computeSum(int[] A, int n)
    int k = binarySearchAB(A, 1, n)
    return k · A[1] + (n − k) · A[n]          % Questa versione per restituire la somma
    return max(k, n − k)                      % Questa versione per restituire la frequenza del valore più frequente
```

```
binarySearchAB(int[] A, int i, int j)
    if j == i + 1 then
        return i
    int m = ⌊(i + j)/2⌋
    if A[m] == A[i] then
        return binarySearchAB(A, m, j)
    else
        return binarySearchAB(A, i, m)
```