Algoritmi e Strutture Dati
10/09/12

Esercizio 1

La complessità è rappresentata dalla seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 3T(\frac{2}{3}n) + 1 & n > 6 \\ 1 & n \leq 6 \end{cases}$$

Utilizzando il teorema delle ricorrenze lineari con partizione bilanciata, si ottiene che $a = 3$, $b = 3/2$, da cui $\alpha = \log_{3/2} 3$; inoltre, $\beta = 0$. Siamo quindi nel caso $T(n) = n^{\alpha}$.

Non avete una calcolatrice e non sapete quanto sia $\log_{3/2} 3$? E' semplice: $\left(\frac{3}{2}\right)^2 = \frac{9}{4} < 3$, mentre $\left(\frac{3}{2}\right)^3 = \frac{27}{8} > 3$. Quindi α è compreso fra 2 e 3, e quindi questo algoritmo è addirittura peggiore di Insertion Sort. Il prof. Sortino non finirà nel mio libro.

Per rispondere alla domanda sulla correttezza dell'algoritmo, ragioniamo in maniera induttiva. Per tutti i valori di $n \leq 6$, utilizziamo Insertion Sort quindi l'algoritmo è corretto.

Supponiamo ora di aver dimostrato che l'algoritmo funziona correttamente per tutti i valori $n' < n$, e cerchiamo di dimostrarlo per n. Affinché $2\lceil n/3 \rceil < n$, è sufficiente che $2(n/3 + 1) < n$, ovvero che $\frac{2}{3}n + 2 < n$, ovvero che $n > 6$ (da cui il nostro algoritmo). Quindi le varie chiamate ricorsive producono ordinamenti corretti sui valori a cui sono applicati.

Dopo il primo e il secondo ordinamento, tutti i valori nel terzo terzile sono correttamente ordinati. Infatti, si consideri un valore che apparterrebbe al terzo terzile ma si trova inizialmente nel primo. Dopo il primo ordinamento, si troverà necessariamente nel secondo; altrimenti, esisterebbero più di un terzo di elementi maggiori di esso, ma questo è assurdo in quanto appartiene al terzo terzile ordinato. Dopo il secondo ordinamento, qualunque valore appartenente al terzo terzile che si trova nel secondo terzile viene spostato nel terzo terzile.

Il terzo ordinamento serve quindi a ordinare i primi due terzili, che contengono solo valori minori del terzo terzile. Il vettore così risultante è ordinato.

Esercizio 2

Consideriamo i punti *come se* fossero ordinati e calcoliamo la distanza fra tutte le $n - 1$ coppie di punti consecutivi. Se per assurdo tutte le distanze fossero maggiori di d, allora $\mathsf{max}(A) - \mathsf{min}(A) > (n - 1)d$, il che contraddice la definizione di d.

Per individuare una coppia, dividiamo la retta reale fra $\mathsf{max}(A)$ e $\mathsf{min}(A)$ in $n - 1$ "caselle" di dimensione d; utilizzando un meccanismo simile a PigeonHole Sort, memorizziamo i punti a seconda di dove si trovano lungo la retta. La prima volta che due punti vengono assegnati alla stessa casella, hanno una distanza necessariamente inferiore a d.

```
(int, int) closePoints(real[ ] A, int n)
    int[ ] B = new real[0 ... n − 2]
    for i = 0 to n − 2 do
        B[i] = ⊥
    int min = min(A)
    int max = max(A)
    real d = (max − min)/(n − 1)
    for i = 1 to n do
        int j = ⌊(A[i] − min)/d⌋
        if B[j] = ⊥ then
            B[j] = i
        else
            return (B[j], i)
```

La complessità è ovviamente $O(n)$, in quanto sono presente due soli cicli non annidati limitati da n.

Esercizio 3

Utilizziamo una matrice $DP[1 \ldots n][1 \ldots n]$ per memorizzare, in $DP[i][j]$, la dimensione del massimo quadrato il cui angolo basso/destra si trova nella cella i, j.

Tutte le celle i, j che hanno valore $A[i][j] = 0$ hanno una dimensione massima pari a 0. Per le celle pari a 1, se si trovano sui bordi alto ($i = 1$) o sinistro ($j = 1$) hanno un valore pari a 1. Altrimenti, per fare un quadrato di dimensione k a partire da i, j è necessario che

esistano quadrati di dimensione almeno $k-1$ nelle celle sopra $(i-1,j)$, sopra a sinistra $(i-1,j-1)$ e a sinistra $(i,j-1)$; prendiamo quindi il valore minimo fra queste celle e sommiamo 1.

$$DP[i][j] = \begin{cases} 0 & A[i][j] = 0 \\ 1 & A[i][j] = 1 \wedge (i = 1 \vee j = 1) \\ \min\{DP[i-1][j], & \\ \quad DP[i-1][j-1], & \\ \quad DP[i][j-1]\} + 1 & \text{altrimenti} \end{cases}$$

È possibile calcolare DP con due cicli annidati; il risultato è dato quindi dal valore massimo memorizzato in DP.

```
int maxSquare(boolean[][] A, int n)
    int[][] DP = new int[1...n][1...n]
    for i = 1 to n do
        DP[i][1] = A[i][1]
        DP[1][i] = A[1][i]
    for i = 2 to n do
        for j = 2 to n do
            if A[i][j] == 0 then
                DP[i][j] = 0
            else
                DP[i][j] = min(DP[i-1][j], DP[i-1][j-1], DP[i][j-1]) + 1
    return max(A, n)                    % Return max value in matrix, Θ(n^2)
```

La complessità di questo algoritmo è ovviamente $\Theta(n^2)$.
L'esecuzione sulla matrice di esempio del compito dà origine alla seguente matrice M, in cui si possono notare due celle da cui può avere origine un quadrato 4×4:

```
1 0 1 0 1 0 0
1 0 1 1 1 1 0
0 1 1 2 2 2 0
0 0 1 2 3 3 0
1 1 1 2 3 4 0
1 2 2 2 3 4 0
```

Esercizio 4

Per quanto riguarda le foglie, esse hanno una lunghezza di sequenza pari ad 1. Nei nodi interni, bisogna verificare se il valore contenuto nei figli è maggiore di quello contenuto nel nodo stesso (sequenza crescente). Se questo avviene, prendiamo la sequenza più lunga e sommiamo 1.

```
int longestSequence(TREE t)
    int max = 1
    if t.left ≠ nil and t.left.key > t.key then
        max = max(max, longestSequence(t.left) + 1)
    if t.right ≠ nil and t.right.key > t.key then
        max = max(max, longestSequence(t.right) + 1)
    return max
```

Il costo è quella di una visita, $O(n)$.