

# 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
<b>6</b>	<b>Quick Sort</b>	<b>10</b>
6.1	Pseudocodice . . . . .	10
6.2	Passaggi . . . . .	10
<b>7</b>	<b>Counting Sort</b>	<b>11</b>
7.1	Funzionamento . . . . .	11
7.2	Pseudocodice . . . . .	11
7.3	Complessità . . . . .	11
<b>8</b>	<b>Radix Sort</b>	<b>12</b>
8.1	Pseudocodice . . . . .	12
8.2	Correttezza . . . . .	12
8.3	Complessità . . . . .	12
<b>9</b>	<b>Hash Table</b>	<b>13</b>
9.1	Esempio . . . . .	14

# 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  $(n - 1), \dots, (n - r)$ .

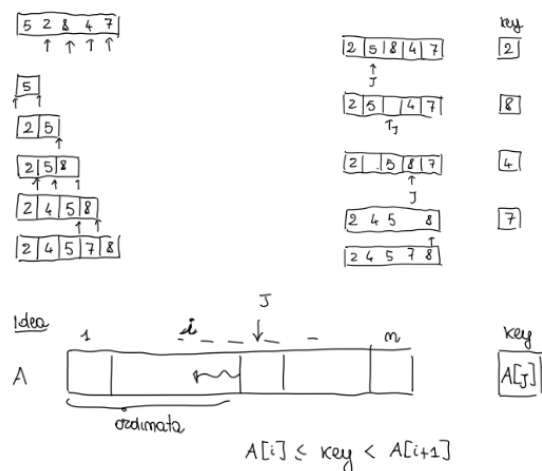
## 2 Insertion Sort

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

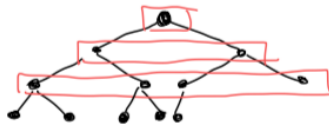
**Albero binario completo:** ogni nodo non foglia ha due figli e ogni cammino radice-foglia ha la stessa lunghezza.

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



### 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

## 6 Quick Sort

L'algoritmo di ordinamento più utilizzato e generalmente più efficiente con  $O(n^2)$  come caso pessimo ed  $O(n \log n)$  come caso medio e migliore è proprio quicksort. Viene definito algoritmo "in-place", cioè un algoritmo che non ha bisogno di spazio extra e produce un output nella stessa memoria che contiene i dati.

Anche questo algoritmo è basato sul paradigma divide-et-impera, infatti, per ordinare  $A[p, r]$ :

- Divide:
  - sceglie un pivot  $x$  in  $A[p, r]$
  - partiziona in  $A[p, q - 1] \leq x$  e  $A[q + 1, r] \geq x$
- Impera:
  - ricorre su  $A[p, q - 1]$  e su  $A[q + 1, r]$

### 6.1 Pseudocodice

```
QUICKSORT(A, p, r)
1  q = PARTITION(A, p, r)
2  QUICKSORT(A, p, q-1)
3  QUICKSORT(A, q+1, r)
```

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p - 1
3  for j = p to r - 1
4      if A[j] <= x
5          i = i + 1
6          A[i] <=> A[j]
7  A[i + 1] <=> A[r]
8  return i + 1
```

### 6.2 Passaggi

Per ordinare un array, seguite i passaggi seguenti:

1. Si sceglie come pivot un qualsiasi valore di indice dell'array.
2. Quindi si partiziona l'array in base al pivot.
3. Quindi si esegue un Quicksort ricorsivo della partizione di sinistra.
4. Successivamente, si esegue un Quicksort ricorsivo della partizione corretta.

Diamo un'occhiata più da vicino alla partizione di questo algoritmo:

1. Si sceglie un pivot qualsiasi, ad esempio il valore più alto dell'indice.
2. Si prenderanno due variabili per puntare a sinistra e a destra dell'elenco, escludendo il pivot.
3. La sinistra punterà all'indice più basso e la destra all'indice più alto.
4. Ora si spostano a destra tutti gli elementi maggiori del pivot.
5. Quindi si sposteranno tutti gli elementi più piccoli del perno nella partizione di sinistra.

## 7 Counting Sort

Il Counting Sort è una tecnica di ordinamento stabile, utilizzata per ordinare gli oggetti in base alle chiavi che sono piccoli numeri. Conta il numero di chiavi i cui valori sono uguali. Questa tecnica di ordinamento è efficiente quando la differenza tra le diverse chiavi non è così grande, altrimenti può aumentare la complessità dello spazio.

### 7.1 Funzionamento

1. Scopri l'elemento  $max$  dall'array specificato.

$max$

8	4	2	2	8	3	3	1
---	---	---	---	---	---	---	---

2. Inizializzare una matrice di lunghezza  $max + 1$  con tutti gli elementi 0. Questa matrice viene utilizzata per memorizzare il conteggio degli elementi nella matrice.

0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8

3. Memorizza il conteggio di ciascun elemento nel rispettivo indice nell'array count.

0	1	2	2	1	0	0	0	1
0	1	2	3	4	5	6	7	8

4. Memorizzare la somma cumulativa degli elementi dell'array count. Aiuta a posizionare gli elementi nell'indice corretto dell'array ordinato.

0	1	3	5	6	6	6	6	7
0	1	2	3	4	5	6	7	8

5. Individuare l'indice di ogni elemento della matrice originale nell'array count. Questo dà il conteggio cumulativo.

1	2	2	3	3	4	8
0	1	2	3	4	5	6

### 7.2 Pseudocodice

```
COUNTING-SORT(A, B, k)
1  C[0..k] = 0
2  for j = 1 to A.length
3      C[A[j]] = C[A[j]] + 1
4  for i = 1 to k
5      C[i] = C[i-1] + C[i]
6  for j = A.length to 1
7      B[C[A[j]]] = A[j]
8      C[A[j]] = C[A[j]] - 1
```

### 7.3 Complessità

$C[0..k] = 0$   $\Theta(k)$   
for j = 1 to A.length  $\Theta(n)$   
...  
for i = 1 to k  $\Theta(k)$   
...  
for j = A.length to 1  $\Theta(n)$   
...

Somma  $\Theta(n + k)$  con  $k = \Theta(1) \Rightarrow \Theta(n)$

## 8 Radix Sort

Il Radix Sort è un algoritmo di ordinamento che ordina gli elementi raggruppando prima le singole cifre tutte nella stessa attuale posizione. Quindi, ordina gli elementi in base al loro ordine crescente/decescente. L'idea è quella di ordinare cifra per cifra con un algoritmo stabile e risolvere i problemi di memoria di Counting Sort.

### 8.1 Pseudocodice

```
Radix-SORT(A, d)
1  for j = 1 to d
2      COUNTING-SORT(A, j)
```

### 8.2 Correttezza

**Invariante:**  $A^{j-1}$  ordinato

**Inizializzazione:**  $j = 1, A^{j-1} = A^0$

**Mantenimento:**  $A^{j-1}$  ordinato e rodino rispetto a  $j$

### 8.3 Complessità

- $d$  volte Counting-Sort  $\Theta(n + b) \Rightarrow \Theta(d(n + b)) = \Theta(n)$
- $d$  cifre  $\Theta(1)$
- $b$  base  $\Theta(n)$

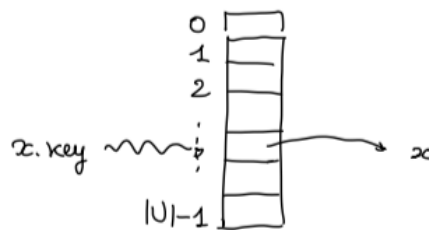
$$\left. \begin{array}{l} m \text{ bit} \Rightarrow m = O(\log n) \\ r \text{ bit per cifra} \Rightarrow r = O(\log_2 n) \\ \text{base } 2^r \end{array} \right\} \Rightarrow \Theta\left(\frac{m}{r}(m + 2^r)\right) = \Theta\left(\frac{m}{\log n}(m + 2^{\log n})\right) = \Theta\left(\frac{m}{\log n}n\right) = \Theta(n)$$

## 9 Hash Table

La Hash Table è una struttura di dati che memorizza i dati in modo associativo. In una tabella, i dati sono memorizzati in un formato array, dove ogni valore di dati ha un proprio valore di indice univoco. L'accesso ai dati diventa molto veloce se si conosce l'indice del dato desiderato.

In questo modo, diventa una struttura di dati in cui le operazioni di inserimento e ricerca sono molto veloci, indipendentemente dalla dimensione dei dati. La Hash Table utilizza un array come supporto di memorizzazione e usa la tecnica dell'Hash per generare un indice in cui un elemento deve essere inserito o da cui deve essere individuato.

In particolare, si considera  $U$  come insieme (universo) della chiavi, visto come  $I = 0, 1, \dots, |U| - 1$ . La Hash Table  $T$  viene vista come array  $T[0, \dots, |U| - 1]$ , in cui un elemento  $x$  è inserito in  $T[x.key]$ .



### Problemi:

- Non è possibile avere oggetti con la stessa chiave
- OK se il numero  $|U|$  è piccolo

```
INSERT(T, x)
1   T[x.key] = x            $\Theta(1)$ 
```

```
DELETE(T, x)
1   T[x.key] = NULL        $\Theta(1)$ 
```

```
SEARCH(k)
1   return T[k]            $\Theta(1)$ 
```

## 9.1 Esempio

**Problema:** consideriamo che la key sia di 8 caratteri (8 bit per rappresentare un carattere). Risulta molto costosa in termini di memoria la tabella hash.

**Obiettivo:** usare quantità di memoria proporzionale al numero di elementi da memorizzare.

**Idea:** creazione di una tabella  $T$  di dimensione  $m \ll |U|$

$$h : U \rightarrow \{0, 1, \dots, m - 1\}$$

Se  $n > m \Rightarrow \exists x_1, x_2 : h(x_1.key) = h(x_2.key) \Rightarrow$  conflitto

Dove:

- $n = \#$  elementi memorizzati nella tabella  $T$
- $m = \#$  celle

La collisione verifica con  $n =$  numero di elementi memorizzati se  $m \ll |U|$  se  $n > m$ .

Il principio è il pigeonhole principle, quindi nella tabella hash, ogni valore trova una corrispondenza. Il pigeonhole principle è una delle idee più semplici ma più utili della matematica e può aiutarci in questo caso. Una versione di base dice che se  $(N + 1)$  piccioni occupano  $N$  buche, allora in qualche buca devono esserci almeno 2 piccioni. Quindi, se 5 piccioni occupano 4 buche, deve esserci una buca con almeno 2 piccioni.