

# Algoritmi

UniVR - Dipartimento di Informatica

**Fabio Irimie**

1° Semestre 2024/2025

# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
1.1	Confronto tra algoritmi . . . . .	2
1.2	Rappresentazione dei dati . . . . .	2
<b>2</b>	<b>Calcolo della complessità</b>	<b>2</b>
2.1	Linguaggi di programmazione . . . . .	2
2.1.1	Blocchi iterativi . . . . .	3
2.1.2	Blocchi condizionali . . . . .	3
2.1.3	Blocchi iterativi . . . . .	3
2.2	Esempio . . . . .	4
2.3	Ordine di grandezza . . . . .	4
2.3.1	Esempi di dimostrazioni . . . . .	7
<b>3</b>	<b>Studio degli algoritmi</b>	<b>9</b>
3.1	Insertion sort . . . . .	10
3.2	Fattoriale . . . . .	11
3.3	Teorema dell'esperto . . . . .	14
3.4	Merge sort . . . . .	16
3.5	Heap . . . . .	17
3.5.1	Proprietà . . . . .	18
3.5.2	Heap sort . . . . .	19
3.6	Quick sort . . . . .	19
3.7	Algoritmi di ordinamento non per confronti . . . . .	20
3.7.1	Counting sort . . . . .	21

# 1 Introduzione

Un algoritmo è una sequenza **finita** di **istruzioni** volta a risolvere un problema. Per implementarlo nel pratico si scrive un **programma**, cioè l'applicazione di un linguaggio di programmazione, oppure si può descrivere in modo informale attraverso del **pseudocodice** che non lo implementa in modo preciso, ma spiega i passi per farlo.

Ogni algoritmo può essere implementato in modi diversi, sta al programmatore capire qual'è l'opzione migliore e scegliere in base alle proprie necessità.

## 1.1 Confronto tra algoritmi

Ogni algoritmo si può confrontare con gli altri in base a tanti fattori, come:

- **Complessità**: quanto ci vuole ad eseguire l'algoritmo
- **Memoria**: quanto spazio in memoria occupa l'algoritmo

## 1.2 Rappresentazione dei dati

Per implementare un algoritmo bisogna riuscire a strutturare i dati in maniera tale da riuscire a manipolarli in modo efficiente.

# 2 Calcolo della complessità

La complessità di un algoritmo mette in relazione il numero di istruzioni da eseguire con la dimensione del problema, e quindi è una funzione che dipende dalla dimensione del problema.

La **dimensione del problema** è un insieme di oggetti adeguato a dare un'idea chiara di quanto è grande il problema da risolvere, ma sta a noi decidere come misurare il problema.

Ad esempio una matrice è più comoda da misurare come il numero di righe e il numero di colonne, al posto di misurarla come il numero di elementi totali.

La complessità di solito si calcola come il **caso peggiore**, cioè il limite superiore di esecuzione dell'algoritmo.

## 2.1 Linguaggi di programmazione

Ogni linguaggio di programmazione è formato da diversi blocchi:

1. **Blocco iterativo**: un tipico blocco di codice eseguito sequenzialmente e tipicamente finisce con un punto e virgola.
2. **Blocco condizionale**: un blocco di codice che viene eseguito solo se una condizione è vera.
3. **Blocco iterativo**: un blocco di codice che viene eseguito ripetutamente finché una condizione è vera.

Questi sono i blocchi base della programmazione e se riusciamo a calcolare la complessità di ognuno di questi blocchi possiamo calcolare più facilmente la complessità di un intero algoritmo.

### 2.1.1 Blocchi iterativi

$$\begin{array}{l} I_1 \quad c_1(n) \\ I_2 \quad c_2(n) \\ \vdots \quad \vdots \\ I_l \quad c_l(n) \end{array}$$

Se ogni blocco ha complessità  $c_1(n)$ , allora la complessità totale è data da:

$$\sum_{i=1}^l c_i(n)$$

### 2.1.2 Blocchi condizionali

$$\begin{array}{l} \text{IF cond} \quad c_{cond}(n) \\ I_1 \quad c_1(n) \\ \text{ELSE} \\ I_2 \quad c_2(n) \end{array}$$

La complessità totale è data da:

$$c(n) = c_{cond}(n) + \max(c_1(n), c_2(n))$$

A volte la condizione è un test sulla dimensione del problema e in quel caso si può scrivere una complessità più precisa.

### 2.1.3 Blocchi iterativi

$$\begin{array}{l} \text{WHILE cond} \quad c_{cond}(n) \\ I \quad c_0(n) \end{array}$$

Si cerca di trovare un limite superiore  $m$  al limite di iterazioni.

Di conseguenza la complessità totale è data da:

$$c_{cond}(n) + m(c_{cond}(n) + c_0(n))$$

## 2.2 Esempio

**Esempio 2.1.** Calcoliamo la complessità della moltiplicazione tra 2 matrici:

$$A_{n \times m} \cdot B_{m \times l} = C_{n \times l}$$

L'algoritmo è il seguente:

```
1
2  for i <- 1 to n // n ( 5 ml + 4l + 2) + n + 1
3    for j <- 1 to l // l (5m + 2 + 1) + 1 + 1
4      c[i][j] <- 0
5      for k <- 1 to m // (m + 1 + m(4))
6        // 3 (moltiplicazione, somma e assegnamento)
7        // 1 (incremento for)
8        c[i][j] += a[i][k] * b[k][j]
9
```

Partiamo calcolando la complessità del ciclo for più interno. Non ha senso tenere in considerazione tutti i dati, ma solo quelli rilevanti. In questo caso avremo:

$$(m + 1 + m(4)) = 5m + 1$$

Questa complessità contiene informazioni poco rilevanti perchè possono far riferimento alla velocità della cpu e un millisecondo in più o in meno non cambia nulla se teniamo in considerazione solo l'incognita abbiamo:

$$m$$

Questo semplifica molto i calcoli, rendendo meno probabili gli errori. Siccome la complessità si calcola su numeri molto grandi, le costanti piccole prima o poi verranno tolte perchè poco influenti.

La complessità totale alla fine sarebbe stata:

$$5nml + 4ml + 2n + n + 1$$

Ma ciò che ci interessa veramente è:

$$5nml + 4ml + 2n + n + 1$$

Se non consideriamo le costanti inutili, la complessità finale è:

$$nml$$

Nella maggior parte dei casi ci si concentra soltanto sull'ordine di grandezza della complessità, e non sulle costanti.

## 2.3 Ordine di grandezza

L'ordine di grandezza è una funzione che approssima la complessità di un algoritmo:

$$f \in O(g)$$

$$\exists c > 0 \exists \bar{n} \forall n \geq \bar{n} \ f(n) \leq cg(n)$$

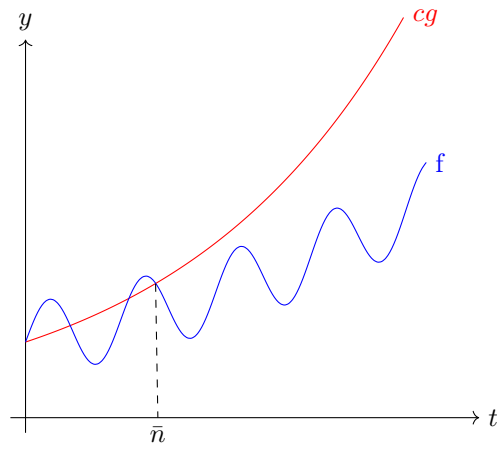


Figura 1: Esempio di funzione  $f \in O(g)$

$$f \in \Omega(g)$$

$$\exists c > 0 \exists \bar{n} \forall n \geq \bar{n} \ f(n) \geq cg(n)$$

$$f \in \Theta(g)$$

$$f \in O(g) \wedge f \in \Omega(g)$$

Per gli algoritmi:

**Definizione 2.1.**

$$A \in O(f)$$

So che l'algoritmo  $A$  termina entro il tempo definito dalla funzione  $f$ . Di conseguenza se un algoritmo termina entro un tempo  $f$  allora sicuramente termina entro un tempo  $g$  più grande. Ad esempio:

$$A \in O(n) \Rightarrow A \in O(n^2)$$

Questa affermazione è **corretta**, ma **non accurata**.

$$A \in \Omega(f)$$

Significa che esiste uno schema di input tale che se  $g(n)$  è il numero di passi necessari per risolvere l'istanza  $n$  allora:

$$g \in \Omega(f)$$

Quindi l'algoritmo non termina in un tempo minore di  $f$ .

Calcolando la complessità si troverà lo schema di input tale che:

$$g \in O(f)$$

cioè il limite superiore di esecuzione dell'algoritmo.

Successivamente ci si chiede se esistono algoritmi migliori e si troverà lo schema di input tale che:

$$g \in \Omega(f)$$

cioè il limite inferiore di esecuzione dell'algoritmo.

Se i due limiti coincidono allora:

$$g \in \Theta(f)$$

abbiamo trovato il tempo di esecuzione dell'algoritmo.

**Teorema 2.1** (Teorema di Skolem). Se c'è una formula che vale coi quantificatori esistenziali, allora nel linguaggio si possono aggiungere delle costanti al posto delle costanti quantificate e assumere che la formula sia valida con quelle costanti.

### 2.3.1 Esempi di dimostrazioni

**Esempio 2.2.** È vero che  $n \in O(2n)$ ?  
Se prendiamo  $c = 1$  e  $\bar{n} = 1$  allora:

$$n \leq c2n$$

Quindi è vero

**Esempio 2.3.** È vero che  $2n \in O(n)$ ?  
Se prendiamo  $c = 2$  e  $\bar{n} = 1$  allora:

$$2n \leq 2n$$

Quindi è vero



**Esempio 2.4.** È vero che  $f \in O(g) \iff g \in \Omega(f)$ ?

Dimostro l'implicazione da entrambe le parti:

- $\rightarrow$ : Usando il teorema di Skolem:

$$\forall n \geq \bar{n} \quad f(n) \leq cg(n)$$

Trasformo la disequazione:

$$\forall n \geq \bar{n} \quad \frac{f(n)}{c} \leq g(n)$$

$$\forall n \geq \bar{n} \quad g(n) \geq \frac{f(n)}{c}$$

$$\forall n \geq \bar{n} \quad g(n) \geq \frac{1}{c}f(n) \quad \square$$

Se la definizione di  $\Omega(g)$  è:

$$\exists c' > 0 \exists \bar{n}' \quad \forall n \geq \bar{n}' \quad f(n) \geq c'g(n)$$

sappiamo che:

$$c' = \frac{1}{c}$$

- $\leftarrow$ : Usando il teorema di Skolem:

$$\forall n \geq \bar{n}' \quad g(n) \geq c'f(n)$$

Trasformo la disequazione:

$$\forall n \geq \bar{n}' \quad \frac{g(n)}{c'} \geq f(n)$$

$$\forall n \geq \bar{n}' \quad f(n) \leq \frac{1}{c'}g(n) \quad \square$$

**Esempio 2.5.**

$$f_1 \in O(g) \quad f_2 \in O(g) \Rightarrow f_1 + f_2 \in O(g)$$

Dimostrazione:

Ipotesi

$$\bar{n}_1 c_1 \quad \forall n > n_1 \quad f_1(n) \leq c_1 g(n)$$

$$\bar{n}_1 c_2 \quad \forall n > n_2 \quad f_2(n) \leq c_2 g(n)$$

$$f_1(n) + f_2(n) \leq (c_1 + c_2)g(n) \quad \square$$

Quindi:

$$c = (c_1 + c_2)$$

$$\bar{n} = \max(\bar{n}_1, \bar{n}_2)$$

**Esempio 2.6.** Se

$$f_1 \in O(g_1) \quad f_2 \in O(g_2)$$

è vero che:

$$f_1 \cdot f_2 \in O(g_1 \cdot g_2)$$

Dimostrazione:

Ipotesi

$$\bar{n}_1 c_1 \quad \forall n > \bar{n}_1 \quad f_1(n) \leq c_1 g_1(n)$$

$$\bar{n}_2 c_2 \quad \forall n > \bar{n}_2 \quad f_2(n) \leq c_2 g_2(n)$$

$$f_1(n) \cdot f_2(n) \leq (c_1 \cdot c_2)(g_1(n) \cdot g_2(n)) \quad \square$$

Quindi:

$$c = c_1 \cdot c_2$$

$$\bar{n} = \max(\bar{n}_1, \bar{n}_2)$$

### 3 Studio degli algoritmi

Il problema dell'ordinamento si definisce stabilendo la relazione che deve esistere tra **input** e **output** del sistema.

- **Input:** Sequenza  $(a_1, \dots, a_n)$  di oggetti su cui è definita una relazione di ordinamento, cioè l'unico modo per capire la differenza tra due oggetti è confrontarli.
- **Output:** Permutazione  $(a'_1, \dots, a'_n)$  di  $(a_1, \dots, a_n)$  tale che:

$$\forall i < j \quad a'_i \leq a'_j$$

L'obiettivo è trovare un algoritmo che segua la relazione di ordinamento definita e risolva il problema nel minor tempo possibile.

### 3.1 Insertion sort

Divide la sequenza in due parti:

- **Parte ordinata:** Sequenza di elementi ordinati
- **Parte non ordinata:** Sequenza di elementi non ordinati

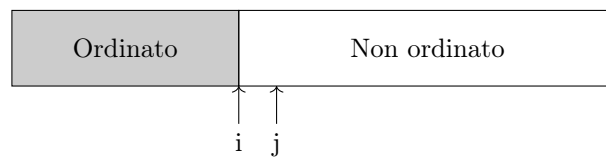


Figura 2: Parte ordinata e non ordinata

Pseudocodice:

```

1 insertion_sort(A)
2   for j <- 2 to length[A] // A sinistra di j e tutto ordinato-
3     key <- A[j]           // |
4     i <- j - 1           // | 0(n)
5     while i > 0 and A[i] > key // -- |
6       A[i + 1] <- A[i]   // | 0(n)
7       i--              // -- |
8     A[i + 1] <- key      // -- -----

```

La complessità di questo algoritmo è:

$$O(n^2)$$

Per capirlo è sufficiente guardare il numero di cicli nidificati e quante volte eseguono il codice all'interno.

Se l'array è già ordinato la complessità è:

$$\Omega(n)$$

Con l'input peggiore possibile la complessità è:

$$\Omega(n^2)$$

di conseguenza, visto che vale  $O(n^2)$  e  $\Omega(n^2)$  vale:

$$\Theta(n^2)$$

Quanto spazio in memoria utilizza questo algoritmo?

- Variabile j
- Variabile i
- Variabile key

A prescindere da quanto è grande l'array utilizzato, di conseguenza la memoria utilizzata è costante.

- **Ordinamento in loco:** se la quantità di memoria extra che deve usare non dipende dalla dimensione del problema allora si dice che l'algoritmo è in loco.
- **Ordinamento non in loco:** se la quantità di memoria extra che deve usare dipende dalla dimensione del problema allora si dice che l'algoritmo è non in loco.
- **Stabilità:** La posizione relativa di elementi uguali non viene modificata

L'insertion sort ordina in loco ed è stabile.

## 3.2 Fattoriale

```

1 Fatt(n)
2   if n = 0
3     ret 1
4   else
5     ret n * Fatt(n - 1)

```

L'argomento della funzione ci fa capire la complessità dell'algoritmo:

$$T(n) = \begin{cases} 1 & \text{se } n = 0 \\ T(n-1) + 1 & \text{se } n > 0 \end{cases}$$

Con problemi ricorsivi si avrà una complessità con funzioni definite ricorsivamente. Questo si risolve induttivamente:

$$\begin{aligned}
 T(n) &= 1 + T(n-1) \\
 &= 1 + 1 + T(n-2) \\
 &= 1 + 1 + 1 + T(n-3) \\
 &= \underbrace{1 + 1 + \dots + 1}_i + T(n-i)
 \end{aligned}$$

La condizione di uscita è:  $n - i = 0 \quad n = i$

$$\begin{aligned}
 &= \underbrace{1 + 1 + \dots + 1}_n + T(n-n) \\
 &= n + 1 = \Theta(n)
 \end{aligned}$$

Questo si chiama passaggio iterativo.

### Esempio 3.1.

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$$

Questa funzione si può riscrivere come:

$$T(n) = \begin{cases} \text{Costante} & \text{se } n < a \\ 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n & \text{se } n \geq a \end{cases}$$

Se la complessità fosse già data bisognerebbe soltanto verificare se è corretta. Usando il metodo di sostituzione:

$$T(n) = cn \log n$$

sostituiamo nella funzione di partenza:

$$\begin{aligned} T(n) &= 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \\ &\leq 2c\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \log \left\lfloor \frac{n}{2} \right\rfloor + n \\ &\leq 2c \frac{n}{2} \log \frac{n}{2} + n \\ &= cn \log n - cn \log 2 + n \\ &\stackrel{?}{\leq} cn \log n \quad \text{se } n - cn \log 2 \leq 0 \end{aligned}$$

$$c \geq \frac{n}{n \log 2} = \frac{1}{\log 2}$$

Il metodo di sostituzione dice che quando si arriva ad avere una disequazione corrispondente all'ipotesi, allora la soluzione è corretta se soddisfa una certa ipotesi.

### Esempio 3.2.

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \in O(n)$$

$$T(n) \leq cn$$

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\ &\leq c\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\ &= c\left(\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil\right) + 1 \\ &= cn + 1 \stackrel{?}{\leq} cn \end{aligned}$$

Il metodo utilizzato non funziona perchè rimane l'1 e non si può togliere in alcun modo. Per risolvere questo problema bisogna risolverne uno più forte:

$$T(n) \leq cn - b$$

$$\begin{aligned}
T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\
&\leq c\left(\left\lfloor \frac{n}{2} \right\rfloor\right) - b + c\left(\left\lceil \frac{n}{2} \right\rceil\right) - b + 1 \\
&= c\left(\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil\right) - 2b + 1 \\
&= cn - 2b + 1 \stackrel{?}{\leq} cn - b \\
&= \underbrace{cn - b}_{\leq 0} + \underbrace{1 - b}_{\leq 0} \leq cn - b \quad \text{se } b \geq 1
\end{aligned}$$

Se la proprietà vale per questo problema allora vale anche per il problema iniziale perchè è meno forte.

### Esempio 3.3.

$$\begin{aligned}
T(n) &= 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + n \\
&= n + 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) \\
&= n + 3\left(\left\lfloor \frac{n}{4} \right\rfloor + 3T\left(\left\lfloor \frac{\left\lfloor \frac{n}{4} \right\rfloor}{4} \right\rfloor\right)\right) \\
&= n + 3\left\lfloor \frac{n}{4} \right\rfloor + 3^2T\left(\left\lfloor \frac{n}{4^2} \right\rfloor\right) \\
&\leq n + 3\left\lfloor \frac{n}{4} \right\rfloor + 3^2\left(\left\lfloor \frac{n}{4^2} \right\rfloor + 3T\left(\left\lfloor \frac{\left\lfloor \frac{n}{4^2} \right\rfloor}{4} \right\rfloor\right)\right) \\
&= n + 3\left\lfloor \frac{n}{4} \right\rfloor + 3^2\left\lfloor \frac{n}{4^2} \right\rfloor + 3^3T\left(\left\lfloor \frac{n}{4^3} \right\rfloor\right) \\
&= n + 3\left\lfloor \frac{n}{4} \right\rfloor + \dots + 3^{i-1}\left\lfloor \frac{n}{4^{i-1}} \right\rfloor + 3^iT\left(\left\lfloor \frac{n}{4^i} \right\rfloor\right)
\end{aligned}$$

Per trovare il caso base poniamo l'argomento di T molto piccolo:

$$\begin{aligned}
\frac{n}{4^i} &< 1 \\
4^i &> n \\
i &> \log_4 n
\end{aligned}$$

L'equazione diventa:

$$\leq n + 3\left\lfloor \frac{n}{4} \right\rfloor + \dots + 3^{\log_4 n - 1}\left\lfloor \frac{n}{4^{\log_4 n - 1}} \right\rfloor + 3^{\log_4 n} c$$

Si può togliere l'approssimazione per difetto per ottenere un maggiorante:

$$\begin{aligned} &\leq n \left( 1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \dots + \left(\frac{3}{4}\right)^{\log_4 n - 1} \right) + 3^{\log_4 n} c \\ &\leq n \left( \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i \right) + c 3^{\log_4 n} \end{aligned}$$

Per capire l'ordine di grandezza di  $3^{\log_4 n}$  si può scrivere come:

$$3^{\log_4 n} = n^{(\log_n 3^{\log_4 n})} = n^{\log_4 n \cdot \log_n 3} = n^{\log_4 3}$$

Quindi la complessità è:

$$= O(n) + O(n^{\log_4 3})$$

Si ha che una funzione è uguale al termine noto della funzione originale e l'altra che è uguale al logaritmo dei termini noti. Se usassimo delle variabili uscirebbe:

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + f(n) \\ &= O(f(n)) + O(n^{\log_b a}) \end{aligned}$$

### 3.3 Teorema dell'esperto

**Teorema 3.1** (Teorema dell'esperto o Master theorem). Per un'equazione di ricorrenza del tipo:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Si distinguono 3 casi:

- $f(n) \in O(n^{\log_b a - \varepsilon})$  allora  $T(n) \in \Theta(n^{\log_b a})$
- $f(n) \in \Theta(n^{\log_b a})$  allora  $T(n) \in \Theta(f(n) \log n)$
- $f(n) \in \Omega(n^{\log_b a + \varepsilon})$  allora  $T(n) \in \Theta(f(n))$

**Esempio 3.4.**

$$T(n) = 9T\left(\frac{n}{3}\right) + n$$

Applico il teorema dell'esperto:

$$a = 9$$

$$b = 3$$

$$f(n) = n$$

$$n^{\log_b a} = n^{\log_3 9} = n^2$$

Verifico se esiste un  $\varepsilon$  tale che:

$$n \in O(n^{2-\varepsilon})$$

prendo  $\varepsilon = -\frac{1}{2}$  e verifico:

$$n \in O(n^2 \cdot n^{-\frac{1}{2}})$$

Quindi ho trovato il caso 1 del teorema dell'esperto.

$$T(n) \in \Theta(n^2)$$

### Esempio 3.5.

$$T(n) = T\left(\frac{2n}{3}\right) + 1$$

Applico il teorema dell'esperto:

$$a = 1$$

$$b = \frac{3}{2}$$

$$f(n) = n^0$$

$$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0$$

Si nota che le due funzioni hanno lo stesso ordine di grandezza, quindi siamo nel secondo caso del teorema dell'esperto.

$$T(n) \in \Theta(\log n)$$

### Esempio 3.6.

$$T(n) = 3T\left(\frac{n}{4}\right) + n \log n$$

Applico il teorema dell'esperto:

$$a = 3$$

$$b = 4$$

$$f(n) = n \log n$$

$$n^{\log_b a} = n^{\log_4 3}$$

$$n \log n \in \Omega(n^{\log_4 3})$$

Esiste un  $\varepsilon$  tale che:

$$n \log n \in \Omega(n^{\log_4 3 + \varepsilon})$$

perchè basta che sia compreso tra  $\log_4 3$  e 1.



Quindi siamo nel terzo caso del teorema dell'esperto.

$$T(n) \in \Theta(n \log n)$$

**Esempio 3.7.**

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

Applico il teorema dell'esperto:

$$a = 2$$

$$b = 2$$

$$f(n) = n \log n$$

$$n^{\log_b a} = n^{\log_2 2} = n$$

$$n \log n \in \Omega(n)$$

Verifico se esiste un  $\varepsilon$ , quindi divido per  $n$ :

$$\log n \in \Omega(n^\varepsilon)$$

Quindi si nota che questa proprietà non è verificata, quindi non si può applicare il teorema dell'esperto.

### 3.4 Merge sort

Questo algoritmo di ordinamento è basato sulla tecnica divide et impera:

- **Divide:** Dividi il problema in sottoproblemi più piccoli
- **Impera:** Risolvi i sottoproblemi in modo ricorsivo
- **Combina:** Unisci le soluzioni dei sottoproblemi per risolvere il problema originale

Questo algoritmo divide la sequenza in due parti uguali e le ordina separatamente, successivamente le unisce in modo ordinato. La complessità, considerando il merge con complessità lineare, risulta:

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

Applicando il teorema dell'esperto si ottiene:

$$a = 2$$

$$b = 2$$

$$f(n) = n$$

$$n^{\log_b a} = n$$

$$n \in \Theta(n)$$

Quindi siamo nel secondo caso del teorema dell'esperto:

$$T(n) \in \Theta(n \log n)$$

Definizione del merge sort:

```

1 // A: Array da ordinare
2 // P: Indice di partenza
3 // r: Indice di arrivo
4 merge_sort(A, p, r)           // --
5     if p < r                  // |
6         q <- floor((p + r) / 2) // |
7         merge_sort(A, p, q)    // | 0(n log n)
8         merge_sort(A, q + 1, r) // |
9         merge(A, p, q, r)      // --

1 // A: Array da ordinare
2 // P: Indice di partenza
3 // q: Indice di mezzo
4 // r: Indice di arrivo
5 merge(A, p, q, r)
6     i <- 1
7     j <- p
8     k <- q + 1
9     // Ordina gli elementi di A in B
10    while(j <= q and k <= r) // --
11        if j <= q and (k > r or A[j] <= A[k]) // |
12            B[i] <- A[j] // |
13            j++ // |
14        else // | 0(n)
15            B[i] <- A[k] // |
16            k++ // |
17            i++ // --
18
19    // Copia gli elementi di B in A
20    for i <- 1 to r - p + 1 // -|
21        A[p + i - 1] <- B[i] // -| 0(n)

```

L'algoritmo è stabile perchè non vengono scambiati elementi uguali e non è in loco perchè utilizza un array di appoggio.

### 3.5 Heap

È un albero semicompleto (ogni nodo ha 2 figli ad ogni livello tranne l'ultimo che è completo solo fino ad un certo punto) in cui i nodi contengono oggetti con relazioni di ordinamento.

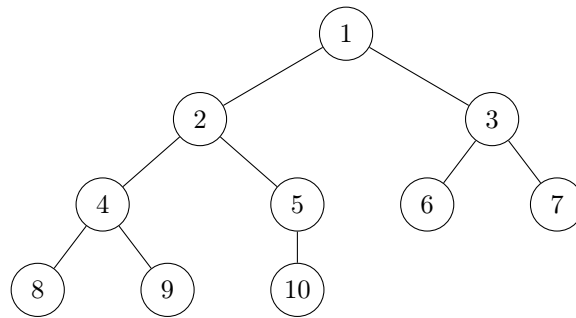


Figura 3: Heap con l'indice di un array associato ai nodi

### 3.5.1 Proprietà

$\forall$  nodo il contenuto del nodo è  $\geq$  del contenuto dei figli. Per calcolare il numero di nodi di un albero binario si usa la formula:

$$N = 2^0 + 2^1 + 2^2 + \dots + 2^{h-1} = \frac{1 - 2^h}{1 - 2} = 2^h - 1$$

dove  $h$  è l'altezza dell'albero. Il numero di foglie di un albero sono la metà dei nodi.

Definiamo una funzione che "aggiusta" i figli di un nodo per mantenere la proprietà di heap:

```

1  heapify(A, i) // O(n)
2  l <- left[i] // Indice del figlio sinistro (2i)
3  r <- right[i] // Indice del figlio destro (2i+1)
4  if l < H.heap_size and H[l] > H[i]
5      largest <- l
6  else
7      largest <- i
8
9  if r < H.heap_size and H[r] > H[largest]
10     largest <- r
11 if largest != i
12     swap(H[i], H[largest])
13     heapify(H, largest)

```

Ora si vuole definire una funzione che costruisce un heap da un array:

```

1  build_heap(A) // O(n)
2  heapsize(a) <- length[A]
3  for i <- floor(length[A]/2) downto 1
4      heapify(A, i)

```

Una volta definito un heap si possono fare diverse operazioni, come ad esempio estrarre il nodo massimo:

```

1  extract_max(A)
2  H[1] <- H[H.heap_size]
3  H.heap_size <- H.heap_size - 1
4  heapify(H, 1)

```

### 3.5.2 Heap sort

Heap sort è un algoritmo di ordinamento basato su heap.

```
1 heap_sort(A) // O(n log n)
2   build_heap(A) // n
3   for i <- length[A] downto 2
4     scambia(A[i], A[1])
5     heapsize(A)--
6     heapify(A, 1) // log i
```

La complessità dell'algoritmo è  $O(n \log n)$ , ma più precisamente:

$$\sum_{i=1}^n \log i = \log \prod_{i=1}^n i = \log n! = \Theta(\log n^n) = \Theta(n \log n)$$

Per la formula di Stirling  $n!$  ha ordine di grandezza  $n^n$ . Questo algoritmo è in loco e stabile.

Il caso pessimo è un array ordinato al contrario.

### 3.6 Quick sort

Il concetto di questo algoritmo è quello di mettere prima in disordine l'algoritmo e poi ordinarlo. L'algoritmo divide l'array in 2 parti e ordina ricorsivamente le due parti; a quel punto l'array è ordinato.

```
1 // A: Array da ordinare
2 // p: Indice di partenza
3 // r: Indice di arrivo
4 quick_sort(A, p, r)
5   if p < r // Ordina solo se l'array ha piu' di un elemento
6     q <- partition(A, p, r) // Dividi l'array in due parti
7     quick_sort(A, p, q) // Ordina sinistra
8     quick_sort(A, q + 1, r) // Ordina destra

1 partition(A, p, r)
2   x <- A[p] // Elemento perno (o pivot)
3   i <- p - 1
4   j <- r + 1
5   while true
6     repeat // Ripete finche' la condizione non e' soddisfatta
7       j-- // n/2
8     until A[j] <= x // Trova un elemento che non puo' stare a
9     destra
10    repeat
11      i++ // n/2
12    until A[i] >= x // Trova un elemento che non puo' stare a
13    sinistra
14    if i < j
15      swap(A[i], A[j]) // n/2
16    return j // alla fine j puntera' all'ultimo elemento di
17    sinistra
```

Questo algoritmo è in loco e non è stabile. La sua complessità nel caso peggiore è:

$$\begin{aligned}
 T(n) &= T(\text{partition}) + T(q) + T(n - q) \\
 &= n + T(q) + T(n - q) \\
 &= n + \cancel{T(1)} + T(n - 1) \\
 &= n + T(n - 1) \\
 &= \Theta(n^2)
 \end{aligned}$$

Il caso peggiore è un array già ordinato.

Mediamente ci si aspetta che l'array venga diviso in 2 parti molto simili, quindi la complessità è  $O(n \log n)$  perchè:

$$0 < c < 1$$

$$T(n) = n + T(cn) + T((1 - c)n)$$

```

1  rand_partition(A, p, r)
2      i <- random(p, r)
3      swap(A[i], A[p])
4      return partition(A, p, r)

```

La complessità è la media di tutte le complessità con probabilità  $\frac{1}{n}$

$$\begin{aligned}
 T(n) &= n + \frac{1}{n} (T(1) + T(n - 1)) + \frac{n - 1}{n} (T(2) + T(n - 2)) \\
 &\quad + \dots + \frac{1}{n} (T(n - 1) + T(1))
 \end{aligned}$$

$$\begin{aligned}
 T(n) &= n + \frac{1}{n} \sum_i (T(i) + T(n - i)) \\
 &= n + \frac{2}{n} \sum_i T(i)
 \end{aligned}$$

### 3.7 Algoritmi di ordinamento non per confronti

Si possono avere algoritmi di ordinamento con complessità  $< n \log n$ ?

Qualsiasi algoritmo che ordina per confronti deve fare almeno  $n \log n$  confronti nel caso pessimo

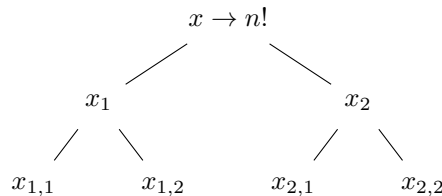


Figura 4: Heap con l'indice di un array associato ai nodi

Le foglie rappresentano ogni singola combinazione possibile. Il numero di foglie è  $n!$  e l'altezza sarà sempre

$$h \geq \log_2 n! = n \log n$$

### 3.7.1 Counting sort

Si vogliono ordinare  $n$  numeri con valori da 1 a  $k$ . L'idea di questo algoritmo è quella di creare un'array che contiene il numero di occorrenze di un certo valore (rappresentato dall'indice).

```
1 counting_sort(A, k)
2   for i <- 1 to k // k
3     C[i] <- 0 // Inizializzazione di un array a 0
4
5   for j <- 1 to length[A] // n
6     C[A[j]]++ // Conteggio delle occorrenze
7
8   // k
9   for i <- 2 to k           // In ogni indice metto il numero di
10    C[i] <- C[i] + C[i-1] // elementi minori o uguali
11                           // al numero dell'indice
12  // Alla fine l'array C conterra' l'ultima posizione di occorrenza
13  // per ogni elemento
14
15  for j <- length[A] downto 1 // n
16    B[C[A[j]]] <- A[j] // Inserimento dell'elemento in posizione
17    C[A[j]]--          // Decremento della posizione di occorrenza
```

La complessità di questo algoritmo è  $O(n + k)$  e siccome sappiamo che  $k$  è una costante fissata a priori la complessità è  $O(n)$ . Non è in loco, ma è stabile