

# Algoritmi & Strutture Dati

Andrea Comar

October 2024

# Contents

<b>I</b>	<b>Concetti Matematici</b>	<b>3</b>
<b>1</b>	<b>Notazione asintotica</b>	<b>3</b>
1.1	Notazione O-grande . . . . .	3
<b>II</b>	<b>Algoritmi</b>	<b>4</b>
<b>2</b>	<b>Algoritmi di ordinamento</b>	<b>4</b>
2.1	Insertion sort . . . . .	4
2.2	Merge . . . . .	5
2.3	Merge sort . . . . .	5
2.4	Heapify (array) . . . . .	6
2.5	Build-Max-Heap (array) . . . . .	6
<b>III</b>	<b>Strutture Dati</b>	<b>7</b>
<b>3</b>	<b>strutture dati lineari</b>	<b>7</b>
3.1	Array . . . . .	7
3.2	Lista . . . . .	7
3.3	Pila . . . . .	7
3.4	Albero Binario . . . . .	7
3.5	Code di priorità . . . . .	8
3.6	Max-Heap . . . . .	8

## Part I

# Concetti Matematici

## 1 Notazione asintotica

Strumento per confrontare quale funzioni divergono all'infinito più velocemente.  
Metodo di confronto tra funzioni di costo degli algoritmi. Caratteristiche:

- $f : \mathbb{N} \rightarrow \mathbb{R}^+$
- funzioni **monotone crescenti**
- $\lim_{n \rightarrow \infty} f(n) = \infty$  (divergenti)

### 1.1 Notazione O-grande

$$O(g(n)) = f(n) \mid \exists c > 0 \exists n$$

## Part II

# Algoritmi

## 2 Algoritmi di ordinamento

**PROBLEMA** Problema data una sequenza  $a_1, a_2, \dots, a_n$  di numeri, trovare una permutazione tale che  $a_1 \leq a_2 \leq \dots \leq a_n$ .

Soluzioni:

### 2.1 Insertion sort

Algorithm 1: InsertionSort
----------------------------

<p><b>Data:</b> A array, i indice, j indice</p> <p><b>for</b> <math>i \leftarrow 2</math> <b>to</b> <math>A.length</math> <b>do</b></p> <table border="0"><tr><td style="padding-right: 10px;"> </td><td>key <math>\leftarrow A[i]</math>;</td></tr><tr><td style="padding-right: 10px;"> </td><td>j <math>\leftarrow j - 1</math>;</td></tr><tr><td style="padding-right: 10px;"> </td><td><b>while</b> <math>j &gt; 0</math> <b>and</b> <math>A[j] &gt; key</math> <b>do</b></td></tr><tr><td style="padding-right: 10px;"> </td><td style="padding-left: 20px;">A[j+1] <math>\leftarrow A[j]</math> ;</td></tr><tr><td style="padding-right: 10px;"> </td><td style="padding-left: 20px;">j <math>\leftarrow j - 1</math> ;</td></tr><tr><td style="padding-right: 10px;"> </td><td>A[j+1] <math>\leftarrow key</math>;</td></tr></table>		key $\leftarrow A[i]$ ;		j $\leftarrow j - 1$ ;		<b>while</b> $j > 0$ <b>and</b> $A[j] > key$ <b>do</b>		A[j+1] $\leftarrow A[j]$ ;		j $\leftarrow j - 1$ ;		A[j+1] $\leftarrow key$ ;
	key $\leftarrow A[i]$ ;											
	j $\leftarrow j - 1$ ;											
	<b>while</b> $j > 0$ <b>and</b> $A[j] > key$ <b>do</b>											
	A[j+1] $\leftarrow A[j]$ ;											
	j $\leftarrow j - 1$ ;											
	A[j+1] $\leftarrow key$ ;											

**Complessità Spaziale** :  $\theta(1)$  in richiede unicamente 3 interi (i, j, A.length) per memorizzare i valori.

**Complessità Temporale** :

- nel caso migliore:  $\theta(n)$  vettore già ordinato
- nel caso peggiore:  $\theta(n^2)$  vettore ordinato al contrario

**Correttezza** :

## 2.2 Merge

Procedura che unisce due vettori ordinati in un unico vettore ordinato. I due vettori di input non devono necessariamente avere la stessa lunghezza.

### Algorithm 2: Merge

```
Input: Array  $A$ , indices  $p, r, q$   
Output: Merged array  $A[p..q]$   
 $i \leftarrow p$ ;  
 $j \leftarrow r + 1$ ;  
 $B \leftarrow$  new array of size  $q - p + 1$ ;  
 $k \leftarrow 1$ ;  
while  $i < r + 1$  and  $j < q + 1$  do  
    if  $A[i] \leq A[j]$  then  
         $B[k] \leftarrow A[i]$ ;  
         $i \leftarrow i + 1$ ;  
    else  
         $B[k] \leftarrow A[j]$ ;  
         $j \leftarrow j + 1$ ;  
     $k \leftarrow k + 1$ ;  
if  $i > r$  then  
    for  $l \leftarrow j$  to  $q$  do  
         $B[k] \leftarrow A[l]$ ;  
         $k \leftarrow k + 1$ ;  
else  
    for  $l \leftarrow i$  to  $r$  do  
         $B[k] \leftarrow A[l]$ ;  
         $k \leftarrow k + 1$ ;
```

## 2.3 Merge sort

Idea: divide et impera. Divido il vettore in due parti, ordino le due parti e poi le unisco.

### Algorithm 3: MergeSort

```
Input: Array  $A$ , indices  $p, q$   
Output: Sorted array  $A[p..q]$   
if  $p < q$  then  
     $r \leftarrow \lfloor \frac{(p+q)}{2} \rfloor$ ;  
    MergeSort( $A, p, r$ );  
    MergeSort( $A, r+1, q$ );  
    Merge( $A, p, q, r$ );
```

MergeSort è un algoritmo basato su ricorsione. Necessita della procedura Merge per unire i due vettori.

**Complessità Spaziale** :  $\theta(n)$  in quanto richiede un vettore di appoggio di dimensione  $n$ .

**Complessità Temporale** :  $\theta(n \log n)$  in quanto il vettore viene diviso in due parti e ogni parte viene ordinata in  $\log n$  passi.

## 2.4 Heapify (array)

Procedura che serve a trasformare una heap in una max-heap.

**pre-condizioni:**  $H[\text{left}(i)]$  e  $H[\text{right}(i)]$  sono max-heap.

### Algorithm 4: Heapify

**Input:** Heap  $H$ , indice  $i$   
**Output:** Max-heap  $H$   
 $l \leftarrow \text{left}(i);$   
 $r \leftarrow \text{right}(i);$   
**if**  $l \leq H.\text{heapsize}$  *and*  $H[l] > H[i]$  **then**  
     $m \leftarrow l$  ( $m$  è  $\text{max}$ );  
**else**  
     $m \leftarrow i;$   
**if**  $r \leq H.\text{heapsize}$  *and*  $H[r] > H[m]$  **then**  
     $m \leftarrow r;$   
**if**  $m \neq i$  **then**  
    scambia ( $H, i, m$ );  
    Heapify( $H, m$ );

**Complessità Spaziale** :

**Complessità Temporale** :

**Correttezza** :

## 2.5 Build-Max-Heap (array)

Date  $n$  chiavi memorizzate in un array, voglio trasformare l'array in una max-heap.

### Algorithm 5: Build-Max-Heap

**Input:** Array  $H$   
**Output:** Max-heap  $H$   
 $H.\text{heapsize} \leftarrow H.\text{length};$   
**for**  $i \leftarrow \lfloor \frac{H.\text{length}}{2} \rfloor$  *downto* 1 **do**  
    Heapify( $H, i$ );

## Part III

# Strutture Dati

## 3 strutture dati lineari

### 3.1 Array

struttura dati **statica** (= suo spazio di memoria non varia) di  $n$  elementi. Sono a **indirizzamento diretto** e l'accesso ha un costo fisso di  $\theta(1)$

### 3.2 Lista

Le liste sono strutture dati **dinamiche** (= il loro spazio di memoria può variare). Possono occupare spazi di memoria non contigui. Tra le operazioni che vogliamo fare con le liste ci sono:

- **inserimento** di un elemento in una posizione arbitraria
- **cancellazione** di un elemento in una posizione arbitraria
- **ricerca** di un elemento in una posizione arbitraria

**liste concatenate** : ogni elemento della lista contiene un campo che punta all'elemento successivo. Nel caso di liste concatenate **push** e **pull** hanno complessità  $\theta(1)$  in quanto conta soltanto la cella individuata dall'indice e **max** ha complessità  $\theta(n)$ .

### 3.3 Pila

La pila è una struttura dati **dinamica** che permette di inserire (**push**) e cancellare (**pull**) elementi con politica **LIFO** (Last In First Out).

### 3.4 Albero Binario

Struttura dati **dinamica** costituita da nodi aventi i seguenti campi:

- chiave:  $x.key$
- puntatore genitore:  $x.parent$
- puntatore figlio sinistro:  $x.left$
- puntatore figlio destro:  $x.right$

Nota: ovviamente se  $x.left$  punta a  $y$ , allora  $y.parent$  punta a  $x$ .

**albero binario completo:** Ogni nodo che non è una foglia ha esattamente due figli e tutti i nodi sono al livello  $h$  o  $h-1$ .

**albero binario quasi completo:** è un albero binario completo fino al penultimo livello, l'ultimo è riempito da sinistra a destra.

**altezza di un nodo:** lunghezza del cammino più lungo che va dal nodo a una foglia. Due convenzioni, in base alla scelta dell'altezza delle foglie, che può essere 0 o 1. Nel nostro caso sarà 0. Un albero può avere altezza massima  $n - 1$ .

### 3.5 Code di priorità

Sono strutture dati **sequenziali** e **dinamiche** i cui elementi sono gestiti con politica **HPFO** (Highest Priority First Out). Ogni elemento è dotato di una key ma anche di una priorità.

- vettori sovradimensionati
- liste concatenate
- vettori sovradimensionati ordinato per priorità

### 3.6 Max-Heap

**Heap:** Ci permettono di implementare code con priorità costo di inserimento e cancellazione pari a  $O(\log n)$ .

**Max-Heap:** è un albero binario completo in cui ogni elemento ha una chiave minore o uguale di quella del proprio genitore.

**proprietà** Data una max-heap di  $n$  nodi:

- altezza:  $\theta(\log n)$
- chiave massima si trova nella radice
- ogni percorso radice-foglia ha le chiavi ordinate in modo decrescente
- la chiave minima si trova su una foglia
- le foglie sono all'incirca  $\frac{n}{2}$

**operazioni:** voglio gestire le code di priorità, pertanto:

- inserimento nuovo nodo
- cancellazione elemento con priorità massima
- ricerca del nodo con priorità massima
- modifica della priorità di un nodo



**implementazione:** per implementare una max-heap posso usare:

- **albero:** struttura dati dinamica
- **vettore sovradimensionato:** struttura dati statica

**procedure di base:**

figlio sinistro:	figlio destro:	nodo genitore:
<pre>left(i){     return 2i }</pre>	<pre>right(i){     return 2i + 1 }</pre>	<pre>parent(i){     return <math>\lfloor \frac{i}{2} \rfloor</math> }</pre>