Algoritmi e Strutture Dati - 06/07/16

Esercizio 1

E' possibile notare che l'equazione può essere limitata superiormente dalla seguente disequazione:

$$\begin{aligned} T(n) &= 2T(\lfloor n/\sqrt{2} \rfloor - 5) + n^{\Pi/2} \\ &\leq 2T(n/\sqrt{2}) + n^{\Pi/2} \end{aligned}$$

Utilizzando il teorema delle Ricorrenze lineari con partizione bilanciata (esteso), è possibile notare che $2T(n/\sqrt{2}) + n^{\Pi/2} = \Theta(n^2)$, in quanto $\alpha = \log_{\sqrt{2}} 2 = 2$ e $\beta = \Pi/2 \approx 1.57$.
Quindi siamo nel caso (1) del teorema, e $n^\beta = O(n^{\alpha-\epsilon})$, che è vera per $\epsilon < 2 - \Pi/2$.

Esercizio 2

E' possibile notare che semplicemente ordinare i valori in ordine crescente, riga per riga, rispetta le regole di ordinamento proposte.

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 7 & 8 & 8 \\ 9 & 10 & 12 & 14 \\ 16 & 18 & 21 & 24 \end{pmatrix}$$

Questo perché ogni numero è necessariamente superiore o uguale ai numeri che lo precedono nelle righe e nelle colonne.
Una soluzione semplice è quindi la seguente: si copiano i valori in un vettore di dimensione $n \times m$, e si ordina tale vettore in tempo $O(nm \log(nm))$. A questo punto, si copia tale vettore di nuovo nella matrice. Lo pseudocodice è il seguente:

```
matrixSort(int[][] A, int m, int n)
    intB ← new int[1 ... m · n]
    for i ← 1 to n do
        for j ← 1 to m do
            B[(i − 1) · n + 1] ← A[i][j]
    sort(B, m · n)
    for i ← 1 to n do
        for j ← 1 to m do
            A[i][j] ← B[(i − 1) · n + 1]
```

Esercizio 3

Il problema si risolve con la programmazione dinamica. Sia $best[j]$ il maggior numero di elettori che saranno presenti considerando le prime i città. $best[j]$ può essere calcolato nel modo seguente:

$$best[j] = \begin{cases} e[1] & j = 1 \\ \max\{best[j-1], e[i] + \max_{1 \leq i < j \wedge m[j] - m[i] \geq D} best[i]\} & j > 1 \end{cases}$$

```
int serveTheDonald(int[] m, int[] e, int n, int D)
    best[1] ← e[1]
    for j ← 2 to n do
        best[j] ← best[j − 1]
        inti ← 1
        while i < j and m[j] − m[i] ≥ D do
            best[j] ← max(best[j], e[j] + best[i])
            i ← i + 1
    return best[n]
```

L'algoritmo, basato su programmazione dinamica, ha una complessità pari a $O(n^2)$ nel caso pessimo.

Esercizio 4

Dato un premio u e detto $Pred_u$ l'insieme dei predecessori di u (nodi che possono raggiungere u tramite un cammino), la probabilità $x[u]$ che u non venga scoperto è pari a:

$$x[u] = (1 - p(u)) \cdot \prod_{v \in Pred_u} (1 - p[v])$$

Per calcolare il vettore x, un modo possibile è quello di rovesciare il ragionamento, facendo partire una visita in profondità da ogni nodo $u \in V$, moltiplicando l'elemento $x[v]$ di ogni nodo che può essere raggiunto da u (compreso u stesso) per il fattore $(1 - p[u])$. Gli elementi di x sono inizializzati ad 1. Così facendo, al termine di questa procedura in $x[u]$ si sono accumulati tutti i fattori provenienti dai nodi predecessori di u, compreso u stesso.
Lo pseudo-codice per questo algoritmo è il seguente:

```
computeProb(GRAPH G, int[] p)
    int[]x ← new int[1...G.n]
    int[]visited ← new int[1...G.n]
    x ← {1, ..., 1}
    foreach u ∈ G.V() do
        visited ← {false, ..., false}
        visitaDFS(G, u, visited, p, x, (1 − p[u]))
    return Σ_{u∈G.V()} (1 − x[u])
```

```
visitaDFS(GRAPH G, NODE u, int[] visited, int[] p, int[]x, int prob,)
    visited[u] ← true
    x[u] ← x[u] · prob
    foreach v ∈ G.adj(u) do
        if not visited[v] then
            visitaDFS(G, v, visited, p, x, prob)
```

Il costo della procedura è $O(nm)$ (n visite in profondità di costo $O(n + m)$). La sommatoria finale è quella che restituisce il valore atteso.