Algoritmi e Strutture Dati – 06/07/16

Esercizio 1

È 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 $T(n) = 2T(n/\sqrt{2}) + n^{\Pi/2} = \Theta(n^2)$, 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

È 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)
    int B = new int[1 ... m · n]
    for i = 1 to n do
        for j = 1 to m do
            B[(i − 1) · n + j] = 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 + j]
```

Esercizio 3

Soluzione $O(n^2)$ Il problema può essere risolto con programmazione dinamica. Sia $DP[j]$ il numero massimo di elettori che potranno essere raggiunti considerando le prime j città. $DP[j]$ può essere calcolato nel modo seguente:

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

In altre parole:

- Se $j = 1$, c'è una sola città e quindi il numero totale di elettori è pari ad $e[1]$

- Nel caso generale, è possibile decidere di saltare la città j, nel qual caso ci si riduce al problema con $j-1$ città, ovvero $DP[j-1]$; altrimenti, si seleziona la città j con $e[j]$ elettori, a cui si somma il massimo numero di elettori fra tutti i sottoproblemi che comprendo i città, $i < j$, tali per cui $m[j] - m[i] \geq D$.

```
int serveTheDonald(int[] m, int[] e, int n, int D)
    int[] DP = new int[1...n]
    DP[1] = e[1]
    for j = 2 to n do
        DP[j] = DP[j − 1]
        int i = 1
        while i < j and m[j] − m[i] ≥ D do
            DP[j] = max(DP[j], e[j] + DP[i])
            i = i + 1
    return DP[n]
```

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

Soluzione $O(n \log n)$ Alternativamente, è possibile ridurre questo problema al problema dell'*Insieme indipendente di intervalli pesati*, dove gli intervalli sono definiti da $[m[i] - D, m[i]$ (in modo da essere incompatibili con tutte le città precedenti più vicine di D) e i pesi sono quelli contenuti nel vettore e. Utilizzando l'algoritmo visto a lezione, è possibile risolvere il problema in tempo $O(n \log n)$, migliore del precedente.

Soluzione $O(n)$ Completo con una soluzione proposta di Samuele Conti nel 2021, basata sulla seguente osservazione: anche gli estremi di inizio degli intervalli sono ordinati. Questo semplifica il problema rispetto a quello generale in cui gli estremi di inizio degli intervalli sono arbitrari e possono causare l'esclusione di un numero arbitrario di altri intervalli (da cui l'operazione di pre-processamento per cercare l'ultimo intervallo compatibile tramite ricerca binaria).
In questa versione, l'indice dell'ultimo intervallo compatibile con j (ovvero che abbia una distanza maggiore di D) è sicuramente maggiore o uguale all'indice dell'ultimo intervallo compatibile con $j-1$, per via della distanza fissa D. Possiamo quindi sfruttare questo fatto per evitare di ripartire sempre dall'intervallo 1, ripartendo invece dall'ultimo intervallo che abbiamo verificato.
Il codice è simile a quello precedente: abbiamo rimosso l'inizializzazione dell'indice i all'interno del ciclo **for**, spostandola all'esterno. Quando si analizza l'intervallo j-esimo, per prima cosa si inizializza $DP[j]$ a $DP[j-1]$, che corrisponde al non prendere j. Dopo di che si cerca di far avanzare i; se i è minore di $j-1$, questo significa che qualche intervallo prima di $j-1$ non era compatabile con esso; ma ora potrebbe essere che i sia compatibile con j; controlliamo tramite la condizione $DP[j] - DP[i] \geq D$. Oppure $i = j-1$: questo significa che tutti gli intervalli prima di $j-1$ sono compatibili con esso. Questo è riflesso nel valore $DP[j-1]$, che abbiamo utilizzato per inizializzare $DP[j]$. Dobbiamo quindi verificare se $j-1$ è compatibile con j, nel qual caso aumenteremo il valore $DP[j]$.

```
int serveTheDonald(int[] m, int[] e, int n, int D)
    int[] DP = new int[1...n]
    DP[1] = e[1]
    int i = 1
    for j = 2 to n do
        DP[j] = DP[j − 1]
        while i < j and m[j] − m[i] ≥ D do
            DP[j] = max(DP[j], e[j] + DP[i])
            i = i + 1
    return DP[n]
```

È facile vedere che nonostante i due cicli annidati, il costo computazionale è $\Theta(n)$, in quanto l'indice i viene fatto scorrere linearmente sui valori compresi fra 1 e $n-1$ e non viene mai arretrato.

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:

```
int computeProb(Graph G, int[] p)
    int[] x = new int[1...G.n] = { 1 }                    % Initialized to 1
    foreach u ∈ G.V() do
        int[] visited = new int[1...G.n] = { false }      % Initialized to false
        DFSVisit(G, u, visited, p, x, (1 − p[u]))
    return Σ_{u∈G.V()} (1 − x[u])
```

```
DFSVisit(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
            DFSVisit(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.