

**Esercizio 1**

L'equazione di ricorrenza è composta da vari casi, che possono essere innanzitutto analizzati separatamente per comprendere il loro comportamento.

**Equazione 1**  $T_1(n) = T_1(\lfloor n/2 \rfloor) + T_1(\lfloor n/4 \rfloor) + n$

È facile vedere che  $T_1(n) = \Theta(n)$ ; è  $\Omega(n)$  per via del termine non ricorsivo, ed è possibile dimostrare che  $T_1(n) = O(n)$  per sostituzione con  $T_1(n) \leq cn$ :

$$\begin{aligned} T_1(n) &= T_1(\lfloor n/2 \rfloor) + T_1(\lfloor n/4 \rfloor) + n \\ &\leq c\lfloor n/2 \rfloor + c\lfloor n/4 \rfloor + n \\ &\leq cn/2 + cn/4 + n \\ &= \frac{3}{4}cn + n \leq cn \end{aligned}$$

L'ultima disequazione è vera per  $c \geq 4$ . Nel caso base  $T_1(1) = 1 \leq c$ , che è vera per  $c \geq 1$ . Abbiamo quindi dimostrato che  $T_1(n) = \Theta(n)$ .

**Equazione 2**  $T_2(n) = T_2(\lfloor n/9 \rfloor) + T_2(\lfloor n/81 \rfloor) + \sqrt{n}$

È facile vedere che  $T_2(n) = \Theta(\sqrt{n})$ . Infatti, è possibile vedere che  $T_2(n) = \Omega(\sqrt{n})$ , per via del termine non ricorsivo. È possibile poi vedere che  $T_2(n) = O(\sqrt{n})$  per sostituzione con  $T_2(n) \leq c\sqrt{n}$ :

$$\begin{aligned} T_2(n) &= T_2(\lfloor n/9 \rfloor) + T_2(\lfloor n/81 \rfloor) + \sqrt{n} \\ &\leq c\sqrt{\lfloor n/9 \rfloor} + c\sqrt{\lfloor n/81 \rfloor} + \sqrt{n} && \text{Per } n \geq 1 \\ &\leq c\sqrt{n/9} + c\sqrt{n/81} + \sqrt{n} \\ &\leq c/3\sqrt{n} + c/9\sqrt{n} + \sqrt{n} \\ &= \frac{4}{9}c\sqrt{n} + \sqrt{n} \leq c\sqrt{n} \end{aligned}$$

L'ultima disequazione è vera per  $c \geq 9/5$ . Nel caso base  $T_2(1) = 1 \leq c$ , che è vera per  $c \geq 1$ . Abbiamo quindi dimostrato che  $T_2(n) = \Theta(\sqrt{n})$ .

**Equazione 3** L'ultima equazione non è ricorsiva, quindi  $T_3(n) = \Theta(\log n)$

**Soluzione finale** La funzione si comporta quindi in questo modo:

- Per valori divisibili per 4 ma non per 81,  $T(n)$  si comporta come una funzione lineare, che è la classe di complessità maggiore fra quelle presenti
- Per valori divisibili per 81  $n = 9^k, n \neq$  (divisibile per 9) con  $k$  pari,  $T(n)$  si comporta come  $\sqrt{n}$
- Per tutti gli altri valori, si comporta come  $\log n$ , che è la classe di complessità minore fra quelle presenti.

È possibile quindi dire che  $T(n) = O(n)$  e  $T(n) = \Omega(\log n)$ .

**Esercizio 2**

Esistono svariati modi per risolvere il problema. Quello che segue è il più efficiente, con complessità  $O(m + n)$ .

Se il grafo contiene un ciclo, si restituisce il valore 0 in quanto il grafo non può essere ordinato topologicamente. Si può utilizzare la funzione `hasCycle()` per identificarlo.

Altrimenti, si calcoli l'in-degree per ognuno dei nodi. Se esiste più di un nodo con in-degree 0, uno qualunque dei nodi può essere scelto per primo nell'ordinamento topologico. Si restituisca 2. Se esiste un solo nodo  $u$  con in-degree 1, si elimini dal grafo (virtualmente) e si riduca di 1 l'in-degree di tutti i vicini di  $u$ . Se l'in-degree di uno di essi scende a zero, verrà considerato al prossimo giro. Se più di un nodo vede l'in-degree scendere a zero, si deve restituire 2, perchè più ordinamenti saranno possibili. Se si esauriscono i nodi, l'algoritmo restituisce 1, perchè esiste un'unico ordinamento possibile.

---

```

count(GRAPH G)
  if hasCycle(G) then    % Se contiene un ciclo, allora non è possibile ordinare topologicamente
    | return 0

  int[] in = new int[1...G.n]          % Calcola in-degree
  for u = 1 to n do
    | in[u] = 0
  foreach u ∈ G.V() do
    | for v ∈ G.adj(u) do
      | | in[v] = in[v] + 1

  QUEUE Q = Queue()                % Cerca nodi iniziali con in-degree zero
  for u ∈ G.V() do
    | if in[u] == 0 then
      | | Q.enqueue(u)

  while Q.size() == 1 do            % Esamina il grafo
    | u = Q.dequeue()
    | for v ∈ G.adj(u) do
      | | in[v] = in[v] - 1
      | | if in[v] == 0 then
        | | | Q.enqueue(u)

  if Q.size() > 1 then              % Restituisci il valore corretto
    | return 2
  else
    | return 1

```

---

Il costo è  $O(m + n)$ .

### Esercizio 3

La seguente funziona ricorsiva restituisce il numero di alberi strutturalmente diversi, senza tener conto dell'altezza, come discusso nelle esercitazioni:

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

Aggiungendo il vincolo dell'altezza, alcuni di questi alberi non devono essere considerati. In particolare, quando l'altezza scenda a zero, solo alberi con zero o uno nodi possono essere considerati. Quando l'altezza scende sotto lo zero, si restituisce 0.

$$DP[n][h] = \begin{cases} \sum_{k=0}^{n-1} DP[k][h-1] \cdot DP[n-1-k][h-1] & n > 1 \wedge h \geq 0 \\ 1 & n \leq 0 \wedge h \geq 0 \\ 0 & h < 0 \end{cases}$$

Scriviamo il codice utilizzando memoization:

---

```

int countTree(int n, int h)


---


  int[][] DP = new int[0...n][0...n] = {-1}           % Initialized to -1
  return ctRec(n, h, DP)


---



int ctRec(int n, int h, int[][] DP)


---


  if h < 0 then
  |   return 0
  else if n ≤ 1 then
  |   return 1
  else
  |   if DP[n][h] < 0 then
  |   |   DP[n][h] = 0
  |   |   for k = 0 to n - 1 do
  |   |   |   DP[n][h] = DP[n][h] + ctRec(k, h - 1, DP) · ctRec(n - 1 - k, h - 1, DP)
  |   return DP[n][h]


---



```

---

La complessità della procedure è  $O(nh)$ .

#### Esercizio 4

Il problema può essere risolto in tempo lineare tramite programmazione dinamica. Costruiamo un vettore  $DP[i]$  che contiene il conto del numero di modi possibili in cui è possibile interpretare le cifre  $T[i \dots n]$ .

- Se incontriamo uno 0 spaiato, si restituisce 0. Questo fa sì che codici numerici malformati come 011 oppure 301 restituiscano 0;
- Nel caso  $i = n + 1$  (vettore terminato), si restituisce 1;
- Nel caso  $i = n$ , ovvero si sta considerando l'ultimo carattere e questo è diverso da 0, si restituisce 1;
- Se incontriamo uno 1 e c'è spazio per una seconda cifra, consideriamo due possibilità: interpretiamo 1 da solo (lettera "A") oppure associato ad un'altra cifra, e sommiamo i conteggi corrispondenti.
- Se incontriamo uno 2 e c'è spazio per una seconda cifra e questa seconda cifra è compresa fra 0 e 6, consideriamo due possibilità: interpretiamo 2 da solo (lettera "B") oppure associato ad un'altra cifra, e sommiamo i conteggi corrispondenti.
- Qualunque cifra diversa da 0,1,2 viene interpretata come singola, e si riporta il valore del conteggio successivo.

$$DP[i] = \begin{cases} 0 & i \leq n \wedge T[i] = 0 \\ 1 & i = n + 1 \vee (i = n \wedge T[i] > 0) \\ DP[i + 1] + DP[i + 2] & i < n \wedge T[i] = 1 \\ DP[i + 1] + DP[i + 2] & i < n \wedge T[i] = 2 \wedge 0 \leq T[i + 1] \leq 6 \\ DP[i + 1] & i < n \wedge T[i] \geq 3 \end{cases}$$

Il codice può essere scritto nel modo seguente:

---

```

int count(int[] T, int n)


---


int[] C = new int[1...n + 1]
DP[n + 1] = 1
DP[n] = if(T[n] = 0, 0, 1)
for i = n - 1 downto 1 do
    if T[i] == 0 then
        | DP[i] = 0
    if T[i] == 1 or (T[i] == 2 and 0 ≤ T[i + 1] ≤ 6) then
        | DP[i] = DP[i + 1] + DP[i + 2]
    else
        | DP[i] = DP[i + 1]
return DP[1]

```

---

La complessità computazionale è ovviamente  $\Theta(n)$ .