# *Algoritmi e Strutture Dati - 04/09/2017*

## Esercizio 1

Andando per tentativi, proviamo con $\Theta(n^2)$. E' facile vedere che la ricorrenza è $\Omega(n^2)$, per via della sua componente non ricorsiva. Proviamo quindi a dimostrare che $T(n) = O(n^2)$.

- Caso base: $T(n) = 1 \leq cn^2$, per tutti i valori di $n$ compresi fra 1 e 12. Tutte queste disequazioni sono soddisfatte da $c \geq 1$.
- Ipotesi induttiva: $T(k) \leq ck^2$, per $k < n$
- Passo induttivo:

$$\begin{aligned} T(n) &= 3T(\lfloor n/3 \rfloor) + 6T(\lfloor n/6 \rfloor) + 54T(\lfloor n/12 \rfloor) + n^2 \\ &\leq 3c\lfloor n/3 \rfloor^2 + 6c\lfloor n/6 \rfloor^2 + 54\lfloor n/12 \rfloor^2 + n^2 \\ &\leq 3cn^2/9 + 6cn^2/16 + 54cn^2/144 + n^2 \\ &\leq 7/8cn^2 + n^2 \leq cn^2 \end{aligned}$$

  L'ultima disequazione è rispettata per $c \geq 8$.

Abbiamo quindi dimostrato che $T(n) = \Theta(n^2)$.

## Esercizio 2

Un approccio di costo $O(n \log n)$ consiste nell'ordinare il vettore; poi, per ognuno degli elementi trovati, si contano quante istanze sono state trovate e le si inserisce in vettore di appoggio $B$. Una volta ordinato anche questo secondo vettore, si prosegue cercando numeri consecutivi

```
doubleLength(ITEM[] A, int n)
    sort(A, n)
    B ← new int[1 ... n]
    for i ← 1 to n do B[i] ← 0
    int prev ← ⊥
    int distinct ← 0
    for i ← 1 to n do
        if A[i] ≠ prev then
            distinct ← distinct + 1
            B[distinct] ← 1
            prev ← A[i]
        else
            B[distinct] ← B[distinct] + 1
    sort(B, distinct)
    for i ← 2 to distinct do
        if B[i] == B[i − 1] then
            return true
    return false
```

Il costo è dominato dal primo ordinamento, di costo $O(n \log n)$. Il secondo ordinamento può essere risolto con un costo di $O(n)$, utilizzando CountingSort (tutti i valori da ordinare sono compresi fra 1 e $n$), ma questo non cambia la complessità finale, che è $O(n \log n)$. Una soluzione simile basata su tabella hash elimina la necessità del primo ordinamento, e riduce il costo computazionale a $O(n)$.

## Esercizio 3

E' possibile risolvere il problema in maniera non efficiente lanciando una visita a partire da ogni nodo, visitando solo archi che collegano un punto più elevato ad un punto meno elevato, e prendendo la distanza massima così ottenuta. Un simile approccio avrebbe costo $O(n(m+n)) = O(mn)$.
Si può invece sfruttuare il fatto che, una volta calcolata la distanza massima percorribile da un punto $v$, questa può essere utilizzata per calcolare la distanza massima raggiungibile da un punto $u$ tale che $(u,v) \in E$ e $z[v] < z[u]$. In particolare, sia $D[u]$ la distanza massima percorribile dal nodo $u$, allora:

$$D[u] = \begin{cases} 0 & \nexists v : (u,v) \in E \wedge z[v] < z[u] \\ \max_{v:(u,v)\in E \wedge z[v]<z[u]} d[u,v] + D[v] & \text{altrimenti} \end{cases}$$

Il problema può essere risolto con una singola visita in profondità, partendo da ogni nodo che non sia già stato raggiunto da una visita precedente, come mostrato nel codice seguente.

```
longestDescendingPath(GRAPH G, int[] z, int[][] d)
    D ← new int[1 ... G.n]
    foreach u ∈ G.V() do
        D[u] ← ⊥
    foreach u ∈ G.V() do
        if D[u] = ⊥ then
            longest(G, z, d, D, u)
    return max(D)
```

```
int longest(GRAPH G, int[] z, int[][] d, int[][] D, int u)
    if D[u] = ⊥ then
        D[u] ← 0
        foreach v ∈ G.adj(u) : z[v] < z[u] do
            D[u] ← max(D[u], longest(G, z, d, D, v) + d(u, v))
    return D[u]
```

Il costo della procedura è quello di una visita in profondità, $O(m+n)$.

## Esercizio 4

E' possibile risolvere il problema utilizzando la tecnica backtrack. L'idea è la seguente: si deve riempire una stringa lunga $2n$. Ad ogni passo di backtrack, sono date due possibilità: si può aggiungere una parentesi aperta, se non si è esaurito il numero di parentesi aperte ancora da aprire; o si può aggiungere una parentesi chiusa per ogni parentesi aperta non ancora chiusa. Si chiama quindi ricorsivamente la procedura, modificando opportunamente il numero di parentesi da aprire o chiudere. Quando non si hanno più parentesi aperte o chiuse da aggiungere, si stampa la stringa così generata.

```
printparrec(ITEM[] L, int i, int open, int close)
    if open + close = 0 then
        print L
    else
        if open > 0 then
            L[i] ← "("
            printparrec(L, i + 1, open − 1, close + 1)
        if close > 0 then
            L[i] ← ")"
            printparrec(L, i + 1, open, close − 1)
```

La procedure ricorsiva viene richiamata dalla seguente funzione, che alloca un vettore di dimensione $2n$ e poi richiama la funzione con $n$ parentesi da aprire e zero da chiudere (inizialmente).

```
printpar(int n)
    L ← new ITEM[1 ... 2n]
    printparrec(L, 0, n, 0)
```

La complessità è almeno esponenziale, visto che ad ogni chiamata ricorsiva è possibile eseguire due chiamate ricorsive.