Algoritmi e Strutture Dati - 21/07/14

Esercizio 1

Notate che la funzione calcola il minimo fra tutti i valori k; quindi, sicuramente

$$T(n) = \min_{1 \leq k \leq n-1} \{T[k] + T(n-k)\} + 1 \leq T(1) + T(n-1) + 1$$

Tramite il Teorema sulle ricorrenze lineari di ordine costante, è facile vedere che $T(n) = O(n)$
Proviamo a dimostrare per induzione che $T(n) = O(n)$; è facile dimostrare che

$$\exists c > 0, \exists m > 0 : T(n) \leq cn, \forall n \geq m$$

non è possibile a causa di un termine di ordine inferiore:

$$\begin{aligned} T(n) &= \min_{1 \leq k \leq n-1} \{T[k] + T(n-k)\} + 1 \\ &\leq \min_{1 \leq k \leq n-1} \{ck + cn - ck\} + 1 \\ &= cn + 1 \\ &\not\leq cn \end{aligned}$$

Proviamo quindi a dimostrare che:

$$\exists c > 0, \exists b > 0, \exists m > 0 : T(n) \leq cn - b, \forall n \geq m$$

- Passo base: $T(1) = 1 \leq c - b$, per cui $c \geq b + 1$
- Ipotesi induttiva: $T(n') \leq cn' - b$, per tutti gli $n' \leq n$;
- Passo induttivo:

$$\begin{aligned} T(n) &= \min_{1 \leq k \leq n-1} \{T[k] + T(n-k)\} + 1 \\ &\leq \min_{1 \leq k \leq n-1} \{ck - b + cn - ck - b\} + 1 \\ &= cn - 2b + 1 \\ &\not\leq cn - b \end{aligned}$$

L'ultima disequazione è vera per $b \geq 1$, e quindi dalla condizione del passo base abbiamo $c \geq 2$. Abbiamo quindi dimostrato che $T(n) = O(n)$; infatti, cresce come $T(n) = 2n - 1$.

Esercizio 2

Per la prima parte, è sufficiente effettuare una visita a partire dal nodo v (in tempo $O(m + n)$), utilizzando per esempio l'algoritmo che abbiamo scritto per identificare le componenti connesse; v è principale se e solo se tutti i nodi sono stati visitati a partire da v.

```
boolean isPrincipal(GRAPH G, NODE v)
    boolean[] id = new int[1 ... G.n]
    foreach u ∈ G.V() do
        id[u] = 0
    ccdfs(G, 1, v, id)
    foreach u ∈ G.V() do
        if id[u] == 0 then
            return false
    return true
```

Per la seconda parte, è ovviamente possibile ripetere la procedura isPrincipal() a partire da ogni nodo, con un costo computazionale $O(n(m + n)) = O(mn)$; ma è comunque possibile risolvere il problema in $O(m + n)$.
Si effettui una visita in profondità toccando tutti i nodi del grafo trasposto, utilizzando il meccanismo di discovery/finish time. Sia v l'ultimo nodo ad essere chiuso. Si utilizzi ora la procedura isPrincipal(G, v) definita sopra; se otteniamo **true**, allora esiste un nodo principale. Altrimenti, non esiste alcun nodo principale in G. La dimostrazione è per assurdo. Supponiamo che esista un nodo w principale; possono darsi due casi:

- se w è stato scoperto prima di v, allora v è un discendente di w e deve essere stato chiuso prima di w, assurdo;
- se v è stato scoperto prima di w, allora possono darsi due casi:
  - w è un discendente di v; ma allora anche v è principale, perchè v può raggiungere w e da esso tutti gli altri nodi; assurdo.
  - w non è un discendente di v; non esiste quindi un cammino di da v a w, e quindi v viene chiuso prima di w, assurdo.

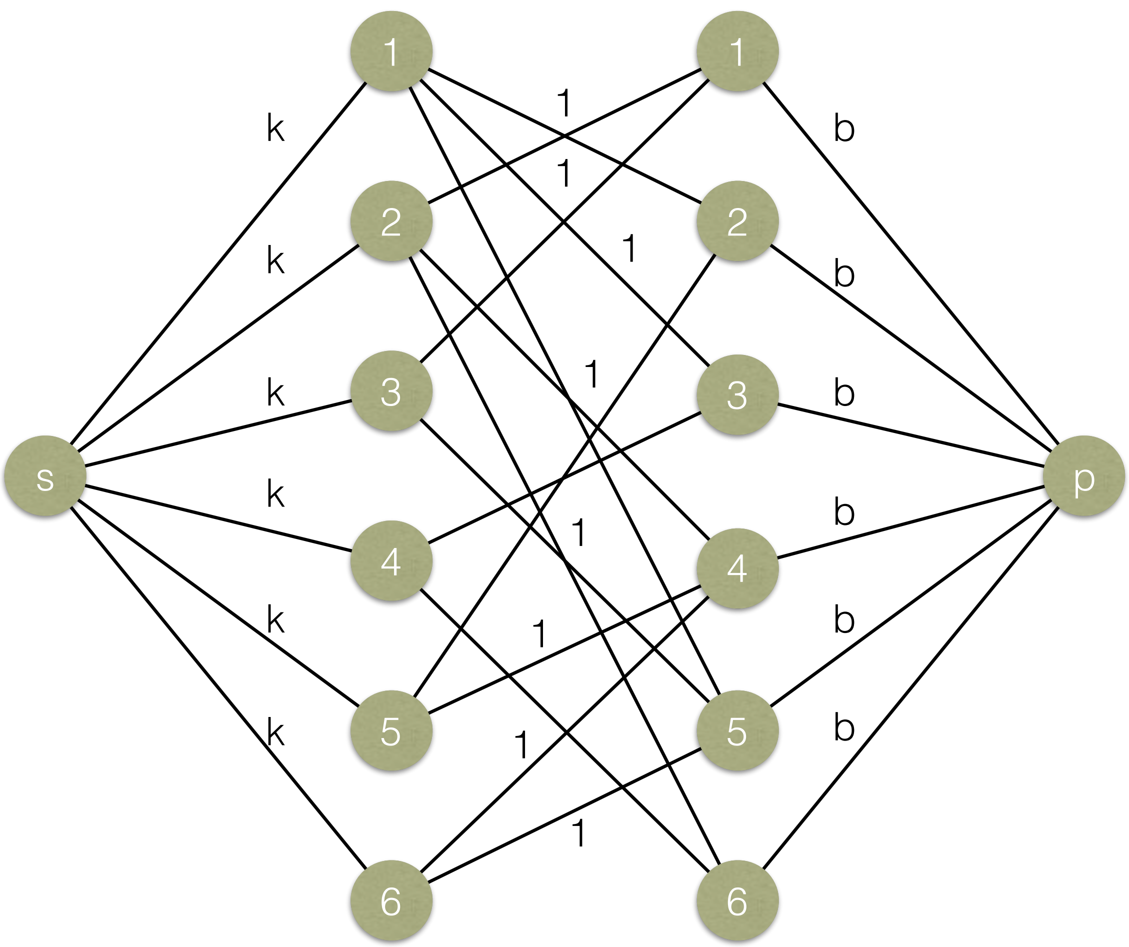
Per scrivere il codice, utilizziamo la procedura topsort() definita nei lucidi.

```
principal(GRAPH G)
    STACK S = topsort(G)
    NODE v = S.pop()
    return isPrincipal(G, v)
```

La procedura risultante è $O(m + n)$.

Esercizio 3

È possibile risolvere questo problema utilizzando una rete di flusso. È sufficiente creare un grafo contenente (i) una (super)sorgente; (ii) n nodi, uno per ogni sensore; (iii) altri n nodi, uno per ogni sensore; (iv) un (super)pozzo. La supersorgente è collegata ad ogni nodo sensore della prima serie con un arco con capacità k (valore limite che vogliamo raggiungere). I nodi sensori della prima serie sono collegati ai nodi sensori della seconda serie con archi con capacità 1, se possono comunicare l'uno con l'altro; (iv) la seconda serie di nodi sensori è collegata al superpozzo con archi di capacità b (valore limite che non vogliamo superare). La disposizione dei nodi è valida se tutti gli archi della supersorgente hanno valore k, ovvero se il flusso massimo è pari a kn.



La complessità è la seguente: esistono $|V| = 2n + 2$ nodi, con $|E| \leq 2n + n(n - 1)$ archi; secondo il limite di Ford-Fulkerson, la complessità è pari a $O(kn(|V| + |E|))$, ovvero $O(kn^3)$.

Esercizio 4

Al solito, per risolvere un problema come questo è utile definire la lunghezza massima in maniera ricorsiva e quindi utilizzare programmazione dinamica o memoization per risolvere il problema.
Definiamo con $DP[i][j]$ la lunghezza della più lunga sottosequenza palindroma contenuta nella sottostringa $s[i \dots j]$.

- Se $j < i$, ovvero se la sottostringa è nulla, allora la più lunga sottosequenza palindroma massimale è lunga 0;
- Se $j = i$, ovvero se la sottostringa è composta da un singolo carattere, allora la sottosequenza palindroma massimale è lunga 1, ovvero il carattere stesso;
- Altrimenti, se $s[i] = s[j]$, ovvero se il primo e l'ultimo carattere sono uguali, la sottosequenza massimale è data da $S[i+1, j-1]+2$, in quanto contiamo tali caratteri e poi cerchiamo la più lunga sottosequenza palindroma massimale contenuta *fra* essi;
- Altrimenti, elimino il primo carattere o l'ultimo, e verifico qual è la più lunga sottosequenza palindroma massimale nelle sottostringhe risultanti

$$DP[i][j] = \begin{cases} 0 & j < i \\ 1 & j = i \\ DP[i+1][j-1] + 2 & j > i \wedge s[i] = s[j] \\ \max\{DP[i+1][j], DP[i][j-1]\} & j > i \wedge s[i] \neq s[j] \end{cases}$$

Per semplicità di scrittura, utilizziamo memoization:

```
longestPalindrome(int[] s, int n)
  int[][] L = new int[1...n][1...n]
  for i = 1 to n do
    for i = 1 to n do
      DP[i][j] = ⊥
  longRec(s, 1, n, DP)
  printRec(s, 1, n, DP)
```

```
int longRec(int[] s, int i, int j, int[][] DP)
  if j > i then
    return 0
  if j == i then
    return 1
  if DP[i][j] = ⊥ then
    if s[i] = s[j] then
      DP[i][j] = longRec(s, i + 1, j − 1, DP) + 2
    else
      DP[i][j] = max(longRec(s, i + 1, j, DP), longRec(s, i, j − 1, DP), )
  return DP[i][j]
```

Dovendo riempire una tabella di dimensione n^2, la complessità dell'algoritmo è $O(n^2)$.

Per la richiesta opzionale di stampare una stringa, il codice seguente utilizza i valori memorizzati nella tabella L per stampare una sottosequenza palindroma massimale.

```
printRec(int[] s, int i, int j, int[][] DP)
  if j > i then
    return
  if j == i then
    print s[i]
    return
  if s[i] == s[j] then
    print s[i]
    printRec(s, i + 1, j − 1)
    print s[j]
  else
    if DP[i + 1][j] > DP[i][j − 1] then
      printRec(s, i + 1, j, DP)
    else
      printRec(s, i, j − 1, DP)
```

Notate che si poteva risolvere il problema ancora più semplicemente cercando la sottosequenza comune massimale fra la stringa s e la stringa s invertita.