Algoritmi e Strutture Dati - 14/07/15

Esercizio 1

Utilizzando il master theorem, è facile vedere che $T(n) = \Theta(\sqrt[3]{n} \log n)$.
Dimostriamo quindi per sostituzione che $T(n) = \Omega(\sqrt[3]{n} \log n)$. Vogliamo dimostrare che

$$\exists c > 0, \exists m \geq 0 : T(n) \geq c\sqrt[3]{n} \log n, \forall n \geq m$$

- Il caso base è facilmente dimostrato:
$$T(n) = 1 \geq c\sqrt[3]{1} \log 1 = 0$$
che è vera per qualsiasi valore di c.
- Ipotesi induttiva: $\forall n' < n : T(n') \geq \sqrt[3]{n'} \log n'$
- Passo induttivo:
$$\begin{aligned} T(n) &\geq 2c\sqrt[3]{n/8} \log n/8 + \sqrt[3]{n} \\ &= c\sqrt[3]{n} \log n/8 + \sqrt[3]{n} \\ &= c\sqrt[3]{n}(\log n - \log 8) + \sqrt[3]{n} \\ &= \sqrt[3]{n}(c \log n - 3c + 1) \geq c\sqrt[3]{n} \log n \end{aligned}$$
L'ultima disequazione è vera per $1 - 3c \geq 0$, ovvero se $c \leq 1/3$. Abbiamo quindi dimostrato che il limite inferiore è $T(n) = \Omega(\sqrt[3]{n} \log n)$.

Esercizio 2

È sufficiente utilizzare due ricerche dei cammini minimi, a partire dai nodi Alice e Bob, in un grafo in cui i pesi sugli archi sono stati modificati in modo tale che tutti gli archi entranti nel nodo Carl abbiano peso infinito. In questo modo, il nodo Carl viene disconesso dal grafo.
Una volta ottenuti i pesi dei cammini minimi, è sufficiente cercare il nodo con peso inferiore. La procedura camminiMinimi() implementa l'algoritmo di Johnson di complessità $O(m \log n)$ e restituisce il vettore delle distanze

---

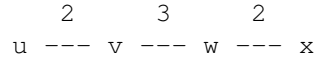
int secretPlace(GRAPH G, **int**[][] w), NODE *Alice*, NODE *Bob*, NODE *Carl*)

```
foreach u ∈ G.V() do
    w[u, Carl] = +∞
int[] d_a = camminiMinimi(G, Alice)
int[] d_b = camminiMinimi(G, Bob)
int min = +∞
foreach u ∈ G.V() do
    if d_a[u] + d_b[u] < min then
        min = d_a[u] + d_b[u]
return min
```

La complessità è dominata da $O(m \log n)$, in quanto gli altri passi hanno costo lineare nel numero di nodi.

Esercizio 3

- Si consideri il grafo seguente:



L'algoritmo greedy seleziona per primo l'arco (v, w), e si trova poi quindi nell'impossibilità di selezionare gli altri archi, perchè coinvolgono i nodi u, v. La soluzione proposta è quindi $\{(v, w)\}$ di peso totale 3. Tuttavia, la soluzione $\{(u, v), (w, x\}$ ha peso totale 4, quindi la risposta data dall'algoritmo greedy non è corretta.

- Per risolverlo, è necessario utilizzare un approccio di tipo backtrack, la cui complessità è $O(2^m)$, dove m è il numero degli archi. Partendo da un nodo qualunque u e utilizzando un vettore *used* inizializzato a **false**, valutiamo se prendere (nel caso entrambi gli estremi abbiano **false** nel vettore *used*) o non prendere uno degli archi (u, v) uscenti da u.

```
int univoco(int u, int[] used)
    int max = 0
    foreach v ∈ u.adj() do
        { Se non è già preso }
        if not used[u] and not used[v] then
            used[u] = used[v] = true
            max = max(max, w(u, v) + univoco(v, used))
            used[u] = used[v] = false
        max = max(max, w(u, v) + univoco(v, used))
    return max
```

Esercizio 4

Utilizzando la programmazione dinamica, e in particolare la tecnica di memoization, è possibile definire il problema nel modo ricorsivo seguente: $DP[m][s][k]$ è un valore booleano che indica se è possibile dividere i primi k giocatori (con $k \leq 2n$) in modo tale che la prima squadra abbia s giocatori e una bravura totale m, mentre la seconda abbia $k - s$ giocatori e una bravura totale $(\sum_{i=1}^{k} b[i]) - m$. Il valore di $DP[m][s][k]$ può essere calcolato nel modo seguente:

$$DP[m][s][k] = DP[m - b[k]][s - 1][k - 1] \vee DP[m][s][k - 1]$$

dove il primo caso rappresenta il sottoproblema generato dall'assegnemento del giocatore k alla prima squadra (restano da trovare $s - 1$ giocatori con bravura totale $m - b_k$, scegliendo tra i primi $k - 1$ giocatori; il secondo caso rappresenta il sottoproblema generato dall'assegnamento del giocatore k alla seconda squadra.
I casi base della ricorsione sono i seguenti:

- $DP[0][0][k] = \mathbf{true}$ (in altre parole, è sempre possibile assegnare tutti i giocatori alla seconda squadra);
- $DP[m][s][k] = \mathbf{false}$, per $m < 0$ o $s < 0$; ;
- $DP[m][s][0] = \mathbf{false}$, per $m > 0$ o $s > 0$; in altre parole, è impossibile ottenere una prima squadra di valore m positivo o con s giocatori, con 0 giocatori disponibili;

Il problema originale è $DP[M/2][n][2n]$, dove $M = \sum_{i=1}^{2n} b[i]$.
Utilizzando memoization, il programma assume la forma seguente:

```
soccer(int[] b, int n)
    int M = Σ_{i=1}^{2n} b[i]
    int[][][] DP = new int[1...M/2, 1...n, 1...2n]P = {nil}        % Initialized to nil
    return soccerRc(b, C, M/2, n, 2n, DP)
```

```
soccerRc(int[] b, int [ ][ ][ ] C, int m, int s, int k, int[][][] DP)
    if s == 0 and m == 0 then
        return true
    if s < 0 or m < 0 then
        return false
    if k == 0 then
        return false
    if C[m][s][k] == nil then
        C[m][s][k] = soccerRec(b, C, m, s, k − 1, DP) or soccerRec(b, C, m − b[k], s − 1, k − 1, DP)
    return DP[m][s][k]
```

La complessità è pari a $O(Mn^2)$