Algoritmi e Strutture Dati - 07/01/16

Esercizio 1

- **MergeSort**: l'algoritmo di MergeSort ha costo $\Theta(n \log n)$, indipendentemente dai valori presenti in input;
- **QuickSort**: l'algoritmo ricorsivo seleziona il primo elemento del sottovettore $V[i \dots j]$ considerato; tale elemento $V[i]$ è sicuramente più piccolo di tutti gli elementi $V[i+5 \dots j]$; nel caso "ottimo", gli elementi $V[i+1 \dots i+4]$ sono più piccoli di $V[i]$. L'equazione di ricorrenza, in tale caso ottimo, è quindi pari a:

$$T(n) = \begin{cases} T(n-5) + O(n) & n > 5 \\ c & n \leq 5 \end{cases}$$

  Tale equazione ha soluzione $T(n) = \Theta(n^2)$; quindi l'algoritmo ha costo $\Theta(n^2)$ nel caso ottimo e pessimo, e conseguentemente anche medio.
- **InsertionSort**: poichè ogni valore $V[i]$ può essere più piccolo dei soli 5 valori che lo precedono, il ciclo interno di Insertion Sort ha complessità costante e quindi la complessità è $\Theta(n)$.

Esercizio 2

E' possibile calcolare la distanza utilizzando una semplice visita in profondità, ma bisogna fare attenzione a non seguire più volte il percorso a partire dalla radice (venendo da una foglia), in quanto può portare il costo ad essere $O(n^2)$. Scriviamo quindi una visita DFS che a partire da un nodo x, cerca un nodo t, senza mai passare per un nodo r; restituendo la distanza minima da x a t, se t è stato trovato; $+\infty$ altrimenti.

```
int dfsSearch(GRAPH g, NODE x, NODE t, NODE r)
    if x = t then
        return 0
    min ← ∞
    foreach y ∈ G.adj(x) do
        if y ≠ r then
            int d ← dfsSearch(G, y, t, r)
            if d + w(x, y) < min then
                min ← d + w(x, y)
    return min
```

Nel caso v sia nel sottoalbero di u, la sua distanza può essere calcolata tramite la procedura dfsSearch(), evitando di passare per il nodo r; poichè esiste un solo cammino da u a v, e la DFS ha termine perchè si blocca nelle foglie, il costo è pari a $O(n+m)$.
Se così non è, dfsSearch() ritorna $+\infty$ e la visita si compone di due parti: il costo minimo (e unico) per andare dalla radice a v (senza mai passare da r), e il costo minimo per andare da u alla radice (senza vincoli di nodi da cui non passare). Queste due visite hanno ancora costo $O(m+n)$.

```
computeDist(GRAPH g, NODE r, NODE u, NODE v)
    d ← dfsSearch(G, u, v, r)
    if d < ∞ then
        return d
    else
        return dfsSearch(G, u, r, −1) + dfsSearch(G, r, v, r)
```

In realtà, poichè ogni nodo ha al massimo 2 archi uscenti (può avere un figlio sinistro e destro; solo un figlio sinistro o solo un figlio destro; oppure un collegamento alla radice, essendo foglia), se ne deduce che $m < 2n$ e la complessità dell'algoritmo è $O(n)$.

Esercizio 3

E' possibile utilizzare un algoritmo greedy, ordinando gli intervalli per estremo di inizio crescente; a parità di inizio, per estremo di fine decrescente. Si inserisce quindi il primo intervallo. Da questo punto in poi, si scartano tutti gli intervalli che sono totalmente contenuti

nell'ultimo intervallo inserito, per poi inserire il primo intervallo successivo che il cui estremo di fine è superiore all'estremo di fine dell'ultimo intervallo inserito.
Il costo della procedura così delineata è $\Theta(n \log n)$, derivante dall'operazione di ordinamento. La soluzione è rappresentata da un sottoinsieme di $\{1, \ldots, n\}$.

```
SET cover(int[] a, int[] b, int n)
    sort(a, b, n)                          % Ordina per estremo di inizio crescente, estremo di fine decrescente
    SET S ← Set()
    S.insert(1)
    int ultimo ← 1                         % Ultimo intervallo inserito
    for i ← 2 to n do
        if b[i] > b[ultimo] then           % Controllo indipendenza
            S.insert(i)
            ultimo ← i
    return S
```

Scritto in questo modo, tuttavia, l'algoritmo non è corretto; fallisce proprio per l'esempio $[2, 4], [3, 6], [4, 7]$, in quanto inserisce tutti e tre gli intervalli quando invece sono sufficienti il primo e l'ultimo. Il problema sta nel fatto che dopo aver inserito $[2, 4]$ e $[3, 6]$, al momento di valutare $[4, 7]$ bisogna ri-valutare $[3, 6]$ ed eventualmente rimuoverlo, in quanto contenuto in $[2, 4] + [4, 7]$. In ogni caso, a fini della valutazione, questo algoritmo è stato considerato corretto, vista la difficoltà del problema.

Una soluzione corretta è la seguente. Invece di un insieme S, manteniamo un vettore S che contiene gli ultimi elementi aggiunti. Ogni volta che si sta per inserire un intervallo (indice i), lo si confronta con il penultimo inserito (indice $S[j-1]$); se il penultimo intervallo inserito e il nuovo intervallo sono in contatto ($a[i] \leq b[S[j-1]]$), si elimina l'ultimo intervallo inserito ($S[j]$), facendo arretrare j; si continua così fino a quando si raggiunge il primo intervallo oppure si trovano due intervalli non in contatto. A quel punto, viene inserito il nuovo intervallo.

```
SET cover(int[] a, int[] b, int n)
    sort(a, b, n)                          % Ordina per estremo di inizio crescente, estremo di fine decrescente
    int[] S = new int[1 ... n]
    S[1] ← 1
    int j ← 1                              % Ultimo intervallo inserito
    for i ← 2 to n do
        if b[i] > b[S[j]] then             % Controllo indipendenza

            while j > 1 and a[i] ≤ b[S[j − 1]] do
                j ← j − 1
            j ← j + 1
            S[j] ← i
    return {S[k] | 1 ≤ k ≤ j}
```

La complessità resta $O(n \log n)$, dovuto all'ordinamento, perchè nonostante i due cicli, ogni intervallo viene inserito e rimosso al massimo una volta, e quindi il corpo principale dell'algoritmo (escluso l'ordinamento) ha costo $O(n)$.
Questa tecnica è chiamata backtrack iterativo e segue questa filosofia: "prova ad aggiungere un'intervallo (perchè coerente con quelli precedenti), ma se un futuro inserimento renderà questo intervallo inutile, lo si rimuoverà (una e una sola volta)".
Per dimostrare la correttezza, dimostriamo il seguente invariante: all'inizio del ciclo i-esimo, la soluzione contenuta in $S[1 \ldots j]$ è ottima per il sottoinsieme dato dai primi $i - 1$ intervalli.
Quando $i = 2$, ovvero prima dell'inizio del ciclo, $S[1 \ldots 1]$ contiene il primo (e unico) intervallo; è quindi una soluzione corretta.
Assumendo che la proprietà sia vera fino al ciclo $i - 1$-esimo, dimostriamo che sia vera all'inizio del ciclo i-esimo.

- Se l'intervallo i ha un estremo superiore più basso o uguale dell'intervallo $S[j]$, non deve essere inserito (in quanto $a[S[j]] \leq a[i]$ – per l'ordinamento degli intervalli – e $b[i] \leq b[S[j]]$ – per ipotesi, ovvero l'intervallo i è totalmente contenuto nell'intervallo $S[j]$).
- Altrimenti, se $b[i] > b[S[j]]$, è necessario che sia inserito, altrimenti il valore $b[i]$ non sarebbe coperto. Ma una volta inserito, può essere che $\exists k < j$ tale che $\bigcup_{k+1}^{j} I[S[k]] \subseteq I_S[k] \cup I_i$; in tal caso, tutti questi intervalli vanno rimossi.
- Infine, al termine dell'esecuzione, gli intervalli contenuti in $S[1 \ldots j]$ sono una soluzione ottima per tutti gli n intervalli.

Esercizio 4

Sia $D[i, j]$ il costo minimo per dividere la stringa dal punto i-esimo al punto j-esimo; si estenda il vettore iniziale con $V[0] = 0$ e $V[n + 1] = L$, utilizzati come sentinelle. Il problema iniziale corrisponde a $V[0, n + 1]$.
Una possibile formulazione ricorsiva:

$$D[i,j] = \begin{cases} 0 & j = i+1 \\ \min_{i<k<j}\{D[i,k] + D[k,j]\} + (V[j] - V[i]) & j > i+1 \end{cases}$$

Quando $j = i + 1$, non c'è nessun taglio intermedio da fare; quindi il costo è pari a zero. Altrimenti, si considerano tutti i possibili tagli intermedi, si ottengono due sottostringhe da trattare separatamente e sommare, e si considera il valore minimo fra tutti i tagli intermedi. Si somma quindi la lunghezza della stringa intera, che è il costo del taglio.
Utilizzando memoization, si ottiene un algoritmo di costo n^3.

```
cutString(int[] V, int n, int L)
    int[][] D ← new int[0...n+1][0...n+1]
    for i = 0 to n + 1 do
        for j = 1 to n + 1 do
            D[i, j] ← ⊥
    return cutStringRec(V, L, D, 0, n + 1)
```

```
cutStringRec(int[] V, int L, int[][] D, int i, int j)
    if j = i + 1 then
        return 0
    if D[i, j] = ⊥ then
        D[i, j] ← +∞
        for k = i + 1 to j − 1 do
            cost ← cutStringRec(V, L, D, i, k) + cutStringRec(V, L, D, k, j) + (V[j] − V[i])
            if cost < D[i, j] then
                D[i, j] ← cost
    return D[i, j]
```