Algoritmi e Strutture Dati - 29/08/2016

Esercizio 1

Dimostriamo per induzione che $T(n) = O(n)$.

- Caso base, $n = 1$: $T(n) = 1 \leq c \cdot 1$ è vero per $c \geq 1$.

  Ad onor del vero, nel caso $n = 0$ non è possibile dimostrare il caso base. Bisognerebbe quindi dimostrare che la disequazione è valida per tutti i casi in cui $n < 15$, perchè danno origine al caso $T(0)$.

  Ad esempio, per $n = 2$, si ottiene:

  $$T(2) = T(\lfloor 2/15 \rfloor) + T(\lfloor 2/10 \rfloor) + 2T(\lfloor 2/6 \rfloor) + \sqrt{2} = T(0) + T(0) + 2T(0) + \sqrt{2} = 4 + \sqrt{2} \leq c \cdot 2$$

  che è vera per $c \geq (4 + \sqrt{2})/2$.

  Proviamo ad esempio $n = 12$; si ottiene

  $$T(12) = T(\lfloor 12/15 \rfloor) + T(\lfloor 12/10 \rfloor) + 2T(\lfloor 12/6 \rfloor) + \sqrt{12} = T(0) + T(1) + 2T(2) + 2\sqrt{3} = 6 + \sqrt{2} + 2\sqrt{3} \leq c \cdot 12$$

  che è vera per $c = (6 + \sqrt{2} + 2\sqrt{3})/12$

  L'esercizio dovrebbe andare avanti per tutti i valori di $n < 15$, ma alla fine la forma è sempre uguale: c deve essere maggiore di qualche valore. Sarà sufficiente prendere il più grande fra questi, e tutte le condizioni sui casi base saranno verificate.

- Passo induttivo, $n > 1$. Per ipotesi induttiva, assumiamo che $T(n') \leq cn'$, per qualsiasi $n' \leq n$.

  $$\begin{aligned} T(n) &= T(\lfloor n/15 \rfloor) + T(\lfloor n/10 \rfloor) + 2T(\lfloor n/6 \rfloor) + \sqrt{n} \\ &\leq c\lfloor n/15 \rfloor + c\lfloor n/10 \rfloor + 2c\lfloor n/6 \rfloor + \sqrt{n} \\ &\leq cn/15 + cn/10 + cn/3 + \sqrt{n} \\ &\leq cn/2 + \sqrt{n} \leq cn \end{aligned}$$

  L'ultima disequazione è la nostra tesi; la condizione è vera per $c \geq 2/\sqrt{n}$. Essendo questa funziona monotona decrescene per i valori di n interi positivi, è sufficiente che $c \geq 2$.

Abbiamo quindi dimostrato che $T(n) = O(n)$.
In realtà il limite è più stretto. Chiedendo un limite superiore, è possibile valutare nel modo seguente:

$$\begin{aligned} T(n) &= T(\lfloor n/15 \rfloor) + T(\lfloor n/10 \rfloor) + 2T(\lfloor n/6 \rfloor) + \sqrt{n} \\ &\leq T(n/15) + T(n/10) + 2T(n/6) + \sqrt{n} \\ &\leq T(n/6) + T(n/6) + 2T(n/6) + \sqrt{n} \\ &= 4T(n/6) + \sqrt{n} \end{aligned}$$

Utilizzando il master theorem, si ottiene che $T(n) = O(n^{\log_6 4})$, dove $\log_6 4 \approx 0.77$.
Si potrebbe tentare di realizzare di dimostrare $O(\sqrt{n})$, ma la disequazione risulta falsa per il termine di ordine superiore, quindi $T(n)$ non è $O(\sqrt{n})$.

$$
\begin{aligned}
T(n) &= T(\lfloor n/15 \rfloor) + T(\lfloor n/10 \rfloor) + 2T(\lfloor n/6 \rfloor) + \sqrt{n} \\
&\stackrel{ip.ind.}{\leq} c\left(\sqrt{\lfloor n/15 \rfloor}\right) + c\left(\sqrt{\lfloor n/10 \rfloor}\right) + 2c\left(\sqrt{\lfloor n/6 \rfloor}\right) + \sqrt{n} \\
&\leq c\frac{\sqrt{n}}{\sqrt{15}} + c\frac{\sqrt{n}}{\sqrt{10}} + 2c\frac{\sqrt{n}}{\sqrt{6}} + \sqrt{n} \\
&\leq \sqrt{n} \cdot c \cdot 1.39... + \sqrt{n} \\
&= \sqrt{n}(c \cdot 1.39... + 1) \stackrel{?}{\leq} c\sqrt{n}
\end{aligned}
$$

$$
\begin{aligned}
&\cancel{\sqrt{n}}(c \cdot 1.39... + 1) \stackrel{?}{\leq} c\cancel{\sqrt{n}} \\
&(c \cdot 1.39... - 1) \stackrel{?}{\leq} -1 \\
&(c \cdot 0.39...) \stackrel{?}{\leq} -1 \\
&c \not\leq -\frac{1}{0.39...}
\end{aligned}
$$

Impossibile per valori di $c > 0$

Questa disequazione è stata contribuita da Andrea Zanotto.

Esercizio 2

Un algoritmo semplice per risolvere il problema effettua una visita a partire da ogni nodo, verificando nodo per nodo e interrompendo la visita qualora le etichette non corrispondano.

```
boolean checkPath(GRAPH G, int[] ℓ, int[] A, int n)
  foreach u ∈ G.V() do
    if checkLabels(G, ℓ, A, n, 1, u) then
      return true
```

```
boolean checkPath(GRAPH G, int[] ℓ, int[] A, int n, int i, int u)
  if ℓ[u] == A[i] then
    if i == n then
      return true
    foreach v ∈ G.adj(u) do
      if checkLabels(G, ℓ, A, n, i + 1, v) then
        return true
  return false
```

La complessità dell'algoritmo è superpolinomiale, in quanto è necessario seguire tutti i possibili percorsi, anche tenendo conto che il grafo è aciclico.

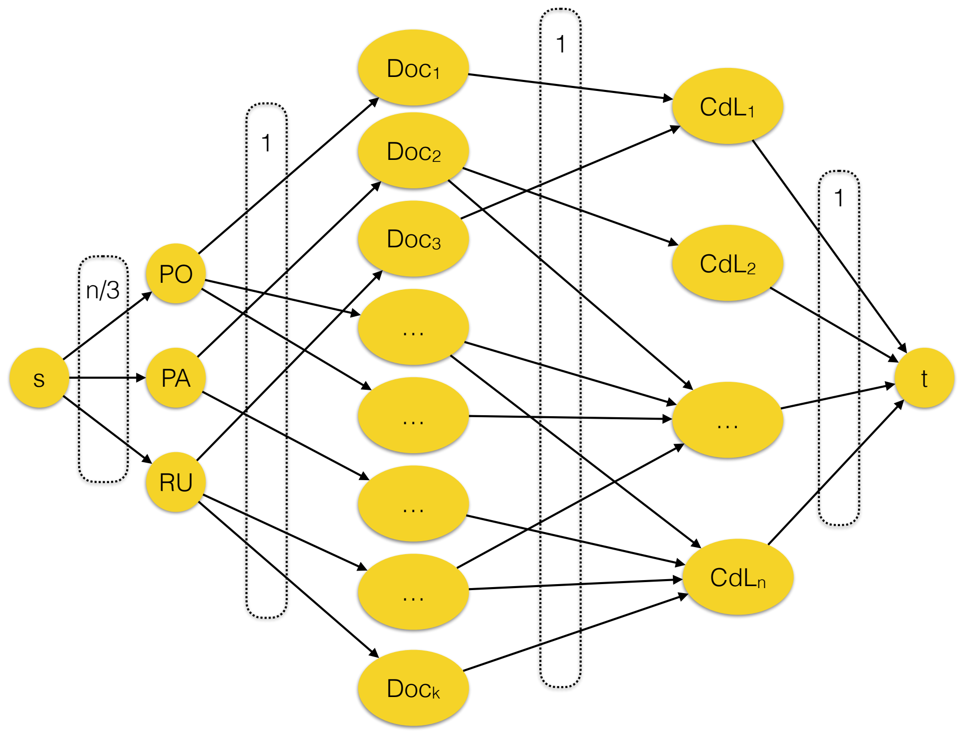
Esercizio 3

Per risolvere questo problema, è possibile utilizzare una rete di flusso.
Si crea un nodo per ogni CdL e un nodo per ogni docente. Si creano inoltre tre nodi PO, PA, RU che rappresentano ognuna delle categorie, più i nodi sorgente e pozzo.

- Si collega il nodo sorgente con ognuna delle categorie, con capacità $n/3$.
- Si collega ogni categoria a tutti i docenti della categoria, con capacità 1.
- Si collega ogni docente ai CdL in cui insegna, con capacità 1.

- Si collega ogni CdL al pozzo, con capacità 1.

Il numero totale di nodi è pari a $n+k+5$; il numero totale di archi è limitato superiormente da $n+4k+3$. Il valore del flusso massimo che si vuole ottenere è pari a n (ogni CdL ha un rappresentante). Il costo totale è quindi limitato superiormente da $O(nk+n^2)$.



Esercizio 4

Utilizziamo programmazione dinamica. Sia $DP[p][t]$ il numero di occorrenze del suffisso $P[1\ldots p]$ all'interno del suffisso $T[1\ldots t]$. Il problema originale è quindi $DP[m][n]$. È possibile calcolare $DP[p][t]$ in modo ricorsivo come segue:

$$DP[p][t] = \begin{cases} 0 & p > t \\ 1 & p = 0 \land t \geq 0 \\ DP[p][t-1] & P[p] \neq T[t] \land 0 < p \leq t \\ DP[p-1][t-1] + DP[p][t-1] & \text{altrimenti} \end{cases}$$

In altre parole, restituiamo 0 se stiamo cercando un pattern più lungo del testo; restituiamo 1 se il pattern è vuoto. Se gli ultimi caratteri del pattern e del testo non coincidono, dobbiamo per forza scartare l'ultimo carattere del testo, e continuare a cercare il pattern, arretrando di 1 nel testo. Se gli ultimi caratteri coincidono, ci sono due possibilità: possiamo considerare questa uguaglianza, e quindi scartare entrambi i caratteri, oppure no, nel qual caso dobbiamo scartare un carattere del testo. Questi due casi vanno sommati.

È possibile scrivere il codice basato su memoization nel modo seguente:

```
int count(int[] P, int[] T, int m, int n)
    int[][] DP = new int[0...m][0...n] = {−1}                % Initialized to −1
    return countRec(P, T, m, n, DP)
```

```
int countRec(int[] P, int[] T, int p, int t, int[][] DP)
    if p > t then
        return 0
    if p == 0 then
        return 1
    if DP[p][t] < 0 then
        if P[p] ≠ T[t] then
            DP[p][t] = countRec(P, T, p, t − 1, C)
        else
            DP[p][t] = countRec(P, T, p − 1, t − 1, C) + countRec(P, T, p, t − 1, C)
    return DP[p][t]
```

L'algoritmo risultante ha complessità $O(mn)$. Si noti che la tabella viene riempita completamente solo se entrambe le stringhe sono composte dallo stesso carattere ripetuto.