

## Esercizio 1

Proviamo a dimostrare che  $T(n) = O(n)$ ; provando a dimostrare subito il caso induttivo, non si riesce a soddisfare la disequazione per un termine di ordine inferiore:

$$\begin{aligned} T(n) &= 6T(n/8) + T(n/4) + 1 \\ &\leq \frac{6}{8}cn + \frac{1}{4}cn + 1 \\ &= cn + 1 \\ &\not\leq cn \end{aligned}$$

Proviamo quindi a dimostrare che  $T(n) \leq cn - b$ , con  $b > 0$ .

- Caso base:  $T(1) = 1 \leq c - b$ , che è vera per  $c \geq b + 1$ .
- Ipotesi induttiva:  $T(n') \leq cn' - b$ , per ogni  $n' < n$ .
- Passo induttivo:

$$\begin{aligned} T(n) &= 6T(n/8) + T(n/4) + 1 \\ &\leq \frac{6}{8}cn - 6b + \frac{1}{4}cn - b + 1 \\ &= cn - 7b + 1 \end{aligned} \qquad \not\leq cn - b$$

L'ultima disequazione è vera per  $b \geq \frac{1}{6}$ .

## Esercizio 2

(1) Questo problema si può risolvere con un algoritmo greedy. Ordiniamo le sezioni in senso non decrescente rispetto alla lunghezza, in modo che il segmento 1 abbia lunghezza minima e il segmento  $n$  lunghezza massima. Procediamo quindi a segare prima il segmento più corto, poi quello successiva e così via finché possibile (cioè fino a quando la lunghezza residua ci consente di ottenere almeno un'altro segmento). Lo pseudocodice può essere descritto in questo modo algoritmo.

---

maxSections(integer  $L$ , integer[]  $S$ , integer  $n$ )

---

```
sort( $S$ )
integer  $i \leftarrow 1$ 
while  $i \leq n$  and  $L \geq S[i]$  do
     $L \leftarrow L - S[i]$ 
     $i \leftarrow i + 1$ 
return  $i - 1$ 
```

---

(2) Per dimostrare che l'algoritmo gode della proprietà greedy, si consideri una soluzione ottima  $S$  che non contiene il segmento più corto  $m$ . Si elimini il segmento più corto  $m'$  contenuto in  $S$ , e si sostituisca con il segmento più corto in assoluto, ottenendo quindi la soluzione  $S' = S - \{m'\} \cup \{m\}$ . Tale soluzione è ovviamente ottima quanto  $S$ , visto che contiene lo stesso numero di segmenti, e rispetta la proprietà greedy.

Si noti che l'algoritmo di cui sopra restituisce l'output corretto sia nel caso in cui gli  $n$  segmenti abbiano complessivamente lunghezza minore o uguale a  $L$ , sia nel caso opposto in cui nessuno abbia lunghezza minore o uguale a  $L$  (in questo caso l'algoritmo restituisce zero).

(3) L'operazione di ordinamento può essere fatta in tempo  $\Theta(n \log n)$  usando un algoritmo di ordinamento generico. Il successivo ciclo while ha costo  $O(n)$  nel caso peggiore. Il costo complessivo dell'algoritmo risulta quindi  $\Theta(n \log n)$ .

### Esercizio 3

Due alberi sono strutturalmente uguali se hanno lo stesso numero di nodi, e i sottoalberi destro e sinistro della radice sono strutturalmente uguali. Possiamo sfruttare questo fatto scrivendo per prima un algoritmo che utilizza un campo  $n$  per memorizzare il numero di nodi di un albero; e poi sfruttando questo valore per confrontare i due alberi.

---

**boolean** find(**TREE**  $T$ , **TREE**  $P$ )

---

```
count( $T$ )
count( $P$ )
return check( $T$ ,  $P$ )
```

---

---

**integer** count(**TREE**  $T$ )

---

```
if  $T = \text{nil}$  then
  return 0
 $T.n \leftarrow 1 + \text{count}(T.\text{left}) + \text{count}(T.\text{right})$ 
return  $T.n$ 
```

---

---

**boolean** check(**TREE**  $T$ , **TREE**  $P$ )

---

```
if  $T = P = \text{nil}$  then
  return true
if  $T.n < P.n$  then
  { Il numero di nodi in  $T$  è minore del numero di nodi in  $P$ , quindi  $T$  non può contenere  $P$  }
  return false
if  $T.n = P.n$  then
  { Avendo lo stesso numero di nodi, verifichiamo che i sottoalberi destro e sinistro abbiano lo stesso numero di nodi }
  return check( $T.\text{right}$ ,  $P.\text{right}$ ) and check( $T.\text{left}$ ,  $P.\text{left}$ )
if  $T.n > P.n$  then
  { Poichè  $T$  ha un numero maggiore di nodi,  $P$  potrebbe essere contenuto nel sottoalbero destro o sinistro }
  return check( $T.\text{right}$ ,  $P$ ) or check( $T.\text{left}$ ,  $P$ )
```

---

La complessità è pari a  $O(n_p + n_t)$ , in quanto tutti i nodi di  $T$  vengono visitati una ed una sola volta, e i nodi di  $P$  vengono visitati solo se la dimensione del sottoalbero di  $T$  esaminato è esattamente uguale a  $n_p$ .

### Esercizio 4

Definiamo con  $S[i]$  il numero minimo di mosse necessarie per andare dalla posizione  $i$  alla posizione  $n$ . È possibile calcolare  $S[i]$  in modo ricorsivo come segue:

$$S[i] = \begin{cases} 0 & i = n \\ \min\{S[i+k] + 1 : 1 \leq k \leq \min(V[i], n-i)\} & i < n \end{cases}$$

In altre parole, sono necessarie 0 mosse per raggiungere la posizione  $n$ , se ci si trova già in posizione  $n$ ; altrimenti, si deve effettuare una mossa e portarsi nella posizione che richiede il minor numero di mosse e che può essere legalmente raggiunta da  $i$ .

Il codice è il seguente:

---

**minMoves**(**integer**[]  $V$ , **integer**  $n$ )

---

```
integer[]  $S \leftarrow \text{new integer}[1 \dots n]$ 
 $S[n] \leftarrow 0$ 
for  $i \leftarrow n - 1$  downto 1 do
   $S[i] \leftarrow +\infty$ 
  for  $k \leftarrow 1$  to  $V[i]$  do
    if  $i + k \leq n$  and  $S[i+k] + 1 < S[i]$  then
       $S[i] \leftarrow S[i+k] + 1$ 
```

---

L'algoritmo ha complessità  $O(n^2)$ , per via dei due cicli annidati.