

# Algoritmi e Strutture Dati

Leonardo Baldo

# Contents

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Induzione . . . . .	3
1.2	Ricorrenza . . . . .	3
<b>2</b>	<b>Insertion Sort</b>	<b>4</b>
2.1	Pseudocodice . . . . .	4
2.2	Correttezza . . . . .	4
<b>3</b>	<b>Merge Sort</b>	<b>5</b>
3.1	Pseudocodice . . . . .	6
3.2	Complessità . . . . .	6
<b>4</b>	<b>Master Theorem</b>	<b>7</b>
<b>5</b>	<b>Heap Sort</b>	<b>8</b>
5.1	Alberi . . . . .	8
5.2	Alberi completi . . . . .	8
5.3	Alberi quasi completi . . . . .	8
5.4	Heap . . . . .	8
5.5	Pseudocodice . . . . .	9
5.6	Complessità . . . . .	9

# 1 Introduzione

**Algoritmo:** procedura che descrive tramite passi elementari come risolvere un problema (tramite un modello di computazione).

Uno stesso problema può essere risolto da diversi algoritmi. Di ogni algoritmo siamo interessati a conoscere:

- correttezza
- stabilità
- complessità

## 1.1 Induzione

L'induzione si struttura con:

- Un caso base  $P(0)$ .
- Un'ipotesi induttiva (se vale per  $P(n)$  allora vale anche per  $P(n + 1)$ ).

Matematicamente parlando significa che:

- Normalmente ho una formula, per esempio  $n = 1$ .
- Se vale per  $n = 1$ , provo con  $n = 2$ .
- Se funziona anche con  $n = 2$ , vuol dire che per  $P(n)$  varrà anche  $P(n + 1)$  e tutti i successivi.

Si ha anche un'ulteriore variante, l'induzione forte:

- $U$  contiene 1 oppure 0.
- Se  $U$  contiene tutti i numeri minori di  $n$  allora contiene anche  $n$ .

La parola "forte" è legata al fatto che questa formulazione richiede delle ipotesi apparentemente più forti e stringenti per inferire che l'insieme  $U$  coincida con  $N$  per ammettere un numero nell'insieme è richiesto infatti che tutti i precedenti ne facciano parte (e non solo il numero precedente).

## 1.2 Ricorrenza

**Relazione di ricorrenza** è una formula ricorsiva che esprime il termine  $n$ -esimo di una successione in relazione ai precedenti. La relazione si dice di ordine  $r$  se il termine  $n$ -esimo è espresso in funzione al più dei termini  $(n - 1), \dots, (n - r)$ .

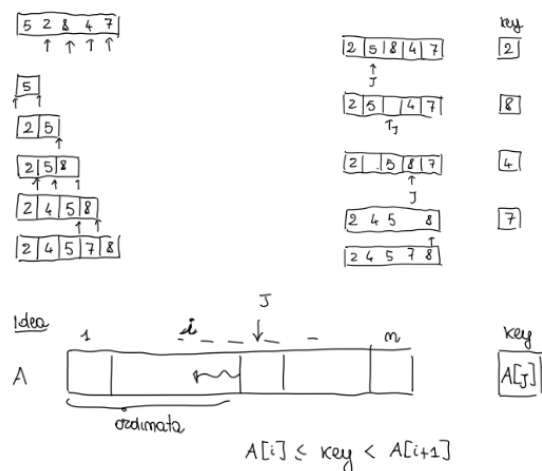
## 2 Insertion Sort

L'array viene virtualmente diviso in una parte ordinata e una non ordinata. I valori della parte non ordinata vengono prelevati e collocati nella posizione corretta della parte ordinata.

Caratteristiche dell'ordinamento per inserzione:

- Questo algoritmo è uno dei più semplici e di semplice implementazione.
- Fondamentalmente, l'ordinamento per inserzione è efficiente per piccoli valori di dati
- L'ordinamento per inserzione è di natura adattativa, cioè è adatto a insiemi di dati già parzialmente ordinati.

Quello che sostanzialmente fa è esaminare gli elementi a coppie e ordinarli gradualmente per come si presentano.



### 2.1 Pseudocodice

```

INSERTION-SORT(A)
1  n = A.length
2  for j = 2 to n
3      key = A[j]
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i - 1
8      A[i + 1] = key
    
```

### 2.2 Correttezza

Definiamo i seguenti passi per A da indice 1 ad indice  $j - 1$

**Inizializzazione:**  $j = 2, A[1, 1]$  ordinato

**Mantenimento:** inserisce  $A[j]$  in  $A[1 \dots j - 1]$  ordinato

**Cancellazione:**  $j = n + 1$

### 3 Merge Sort

Adotta l'approccio divide et impera, quindi:

- divide: prende il problema originale e lo divide in problemi più piccoli.
- impera: ricorsivamente, risolve i sottoproblemi; se abbastanza piccolo, si risolve subito.
- combina: le soluzioni di  $P_1, \dots, P_k$  si combinano in una soluzione di  $P$ .

Quindi, mergesort adotta i seguenti passi:

- dividere l'array in due parti
- ordina i sottoarray
- fonde i sottoarray ordinati



### 3.1 Pseudocodice

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = (p+r)/2
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q+1, r)
5      MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
1  n1 = q - p + 1
2  n2 = r - q
3  for i = 1 to n1
4      L[i] = A[p+i-1]
5  for j = 1 to n2
6      R[j] = A[q+j]
7  L[n1+1] = R[n2+1] = infinity
8  i = j = 1
9  for k = p to r
10     if L[i] <= R[j]
11         A[k] = L[i]
12         i = i + 1
13     else // L[i] > R[j]
14         A[k] = R[j]
15         j = j + 1
```

### 3.2 Complessità

$A[p, \dots, k-1]$  contiene  $L[1, \dots, i-1]$  e  $R[i, \dots, j-1]$   $A[p, \dots, k-1] \leq L[1, \dots, n_1-1]$  e  $R[i, \dots, n_2-1]$  è ordinato

**Inizializzazione:**  $k = p$  e  $A[p, \dots, k-1] = A[p, p-1]$

**Mantenimento:** Divisione in due metà progressive dell'array

**Conclusione:**  $K = r + 1$  e  $A[p, \dots, r]$  è ordinato e contiene i più piccoli elementi  $L[1, \dots, n_1 + 1]$  e  $R[1, \dots, n_2 + 1]$

La dimostrazione del perché sia corretto viene fatta per induzione: se vale per il primo elemento, vale per tutti i casi successivi.

## 4 Master Theorem

Normalmente, dato un problema con size  $n$  lo dividiamo in sottoproblemi con dimensione  $n/b$  ed  $f(n)$  costo della "combina" di entrambe le cose. Otteniamo come ricorrenza (con  $a \geq 1, b > 1$ ):

$$T(n) = aT(n/b) + f(n)$$

Esso stabilisce che, avendo a che fare con le ricorrenze di questo tipo, avremmo tre casi:

- Se il costo della risoluzione dei sotto-problemi ad ogni livello aumenta di un certo fattore, il valore di  $f(n)$  diventerà polinomialmente più piccolo di  $n^{\log_b a}$ . Pertanto, la complessità temporale è opprressa dal costo dell'ultimo livello, vale a dire  $n^{\log_b a}$ .

$$f(n) = O(n^{\log_b a - \epsilon}) \quad (\epsilon > 0)$$

- Se il costo per risolvere il sotto-problema ad ogni livello è quasi uguale, allora il valore di  $f(n)$  sarà  $n^{\log_b a}$ . Pertanto, la complessità temporale sarà  $f(n)$  volte il numero totale di livelli  $n^{\log_b a} \cdot \log(n)$ .

$$f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log n)$$

- Se il costo della risoluzione dei sottoproblemi ad ogni livello diminuisce di un certo fattore, il valore di  $f(n)$  diventerà polinomialmente più grande di  $n^{\log_b a}$ . Pertanto, la complessità temporale è opprressa dal costo di  $f(n)$ .

$$\left. \begin{array}{l} f(n) = \Omega(n^{\log_b a + \epsilon}) \quad (\epsilon > 0) \\ \exists 0 < k < 1 : a \cdot f(n/b) \leq k \cdot f(n) \end{array} \right\} \Rightarrow T(n) = \Theta(f(n))$$

L'idea è che la funzione si ripeta come rapporto ricorsivamente, questo corrisponde alla divisione in due parti e ancora in due parti, ecc.

$$T(n) = f(n) + a \cdot T\left(\frac{n}{b}\right) = f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \dots + a^{\log_b n} f\left(\frac{n}{b^{\log_b n}}\right) = n^{\log_b a}$$



Confronta tra di loro  $f(n)$  e  $n^{\log_b a}$ :

- se vince  $f(n) \Rightarrow \Theta(f(n))$
- se pareggiano  $\Rightarrow \Theta(n^{\log_b a} \cdot \log n)$
- se vince  $n^{\log_b a} \Rightarrow \Theta(n^{\log_b a})$

## 5 Heap Sort

### 5.1 Alberi

- binario: ogni nodo ha al massimo due figli.
- altezza: distanza dalla radice alla foglia più distante.
- ordinato: ogni nodo ha un figlio minore o uguale a se stesso.

### 5.2 Alberi completi

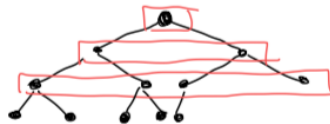
**Albero binario completo:** ogni nodo non foglia ha due figli e ogni cammino radice-foglia ha la stessa lunghezza.

$$\# \text{ nodi} = \sum_{i=0}^h 2^i - 1 \quad (h = \text{altezza})$$



### 5.3 Alberi quasi completi

**Albero quasi completo:** ogni livello è completo, tranne eventualmente l'ultimo con foglie tutte a sx.



### 5.4 Heap

**Heap:** albero binario ordinato quasi completo, ma implementato come se fosse un array.

**Caratteristiche:**

- $A[1]$  è radice
- Un generico nodo sta in posizione  $A[i]$ .
- Un nodo  $A[2i]$  è il figlio sinistro del nodo  $A[i]$
- Un nodo definito come nodo parent/genitore sta in posizione  $A[i/2]$
- Lo spazio occupato effettivamente è  $A.size$ , la lunghezza dell'array è  $A.length$ .





**Max Heap** Il Max Heap è un Heap dove:

- $\forall A[i], A[i] \geq$  nodi discendenti (quindi,  $A.Left[i], A.Right[i]$ )
- $\forall A[i], A[i] \leq$  nodi antenati (quindi,  $A[i] \leq A.Parent[i]$ )



## 5.5 Pseudocodice

```

MAX-HEAPIFY(A)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heapsize and A[l] > A[i]
4      max = l
5  else
6      max = i
7  if r ≤ A.heapsize and A[r] > A[max]
8      max = r
9  if max ≠ i
10     A[i] ↔ A[max]
11     MAX-HEAPIFY(A, max)
    
```

## 5.6 Complessità

Dipende dall'albero, il quale avrà  $n$  elementi possibili:

- la complessità  $O(h)$ , con  $h$  altezza del sottoalbero, alternativamente  $O(\log(n))$ .

$$\left. \begin{array}{l} n \geq 2^{h-1} + 1 = 2^h \\ h \leq \log_2 n \end{array} \right\} \Rightarrow O(h) \cong O(\log n)$$

- se invece fosse visto come ricorrenza, si può dimostrare con il Master Theorem