Algoritmi e Strutture Dati - 19/12/14

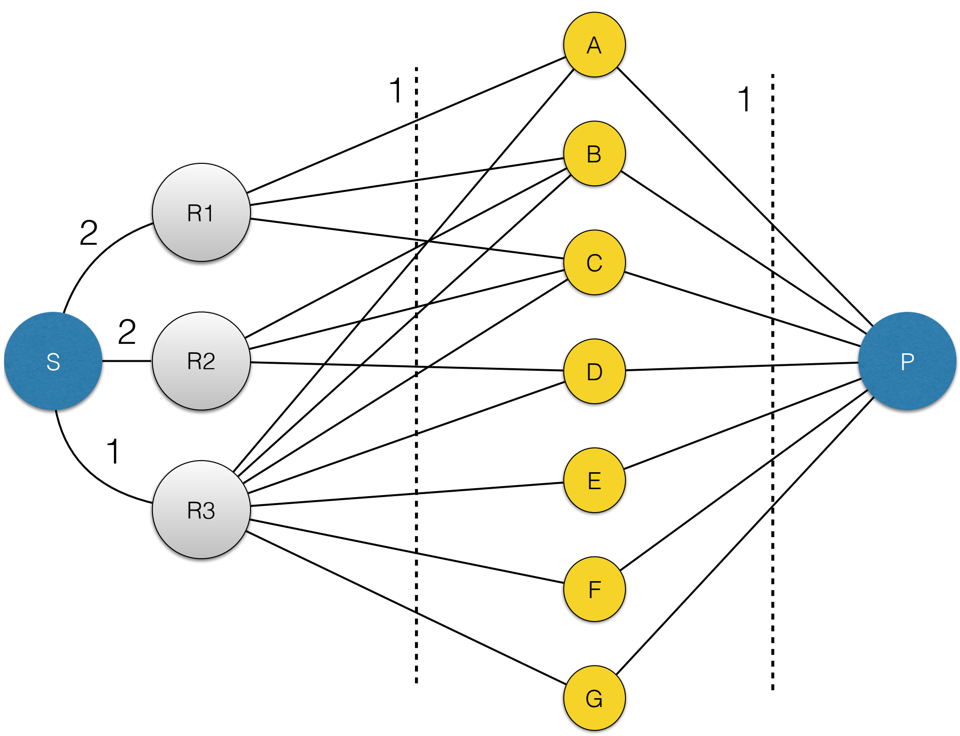
Esercizio 1

Assumiamo che il vettore sia ordinato. Se così non è, è sufficiente ordinarlo in tempo $O(n \log n)$.
Vogliamo dimostrare che un intervallo l'intervallo $[V[1], V[1]+1]$ (l'intervallo unitario che inizia in $V[1]$) fa sempre parte di una soluzione ottima. Si prenda una soluzione ottima S e si consideri l'intervallo $[x, x+1] \in S$ che "ricopre" $V[1]$, ovvero tale per cui $x \leq V[1] \leq x+1$; tale intervallo deve esistere, in quanto $V[1]$ deve essere ricoperto. Poichè $V[1] \geq x$ e $V[1]$ è il primo punto del vettore, è possibile ottenere una soluzione $C - \{[x, x+1]\} \cup \{[V[1], V[1]+1]\}$ che ha la stessa dimensione di C, ricopre $V[1]$ e tutti i punti precedentementi ricoperti da $[x, x+1]$.
L'algoritmo è quindi il seguente:

```
SET covering(real[] V, int n)
    sort(V, n)                                  % O(n log n)
    SET S = {1}
    int last = 1
    for i = 2 to n do
        if V[i] > V[last] + 1 then
            S.append(i)
            last = i
    return S
```

Esercizio 2

È possibile utilizzare l'algoritmo per identificare il flusso massimo, inserendo un nodo per ogni requisito, un nodo per ogni corso, una supersorgente e un superpozzo. Il superpozzo è collegato al requisito i-esimo con capacità m_i; i requisiti sono collegati ai corsi con archi di peso 1; i corsi sono collegati al superpozzo con archi di peso 1. Il regolamento è soddisfacibile se è possibile trovare un flusso di valore t; si noti che condizione necessaria (ma non sufficiente) perchè questo avvenga è che $t = \sum_{i=1}^{k} m_i$. Il grafo risultante per l'esempio del compito è riportato di seguito.



Il numero di nodi è $|V| = n + k + 2$; il numero di archi è limitato superiormente da $|E| = O(k + nk + n)$. La complessità è quindi pari a:

$$O(t \cdot [(n + k + 2) + (n + k + nk)]) = O(tnk)$$

Esercizio 3

Definiamo la matrice DP, dove $DP[i][c]$ contiene la lunghezza della più lunga sottosequenza ordinata-distinta contenuta nella prefisso $S(i)$ (ovvero i primi i caratteri di S) composta dai primi c caratteri dell'alfabeto. Il problema originale corrisponde quindi a $DP[n][26]$. È possibile calcolare DP nel modo ricorsivo seguente:

$$DP[i][c] = \begin{cases} 0 & i = 0 \\ DP[i-1][c] & i > 0 \land S[i] > c \\ \max\{DP[i-1][c], DP[i-1][S[i]-1]+1\} & i > 0 \land S[i] \leq c \end{cases}$$

- Se stiamo considerando un prefisso di 0 caratteri, la sottosequenza ha lunghezza nulla.
- Se il carattere i-esimo non rientra nei primi c caratteri dell'alfabeto, lo scartiamo considerando il problema con $i-1$ caratteri.
- Se il carattere i-esimo rientra nei primi c caratteri dell'alfabeto, abbiamo due possibilità: o lo scartiamo, e allora consideriamo il problema con $i-1$ caratteri e lo stesso alfabeto; o lo prendiamo, nel qual caso dobbiamo comunque considerare $i-1$ caratteri, eliminando tuttavia $S[i]$ e tutti i caratteri seguenti dall'alfabeto.

Questa definizione si traduce in questo algoritmo basato su memoization:

```
int maxOD(int[] S, int n)
    int[][] DP = new int[0...n, 0...26] = { -1 }          % Inizializzato a −1
    maxRec(S, n, 26, DP)
    return DP[n][c]
```

```
int maxRec(int[] S, int i, int c, int[][] DP)
    if i == 0 then
        return 0
    if DP[i][c] < 0 then
        if S[i] > c then
            DP[i][c] = maxRec(S, i − 1, c, DP)
        else
            DP[i][c] = max(maxRec(S, i − 1, c, DP), maxRec(S, i − 1, S[i] − 1, DP) + 1)
    return DP[i][c]
```

Tuttavia, il modo più semplice per risolvere questo problema è rendersi conto che è sufficiente utilizzare l'algoritmo LCS, chiamato passando questa stringa e la stringa ABCDEFGHIJKLMNOPQRSTUVWXYZ. Il costo di questa soluzione è pari a $\Theta(n)$, in quanto deve essere riempita una matrice $n \times 26$.

```
int maxOrdinataDistinta(ITEM[] S, int n)
    int[][] DP = new int[0...n, 0...26]
    lcs(M, S, "ABCDEFGHIJKLMNOPQRSTUVWXYZ", n, 26)
    return DP[m, n]
```

Esercizio 4

Questo esercizio si risolve in maniera simile agli algoritmi di confronto stringhe che abbiamo visto a lezione. Sia $DP[i][j]$ il numero di caratteri dash necessari per allineare le stringhe prefisso $P(i)$ (i primi i caratteri di P) e $T(j)$ (i primi j caratteri di T). $DP[i][j]$ può essere calcolata in modo ricorsivo nel modo seguente:

$$DP[i][j] = \begin{cases} j & i = 0, \text{oppure} \\ i & j = 0, \text{oppure} \\ DP[i-1][j-1] & P(i) = T(j) \\ \mathit{min}\{DP[i-1,j], DP[i,j-1]\} + 1 & \text{altrimenti} \end{cases}$$

- Se una delle stringhe è vuota, bisognerà inserire un carattere dash per oguno dei caratteri dell'altra string;

- Se gli ultimi due caratteri sono uguali, possono essere allineati senza dover inserire caratteri dash;
- Altrimenti, se gli ultimi due caratteri sono diversi, prendiamo il minimo di due casi: il caso in cui l'ultimo carattere di $P(i)$ deve essere allineato con un dash, oppure il caso in cui l'ultimo carattere di $T(j)$ deve essere allineato con un dash. In entrambi i casi, bisogna aggiungere $+1$ per tener conto del carattere aggiunto.

La formula ricorsiva può essere risolta tramite memoization; qui presento invece una versione basata su programmazione dinamica.

```
int minAllineamento(ITEM[] P, ITEM[] S, int n, int m)
    int[][] DP = newint[0...n][0...m]
    for i = 0 to n do
        DP[i][0] = i
    for j = 0 to m do
        DP[j][0] = i
    for i = 1 to n do
        for j = 1 to m do
            if P[i] = T[j] then
                DP[i][j] = DP[i-1][j-1]
            else
                DP[i][j] = min(DP[i-1][j], DP[i][j-1]) + 1
```

La complessità dell'algoritmo risultante è $O(mn)$.