Algoritmi e Strutture Dati - 01/09/15

Esercizio 1

È possibile utilizzare una struttura dati di tipo red-black tree come dizionario per memorizzare, per ognuno dei valori distinti presenti, il numero di occorrenze. È poi sufficiente scorrere l'albero binario di ricerca utilizzando un iteratore ordinato (operazione possibile grazie alle funzioni minimo e successore presenti in un albero ABR), in modo da ottenere i valori in ordine e quindi scrivere ripetutamente il valore, tante volte quante sono le sue ripetizioni. Il costo computazionale è pari a $O(n \log m)$, in quanto è necessario cercare gli n valori in un albero che contiene al più m elementi.

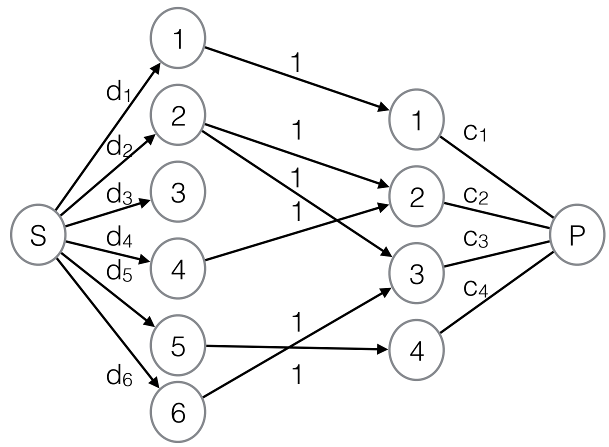
```
repeatSort(int[] A, int n)
    DICTIONARY D = new RedBlackTree()
    for i = 1 to n do
        ITEM v = D.lookup(A[i])
        if v == nil then
            v = 0
        D.insert(A[i], v + 1)
    int j = 1
    for v ∈ D.keys() do                          % Itera sulle chiave in ordine crescente
        for k = 1 to D.lookup(v) do
            A[j] = v
            j = j + 1
```

Una versione più veloce utilizza una tabella hash; tuttavia, la tabella hash non garantisce l'ordinamento; è quindi necessario copiare tutte le chiavi (non ordinate) in un vettore e quindi ordinare tale vettore. Il costo computazionale è quindi $O(n + m \log m)$, dove $O(n)$ deriva dall'inserimento di tutti gli elementi nella tabella hash, mentre $O(m \log m)$ deriva dall'ordinamento di tali valori.

```
repeatSort(int[] A, int n)
    DICTIONARY D = new HashTable()
    for i = 1 to n do
        ITEM v = D.lookup(A[i])
        if v == nil then
            v = 0
        D.insert(A[i], v + 1)
    int m = D.size()
    int B = new int[1 ... m] int j = 1
    for v ∈ D.keys() do                          % Itera sulle chiave in ordine crescente
        B[j] = v
        j = j + 1
    int k
    for j = 1 to m do
        for k = 1 to D.lookup(B[j]) do
            A[k] = B[j]
            k = k + 1
```

Esercizio 2

Il problema è risolvibile con una rete di flusso come quella mostrata in figura.



Definiamo:

- $D = \sum_{i=1}^{n} d_i$, il numero totale di partecipanti
- $W = \sum_{i=1}^{n} |w_i|$, il numero totale di richieste
- $C = \sum_{j=1}^{m} c_j$, la capacità totale dei workshop

L'insieme dei nodi è così costituito:

- una supersorgente,
- un nodo per ognuna delle società,
- un nodo per ognuno dei workshop;
- un superpozzo,

per un totale di $|V| = m + n + 2$ nodi.
L'insieme degli archi è così costituito:

- un arco da supersorgente ad ogni società i, con capacità pari a d_i (il numero di dipendenti mandati alla conferenza),
- un arco da ogni società i all'insieme dei workshop contenuti in W_i, con peso 1,
- un arco da ogni workshop j al superpozzo, con capacità pari al numero massimo di partecipanti c_j,

per un totale di $|E| = m + n + W$ archi.
Il flusso massimo è limitato superiormente da $\min(D, W, C)$ (per via del teorema massimo flusso, minimo taglio). Il costo totale è quindi limitato superiormente da $\min(D, W, C)(2m + 2n + W + 2)$.

Esercizio 3

La dimostrazione è semplice: il valore massimo del vettore è necessariamente più grande dei suoi vicini, essendo questi distinti, e quindi è un picco.
Dato un sottovettore $A[i \dots j]$ con almeno tre elementi in cui è contenuto sicuramente un picco, si consideri l'elemento centrale $m = \lfloor (i + j)/2 \rfloor$. Se è un picco, abbiamo trovato la nostra soluzione.
Altrimenti, si considerino i suoi vicini.

- Se $A[m-1] > A[m] > A[m+1]$, è possibile che nel sottovettore destro $A[m+1 \dots j]$ non vi sia alcun picco, in quanto il sottovettorepotrebbe essere monotono decrescente. Il picco si troverà quindi nel sottovettore sinistro $A[i \dots m-1]$.
- Se $A[m-1] < A[m] < A[m+1]$, è possibile che nel sottovettore sinistro $A[i \dots m-1]$ non vi sia alcun picco, in quanto il sottovettore potrebbe essere monotono crescente. Il picco si troverà quindi nel sottovettore destro $A[m+1 \dots j]$.
- Se $A[m-1] > A[m] < A[m+1]$, ci troviamo in una valle, e quindi esiste sicuramente un picco sia a destra che a sinistra; è quindi sufficiente scegliere un lato.

Se il sottovettore $A[i \dots j]$ in cui è contenuto sicuramente un picco ha due elementi ($m = i$ per via delll'operazione $\lfloor\rfloor$); se questo non è un picco, allora il picco si trova nell'elemento j.
Se il sottovettore $A[i \dots j]$ in cui è contenuto sicuramente un picco ha un elemento ($m = i = j$), allora m è sicuramente un picco e non necessita di caso speciale.
È possibile scrivere il codice nel modo seguente:

```
int findPeak(int[] A, int n, int i, int j)
    int m = ⌊(i + j)/2⌋
    if (m == 1 or A[m − 1] < A[m]) and (m == n or A[m] > A[m + 1]) then
        return m
    if j − i == 1 then
        return j
    if A[m − 1] > A[m] then
        return findPeak(A, n, i, m − 1)
    else
        return findPeak(A, n, m + 1, j)
```

La complessità è $O(\log n)$, in quanto simile alla ricerca dicotomica.

Esercizio 4

Questo è un problema interessante, che è stato studiato a lungo in ambito di ricerca e per cui esistono soluzioni in tempo lineare.[1]
Per quanto riguarda questo compito, è sufficiente trovare una soluzione in tempo quadratico utilizzando la programmazione dinamica.
Sia $DP[i][j]$ la lunghezza della più lunga sottostringa palindroma contenuta in nella stringa $S[i \dots j]$. Si noti che $j - i + 1$ è la lunghezza della sottostringa, e quindi se $DP[i][j] = j - i + 1$, allora l'intera sottostringa è palindroma.
Possono darsi i seguenti casi:

- Il caso base corrisponde a sottostringhe con 0 caratteri ($j - i + 1 = 0$) oppure sottostringhe con 1 caratteri ($j - i + 1 = 1$), che ovviamente sono palindrome. In questi casi, $DP[i][j] = j - i + 1$.
- Se $s[i] = s[j]$ sono uguali, e $s[i+1\ ldots j-1]$ è palindroma (questo avviene se $DP[i+1][j-1] = (j-1)-(i+1)+1 = j-i-1$), allora la stringa $s[i \dots j]$ è palindroma, e quindi $DP[i][j] = j - i + 1$.
- Altrimenti, prendiamo la sottostringa palindroma massimale scegliendo fra $DP[i + 1][j]$ e $DP[i][j - 1]$.

Riassumendo:

$$DP[i][j] = \begin{cases} j - i + 1 & j - i + 1 \leq 1 \\ j - i + 1 & s[i] = s[j] \land DP[i+1][j-1] = j - i - 1 \\ \max\{DP[i][j-1], DP[i+1][j]\} & \text{altrimenti} \end{cases}$$

Utilizzando memoization, traduciamo la formula ricorsiva nel seguente codice che prende in input una tabella M inizializzata a **nil**.

```
maxPalindromeSubstring(ITEM[] s, int n, int i, int j, int[][] DP)
    if j − i + 1 ≤ 1 then
        return j − i + 1
    if s[i] == s[j] and maxPalindromeSubstring(s, n, i + 1, j − 1, DP) = j − i + 1 then
        return j − i + 1
    return max{maxPalindromeSubstring(s, n, i + 1, j, DP), maxPalindromeSubstring(s, n, i, j − 1, DP)}
```

La complessità è $O(n^2)$, dovendo al limite riempire tutta la tabella M di dimensione $n \times n$.

[1] https://en.wikipedia.org/wiki/Longest_palindromic_substring