# *Algoritmi e Strutture Dati - Seconda provetta*
# *18/06/12*

## Esercizio 1

Una semplice visita posticipata risolve il problema. La funzione wrapper chiama una procedura ricorsiva di visita, a cui viene passato il livello del nodo visitato, incrementato di uno ad ogni chiamata ricorsiva. Vengono calcolati il numero di nodi "allo stesso livello" nel sottoalbero destro e nel sottoalbero sinistro. Se il valore del nodo è pari al livello, tale numero è incrementato di 1. La complessità è ovviamente $\Theta(n)$.

```
sameLevel(TREE T)
    return sameLevelRec(T, 0)
```

```
sameLevelRec(TREE T, int level)
    if T == nil then
        return 0
    int count = sameLevel(T.right(), level + 1) + sameLevel(T.left(), level + 1)
    if T.key == level then
        count = count + 1
    return count
```

## Esercizio 2

La procedure Merge() (non richiesta dall'esercizio, l'aggiungo solo per completezza) può essere scritta nel modo seguente. Notate che ne esiste una versione più efficiente che sceglie il minimo tramite un heap binario, ma ci complica solo la vita dopo.

```
Merge(int[] A, int i, int j, int step)
    int m = ⌊(i − j + 1)/step⌋
    int[] pos = new int[1 . . . m]
    int[] end = new int[1 . . . m]
    foreach k = 1 to m do
        pos[k] = i + step · (k − 1)
        end[k] = i + step · k − 1
    int t = i
    while t ≤ j do
        int min = 1
        for k = 2 to m do
            if pos[k] ≤ end[k] and pos[k] < pos[min] then
                min = k
        B[t] = A[pos[min]]
        pos[min] = pos[min] + 1
    for t = i to j do
        A[t] = B[t]
```

La sua complessità è pari a $O(n\sqrt{n})$, in quanto per riempiere ognuno degli $n$ elementi è necessario cercare il minimo fra $\sqrt{n}$ posizioni. L'equazione di ricorrenza risultante è

$$T(n) = \begin{cases} \sqrt{n}T(\sqrt{n}) + n\sqrt{n} & n \geq 4 \\ 1 & n < 4 \end{cases}$$

Questo perchè per ognuno degli $n$ valori, devo guardare in ognuno dei $\sqrt{n}$ sottovettori qual è il minore.

Ponendo $n = 2^k$ (oppure $k = \log n$) nella ricorrenza, otteniamo:

$$\begin{aligned} T(n) &= \sqrt{n}T(\sqrt{n}) + n\sqrt{n} \Rightarrow \\ T(2^k) &= 2^{\frac{1}{2}k}T(2^{\frac{1}{2}k}) + 2^{\frac{3}{2}k} \end{aligned}$$

Dividiamo tutto per $2^k$, e otteniamo:

$$\frac{T(2^k)}{2^k} = \frac{T(2^{\frac{1}{2}k})}{2^{\frac{1}{2}k}} + 2^{\frac{1}{2}k}$$

Poniamo ora $S(k) = \frac{T(2^k)}{2^k}$; otteniamo quindi:

$$S(k) = S(k/2) + 2^{k/2}$$

Tramite il master theorem, si vede che $S(k) = O(2^{k/2})$. Riscrivendo l'espressione come funzione $T$, si ottiene

$$\frac{T(2^k)}{2^k} = 2^{k/2}$$

da cui si ottiene $T(n) = n\sqrt{n}$. Questa versione di MergeSort è peggiore di quella originale.

## Esercizio 3

Si utilizza Dijkstra per ottenere il costo dei cammini minimi che vanno da $q$ a tutti gli altri nodi; il costo di un $q$-cammino minimo da $u$ a $v$ è dato dal costo di un cammino minimo da $q$ ad $u$ (che è pari al costo del cammino minimo da $u$ a $q$, essendo il grafo non orientato) più il costo del cammino minimo da $q$ a $v$.
Assumendo di avere una versione dell'algoritmo di Dijkstra che restituisce la distanza minima dal nodo sorgente, il codice si scrive semplicemente in questo modo:

```
int [ ][ ] qcammini(GRAPH G, NODE q, )
    int[ ][ ] D = new int[1 ... G.n][1 ... G.n]
    int[ ] d = new int[1 ... G.n]
    camminiMinimi(G, q, d)
    foreach u ∈ G.V() do
        foreach v ∈ G.V() do
            D[u][v] = d[u] + d[v]
    return D
```

dove $d$ è il vettore utilizzato per memorizzare il costo del cammino minimo da $q$ agli altri nodi, e $D$ è la matrice dei costi finali.
Il costo di Dikstra è $O(n^2)$, il costo dei due cicli annidati è $O(n^2)$, quindi il costo finale è $O(n^2)$.

## Esercizio 4

Per ogni riga, esistono solo cinque modi per collocare le pedine, rappresentate dal simbolo 1 nella tabella seguente:

| | | | |
|---|---|---|---|
| | | | = 0 |
| | | 1 | = 1 |
| | 1 | | = 2 |
| 1 | | | = 4 |
| 1 | | 1 | = 5 |

Questi cinque modi possono essere interpretati come altrettanti numeri binari, come evidenziato a destra della tabella. Invece, i numeri 3,6,7 non possono essere scelti in un piazzamento regolare perché hanno due o tre bit 1 adiacenti. I piazzamenti di due righe successive sono regolari se e solo se effettuando un **and** binario dei valori corrispondenti si ottiene 0 (nessun bit coincidente).
Sia $M[i,t]$ il valore massimo che si può ottenere assegnando le prime $i$ righe e utilizzando il numero binario $t \in 0,1,2,4,5$ come assegnamento per l'$i$-esima riga. Il valore $M[i,t]$ può essere calcolato nel modo seguente:

$$M[i][t] = \begin{cases} 0 & i = 0 \\ \max\left\{ M[i-1][t'] + \sum_{x=0}^{2} A[i][x] \cdot 2^x \cdot (t >> x \;\&\; 1) : t' \in \{0,1,2,4,5\} \textbf{ and } (t' \;\&\; t) = 0 \right\} & i > 0 \end{cases}$$

dove $(t >> x \;\&\; 1)$ è l'espressione C/Java che ritorna 1 se l'x-esimo bit di $t$ è acceso, 0 altrimenti. L'idea è considerare tutte i possibili piazzamenti per la riga $i$-esima, ed abbinarle al valore massimo che si ottiene nei piazzamenti regolari della riga $i-1$-esima.

Tradotto in pseudo-codice utilizzando programmazione dinamica e non memoization, si ottiene il codice seguente.

```
int scacchiera(int [ ][ ] A, int n)
    int[][] M = new int[0...n][0...7]
    foreach t ∈ {0, 1, 2, 4, 5} do
        M[0][t] = 0
    for i = 1 to n do
        foreach t ∈ {0, 1, 2, 4, 5} do
            M[i][t] = −∞
            foreach t' : t' ∈ {0, 1, 2, 4, 5} and (t' & t) == 0 do
                int tot = M[i − 1][t']
                for x = 0 to 2 do
                    tot = tot + A[i][x] · 2^x · (t >> x & 1)
                if tot > M[i][t] then
                    M[i][t] = tot
    return max{M[n][t] : t ∈ {0, 1, 2, 4, 5}}
```

Il costo computazionale è $O(n)$, dato dal ciclo **for** sulla variabile $i$; tutti gli altri cicli infatti hanno dimensione costante.