Algoritmi e Strutture Dati - 06/06/16

Esercizio 1

È facile vedere che $T(n) = \Omega(n)$, a causa della parte non ricorsiva. Proviamo quindi a dimostrare che $T(n) = O(n)$:

- Caso base: Per $n = 1$, $T(1) = 1 \leq c - b$, ovvero $c \geq b + 1$;
- Ipotesi induttiva: $T(n') \leq cn', \forall n' < n$;
- Passo induttivo:

$$\begin{aligned}
T(n) &= T(\frac{1}{10}n) + T(\frac{5}{6}n) + T(\frac{1}{16}n)) + n \\
&\leq \frac{1}{10}cn + \frac{5}{6}cn + \frac{1}{16}cn + n \\
&= \frac{24 + 200 + 15}{240}cn \\
&= \frac{239}{240}cn \\
&\leq cn
\end{aligned}$$

  L'ultima disequazione è vera per $c \geq 240$; abbiamo quindi dimostrato che $T(n) = O(n)$.

  Ne consegue che $T(n) = \Theta(n)$.

Esercizio 2

È possibile risolvere il problema con una visita in profondità, restituendo ad ogni chiamata ricorsiva su un nodo T una coppia di valori: il profitto dell'albero radicato in T, e il minimo profitto di tutti i sottoalberi contenuti nel sottoalbero radicato in T. La complessità è quella di una visita in profondità – $O(n)$.

```
(int, int)minProfit(TREE u)
  if u == nil then
    return (0, ∞);
  profit = u.productivity − u.salary
  minProfit = +∞
  f = u.leftmostChild()
  while f ≠ nil do
    tot, min = minProfit(f)
    profit = profit + tot
    minProfit = min(minProfit, min)
    f = f.rightSibling()
  minProfit = min(minProfit, profit)
  return (profit, minProfit)
```

Esercizio 3

Questo esercizio è identico all'esercizio di laboratorio "Node cover su albero non pesato" proposto da Guerrieri. È possibile risolverlo con due equazioni di ricorrenza. $S[u]$ restituisce il numero di nodi necessari per coprire l'albero radicato in u, con u scelta obbligata. $L(u)$ restituisce il numero di nodi necessari per coprire l'albero radicato in u, con il nodo u che può essere scelto oppure no. Utilizziamo $C(u)$ per denotare i figli di u.

$$S[u] = \begin{cases} 1 + \sum_{f \in C(u)} L[f] & u \neq \textbf{nil} \\ 0 & u = \textbf{nil} \end{cases}$$

$$L[u] = \begin{cases} \min(S[u], \sum_{f \in C(u)} S[f] & u \neq \textbf{nil} \\ 0 & u = \textbf{nil} \end{cases}$$

Per risolvere il problema, si calcola il valore di $S[T]$, dove T è la radice dell'albero. Essendo un albero generale, utilizziamo la notazione figlio sinistro - fratello destro. Vista la doppia ricorsione, è possibile che lo stessa chiamata più volte, ed è quindi necessario utilizzare memoization. La complessità è quella di una visita, $O(n)$ per un albero di n nodi.

```
int computeS(TREE u, int[] S, int[] L)
  if u == nil then
    return 0
  if S[u] == nil then
    int tot = 0
    f = u.leftmostChild()
    while f ≠ nil do
      tot = tot + computeL(f, S, L)
      f = f.rightSibling()
    S[u] = 1 + tot
  return S[u]
```

```
int computeL(TREE u, int[] S, int[] L))
  if u == nil then
    return 0
  if L[u] == nil then
    int tot = 0
    f = u.leftmostChild()
    while f ≠ nil do
      tot = tot + computeS(f, S, L)
      f = f.rightSibling()
    L[u] = min(computeS(u), tot)
  return L[u]
```

Esercizio 4

Sia $DP[n][k]$ il numero di vettori ordinati di lunghezza n, contenenti k valori distinti (compresi fra 1 e k). $DP[n][k]$ può essere calcolato in maniera ricorsiva come segue:

$$DP[n][k] = \begin{cases} 1 & n = 0 \\ \sum_{i=1}^{k} DP[n-1][i] & \end{cases}$$

In altre parole, è possibile scegliere il valore più basso, ed avere ancora k oggetti possibili; il secondo valore più basso, ed avere $k-1$ oggetti possibili; e così via fino a scegliere il valore più alto, limitando ogni futura scelta a quel valore, per cui si ha 1 solo valore possibile. L'equazione ricorsiva di cui sopra può essere trasformata nel codice seguente, basato su memoization:

```
int orderedPermutation(int n, int k)
  int[][] DP = new int[1...n][1...k] = {−1}        % Initialized to −1
  return opRec(n, k, DP)
```

```
int opRec(int n, int k, int[][] DP)
  if n == 0 then
    return 1
  if DP[n][k] < 0 then
    DP[n][k] = 0
    for i = 1 to k do
      DP[n][k] = DP[n][k] − opRec(n − 1, i, DP)
  return DP[n][k]
```

Ovviamente, questo richiede una tabella $O(nk)$, per calcolare ogni elemento delle quale saranno necessarie $O(k)$ operazioni, per un costo totale di $O(nk^2)$.
Una soluzione alternativa, più efficiente, calcola $DP[n][k]$ nel modo seguente:

$$DP[n][k] = \begin{cases} k & n = 1 \\ 0 & k = 0 \\ DP[n-1][k] + DP[n][k-1] & \text{altrimenti} \end{cases}$$

In altre parole, se ho un solo elemento, ho k possibili valori; se non mi sono rimasti più valori disponibile, restituisco 0, perchè non è possibile formare il vettore. Altrimenti, possono darsi due casi: posso considerare sempre n valori, ma utilizzando un numero ridotto di valori $(k-1)$ oppure posso tenere fisso k e ridurre il numero di elementi del vettore.
Lo pseudocodice basato su memoizaton che implementa l'equazione ricorsiva di cui sopra è il seguente. La funzione wrapper è identica alla precedente.

```
int opRec(int n, int k, int[][] DP)
    if n == 1 then
        return k
    if k == 0 then
        return 0
    if DP[n][k] < 0 then
        DP[n][k] = opRec(n − 1, k, DP) + opRec(k, n − 1, DP)
    return DP[n][k]
```

Ovviamente, questo richiede una tabella $O(nk)$, per calcolare ogni elemento delle quale saranno necessarie $O(1)$ operazioni, per un costo totale di $O(nk)$.