**Studenti:**
Martedì Gaetano (M63001226)
Salzillo Biagio (M63001227)

**Corso di Algoritmi e Strutture Dati**
Homework #1

---

1. **Esercizio 1.1. Notazione asintotica e crescita delle funzioni**

   Per ogni gruppo di funzioni, ordina le funzioni in ordine crescente di complessità asintotica (big-O):

   (a)

$$f_1(n) = n^{0,999999} \log n$$
$$f_2(n) = 10000000n$$
$$f_3(n) = 1,000001^n$$
$$f_4(n) = n^2$$

   *Svolgimento:*

   Le notazioni $O$, $\Omega$, $\Theta$ godono delle seguenti proprietà:

$$O(f_1) \cdot O(f_2) = O(f_1 \cdot f_2)$$
$$\Omega(f_1) \cdot \Omega(f_2) = \Omega(f_1 \cdot f_2)$$
$$\Theta(f_1) \cdot \Theta(f_2) = \Theta(f_1 \cdot f_2)$$

   Dunque,

   - $f_1(n) = n^{0,999999} \log n$

$$\log n = O(n^a) \qquad \forall a \in \mathbb{R}^+$$

$$\begin{cases} n^{0,999999} = O(n^{0,999999}) \\ \log n = O(n^{0,000001}) \end{cases} \implies \begin{aligned} f_1(n) &= n^{0,999999} \log n = \\ &= O(n^{0,999999}) \cdot O(n^{0,000001}) = \\ &= O(n^{0,999999} \cdot n^{0,000001}) = O(n) \end{aligned}$$

$$f_1(n) = O(n)$$

   - $f_2(n) = 10000000n$

$$f_2(n) = \Theta(n)$$

   - $f_3(n) = 1,000001^n$

$$f_3(n) = \Theta(1,000001^n)$$

- $f_4(n) = n^2$

$$f_4(n) = \Theta(n^2)$$

Concentriamoci sulla relazione che sussiste tra $f_1(n)$ e $f_2(n)$:

$$f_2(n) = \Theta(n) \implies f_2(n) = \Omega(n) \implies n = O(f_2(n))$$
$$\begin{cases} f_1(n) = O(n) \\ n = O(f_2(n)) \end{cases} \implies f_1(n) = O(f_2(n))$$

Invece, per trovare la relazione che sussiste tra $f_3(n)$ e $f_4(n)$ utilizziamo il risultato:

$$n^b = O(a^n) \qquad \forall a, b \in \mathbb{R}, \text{con } a > 1.$$

dunque,

$$f_4(n) = O(f_3(n))$$

Utilizzando la notazione $g(n) \ll f(n)$ per indicare che $g(n) = O(f(n))$ riportiamo il risultato:

$$f_1(n) \ll f_2(n) \ll f_4(n) \ll f_3(n)$$

(b)

$$f_1(n) = 2^{2^{1000000}}$$
$$f_2(n) = 2^{10000n}$$
$$f_3(n) = \binom{n}{2}$$
$$f_4(n) = n\sqrt{n}$$

*Svolgimento:*

$$f_1(n) = 2^{2^{1000000}} = \Theta(1)$$
$$f_2(n) = 2^{10000n} = 2^{10000} \cdot 2^n = \Theta(2^n)$$
$$f_3(n) = \binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)(n-2)!}{2(n-2)!} = \frac{n(n-1)}{2} = \frac{n^2 - n}{2} = \Theta(n^2)$$
$$f_4(n) = n\sqrt{n} = n \cdot n^{1/2} = n^{3/2} = \Theta(n^{1,5})$$

Dunque:

$$f_1(n) \ll f_4(n) \ll f_3(n) \ll f_2(n)$$

(c)

$$f_1(n) = n^{\sqrt{n}}$$
$$f_2(n) = 2^n$$
$$f_3(n) = n^{10} \cdot 2^{n/2}$$
$$f_4(n) = \sum_{i=1}^{n}(i+1)$$

*Svolgimento:*

Cominciamo analizzando $f_2(n)$ e $f_4(n)$, che sono le funzioni più semplici da caratterizzare:

$$f_2(n) = 2^n \qquad = \Theta(2^n)$$
$$f_4(n) = \sum_{i=1}^{n}(i+1) = \sum_{i=1}^{n} i + \sum_{i=1}^{n} 1 = \frac{n(n+1)}{2} + n = \frac{n^2+3n}{2} \qquad = \Theta(n^2)$$

$$f_4(n) \ll f_2(n)$$

Analizziamo ora $f_3(n)$:

$$f_3(n) = n^{10} \cdot 2^{n/2} = n^{10} \cdot (\sqrt{2}\,)^n$$

$$\begin{cases} n^{10} = O\Big((\sqrt{2}\,)^n\Big) = O(2^{n/2}) \\ 2^{n/2} = \Theta(2^{n/2}) \end{cases} \implies f_3(n) = O(2^{n/2}) \cdot O(2^{n/2}) = O(2^{n/2} \cdot 2^{n/2}) \quad = O(2^n)$$

$$\begin{cases} n^{10} = \Omega(n^2) \\ 2^{n/2} = \Omega(1) \end{cases} \implies f_3(n) = \Omega(n^2) \cdot \Omega(1) = \Omega(n^2 \cdot 1) \qquad = \Omega(n^2)$$

$$f_4(n) \ll f_3(n) \ll f_2(n)$$

Passiamo ora da analizzare $f_1(n)$:

$$f_1(n) = n^{\sqrt{n}} = n^{n^{1/2}}$$

Dimostriamo che $f_1(n) = \Omega(n^2)$:

$$n^{n^{1/2}} \geq n^2 \implies n^{1/2} \geq 2 \implies n \geq 4$$

dunque,

$$n^{\sqrt{n}} \geq c \cdot n^2 \qquad \forall n \geq 4, \text{ per } c = 1 \implies f_1(n) = \Omega(n^2)$$

Confrontiamo adesso $f_1(n)$ ed $f_3(n)$.
Per farlo confronteremo le due funzioni $g_1(n) = \lg(f_1(n))$ e $g_3(n) = \lg(f_3(n))$.

$$g_1(n) = \lg(f_1(n)) = \lg(n^{\sqrt{n}}) = \sqrt{n} \cdot \lg n$$
$$\begin{cases} n^{\sqrt{n}} = n^{1/2} = O(n^{1/2}) \\ \lg n = O(n^{1/2}) \end{cases} \implies g_1(n) = O(n^{1/2} \cdot n^{1/2}) = O(n)$$

$$g_3(n) = \lg(f_3(n)) = \lg(n^{10} \cdot 2^{n/2}) = \lg n^{10} + \lg 2^{n/2} = 10 \lg n + \frac{n}{2} = \Theta(n)$$
$$g_3(n) = \Theta(n) \implies g_3(n) = \Omega(n)$$

$$\begin{cases} g_1(n) = O(n) \\ g_3(n) = \Omega(n) \end{cases} \implies g_1(n) = O(g_3(n)) \implies f_1(n) = O(f_3(n))$$

Dunque,

$$f_4(n) \ll f_1(n) \ll f_3(n) \ll f_2(n)$$

2. **Esercizio 1.2. Notazione asintotica e crescita delle funzioni**
Per ognuna delle seguenti funzioni si determini se $f(n) = O(g(n))$, $g(n) = O(f(n))$ o entrambe.

(a)
$$f(n) = \frac{(n^2 - n)}{2} \qquad g(n) = 6n$$

*Svolgimento:*
$$\begin{cases} f(n) = \Theta(n^2) \\ g(n) = \Theta(n) \end{cases} \implies g(n) = O\Big(f(n)\Big)$$

(b)
$$f(n) = n + 2\sqrt{n} \qquad g(n) = n^2$$

*Svolgimento:*
$$\begin{cases} f(n) = \Theta(n) \\ g(n) = \Theta(n^2) \end{cases} \implies f(n) = O\Big(g(n)\Big)$$

(c)
$$f(n) = n \log n \qquad g(n) = n \cdot \frac{\sqrt{n}}{2}$$

*Svolgimento:*
$$f(n) : \begin{cases} n = \Theta(n) \\ \log n = O(n^{1/2}) \end{cases} \implies f(n) = O(n \cdot n^{1/2}) = O(n^{3/2})$$
$$g(n) : \begin{cases} \frac{n}{2} = \Theta(n) \\ n^{1/2} = \Theta(n^{1/2}) \end{cases} \implies g(n) = \Theta(n \cdot n^{1/2}) = \Theta(n^{3/2}) \implies g(n) = \Omega(n^{3/2})$$

$$\begin{cases} f(n) = O(n^{3/2}) \\ n^{3/2} = O(g(n)) \end{cases} \implies f(n) = O\Big(g(n)\Big)$$

(d)

$$f(n) = n + \log n \qquad g(n) = \sqrt{n}$$

*Svolgimento:*

$$\begin{cases} f(n) = \Theta(n) \\ g(n) = \Theta(n^{1/2}) \end{cases} \implies g(n) = O\Big(f(n)\Big)$$

(e)

$$f(n) = 2\log^2 n \qquad g(n) = \log n + 1$$

*Svolgimento:*

$$\begin{cases} f(n) = \Theta(\log^2 n) \\ g(n) = \Theta(\log n) \end{cases}$$

ponendo $n = 2^m$,

$$\begin{cases} h(m) = f(2^m) = \Theta(m^2) \\ k(m) = g(m) = \Theta(m) \end{cases} \implies k(m) = O\Big(h(m)\Big) \implies g(n) = O\Big(f(n)\Big)$$

(f)

$$f(n) = 4n\log n + n \qquad g(n) = \frac{n^2 - n}{2}$$

*Svolgimento:*

$$\begin{cases} f(n) = \Theta(n\log n) = O(n^2) \\ g(n) = \Theta(n^2) \end{cases} \implies f(n) = O\Big(g(n)\Big)$$

3. **Esercizio 1.3. Notazione asintotica e crescita delle funzioni**
   Si indichi se le seguenti affermazioni sono vere o false.

(a)

$$2^{n+1} = O(2^n)$$

*Svolgimento:*

$$\begin{aligned} &2^{n+1} = 2 \cdot 2^n \\ &2 \cdot 2^n \leq c \cdot 2^n \qquad \forall n \in \mathbb{N}, \text{ con } c \geq 2 \implies \\ \implies &2^{n+1} = O(2^n) \end{aligned}$$

L'affermazione è VERA.

(b)

$$2^{2n} = O(2^n)$$

*Svolgimento:*

$$2^{2n} = (2^2)^n = 4^n$$
$$\nexists\, c \in \mathbb{R}^+, \nexists\, n_0 \in \mathbb{N} : \qquad 4^n \leq c \cdot 2^n \qquad \text{con } n \geq n_0 \qquad \Longrightarrow \qquad 2^{2n} \neq O(2^n)$$

o altrimenti,

$$4^n = \omega(2^n) \qquad \Longrightarrow \qquad 2^{2n} \neq O(2^n)$$

L'afferamzione è FALSA.

4. **Esercizio 1.4. Ricorrenze**

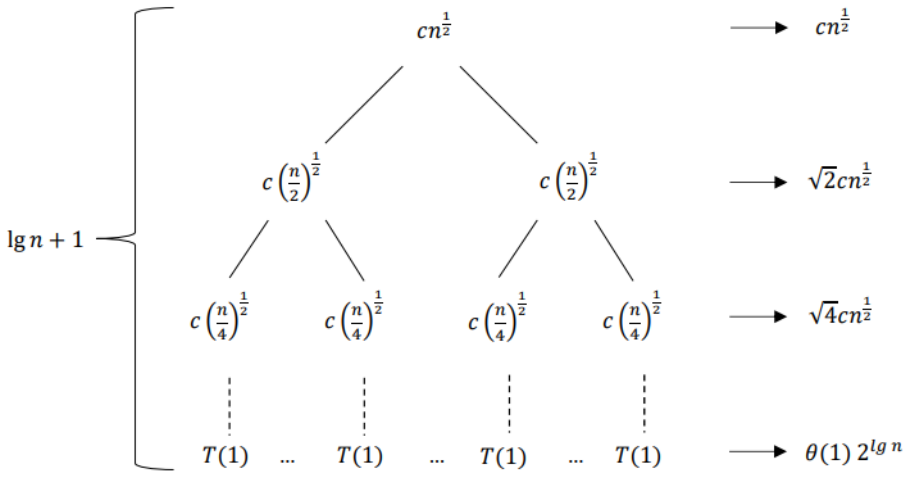
Fornire il limite inferiore e superiore per T(n) nella seguente ricorrenza, usando il metodo dell'albero delle ricorrenze ed il teorema dell'esperto se applicabile. Si fornisca il limite più stretto possibile giustificando la risposta.

(a)

$$T(n) = 2T\left(\frac{n}{2}\right) + O(\sqrt{n})$$

*Svolgimento:*

$$T(n) = 2T\left(\frac{n}{2}\right) + c\sqrt{n}$$




$$\frac{n}{2^i} = 1 \implies n = 2^i \implies i = \lg n \implies i = 0, \ldots, \lg n \implies h = \lg n + 1$$

Dunque il costo per ogni livello dell'albero è di:

$$c \cdot 2^{\frac{i}{2}} \cdot n^{\frac{1}{2}}$$

$$T(n) = \sum_{i=0}^{\lg n-1} \left( c \cdot 2^{\frac{i}{2}} \cdot n^{\frac{1}{2}} \right) + \Theta(1) \cdot 2^{\lg n} =$$

$$= c\, n^{\frac{1}{2}} \sum_{i=0}^{\lg n-1} \left( 2^{\frac{i}{2}} \right) + \Theta(1) \cdot 2^{\lg n} =$$

$$= c\, n^{\frac{1}{2}} \frac{\sqrt{2}^{\lg n} - 1}{\sqrt{2} - 1} + \Theta(1) \cdot 2^{\lg n} =$$

$$= c\, n^{\frac{1}{2}} \frac{\left(2^{\lg n}\right)^{\frac{1}{2}} - 1}{\sqrt{2} - 1} + \Theta(1) \cdot 2^{\lg n} =$$

$$= c\, n^{\frac{1}{2}} \frac{\sqrt{n} - 1}{\sqrt{2} - 1} + \Theta(1) \cdot 2^{\lg n} =$$

$$= \frac{c}{\sqrt{2} - 1}\, n^{\frac{1}{2}} (n^{\frac{1}{2}} - 1) + \Theta(1) \cdot 2^{\lg n} =$$

$$= \frac{c}{\sqrt{2} - 1} (n - n^{\frac{1}{2}}) + \Theta(1) \cdot 2^{\lg n} = \Theta(n).$$

Per questa ricorrenza abbiamo $a = 2$, $b = 2$, $f(n) = O(\sqrt{n})$, quindi $n^{\log_b a} = n^{\lg 2} = n$. Poichè $f(n) = O(\sqrt{n}) = O(n^{1/2}) = O(n^{1-1/2})$, in particolare $O(n^{1-\epsilon})$ dove , $\epsilon = 1/2$, possiamo applicare il caso 1 del teorema dell'esperto e concludere che la soluzione è $T(n) = \Theta(n)$

(b)

$$T(n) = T(\sqrt{n}) + \Theta(\log \log n)$$

*Svolgimento:*

$$\begin{cases} T(n) = T(\sqrt{n}) + \Theta(\log \log n) \\ T(1) = \Theta(1) \end{cases}$$

$$\Theta(\log \log n) = \Theta(\lg \lg n)$$

pongo $n = 2^m \iff m = \lg n$

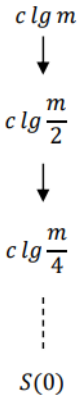
$T(2^m) = T(\sqrt{2^m}) + \Theta(\lg \lg 2^m) = T(2^{\frac{m}{2}}) + \Theta(\lg m)$

$S(m) = T(2^m) = S(\frac{m}{2}) + \Theta(\lg m)$

$n = 1 \implies 2^m = 1 \implies m = 0$

$T(1) = \Theta(1) \implies S(0) = \Theta(1)$

$$\begin{cases} S(0) = \Theta(1) \\ S(m) = S(\frac{m}{2}) + \Theta(\lg m) \end{cases} \implies \begin{cases} S(0) = \Theta(1) \\ S(m) = S(\frac{m}{2}) + c \lg m) \end{cases}$$

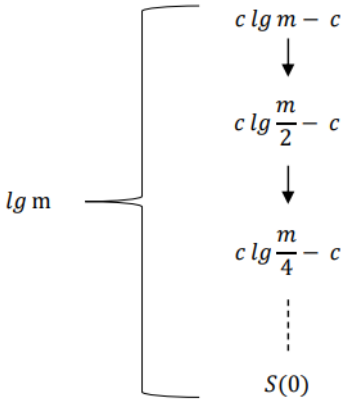



$c \lg \frac{m}{2^i} = 0$ *impossibile*

A questo punto possiamo percorrere due strade alternative:
1) Cambiare il caso base
2) Modificare il costo associato alle operazioni di Divide e Combine

Per questo esercizio la scelta ricade sulla seconda strada.

$$\begin{cases} S(0) = \Theta(1) \\ S(m) = S(\frac{m}{2}) + \Theta(\lg m) \end{cases} \implies \begin{cases} S(0) = \Theta(1) \\ S(m) = S(\frac{m}{2}) + c \lg m - c \end{cases}$$




$c \lg \frac{m}{2^i} - c = 0 \implies c \lg \frac{m}{2^i} = c$

$\implies \lg \frac{m}{2^i} = 1 \implies \frac{m}{2^i} = 2 \implies m = 2^{i+1} \implies i + 1 = \lg m \implies i = \lg m - 1$

$i = 0, 1, \ldots, \lg m - 1 \implies h = \lg m$

$$S(m) = \sum_{i=0}^{\lg m - 2} (c \lg \frac{m}{2^i}) - c + \Theta(1) =$$

$$= c \sum_{i=0}^{\lg m - 2} (\lg \frac{m}{2^i}) - c \sum_{i=0}^{\lg m - 2} 1 + \Theta(1) =$$

$$= c \sum_{i=0}^{\lg m - 2} (\lg m - \lg 2^i) - c(\lg m - 2) + \Theta(1) =$$

$$= c \lg m \sum_{i=0}^{\lg m - 2} 1 - c \sum_{i=0}^{\lg m - 2} i - c \lg m + 2c + \Theta(1) =$$

$$= c \lg m (\lg m - 2) - c \frac{(\lg m - 1)(\lg m - 2)}{2} - c \lg m + \Theta(1) =$$

$$= c \lg^2 m - 2c \lg m - c \frac{\lg^2 m - 3 \lg m + 2}{2} - c \lg m + \Theta(1) = \Theta(\lg^2 m)$$

$$\begin{cases} S(m) = \Theta(\lg^2 m) \\ m = \lg n \end{cases} \implies T(n) = \Theta(\lg^2 \lg n)$$

*Metodo dell'esperto:*

$$S(m) = S(\frac{m}{2}) + \Theta(\lg m)$$

Per questa ricorrenza abbiamo $a = 1$, $b = 2$, $f(m) = \Theta(\lg m)$, quindi $m^{\log_b a} = m^{\lg 1} = m^0 = 1$. Dunque, avendo $f(m) = c \lg m$, possiamo dire che:

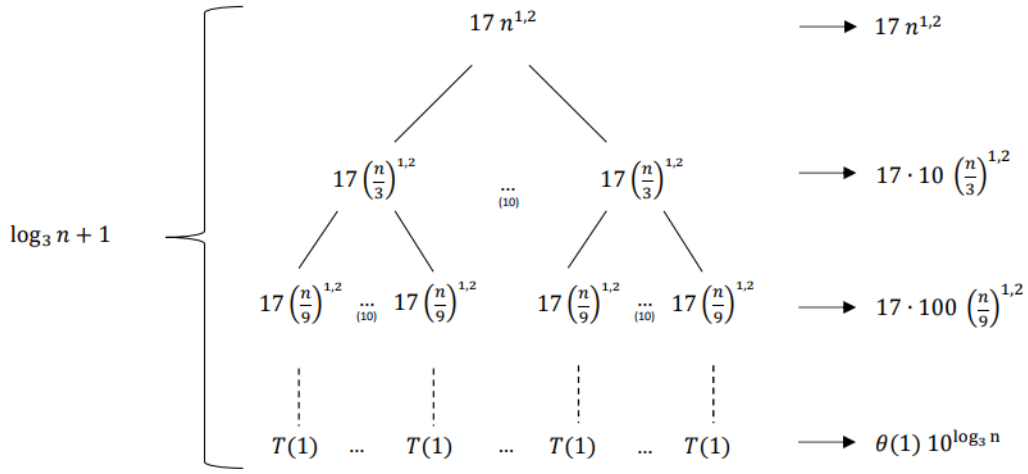
1) $f(m) = O(m^{0-\epsilon})$? No, perchè $\lg m = \omega(m^0)$
2) $f(m) = \Theta(m^0) = \Theta(1)$? No
3) $f(m) = \Omega(m^{0+\epsilon})$? No

Dunque per questa ricorrenza il metodo dell'esperto non può essere applicato.

(c)

$$T(n) = 10\, T\Big(\frac{n}{3}\Big) + 17\, n^{1,2}$$

*Svolgimento:*




$$\frac{n}{3^i} = 1 \implies n = 3^i \implies i = \log_3 n \implies i = 0, \ldots, \log_3 n \implies h = \log_3 n + 1$$

Dunque il costo per ogni livello dell'albero è di:

$$17 \cdot 10^i \left(\frac{n}{3^i}\right)^{1,2} \qquad con\ i = 0, \ldots, \log_3 n - 1$$

$$T(n) = \sum_{i=0}^{\log_3 n - 1} \left[17 \cdot 10^i \left(\frac{n}{3^i}\right)^{1,2}\right] + \Theta(1) \cdot 10^{\log_3 n} =$$

$$= 17\,n^{1,2} \sum_{i=0}^{\log_3 n - 1} \left(\frac{10}{3^{1,2}}\right)^i + \Theta(1) \cdot 10^{\log_3 n} =$$

$$= 17\,n^{1,2} \frac{1 - \left(\frac{10}{3^{1,2}}\right)^{\log_3 n}}{1 - \left(\frac{10}{3^{1,2}}\right)} + \Theta(1) \cdot 10^{\log_3 n} =$$

$$= \frac{17}{\frac{10}{3^{1,2}} - 1} n^{1,2} \left[\left(\frac{10}{3^{1,2}}\right)^{\log_3 n} - 1\right] + \Theta(1) \cdot 10^{\log_3 n} =$$

$$= \alpha\, n^{1,2} \left[\frac{10^{\log_3 n}}{(3^{\log_3 n})^{1,2}} - 1\right] + \Theta(1) \cdot 10^{\log_3 n} = \qquad (dove\ \alpha \approx 10,12)$$

$$= \alpha\, n^{1,2} \left[\frac{10^{\log_3 n}}{n^{1,2}} - 1\right] + \Theta(1) \cdot 10^{\log_3 n} =$$

$$= \alpha\, 10^{\log_3 n} - \alpha\, n^{1,2} + \Theta(1) \cdot 10^{\log_3 n} =$$

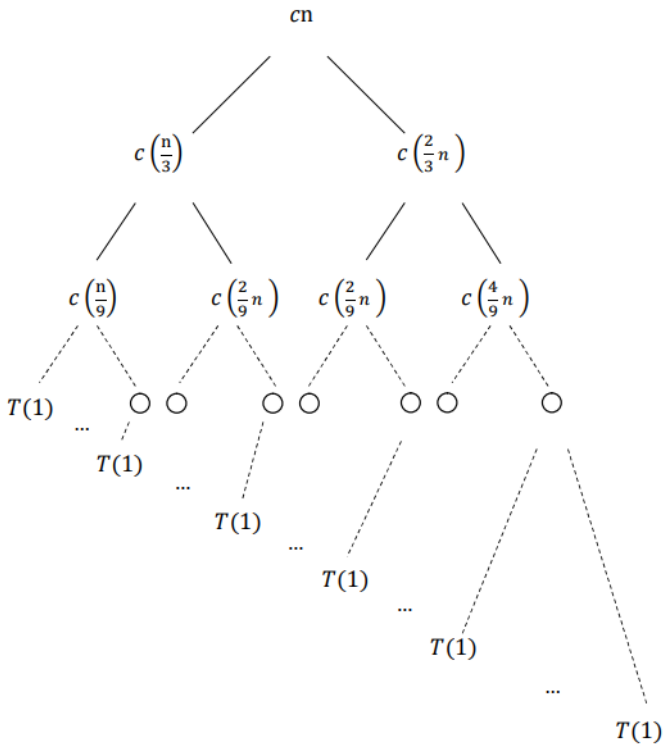
$$= \alpha\, n^{\log_3 10} - \alpha\, n^{1,2} + \Theta(1) \cdot 10^{\log_3 n} = \Theta(n^{\log_3 10}) \qquad (dove\ \log_3 10 \approx 2,1)$$

Per questa ricorrenza abbiamo $a = 10$, $b = 3$, $f(n) = 17\,n^{1,2} = \Theta(n^{1,2})$, quindi $n^{\log_b a} = n^{\log_3 10}$, dove $\log_3 10 \approx 2,1$. In particolare $f(n) = O(n^{\log_3 10 - \epsilon})$ dove, $0 < \epsilon \leq 0,8$, possiamo

applicare il caso 1 del teorema dell'esperto e concludere che la soluzione è $T(n) = \Theta(n^{\log_3 10})$.

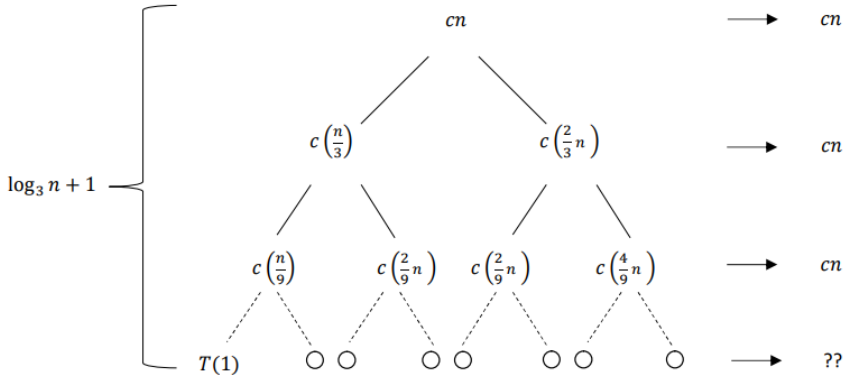
(d) Utilizzando l'albero di ricorsione, dimostrate che la soluzione della ricorrenza $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2}{3}n\right) + cn$, dove c è una costante, è $\Omega(n \lg n)$

*Svolgimento:*

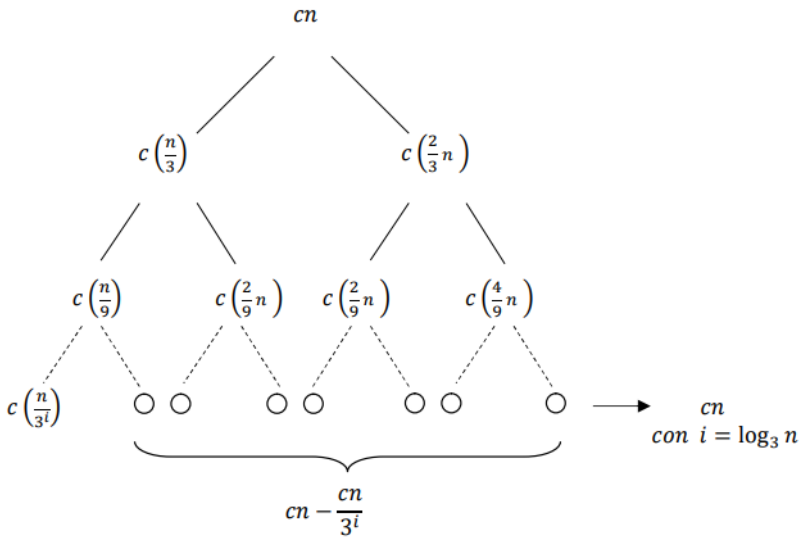



L'albero è un albero binario non completo. L'analisi precisa dei costi su un tale albero è alquanto complicata, ma dato che cerchiamo un limite asintotico inferiore ($\Omega$), eseguiamo un'analisi dei costi sul tracciamneto di tale albero binario, in particolare dalla radice fino al livello in cui l'albero è ancora un albero binario completo. Tale livello coincide con il livello in cui la quantità $\frac{n}{3^i} = 1$, dato che siamo sicuri che tale percorso sarà il primo ad arrivare ad un nodo foglia $T(1)$.

$$\frac{n}{3^i} = 1 \implies n = 3^i \implies i = \log_3 n \implies i = 0, \ldots, \log_3 n \implies h = \log_3 n + 1$$



Vediamo se con tale semplificazione riusciamo a dimostare che $T(n) = \Omega(n \log n)$. Calcoliamo i costi dell'ultimo livello:




*Osservazione: se non avessimo il nodo foglia ma tutti nodi interni allora avremmo un costo pari a cn*

$$i = \log_3 n \implies cn - \frac{cn}{3^{\log_3 n}} = cn - c = c(n-1)$$

A seguito del troncamento dell'albero avremo la seguente disequazione:

$$T(n) \geq \sum_{i=0}^{\log_3 n - 1} (cn) + c(n-1) + \Theta(1) =$$

$$= cn \sum_{i=0}^{\log_3 n - 1} (1) + c(n-1) + \Theta(1) =$$

$$= cn \log_3 n + cn - c + \Theta(1) \qquad dove\ [cn - c + \Theta(1)] \geq 0$$

$$\implies T(n) \geq cn \log_3 n \implies T(n) = \Omega(n \log_3 n) = \Omega(n \log n)$$

5. **Problema 1.1**

Sia data una lista di 2n nomi memorizzati in una Linked List. Si implementi un algoritmo con complessità O(n) che modifichi la lista in modo tale da invertire l'ordine degli elementi della seconda metà della lista (dunque, in modo tal che l'(n+1)-esimo elemento diventi l'ultimo elemento (posizione 2n), l'"(n+2)-esimo diventi il penultimo e così via, l'ultimo elemento diventi l'(n+1)-esimo. L'algoritmo non deve creare nuovi nodi nella lista né istanziare nuove strutture dati.

Si alleghi al PDF un file editabile riportante l'implementazione in un linguaggio a scelta, corredato da almeno tre casi di test (dunque, una sequenza di stringhe in questo caso, con il corrispondente output atteso).

*Svolgimento:*
La soluzione fa uso di una lista lineare singolarmente concatenata.

L'algoritmo risolutivo si basa sull'idea che se si è in grado di invertire una lista per intero, in un tempo $O(n)$, allora è possibile risolvere il problema come segue:

(a) scorro la lista fino all'elemento $n$-esimo;

(b) considero la sottolista che va dal nodo $n+1$ al nodo $2n$ come una lista a sé stante, e richiamo su di essa la funzione REVERSE-LIST($head$), in grado di invertire una lista passandole la testa, e restituendo la nuova testa della lista invertita;

(c) collego il nodo $n$-esimo alla testa della sottolista invertita, ossia il nodo $2n$-esimo.

*Algoritmo:*

```
1: function REVERSE-HALF-LIST(head, n)                                  Costo
2:     x ← head                                                           1
3:     ▷ Scorro la lista fino al nodo n
4:     for 1 to n − 1 do                                                  n
5:         x ← next[x]                                                    n − 1
6:     end for
7:
8:     ▷ Richiamo l'algoritmo di reverse-list passando come testa il nodo n+1
9:     k ← Reverse-List(next[x])                                          ?
10:
11:    ▷ Aggiorno il successivo del nodo n con la testa della sottolista invertita
12:    next[x] ← k                                                        1
13: end function
```

L'algoritmo REVERSE-LIST utilizzato per invertire una lista, è un algoritmo ricorsivo che funziona nel seguente modo:

(a) si stacca il nodo testa dal resto della lista;

(b) si richiama l'algoritmo sulla parte rimanente della lista;

(c) si collega l'ultimo nodo della lista invertita con la testa estratta in precedenza;

(d) si pone il nodo successivo alla testa pari a NIL in modo da trasformare la vecchia testa nella coda della nuova lista.

*Algoritmo:*

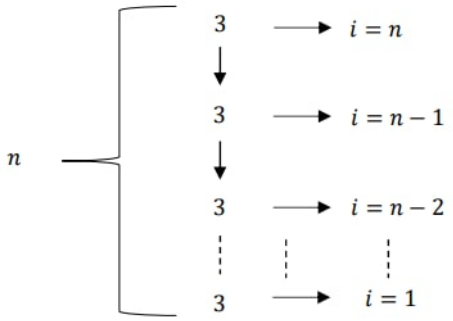
```
1: function REVERSE-LIST(head)                                          Costo
2:     ▷ Caso base: ho una lista con un solo elemento che diventa la nuova testa
3:     if next[h] = NIL then                                              1
4:         k ← head                                                       1
5:     else
6:         ▷ (b):
7:         k ←Reverse-List(next[head])
8:         ▷ (c):
9:         next[next[head]] ← head                                        1
10:        ▷ (d):
11:        next[h] = NIL                                                  1
12:    end if
13:    ▷ Ritorna la testa della sotto-lista invertita
14:    return k                                                           1
15: end function
```

Se l'algoritmo ricorsivo REVERSE-LIST ha una complessità pari a $\Theta(n)$ allora il problema è risolto.

Scriviamo la ricorrenza che modella la complessità dell'algoritmo:

$$\begin{cases} T(1) = 3 \\ T(n) = T(n-1) + 3 \end{cases} \implies \begin{cases} T(1) = \Theta(1) \\ T(n) = T(n-1) + \Theta(1) \end{cases}$$

Risolviamo la ricorrenza utilizzando l'albero delle ricorrenze:




$$T(n) = \sum_{i=1}^{n} 3 = 3n = \Theta(n)$$

Abbiamo dimostrato che REVERSE-LIST ha una complessità pari a $\Theta(n)$ , dunque anche REVERSE-HALF-LIST ha complessità $\Theta(n)$.

6. **Problema 1.2**

Sia data una sequenza di $n$ numeri interi, memorizzata in un vettore A. Si implementi un algoritmo che esegua in tempo $O(nlogn)$ per determinare il numero di elementi minori uguali dell'elemento $i$-esimo, riportando il risultato corrispondente in un secondo vettore B (sempre di dimensione $n$) alla posizione $i$.

Si alleghi al PDF un file editabile riportante l'implementazione in un linguaggio a scelta, corredato da almeno tre casi di test (dunque, una sequenza di numeri interi in questo caso, con il corrispondente output atteso).

*Svolgimento:*
Iniziamo a ragionare nell'ipotesi in cui tutti gli elementi di A siano distinti. Rimuoveremo tale ipotesi successivamente.

La soluzione prevede di utilizzare un terzo vettore di appoggio C e procedere con i seguenti passi:

(a) copiare il vettore A in C;

(b) ordinare il vettore C con un algoritmo di complessità $O(nlogn)$ (es. MERGE-SORT);

(c) eseguire un ciclo che per ogni elemento $i$-esimo del vettore A, ne ricerchi la sua posizione all'interno del vettore C e copi tale valore nella posizione $i$ di B.

Tale algoritmo si basa sull'osservazione che la posizione di un elemento all'interno di un vettore ordinato coincide con il numero di elementi minori o uguali ad esso (sotto l'ipotesi di elementi distinti in A).

Attenzione: il passo (c) dell'algoritmo prevede di scorrere tutto il vettore A, e per ogni elemento eseguire una ricerca all'interno del vettore C; siamo sicuri di rispettare il vincolo che l'algoritmo

sia un $O(nlogn)$? Lo scorrimento del vettore A comporta una complessità pari a $\Theta(n)$, dunque ci occorre un algoritmo di ricerca che sia al più un $O(\log n)$. Utilizzeremo dunque l'algoritmo di ricerca binaria che è proprio un $O(\log n)$, ed è applicabile perché la ricerca avviene all'interno del vettore C, che è ordinato.

*Algoritmo:*

| | | Costo |
|---|---|---|
| 1: | **function** NUM-ELEMENTI-MIN$(A, B, n)$ | |
| 2: | dichiara $C[1..n]$ | 1 |
| 3: | **for** $i \leftarrow 1$ to $n$ **do** | $n+1$ |
| 4: | $C[i] \leftarrow A[i]$ | $n$ |
| 5: | **end for** | |
| 6: | | |
| 7: | Merge-Sort$(C, 1, n)$ | $\Theta(n \log n)$ |
| 8: | | |
| 9: | **for** $i \leftarrow 1$ to $n$ **do** | $n+1$ |
| 10: | $j \leftarrow$Binary-Search$(A[i], C, 1, n)$ | $n \cdot O(\log n) = O(n \log n)$ |
| 11: | $B[i] \leftarrow j$ | $n$ |
| 12: | **end for** | |
| 13: | **end function** | |

```
1: function MERGE(A, p, q, r)
2: end function
```

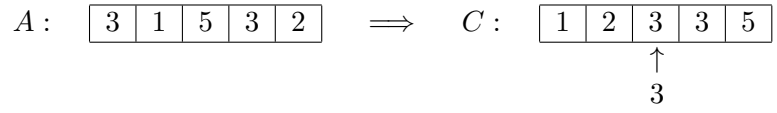
[PSEUDOCODICE MERGE-SORT]

Sommando i costi delle singole istruzioni dell'algoritmo otteniamo una complessità pari ad $O(\log n)$.

Cosa succede se rimuoviamo l'ipotesi che tutti gli elementi di A siano distinti?

Non è più vero che la posizione di un elemento all'interno di un vettore ordinato è pari al numero di elementi minori o uguali ad esso.

*Esempio:*




Il numero di elementi minori o uguali di 3 in A non è pari a 3, ma a 4.

Questo avviene perché prendendo la posizione di uno qualsiasi degli elementi ripetuti all'interno di C, si potrebbero non considerare nel conteggio tutti gli eventuali elementi con lo stesso valore presenti nelle locazioni successive.

*Soluzione:* in caso di elementi ripetuti, quando effettuo la ricerca all'interno di C, restituisco la posizione dell'elemento con indice posizionale più alto tra quelli di ugual valore.

*Esempio:*


Per fare ciò occorre modificare leggermente l'algoritmo di BINARY-SEARCH, in modo tale che la ricerca non termini appena trovo l'elemento cercato, ma continui sulla metà superiore dell'array per cercare se ci sono altri elementi uguali a quello cercato in locazioni successive.

*Algoritmo:*

```
1: function BINARY-SEARCH(x, A, p, r)
2:     i ← −1
3:     while p ≤ r do
4:         q ← ⌊(p+r)/2⌋
5:         if x < A[q] then
6:             r ← q − 1
7:         else
8:             p ← q + 1
9:             if x = A[q] then
10:                i ← q
11:            end if
12:        end if
13:    end while
14:    return i
15: end function
```

Questa modifica comporta una variazione nella complessità dell'algoritmo. Nella sua versione originale è un $O(\log n)$, in quanto nel caso peggiore continua a dividere l'array fino a $\lg n$ volte, fino ad ottenere dei sotto-array di dimensione 1. Ma ciò succede solo nel caso peggiore, perché nel caso l'algoritmo riesca a trovare prima l'elemento, il numero suddivisioni sarà minore di $\lg n$. Nella versione modificata, tuttavia, l'algoritmo non termina quando trova l'elemento cercato, ma solo e unicamente quando si arriva a sotto-array di dimensione pari a 1. Quindi la complessità sarà un $\Theta(\log n)$.

Tale variazione non altera la complessità dell'algoritmo NUM-ELEMENTI-MIN, che resta un $O(n \log n)$.

7. **Problema 1.3**

*Descrizione.*
Se inseriamo un insieme di n elementi in un albero di ricerca binario (binary search tree, BST) utilizzando TREE-INSERT, l'albero risultante potrebbe essere molto sbilanciato. Tuttavia, ci si aspetta che i BST costruiti casualmente siano bilanciati (ossia ha un'altezza attesa O(lg n)). Pertanto, se vogliamo costruire un BST con altezza attesa O(lg n) per un insieme fisso di elementi, potremmo permutare casualmente gli elementi e quindi inserirli in quell'ordine nell'albero. Cosa succede se non abbiamo tutti gli elementi a disposizione in una sola volta? Se riceviamo gli elementi uno alla volta, possiamo ancora costruire casualmente un albero di ricerca binario da essi? Nel seguito è proposta una struttura dati che risponde affermativamente a questa domanda. Un treap è un albero binario di ricerca che usa una strategia diversa per ordinare i nodi. Ogni elemento x nell'albero ha una chiave key[x]. Inoltre, assegniamo priority[x], che è un numero casuale scelto indipendentemente per ogni x. Assumiamo che tutte le priorità siano distinte e anche che tutte le chiavi siano distinte. I nodi del treap sono ordinati in modo che (1) le chiavi obbediscano alla proprietà del binary search tree e (2) le priorità obbediscano alla proprietà min-heap order dell'heap. In altre parole:

- se v è un figlio sinistro di u, allora $key[v] < key[u]$;
- se v è un figlio destro di u, allora $key[v] > key[u]$;
- se v è un figlio di u, allora $priority(v) > priority(u)$.

(Questa combinazione di proprietà è il motivo per cui l'albero è chiamato "treap": ha caratteristiche sia di un albero di ricerca binario che di un heap). È utile pensare ai treaps in questo modo: supponiamo di inserire i nodi $x_1, x_2, \ldots x_n$, ciascuno con una chiave associata, in un treap in ordine arbitrario. Quindi il treap risultante è l'albero che si sarebbe formato se i nodi fossero stati inseriti in un normale albero binario di ricerca nell'ordine dato dalle loro priorità (scelte casualmente). In altre parole, $priority[x_i] < priority[x_j]$ significa che $x_i$ è effettivamente inserito prima di $x_j$. Per inserire un nuovo nodo x in un treap esistente, si assegna dapprima ad

x una priorità casuale priority[x]. Quindi si chiama l'algoritmo di inserimento, che chiameremo TREAPINSERT, il cui funzionamento è illustrato nella Figura 1.

*Quesito.* Fornire il codice della procedura TREAP-INSERT in un linguaggio a scelta, allegando un file editabile. Suggerimento: effettuare il consueto inserimento del BST, ed eseguire le rotazioni per ripristinare la proprietà del min-heap (min-heap order).

*Svolgimento:*
L'algoritmo non fa altro che inserire il nodo $x$ come in un semplice BST, e nel caso sia violata la proprietà del *min-heap*, esegue una serie di rotazioni per far risalire il nodo $x$ attraverso l'albero, finché la proprietà non è ripristinata.

*Algoritmo:*

```
1: function TREAP-INSERT(T, x)
2:     y ← NIL
3:     z ← root[T]
4:     while z ≠ NIL do
5:         y ← z
6:         if key[x] < key[z] then
7:             z ← left[z]
8:         else
9:             z ← right[z]
10:        end if
11:        parent[x] ← y
12:        if y = NIL then
13:            root[T] ← x
14:        else
15:            if key[x] < key[y] then
16:                left[y] ← x
17:            else
18:                right[y] ← x
19:            end if
20:        end if
21:        Treap-Insert-Fixup(T,x)
22:    end while
23: end function
```

```
1: function TREAP-INSERT-FIXUP(T, x)
2:     y ← parent[x]
3:     while y ≠ NILANDpriority[x] < priority[y] do
4:         if x = left[y] then
5:             Right-rotate(T,y)
6:         else
7:             Left-rotate(T,y)
8:         end if
9:         y ← parent[x]
10:    end while
11: end function
```

```
1: function LEFT-ROTATE(T, x)
2:     y ← right[x]
3:     right[x] ← left[y]
4:     if left[y] ≠ NIL then
5:         parent[left[y]] ← x
```

```
    end if
    parent[y] ← parent[x]
    if parent[x] ≠ NIL then
        if x = left[parent[x]] then
            left[parent[x]] ← y
        else
            right[parent[x]] ← y
        end if
    else
        root[T] ← y
    end if
    left[y] ← x
    parent[x] ← y
end function
```

```
function RIGHT-ROTATE(T, x)
    y ← left[x]
    left[x] ← right[y]
    if right[y] ≠ NIL then
        parent[right[y]] ← x
    end if
    parent[y] ← parent[x]
    if parent[x] ≠ NIL then
        if x = left[parent[x]] then
            left[parent[x]] ← y
        else
            right[parent[x]] ← y
        end if
    else
        root[T] ← y
    end if
    right[y] ← x
    parent[x] ← y
end function
```

*Nota:* nella traccia del problema non è richiesto di produrre alcun output, né di fornire alcun input. Tuttavia, nel codice C relativo a tale problema sono presenti delle procedure dedicate alla stampa di un output, utilizzate a fine di testing da chi ha sviluppato il codice. Tali procedure sono state lasciate nel file editabile da consegnare, insieme ad un caso di test, nel caso in cui possano rivelarsi utili.

La formattazione dell'input è si fatta:
$k_1 \quad k_2 \quad k_3 \quad \ldots \quad k_n \quad 0$
dove $k_i \in \{a, b, c, \ldots, z\}$, e 0 è usato come carattere di terminazione.

In output viene stampata una riga per ogni livello dell'albero, e ogni coppia di nodi di una riga inferiore rappresenta la coppia di figli di un nodo della riga superiore. L'assenza di un nodo figlio nell'albero è indicata con il nodo fittizio (NULL). Ogni nodo è stampato nel formato:

(<chiave>:<priorità>, <chiave-del-padre>)

*Esempio:*

(e:13,NIL)
(b:21,e) (f:60,e)
(a:64,b) (d:63,b) (NULL) (NULL)
(NULL) (NULL) (c:77,d) (NULL) (NULL) (NULL) (NULL) (NULL)