Algoritmi e Strutture Dati - 31/10/14

Esercizio 1

Visto che non è richiesto di utilizzare particolari metodi, anche i teoremi dell'esperto sono utilizzabili.

1. $T(n) = T(2n/3) + 2n - 4$

   Poichè $2n - 4 \leq 2n = \Omega(n^{\log_{3/2} 1 + \epsilon})$, per tutti gli ϵ compresi fra 0 e $1 - \log_{3/2} 1 = 1$ (esclusi), possiamo applicare il caso (3) e affermare che $T(n) = \Theta(n)$, a condizione che: $\exists c < 1 : af(n/b) \leq cf(n)$. Ovvero:

   $$2 \cdot 2n/3 \leq c \cdot 2n$$

   condizione che è vera per $c \geq 2/3$.

2. $T(n) = 4T(n/2) + n^2\sqrt{n}$

   Poichè $n^2\sqrt{n} = \Omega(n^{2+\epsilon})$, per $0 < \epsilon < 1/2$, possiamo applicare il caso (3) e affermare che $T(n) = \Theta(n^2\sqrt{n})$, a condizione che: $\exists c < 1 : af(n/b) \leq cf(n)$. Ovvero:

   $$4 \cdot \frac{n^2}{4} \frac{\sqrt{n}}{\sqrt{2}} \leq cn^2\sqrt{n}$$

   condizione che è vera per $c \geq \frac{1}{\sqrt{2}}$, un valore che è minore di 1.

3. $T(n) = 2T(n/4) + \sqrt{n} + 10 \log n$

   Poiché $\sqrt{n} + 10 \log n = \Theta(n^{\log_4 2}) = \Theta(n^{\frac{1}{2}}) = \Theta(\sqrt{n})$, possiamo applicare il caso (2) del teorema dell'esperto e la complessità è $\Theta(\sqrt{n} \log n)$.

4. $T(n) = 3T(n/2) + 2n \log n + 10n$

   Poichè $2n \log n + 10n = O(n^{\log_2 3 - \epsilon})$ per tutti gli ϵ compresi fra 0 (escluso) e $1 - \log_2 3$, possiamo applicare il caso (1) del teorema dell'esperto e la complessità è $\Theta(n^{\log_2 3})$.

5. $T(n) = T(n - 6) + n^{5/6}$

   Si applica il teorema delle Ricorrenze lineari di ordine costante, e si ottiene che $T(n) = \Theta(n^{1+5/6}) = \Theta(n^{11/6})$.

L'algoritmo migliore è il terzo, con complessità $\Theta(\sqrt{n} \log n)$.

Esercizio 2

L'algoritmo proposto ordina gli ultimi $n^{4/5}$ elementi del vettore utilizzando MergeSort(), con costo $\Theta(n^{4/5} \log n^{4/5})$ Poi utilizza la funzione Merge() per ordinare gli elementi del vettore già ordinato e di quello appena ordinato, con costo $\Theta(n)$. Il costo finale è $\Theta(n)$ in quanto $\Theta(n^{4/5} \log n^{4/5})$ ha un costo sublineare.

```
SquareSort(integer[] V, integer n)
    MergeSort(V, n − ⌊n^{4/5}⌋ + 1, n)
    Merge(V, 1, n − ⌊n^{4/5}⌋, n)
```

Esercizio 3

(1) Un grafo orientato debolmente connesso è un grafo in cui esiste un cammino non orientato fra ogni coppia di nodi. In altre parole, è sufficiente costruire un grafo non orientato a partire dal grafo orientato, ed eseguire l'algoritmo che verifica se il grafo è connesso.
E' sufficiente rendere la matrice simmetrica, facendo in modo che se esiste l'arco (u, v), allora esista anche l'arco (v, u). Il costo di tale operazione è $O(n^2)$. La versione presentata qui modifica direttamente il grafo originale.

```
undirected(GRAPH G)
    foreach u ∈ G.V() do
        foreach v ∈ G.adj(u) do
            G.insertEdge(v, u)
```

A questo punto, è sufficiente calcolare le componenti connesse del grafo e verificare che ne sia stata trovata al massimo una. Il costo è ancora $O(n^2)$ perchè identificare le componenti connesse costa quanto una visita in profondità.

```
weaklyConnected(GRAPH G)
    undirected(G)
    integer[] id ← cc(G)
    foreach u ∈ G.V() do
        if id[u] > 1 then return false
    return true
```

(2) Per quanto riguarda i grafi singolarmente connessi, il grafo originale G è singolarmente connesso se è debolmente connesso e non esistono cicli nel grafo non orientato ottenuto da G; in altre parole, se è un albero non radicato! Quindi, calcoliamo ancora una volta il grafo connesso, verifichiamo che sia debolmente connesso e infine verifichiamo se esistono cicli.

```
singularlyConnected(GRAPH G)
    undirected(G)
    return weaklyConnected(G) and not ciclico(G, 1)
```

Esercizio 4

Il problema è risolvibile con un algoritmo di complessità $\Theta(n^3)$, semplicemente considerando tutti i sottovettori non vuoti possibili ($n(n+1)/2$), calcolando il minimo e massimo in essi (utilizzando una funzione di costo lineare) e quindi identificando il sottovettore più lungo fra quelli il cui spessore è inferiore o uguale a C.

```
spessore(integer[] V, integer n, integer C)
    integer maxlen ← 0
    for i ← 1 to n do
        for j ← i to n do
            integer min ← min(V, i, j)
            integer max ← max(V, i, j)
            if max − min ≤ C then
                maxlen ← max(maxlen, j − i + 1)
    return maxlen
```

A questo punto, si può notare in maniera simile a quanto fatto con l'algoritmo maxsum visto il primo giorno di lezione, che è inutile calcolare ripetutamente il massimo e il minimo in sottovettori crescenti. E' sufficiente calcolare aggiornare due variabili *min* e *max* rispetto al minimo/massimo calcolato in precedenza. Il costo è quindi $\Theta(n^2)$.

```
spessore(integer[] V, integer n, integer C)
    integer maxlen ← 0
    for i ← 1 to n do
        integer min ← +∞
        integer max ← −∞
        integer j ← i
        while j ≤ n and max − min ≤ C do
            min ← min(min, A[j])
            max ← max(max, A[j])
            if max − min ≤ C then
                maxlen ← max(maxlen, j − i + 1)
                j ← j + 1
    return maxlen
```

E' possibile usare un approccio divide-et-impera. Dato un vettore $V[i \dots j]$, si calcola $m = \frac{i+j}{2}$ e si divide il vettore in due parti: $V[i \dots m]$ e $V[m+1 \dots j]$. Si richiama l'algoritmo sulle due metà, ottenendo la lunghezza dei più grandi sottovettori contenuti nelle due metà, di spessore al più C. A questo punto, si deve cercare il più grande sottovettore contenuto in $V[i \dots j]$ di spessore inferiore a C che inizia nella prima metà e finisce nella seconda metà. Si noti che $V[m]$ e $V[m+1]$ devono appartenere a tale vettore. Possiamo quindi utilizzare due sottovettori $mins$ e $maxs$, così definiti:

$$mins[k] = \begin{cases} \mathsf{min}(V, k, m) & i \le k \le m \\ \mathsf{min}(V, m+1, k) & m+1 \le k \le j \end{cases}$$

$$maxs[k] = \begin{cases} \mathsf{max}(V, k, m) & i \le k \le m \\ \mathsf{max}(V, m+1, k) & m+1 \le k \le j \end{cases}$$

ovvero $mins[k]$ ($maxs[k]$) contiene il più piccolo (più grande) valore che si incontra tra gli indici i ed m (nel sottovettore di sinistra) e tra $m+1$ e j (nel sottovettore di destra).
Una volta calcolato $mins$ e $maxs$ (cosa possibile in tempo lineare), è possibile analizzare il sottovettore dagli indici $start = i$ fino all'indice $stop = m+1$. Se il sottovettore ha spessore al più C, si aggiorna se possibile la lunghezza massima e si cerca di espanderlo incrementando $stop$; altrimenti, si riduce la sua ampiezza incrementando $start$. Si termina quando l'indice $start$ supera m (cosa non possibile in quanto il sottovettore deve contenere m) o quando $stop$ supera j (ovvero siamo fuori dal sottovettore considerato). Poichè ad ogni iterazione del ciclo si incrementa $start$ o $stop$, il costo di questa operazione è anch'esso lineare.
Il costo è pari a:

$$T(n) = \begin{cases} 1 & n \le 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

e quindi pari a $\Theta(n \log n)$.
Nel codice seguente, assumiamo che i vettori di appoggio siano dichiarati globalmente; altrimenti, è possibile passarli in input. La chiamata iniziale è $\mathsf{spessore}(V, 1, n, C)$.

```
spessore(integer[] V, integer i, integer j, integer C)
if i = j then
    return 1
integer m ← (i + j)/2
integer maxlen ← max(spessore(V, i, m, C), spessore(V, m + 1, j, C))
mins[m] ← maxs[m] = V[m]
for k ← m − 1 downto i do
    mins[k] ← min(mins[k + 1], V[k])
    maxs[k] ← max(maxs[k + 1], V[k])
mins[m + 1] ← maxs[m + 1] = V[m + 1]
for k ← m + 2 to j do
    mins[k] ← min(mins[k − 1], V[k])
    maxs[k] ← max(maxs[k − 1], V[k])
integer start ← i
integer stop ← m + 1
while start ≤ m and stop ≤ j do
    if max(maxs[start], maxs[stop]) − min(mins[start], mins[stop]) ≤ C then
        maxlen ← max(maxlen, stop − start + 1)
        start ← start + 1
    else
        stop ← stop + 1
return maxlen
```