Algoritmi e Strutture Dati – 08/01/2018

Esercizio 1

Andando per tentativi, proviamo con $\Theta(n^2\sqrt{n})$. È facile vedere che la ricorrenza è $\Omega(n^2\sqrt{n})$, per via della sua componente non ricorsiva. Proviamo quindi a dimostrare che $T(n) = O(n^2\sqrt{n})$.

- Caso base: $T(k) = 1 \leq ck^2\sqrt{k}$, per tutti i valori di k compresi fra 1 e 15. Tutte queste disequazioni sono soddisfatte da $c \geq \frac{1}{k^2\sqrt{k}}$. Qualunque valore $c \geq 1$ soddisfa *tutte* queste disequazioni.
- Ipotesi induttiva: $T(k) \leq ck^2\sqrt{k}$, per $k < n$
- Passo induttivo:

$$\begin{aligned} T(n) &= 16T(\lfloor n/4 \rfloor) + 256T(\lfloor n/16 \rfloor) + n^2\sqrt{n} \\ &\leq 16c\lfloor n/4 \rfloor^{2.5} + 256c\lfloor n/16 \rfloor^{2.5} + n^2\sqrt{n} \\ &\leq cn^2\sqrt{n}/2 + cn^2\sqrt{n}/4 + n^2\sqrt{n} \\ &\leq 3/4cn^2\sqrt{n} + n^2\sqrt{n} \leq cn^2\sqrt{n} \end{aligned}$$

  L'ultima disequazione è rispettata per $c \geq 4$.

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

Esercizio 2

Per calcolare il grafo trasposto è necessario invertire tutti gli archi, ossia trasformare ogni arco (i, j) nell'arco (j, i). Per fare ciò è sufficiente scambiare tra loro i valori $M[i][j]$ e $M[j][i]$. Attenzione al ciclo interno: la variabile j ha come valore iniziale $i + 1$: è sbagliato far partire j da 1, in quanto ogni elemento verrebbe scambiato due volte, ottenendo nuovamente la matrice iniziale.

```
transpose(boolean[] M, int n)
for i = 1 to n − 1 do
    for j = i + 1 to n do
        M[i][j] ↔ M[j][i]
```

La complessità è pari a $\Theta(n^2)$.

Esercizio 3

Esistono varie soluzioni, una con costo $O(n^2)$ (basata su un doppio ciclo), una con costo $O(n \log n)$ (basata su divide-et-impera) e una con costo $O(n)$, presentata qui.

```
int maxcrime(int[] A, int n, int t)
int i, j, count, maxsofar
i = j = 1
maxsofar = 0
while j ≤ n do
    if A[j] − A[i] + 1 ≤ t then
        maxsofar = max(maxsofar, j − i + 1)
        j = j + 1
    else
        i = i + 1
return maxsofar
```

L'idea è semplice: si difiniscono due indici, i e j, che rappresentano gli estremi del periodo che stiamo considerando.

- Se $A[j] - A[i] + 1$ è più piccolo o uguale del periodo t, si può tentare di allargare gli estremi, incrementando l'indice j. Quando questo viene fatto, viene contato un nuovo omicidio tramite il contatore *count* ed eventualmente aggiornato il numero massimo di omicidi trovati finora tramite la variable *maxsofar*.
- Se $A[j] - A[i] + 1$ è più grande del periodo t, si restringono gli estremi incrementando i; bisogna anche togliere 1 da *count*.

L'algoritmo termina quando l'estremo j raggiunge la fine dell'array. All'inizio, gli estremi sono entrambi pari a 1, ovvero contengono solo il primo elemento.
La complessità è pari a $O(n)$, come detto.

Esercizio 4

L'idea è la seguente: poichè la stringa rappresenta una visita in profondità anticipata, il primo carattere rappresenta la radice dell'albero. Se questo elemento è una `I`, il carattere successivo sarà la radice del sottoalbero sinistro. Bisognerà quindi leggere tutto il sottoalbero sinistro, poi tutto il sottoalbero destro, e l'albero sarà letto interamente. Questo algoritmo potrà essere applicato ricorsivamente ad entrambi i sottoalberi: si legge l'elemento radice, e poi i sotto-alberi sinistri e destri. Se si incontra una lettera `L`, la lettura del sottoalbero è completata.
L'altezza può essere quindi calcolata ricorsivamente come abbiamo visto ad esercitazione.
L'algoritmo può essere scritto in tanti modi. Ricorsivamente, restituendo una coppia di valori interi che rappresentano il prossimo carattere da leggere e l'altezza del sottoalbero analizzato.

```
int nicetreeRec(ITEM[] S, int n, int pos)
    if S[pos] = "L" then
        return (pos + 1, 0)
    else
        newpos, max_L = nicetreeRec(S, n, pos + 1)
        newpos, max_R = nicetreeRec(S, n, newpos)
        return (newpos, max(max_L, max_R) + 1)
```

```
int nicetree(ITEM[] S, int n)
    return nicetreeRec(S, n, 0)
```

Una soluzione alternativa lavora ricorsivamente, restituendo l'altezza del sottoalbero analizzato e utilizzando una variable globale passata per riferimento

```
int nicetreeRec(ITEM[] S, int n, int &pos)
    if S[pos] == "L" then
        pos = pos + 1
        return 0
    else
        pos = pos + 1
        max_L = nicetreeRec(S, n, pos)
        max_R = nicetreeRec(S, n, pos)
        return max(max_L, max_R) + 1
```

```
int nicetree(ITEM[] S, int n)
    pos = 0
    return nicetreeRec(S, n, pos)
```

Ogni carattere viene letto al più una volta, quindi il costo è pari a $O(n)$.