Algoritmi e Strutture Dati - 03/11/2016

C'erano due versioni molto simili dell'esercizio.

Esercizio 1 - Versione 1

Limite superiore di:

$$T(n) = \begin{cases} 27\,T(\lfloor n/9 \rfloor) + n\sqrt{n} & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Utilizzando il Master Theorem, vediamo che $\alpha = \log_9 27 = 3/2$; $\beta = 3/2$. Siamo quindi nel secondo caso, e sappiamo quindi che $T(n) = n\sqrt{n}\log n$. Proviamo il limite superiore per induzione.

- **Caso base**: $T(n) = 1 \leq c \cdot 1 \cdot \sqrt{1}\log 1 = 0$.

  Purtroppo $1 \not\leq 0$ e quindi dobbiamo considerare altri casi base. Per $2 \leq i \leq 17$, $\lfloor i/9 \rfloor$ è pari a 0 oppure 1, e quindi rientrano nel caso $n \leq 1$ della ricorrenza. Possiamo quindi ottenere le seguenti disequazioni:

  $$T(i) = 27 \cdot 1 + i\sqrt{i} \leq ci\sqrt{i}\log i$$
  $$c \geq \frac{i\sqrt{i} + 27}{i\sqrt{i}\log i} = \frac{1}{\log i} + \frac{27}{i\sqrt{i}\log i}$$

  Poichè $\frac{1}{\log i} + \frac{27}{i\sqrt{i}\log i}$ è una funzione decrescente per $i \geq 2$, possiamo prendere $c \geq 1/2 + \frac{27}{2\sqrt{2}}$ come valore che soddisfa tutte le disequazioni di cui sopra.

- **Ipotesi induttiva**: Supponiamo di aver dimostrato che $T(n') \leq cn'\sqrt{n'}\log n'$, per tutti i valori n' tali che $2 \leq n' < n$.

- **Passo induttivo**:

  $$\begin{aligned} T(n) &= 27\,T(\lfloor n/9 \rfloor) + n\sqrt{n} & \\ &\leq 27c\lfloor n/9 \rfloor\sqrt{\lfloor n/9 \rfloor}\log\lfloor n/9 \rfloor + n\sqrt{n} & \text{Sostituzione} \\ &\leq 27c(n/9)\sqrt{n/9}\log n/9 + n\sqrt{n} & \text{Rimozione interi inferiori} \\ &= cn\sqrt{n}(\log n - \log 9) + n\sqrt{n} & \text{Passo algebrico} \\ &\leq cn\sqrt{n}\log n & \text{Obiettivo} \end{aligned}$$

  L'ultima disequazione è vera per $c \geq \frac{1}{\log 9}$.

Combinando insieme il caso base e il passo induttivo, otteniamo che $T(n) = O(n\sqrt{n}\log n)$ con costanti $m = 2$ e $c \geq 1 + \frac{27}{2\sqrt{2}}$

Esercizio 1 - Versione 2

Limite inferiore di:

$$T(n) = \begin{cases} 64\,T(\lceil n/16 \rceil) + n\sqrt{n} & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Utilizzando il Master Theorem, vediamo che $\alpha = \log_{16} 64 = 3/2$; $\beta = 3/2$. Siamo quindi nel secondo caso, e sappiamo quindi che $T(n) = n\sqrt{n}\log n$. Proviamo il limite inferiore per induzione.

- **Caso base**: $T(n) = 1 \geq c \cdot 1 \cdot \sqrt{1}\log 1 = 0$.

  Questa disequazione è vera per ogni valore di c.

- **Ipotesi induttiva**: Supponiamo di aver dimostrato che $T(n') \geq cn'\sqrt{n'}\log n'$, per tutti i valori n' tali che $2 \leq n' < n$.

- **Passo induttivo**:

$$
\begin{aligned}
T(n) &= 64\,T(\lfloor n/16\rfloor) + n\sqrt{n} & \\
&\geq 64c\lfloor n/16\rfloor\sqrt{\lfloor n/16\rfloor}\log\lfloor n/16\rfloor + n\sqrt{n} & \text{Sostituzione} \\
&\geq 64c(n/16)\sqrt{n/16}\log n/16 + n\sqrt{n} & \text{Rimozione interi inferiori} \\
&= cn\sqrt{n}(\log n - \log 16) + n\sqrt{n} & \text{Passo algebrico} \\
&\geq cn\sqrt{n}\log n & \text{Obiettivo}
\end{aligned}
$$

L'ultima disequazione è vera per $c \leq \frac{1}{\log 16} = 1/4$.

Combinando insieme il caso base e il passo induttivo, otteniamo che $T(n) = O(n\sqrt{n}\log)$ con costanti $m = 1$ e $c = 1/4$.

Esercizio 2

Sia V un vettore di $n \geq 3$ interi, con la seguente proprietà:

- per ogni indice $i \in \{2 \ldots n\}$, $|V[i-1] - V[i]| \leq 1$
- $V[1] < 0$ e $V[n] > 0$

Questo esercizio si risolve tramite divide-et-impera. Innanzitutto, è importante notare che esiste sicuramente un elemento con valore 0 all'interno del vettore. Per assurdo, supponiamo che 0 non sia presente. Poichè nel primo elemento c'è un valore negativo e nell'ultimo elemento c'è un valore positivo, esiste sicuramente una coppia di elementi consecutivi in cui il primo elemento è negativo e il secondo positivo. Essendo interi, la loro differenza relativa è maggiore di 1, il che è in contraddizione con la descrizione del vettore.
L'algoritmo funziona in questo modo: dati un intervallo $A[i \ldots j]$, si calcola $m = \lfloor (i+j)/2 \rfloor$. Si verifica il contenuto di $A[m]$, possono darsi tre casi:

- $A[m] = 0$, abbiamo trovato lo zero.
- $A[m] < 0$, allora nell'intervallo $A[m \ldots j]$ esiste sicuramente uno zero in quanto $A[m] < 0$ e $A[j] > 0$.
- $A[m] > 0$, allora nell'intervallo $A[i \ldots m]$ esiste sicuramente uno zero in quanto $A[i] < 0$ e $A[m] > 0$.

Poichè in tutte le chiamate siamo sicuri che $A[i] < 0$ e $A[j] > 0$ e fra i e j esiste uno zero, il caso base è un vettore di tre elementi $-1, 0, 1$. Viene calcolato il valore mediano e si restituisce tale valore.

```
int zero(int[] V, int n)
    return zeroRec(V, 1, n)
```

```
int zeroRec(int[] V, int i, int j)
    m = ⌊(i + j)/2⌋
    if A[m] == 0 then
        return m
    else if A[m] < 0 then
        return zeroRec(V, m, j)
    else
        return zeroRec(V, i, m)
```

La complessità è $O(\log n)$, per via della ricerca dicotomica effettuata sul vettore.

Esercizio 3

In questo problema, il costo di una foglia, dato dalla somma dei pesi dei nodi sul cammino radice foglia, può essere calcolato tramite una visita in profondità con pre-visita, in cui il valore calcolato finora viene passato nella chiamata ricorsiva, e a cui viene aggiunto il peso del nodo su cui viene invocata.
La funzione minCountRec(t, sum) restituisce una coppia di valori; il primo è il costo minimo delle foglie nel sottoalbero radicato in t, il secondo è il numero di foglie che presentano quel costo. Sono dati vari casi:

- Se il nodo è **nil**, restituiamo un costo $+\infty$, in modo da non sceglierlo mai
- Se il nodo è una foglia, restituiamo il costo della foglia, e riportiamo che in questo sottoalbero solo un nodo ha quel valore
- Se il costo minimo nel sottoalbero sinistro e destro di t è uguale, il costo minimo resta uguale e bisogna sommare insieme il numero di foglie che presentano quel valore
- Altrimenti, si restituisce il valore del sottoalbero con valore più basso.

Il costo della procedure è $O(n)$, in quanto si tratta di una semplice visita dell'albero.

```
int minCount(TREE T)
    int, int c, n = minCountRec(T, 0)
    return c
```

```
int minCountRec(TREE t, int sum)
    if t == nil then
        return (+∞, 1)                                   % Rami nil non vanno considerati
    sum = sum + t.p
    if t.left() == nil and t.right() == nil then
        return (sum, 1)                                  % Foglia
    else
        c_L, n_L = minCountRec(t.left(), sum)
        c_R, n_R = minCountRec(t.right(), sum)
        if c_L == c_R then
            return (c_L, n_L + n_R)
        else if c_L < c_R then
            return (c_L, n_L)
        else
            return (c_R, n_R)
```

Esercizio 4

Nella soluzione che segue, basata su DFS, si utilizza un vettore *longest* che conterrà il più lungo cammino crescente che inizia in ogni nodo. Questo vettore assume anche il ruolo di *visited*, ovvero ci permette di capire se un nodo è già stato visitato.
Si noti che essendo diretto aciclico, un nodo può essere visitato solo da archi dell'albero di visita, in avanti o di attraversamento – niente archi all'indietro. Quindi, tutte le volte che viene visitato un nodo per la seconda volta, il valore *longest* corrispondente è già stato calcolato e può essere utilizzato per calcolare il valore locale.
Il costo è quello di una visita in profondità, $O(m + n)$.

```
int longestIncreasing(GRAPH G)
    int[] longest = new int[1 ... G.n] = { -1 }          % Initialized to −1
    foreach u ∈ G.V() do
        if longest[u] < 0 then
            longestIncRec(G, u, longest)
    return max(longest, G.n)
```

```
int longestIncRec(GRAPH G, NODE u, int[] longest)
    longest[u] = 0
    foreach v ∈ G.adj(u) do
        if longest[v] < 0 then
            longestIncRec(G, v, longest)
        if u.p < v.p and longest[u] < longest[v] + 1 then
            longest[u] = longest[v] + 1
```