

# Algoritmi & Strutture Dati

Andrea Comar

October 2024

# Contents

<b>I</b>	<b>Concetti Matematici</b>	<b>4</b>
<b>1</b>	<b>Notazione asintotica</b>	<b>4</b>
1.1	Notazione O-grande (e o-piccolo)	4
1.2	Notazione Omega-grande (e omega-piccolo)	5
1.3	Notazione Theta	5
1.4	Proprietà di base	6
1.5	Comportamento rispetto alle operazioni	6
<b>2</b>	<b>Cenni utili sui limiti</b>	<b>8</b>
<b>3</b>	<b>Stime di Somme</b>	<b>8</b>
<b>II</b>	<b>Algoritmi</b>	<b>9</b>
<b>4</b>	<b>Tabella riassuntiva costi temporali</b>	<b>9</b>
<b>5</b>	<b>Cenni introduttivi da SISTEMARE</b>	<b>10</b>
<b>6</b>	<b>Algoritmi di ordinamento e costruzione</b>	<b>10</b>
6.1	Insertion sort	10
6.2	Merge sort	12
6.3	Merge	13
6.4	Build-Heap (array)	14
6.5	Heapify (array)	14
6.6	Extract-Max-Heap	15
6.7	Heap Sort	15
6.8	Quick Sort	16
6.9	Partition	16
6.10	Selection	17
6.11	Select	17
<b>7</b>	<b>Algoritmi di ordinamento non basati su scambi e confronti</b>	<b>17</b>
7.1	Breve intro	17
7.2	Counting Sort	17
7.3	Radix Sort	18
7.4	Bucket Sort	18
<b>III</b>	<b>Strutture Dati</b>	<b>19</b>

<b>8</b>	<b>strutture dati lineari</b>	<b>19</b>
8.1	Array . . . . .	19
8.2	Lista . . . . .	19
8.3	Pila . . . . .	19
8.4	Code di priorità . . . . .	19
8.5	Albero Binario . . . . .	20
8.6	Max-Heap . . . . .	20
8.7	Tabelle di Hash . . . . .	21
8.8	Tabelle di Hash con Chaining . . . . .	22
8.9	funzioni di Hash . . . . .	23
8.10	Tabelle di Hash con Open Addressing . . . . .	24
8.11	Tipologie di scansione . . . . .	24
<b>IV</b>	<b>Glossario</b>	<b>24</b>
<b>9</b>	<b>Termini e definizione</b>	<b>25</b>
<b>V</b>	<b>Esercizi e sfide</b>	<b>25</b>
<b>10</b>	<b>esercizi scrittura di algoritmi</b>	<b>25</b>
<b>11</b>	<b>equazioni ricorsive</b>	<b>25</b>
<b>12</b>	<b>esercizi di induzione e correttezza</b>	<b>26</b>
<b>13</b>	<b>sfide</b>	<b>26</b>
13.1	Possibility Majority Candidate . . . . .	26
13.2	Liste Circolari . . . . .	26
13.3	Matrice . . . . .	26

## 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)

In breve la notazione asintotica si basa su tre simboli:

- $O$  (O-grande): rappresenta il caso peggiore, ovvero il **limite asintotico superiore**. "cresce al più come"
- $\Omega$  (Omega): rappresenta il caso migliore, ovvero il **limite asintotico inferiore**. "cresce al meno come"
- $\Theta$  (theta): rappresenta il caso medio, ovvero il **limite asintotico stretto**. "stesso ordine di grande"

La notazione asintotica si fonda sul concetto di limite, in particolare sul **limite del rapporto**. Di base possiamo riscrivere

### 1.1 Notazione O-grande (e o-piccolo)

**Notazione O-grande**  $O$  Abbiamo due definizioni equivalenti:

- $O(g(n)) = \{f(n) \mid \exists c > 0 \exists \bar{n} \forall n \geq \bar{n} f(n) \leq c \cdot g(n)\}$
- Date  $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$  **monotone crescenti**, diciamo che  $f(n) \in O(g(n))$  se  $\exists c > 0 \text{ e } \exists \bar{n} : \forall n \geq \bar{n} f(n) \leq c \cdot g(n)$

Di base entrambe fanno riferimento al concetto di **limite del rapporto**

- $\limsup_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| < \infty \Leftrightarrow f(n) = O(g(n))$

Diciamo che  $g(n)$  domina  $f(n)$ , ovvero  $f(n)$  ha un ordine di grandezza minore o uguale a  $g(n)$ .

**notazione o-piccolo**  $o$  Se nelle definizioni invece di  $\exists c$  vale per  $\forall c$  si parla di notazione o-piccolo. Quindi vale il seguente limite:

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Leftrightarrow f(n) = o(g(n))$

Di conseguenza  $f(n) = o(g(n)) \Rightarrow f(n) = O(g(n))$ , NON il contrario.

## 1.2 Notazione Omega-grande (e omega-piccolo)

**Notazione Omega-grande  $\Omega$**  Abbiamo due definizioni equivalenti:

- $\Omega(g(n)) = \{f(n) \mid \exists c > 0 \exists \bar{n} \in \mathbb{N} \forall n \geq \bar{n} f(n) \geq c \cdot g(n)\}$
- Date  $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$  **monotone crescenti**, diciamo che  $f(n) \in \Omega(g(n))$  se  $\exists c > 0$  e  $\exists \bar{n} : \forall n \geq \bar{n} f(n) \geq c \cdot g(n)$

Di base entrambe fanno riferimento al concetto di **limite del rapporto**

- $\liminf_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| > 0 \Leftrightarrow f(n) = \Omega(g(n))$

**notazione omega-piccolo  $\omega$**  Se nelle definizioni invece di  $\exists c$  vale per  $\forall c$  si parla di notazione omega-piccolo. Quindi vale il seguente limite:

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \Leftrightarrow f(n) = \omega(g(n))$

Di conseguenza  $f(n) = \omega(g(n)) \Rightarrow f(n) = \Omega(g(n))$ , NON il contrario.

## 1.3 Notazione Theta

**Notazione Theta  $\Theta$**  Abbiamo due definizioni equivalenti:

- $\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0 \exists \bar{n} \in \mathbb{N} \forall n \geq \bar{n} c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$
- Date  $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$  **monotone crescenti**, diciamo che  $f(n) \in \Theta(g(n))$  se  $\exists c_1, c_2 > 0$  e  $\exists \bar{n} : \forall n \geq \bar{n} c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

Di base entrambe fanno riferimento al concetto di **limite del rapporto**

- $0 < \liminf_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| \leq \limsup_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| \leq \infty$

**notazione  $\sim$**  Se le due funzioni sono sia  $o$  che  $\Theta$  allora si può scrivere  $f(n) \sim g(n)$  e vale che:

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c, c \neq 0 \in \mathbb{R} \Leftrightarrow f(n) \sim g(n)$

Di conseguenza  $f(n) \sim g(n) \Rightarrow f(n) = \Theta(g(n))$ , NON il contrario.

NOTA: per parlare di equivalenza asintotica  $c$  dovrebbe essere esattamente uguale a 1.

## 1.4 Proprietà di base

Se  $f(n)$  è  $O$  di  $g(n)$  allora  $g(n)$  è  $\Omega$  di  $f(n)$ . Significa che  $g(n)$  cresce di più asintoticamente.

$$\bullet f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$$

Se  $f(n)$  cresce allo stesso modo di  $g(n)$ ,  $g(n)$  rappresenta sia il limite superiore che inferiore.

$$\bullet f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$$

Analogamente per la notazione o-piccolo e omega-piccolo:

$$\bullet f(n) \in o(g(n)) \Leftrightarrow g(n) \in \omega(f(n))$$

Infine due funzioni piuttosto basilari, ovvero:

- $\bullet f = O(f)$
- $\bullet f = o(f) \Rightarrow f \equiv 0$

## 1.5 Comportamento rispetto alle operazioni

Le seguenti regole valgono indistintamente per  $O$ ,  $\Omega$  e  $\Theta$ , si ereditano dalle proprietà dei limiti.

### Abuso di notazione

- $\bullet f(x) = O(g(x))$  non è corretto in quanto non sono effettivamente uguali, tuttavia si usa per comodità al posto di  $f(x) \in O(g(x))$

### Transitività

- $\bullet f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$
- $\bullet f \in \Omega(g) \wedge g \in \Omega(h) \Rightarrow f \in \Omega(h)$
- $\bullet f \in \Theta(g) \wedge g \in \Theta(h) \Rightarrow f \in \Theta(h)$
- $\bullet f \in o(g) \wedge g \in o(h) \Rightarrow f \in o(h)$
- $\bullet f \in \omega(g) \wedge g \in \omega(h) \Rightarrow f \in \omega(h)$

### Additività

- $\bullet f \in O(h) \wedge g \in O(h) \Rightarrow f + g \in O(h)$
- $\bullet f \in \Omega(h) \wedge g \in \Omega(h) \Rightarrow f + g \in \Omega(h)$
- $\bullet f \in \Theta(h) \wedge g \in \Theta(h) \Rightarrow f + g \in \Theta(h)$

**Riflessività**

- $f \in O(f)$ , con abuso di notazione  $f(n) = O(f(n))$
- $f \in \Omega(f)$ , con abuso di notazione  $f(n) = \Omega(f(n))$
- $f \in \Theta(f)$ , con abuso di notazione  $f(n) = \Theta(f(n))$

**Simmetria**

- $f = \Theta(g) \Rightarrow g = \Theta(f)$

**Simmetria trasposta**

- $f = O(g) \Rightarrow g = \Omega(f)$
- $f = o(g) \Rightarrow g = \omega(f)$

**Somma di due funzioni :**

- $f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2)$
- $f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) \Rightarrow f_1 + f_2 \in \Omega(g_1 + g_2)$
- $f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2) \Rightarrow f_1 + f_2 \in \Theta(g_1 + g_2)$

**Prodotto di due funzioni**

- $f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 \cdot f_2 \in O(g_1 \cdot g_2)$
- $f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) \Rightarrow f_1 \cdot f_2 \in \Omega(g_1 \cdot g_2)$
- $f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2) \Rightarrow f_1 \cdot f_2 \in \Theta(g_1 \cdot g_2)$

Le precedenti regole NON valgono per operazioni di sottrazione e divisione.

**Costante moltiplicativa**

- $O(c \cdot f) = O(f), \forall c \in \mathbb{R}_0$
- $\Omega(c \cdot f) = \Omega(f), \forall c \in \mathbb{R}_0$
- $\Theta(c \cdot f) = \Theta(f), \forall c \in \mathbb{R}_0$

Semplicemente possiamo dire che la costante moltiplicativa non influisce sul comportamento asintotico.

**Trascurare termini additivi di ordine inferiore**

- $g = O(f) \Rightarrow f + g = \Theta(f)$

Overver possiamo considerare unicamente il termine di ordine maggiore.

**Trascurare le costanti moltiplicative**

- $\forall a > 0 \Rightarrow a \cdot f = \Theta(f)$

**2 Cenni utili sui limiti**

**3 Stime di Somme**



## Part II

# Algoritmi

### 4 Tabella riassuntiva costi temporali

Algoritmo	peggiore	migliore medio	stabile	Inplace
<b>Insertion Sort</b>	$\Theta(n^2)$	$\Theta(n)$	stabile	inplace
Merge	$\Theta(n)$	$\Theta(n)$	stabile	non inplace
<b>Merge Sort</b>	$\Theta(n \log n)$	$\Theta(n \log n)$		non inplace
Heapify	$O(\log n)$			
Build Max Heap	$\Theta(n)$			inplace
<b>Heap Sort</b>	$\Theta(n \log n)$	$\Theta(n)$ ripetizioni	non stabile	inplace
Quick Sort	$\Theta(n^2)$	$\Theta(n \log n)$	non stabile	non inplace
Selection Sort	$\Theta(n^2)$			
Counting Sort	$\Theta(n^2)$	$\Theta(n + k)$ , $k \in O(n)$	stabile	non inplace
Radix Sort		$d \cdot \Theta(n)$	stabile	inplace
Bucket Sort	$\Theta(n^2)$	$\Theta(n)$	dipende	non inplace

## 5 Cenni introduttivi da SISTEMARE

**valutazione complessità :**

Ogni istruzione di base (assegnamenti, confronti, operazioni algoritmiche) ha un costo costante  $c_h$ . Per semplicità, la funzione di complessità è così strutturata:

- $Time : \mathbb{N} \rightarrow \mathbb{R}^+$

L'algoritmo ovviamente reagisce in modo diverso a seconda dell'input, pertanto parleremo di tempo assoluto nel:

- **caso migliore:**  $T_p : \mathbb{N} \leftarrow \mathbb{R}^+$ , ovvero tempo impiegato al massimo
- **caso peggiore:** input che massimizza il tempo di esecuzione
- **caso medio:** input che si presenta con probabilità uniforme

## 6 Algoritmi di ordinamento e costruzione

**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:

### 6.1 Insertion sort

Dato un vettore, inizio a scorrere da sinistra verso destra. Ad ogni passo, confronto l'elemento corrente con quelli precedenti, e lo inserisco nella giusta posizione.

$A = [5, 2, 4, \dots]$

$A = [5, 2, 4, \dots]$

$A = [2, 5, 4, \dots]$

$A = [2, 4, 5, \dots]$

#### Algorithm 1: InsertionSort

**Data:** A array, i indice, j indice

**begin**

**for**  $i \leftarrow 2$  to  $A.length$  **do**

    key  $\leftarrow A[i]$ ;

    j  $\leftarrow j - 1$ ;

**while**  $j > 0 \ \&\& \ A[j] > key$  **do**

$A[j+1] \leftarrow A[j]$  ;

      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. Questo lo rende un algoritmo **inplace**.  
È un algoritmo **stabile** con  $A[j] > key$ , in quanto non scambia elementi uguali.

**Complessità Temporale** :

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

Nel **caso medio** l'algoritmo ha complessità  $\Theta(n^2)$ .

**Correttezza** :

## 6.2 Merge sort

Problema: devo riordinare un generico array  $A$  di  $n$  elementi. Immagino di voler riordinare una sottosezione di  $A$ , ovvero  $A[p..q]$ . Trovo  $r$  pari a  $\lfloor \frac{(p+q)}{2} \rfloor$  e ordino ricorsivamente le due sottosezioni  $A[p..r]$  e  $A[r+1..q]$ . Infine unisco le due sottosezioni mediante la procedura Merge.

$p = 4, q = 11, r = 7, m = q - p + 1 = 8$

[..., 20, 10, 21, 30, 9, 8, 26, 25, ...]

[..., 20, 10, 21, 30, 9, 8, 26, 25, ...]

[..., 20, 10, 21, 30, 9, 8, 26, 25, ...]

### Algorithm 2: MergeSort

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

**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.

**Equazione ricorsiva di complessità:** 
$$\begin{cases} \Theta(1) & n \leq 1 \\ 2T(\frac{n}{2}) + \Theta(n) & n > 1 \end{cases}$$

### 6.3 Merge

La procedura prende in input o due vettori oppure un vettore con due sottosezioni **già ordinate** e le unisce in un unico vettore ordinato. Si basa sul confronto tra i valori più piccoli delle due sottosezioni, operazione poco costosa in quanto sono ordinate.

[..., 20, 10, 21, 30, 9, 8, 26, 25, ...]  
 [..., 10, 20, 21, 30, 8, 9, 25, 26, ...]  
 [..., 10, 20, 21, 30, 8, 9, 25, 26, ...]  
 [..., 10, 20, 21, 30, 8, 9, 25, 26, ...] → [8, ???]  
 [..., 10, 20, 21, 30, 8, 9, 25, 26, ...] → [8, 9, ???]  
 [..., 10, 20, 21, 30, 8, 9, 25, 26, ...] → [8, 9, 10, ???]  
 [..., 10, 20, 21, 30, 8, 9, 25, 26, ...] → [8, 9, 10, 20, ???]

#### Algorithm 3: Merge

**Input:** Array  $A$ , indices  $p, r, q$   
**Output:** Merged array  $A[p..q]$   
**begin**  
    $i \leftarrow p$ ;  
    $j \leftarrow r + 1$ ;  
    $B \leftarrow$  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**  
        $B[k] \leftarrow A[l]$ ;  
        $k \leftarrow k + 1$ ;

**procedura:** La procedura Merge si basa sul **confrontare** i primi due valori delle rispettive sottosezioni (ovvero i **valori più piccoli**), inserendo il minore in un vettore di appoggio B. L'elemento maggiore dei due viene confrontato con l'elemento successivo dell'altra sottosezione, e così via. Se una delle due sottosezioni si esaurisce, allora si copiano tutti gli elementi rimanenti dell'altra sottosezione. Il procedimento viene effettuato tramite due indici  $i$  e  $j$ .

**complessità temporale:**  $\Theta(n)$  in quanto ogni elemento viene confrontato una sola volta, proprietà che deriva dal fatto che sono ordinate.

## 6.4 Build-Heap (array)

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

### Algorithm 4: Build-Max-Heap

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

## 6.5 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 5: Heapify

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

**Complessità Spaziale** :

**Complessità Temporale** : generalmente la complessità è  $O(n)$ , nel caso peggiore  $\Theta(n)$

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

**Correttezza** :

## 6.6 Extract-Max-Heap

### Algorithm 6: Extract-Max-Heap

```
Input: Heap  $H$   
Output: Max-heap  $H$   
begin  
  scambia( $H, 1, H.\text{heapsize}$ );  
   $H.\text{heapsize} \leftarrow H.\text{heapsize} - 1$ ;  
  if  $H.\text{heapsize} \geq 1$  then  
     $\perp$  Heapify( $H, 1$ );  
   $\perp$  return  $H[H.\text{heapsize} + 1]$ ;
```

## 6.7 Heap Sort

Algoritmo per ordinare  $n$  chiavi in ordine crescente usando una max heap. L'idea è di costruire una max heap estraendo il massimo e ripetendo il procedimento.

### Algorithm 7: HeapSort

```
Input: Array  $A$   
Output: Sorted array  $A$   
begin  
  Build-Max-Heap( $A$ );  
  for  $i \leftarrow A.\text{length}$  downto 2 do  
     $\perp$  scambia( $A, 1, i$ );  
     $\perp$   $A.\text{heapsize} \leftarrow A.\text{heapsize} - 1$ ;  
     $\perp$  Heapify( $A, 1$ );
```

Notiamo come l'algoritmo sia una sorta di fusione tra BuildHeap e Extract Max Heap.

**Complessità temporale** :  $O(n \log n)$ , con caso peggiore  $\Theta(n \log n)$  e caso migliore  $\Theta(n)$  quando sono presenti **molte ripetizioni**.

## 6.8 Quick Sort

Ho un vettore da ordinare, scelgo un elemento che fa da perno e divido il vettore in due parti, uno contenente elementi minori del perno e uno con elementi maggiori. Partition serve a trovare il perno e a dividere il vettore in due parti. Utilizzo ricorsivamente partition per ordinare le due sottosezioni.

### Algorithm 8: QuickSort

```
Input: Array  $A$ , indice  $p$ , indice  $q$   
Output: Sorted array  $A[p..q]$   
begin  
  if  $p < q$  then  
     $r \leftarrow \text{Partition}(A, p, q);$   
    QuickSort( $A, p, r-1$ );  
    QuickSort( $A, r+1, q$ );  
end
```

**Complessità temporale** : caso peggiore  $\Theta(n^2)$ , caso medio  $\Theta(n \log n)$

$$T(n) \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T(m) + T(m - n - 1) + \Theta(n) & \text{se } n > 1 \end{cases}$$

**complessità spaziale:** l'algoritmo NON è inplace e NON è stabile.

## 6.9 Partition

**idea:** prendo l'ultimo elemento come perno e lo inserisco nella posizione corretta, posizionando tutti gli elementi minori o uguali del perno a sinistra di quest'ultimo e quelli maggiori a destra. Restituisco l'indice della posizione corretta del perno. Le due porzioni dell'array NON sono ordinate.

### Algorithm 9: Partition

```
Input: Array  $A$ , indice  $p$ , indice  $q$   
Output: Pivot index  $i$   
begin  
   $x \leftarrow A[q];$   
   $i \leftarrow p - 1;$   
  for  $j \leftarrow p$  to  $q$  do  
    if  $A[j] \leq x$  then  
       $i \leftarrow i + 1;$   
      scambia ( $A, i, j$ );  
  return  $i;$   
end
```

**Complessità temporale** :  $\Theta(n)$  in quanto si tratta di un ciclo for per  $n$  elementi.



## 6.10 Selection

Utilizzo del perno ottimale per partition, in questo modo l'ordinamento è sempre efficiente

## 6.11 Select

Dato un vettore  $A$  di lunghezza  $n$  e dato  $i \in [1, \dots, n]$ , determinare l'elemento che finirebbe in posizione  $i$ -esima se ordinassi  $A$ .

### Algorithm 10: Select

**Input:** Array  $A$ , indice  $p$ , indice  $q$ , indice  $i$

**Output:** Elemento  $x$

**begin**

**if**  $p = q$  **then**

        return  $A[p]$ ;

**else**

$r \leftarrow \text{Partition}(A, p, q, );$

**if**  $i = r$  **then**

            return  $A[r]$ ;

**else if**  $i < r$  **then**

            return  $\text{Select}(A, p, r-1, i)$ ;

**else**

            return  $\text{Select}(A, r+1, q, i-k)$ ;

**idea:** Bisogna trovare un buon perno.

## 7 Algoritmi di ordinamento non basati su scambi e confronti

### 7.1 Breve intro

Ogni algoritmo di ordinamento **basato su scambi e confronti** nel **caso peggiore** ha complessità pari a  $\Omega(n \log n)$ . È possibile ridurre la complessità introducendo delle ipotesi sull'input che permettano l'utilizzo di algoritmi di ordinamento non basati su confronti.

### 7.2 Counting Sort

Richiede delle **ipotesi sulle chiavi** (i valori assunti), ovvero che siano intere e comprese tra 0 e  $k$ , con  $k \in O(n)$ . Una nota importante è che  $k$  *non* è *necessariamente una costante*. Anche valori come  $k = \frac{n}{2}$ ,  $k = 10 \cdot n$ ,  $k = \log n$  sono validi.

**Algorithm 11:** CountingSort**Input:** Array  $A$ , Array  $B$ ,  $k$ **Output:** Sorted array  $A$ **begin**     $C \leftarrow$  new array of size  $k + 1$ ;    **for**  $j \leftarrow 0$  *to*  $k$  **do**         $C[j] \leftarrow 0$ ;    **for**  $i \leftarrow 1$  *to*  $A.length$  **do**         $C[A[i]] \leftarrow C[A[i]] + 1$ ;    **for**  $j \leftarrow 1$  *to*  $k$  **do**         $C[j] \leftarrow C[j] + C[j-1]$ ;    **for**  $i \leftarrow A.length$  *down to*  $1$  **do**         $B[C[A[i]]] \leftarrow A[i]$ ;         $C[A[i]] \leftarrow C[A[i]] - 1$ ;

**Complessità temporale** :  $\Theta(n + k)$  se ho per ipotesi che  $k = O(n)$ . Allora per possiamo ignorare i termini di ordine inferiore e otteniamo  $\Theta(n)$ .

**Complessità spaziale** :

### 7.3 Radix Sort

Utilizzato per ordinare  $n$  numeri di  $d$  cifre. Procede a partire dalla cifra meno significativa fino a quella più significativa. A ogni iterazione applico un algoritmo di ordinamento stabile su una sola cifra. (es. counting sort).

### 7.4 Bucket Sort

## Part III

# Strutture Dati

## 8 strutture dati lineari

### 8.1 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  $\Theta(1)$

**operazioni e costo** :

- **accesso e modifica**:  $A[i]$ , costo  $\Theta(1)$

### 8.2 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  $\Theta(n)$ .

### 8.3 Pila

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

### 8.4 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

Il tipo di implementazione influisce sulla complessità delle operazioni.

- implementazione tramite **lista concatenata**:
  - **inserimento** chiave  $k$  in posizione  $h$ : costo  $O(n)$ , caso peggiore  $\Theta(n)$
  - inserimento (**push**) e cancellazione(**pop**) di  $k$  **in testa**: costo  $\Theta(1)$
  - **ricerca** dell' $h$ -esimo elemento: costo  $O(n)$ , caso peggiore  $\Theta(n)$
- implementazione tramite **vettore sovradimensionato**:
  - **normale**: inserimento costo  $O(n)$ , peggiore  $\Theta(n)$
  - **ordinato per priorità**: cancellazione  $\Theta(1)$ , inserimento  $\Theta(n)$
  - **heap** cancellazione e inserimento con  $O(\log n)$

## 8.5 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  $n-1$ .

## 8.6 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.

**costruzione (rivedere):** posso costruire una max heap tramite diversi metodi:

- da H inserisco una per una le chiavi in K con insert-mex-heap
- tramite Merge-Sort ordino al contrario, in questo modo ottengo una max-heap ( $\Theta(n \log n)$ )
- sfrutto Heapify per trasformare un array in una max-heap  $O(n \log n)$

**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>

## 8.7 Tabelle di Hash

**problema:** dato un insieme  $U$  di elementi identificati da chiavi, voglio memorizzare un sottoinsieme  $k \subseteq U$  che **varia dinamicamente** nel tempo.

**implementazione:** Notiamo che  $|U| = M$  è un numero molto grande rispetto  $K = n$ . L'utilizzo di un vettore renderebbe le operazioni efficienti, ma comporterebbe un grosso quantitativo di spazio  $\Theta(M)$ . Il vettore dovrebbe contenere tutti gli elementi di  $U$ , in quanto  $K$  varia. L'utilizzo di una lista concatenata degli elementi di  $k$  ridurrebbe lo spazio ( $\Theta(n)$ ), ma comporterebbe un costo per le operazioni di ricerca e cancellazione  $O(n)$ . Quello che vogliamo noi è:

**operazioni:**

- **inserimento** di un elemento con costo medio  $\Theta(1)$
- **cancellazione** di un elemento con costo medio  $\Theta(1)$
- **ricerca** di un elemento con costo medio  $\Theta(1)$

Il tutto con costo spaziale  $\Theta(n)$ .

**implementazione:** due possibili implementazioni delle tabelle di Hash:

- **chaining**, ovvero uso di vettore e liste concatenate
- **open addressing**, ovvero uso di un vettore

## 8.8 Tabelle di Hash con Chaining

**idea:** uso di un vettore  $T$  di dimensione  $m$ , in cui ogni cella contiene una lista concatenata di elementi.

**funzione di Hash:**

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

L'elemento  $x$  viene inserito nella lista in  $T[h(x.key)]$ . Calcolare  $h(k)$  costa  $\Theta(1)$ .

**collisioni** Quando si verifica la seguente condizione:

- $x \neq y \in U$  con  $h(x.key) = h(y.key)$

Entrambe gli elementi vengono inseriti nella stessa lista concatenata, indirizzata dalla rispettiva cella in  $T$ . Ovviamente ogni nuovo elemento viene inserito in testa alla lista. Questo garantisce  $\Theta(1)$  per l'inserimento ma  $O(m)$  per ricerca e rimozione.

**implementazione efficiente:** affinché le operazioni rispettino le ipotesi di costo  $\Theta(1)$  nel caso medio, è necessario che le liste abbiano mediamente la stessa lunghezza, ovvero dati  $|T| = m$ ,  $|k| = n$ , allora la lunghezza media di ogni lista dovrebbe essere:

- $\alpha = \frac{n}{m}$ , detto **fattore di carico**

**teorema:** Se  $T$  è una tabella di Hash con chaining e vale l'ipotesi di hashing uniforme sempliceme, allora le operazioni di ricerca e cancellazione hanno costo  $\Theta(1 + \alpha)$  nel caso medio.

**ipotesi di hashing uniforme semplice:** ogni elemento di  $U$  ha la stessa probabilità di essere mappato in una qualsiasi delle  $m$  celle di  $T$ , se non conosco  $x.key$ , ovvero

- $\forall i \in [0, m - 1]$  probabilità di  $(h(x.key) = i) = \frac{1}{m}$
- $\dim(..)$

## 8.9 funzioni di Hash

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

**caratteristiche:**

- **non iniettività:** in quanto due chiavi possono collidere
- **suriettività:** in  $T$  non devono esserci celle non mappate
- **uniformità:** ogni cella di  $T$  deve avere la stessa probabilità di essere mappata
- controllo del dominio e del codominio

**metodo della divisione:**

- $h(key) = key \bmod m$

suggerimenti: meglio che  $m$  sia un numero primo e lontano da una potenza di 2 (per migliore distribuzione)

**metodo bucketsort:**  $A[i]$  elementi vettore  $\in [0, \dots, 1)$  distribuiti in modo uniforme.  $\lfloor A[i] \cdot n \rfloor \in \{0, \dots, m - 1\}$   $B$  vettore di liste, con funzione:

- $h(k) = \lfloor k \cdot n \rfloor$

**metodo universale:** generalizza il caso precedente per qualsiasi  $m$ , consideriamo  $0 < A < 1$ .

- $h(k) = \lfloor m \cdot (k \cdot A - \lfloor k \cdot A \rfloor) \rfloor$

Notiamo che  $k \cdot A - \lfloor k \cdot A \rfloor$  è la parte decimale di  $k \cdot A$ , in quanto  $\lfloor k \cdot A \rfloor$  è unicamente la parte intera.

## 8.10 Tabelle di Hash con Open Addressing

Tutti gli elementi vengono memorizzati in un vettore  $T$  di dimensione  $m$ . Le collisioni vengono risolte tramite **sequenze di scansione**.

**funzione di Hash:**

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

il secondo argomento della funzione rappresenta il tentativo di inserimento.

**proprietà:**  $\forall h(x.key, 0), h(x.key, 1), \dots, h(x.key, m - 1)$

- **iniettiva:** ogni chiave viene mappata in una sola cella
- **suriettiva:** ogni cella di  $T$  è mappata

In questo modo la funzione è **biiettiva**, e rappresenta una **permutazione** di  $\{0, 1, \dots, m - 1\}$ . Per favorire la ricerca e la cancellazione, utilizzo due costanti per identificare le celle vuote:

- **NIL:** cella mai usata
- **DEL:** cella usata e svuotata, vale come occupata per ricerca/cancellazione

**costi delle operazioni (inserimento, ricerca, cancellazione):**

- **caso medio:**  $\Theta(1)$  se vale ipotesi di **hashing uniforme**
- **caso peggiore:**  $\Theta(m)$

**ipotesi di hashing uniforme:** tutte le possibili permutazioni di  $\{0, 1, \dots, m - 1\}$  sono ugualmente probabili, ovvero  $\frac{1}{m!}$ , in quanto dati  $m$  elementi, le possibili permutazioni sono  $m!$

## 8.11 Tipologie di scansione

**scansione lineare:**

$$\bullet h(k, i) = (h(k) + i) \bmod m$$

Se una cella è occupata, si passa alla successiva. Non rispetta l'ipotesi di hashing uniforme, genera soltanto  $m$  possibili scansioni sulle  $m$ !



## Part IV

# Glossario

## 9 Termini e definizione

**inplace** algoritmo che non richiede spazio aggiuntivo per l'esecuzione, quindi  $\Theta(1)$

**stabile** algoritmo che mantiene l'ordine relativo degli elementi con chiavi uguali

**dinamico** struttura dati che può variare nel tempo

**sequenziale** struttura dati in cui gli elementi sono memorizzati in sequenza

**statico** struttura dati in cui lo spazio di memoria non varia

## Part V

# Esercizi e sfide

## 10 esercizi scrittura di algoritmi

## 11 equazioni ricorsive

esercizi di equazioni ricorsive:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ 2T(\frac{n}{2}) + \Theta(1) & \text{se } n > 1 \end{cases}$$

Posso riscrivere l'equazione sostituendo  $\Theta(1)$  con delle costanti, ottenendo:

$$T(n) = \begin{cases} a & \text{se } n \leq 1 \\ 2T(\frac{n}{2}) + b & \text{se } n > 1 \end{cases}$$

Tabella dei costi per livello:

numero nodi	dimensione nodo	costo unitario nodo	costo livello
1	n	b	b
2	$\frac{n}{2}$	b	$2 \cdot b$
$2^2$	$\frac{n}{2^2}$	b	$2^2 \cdot b$
...	...	...	...
$2^i$	$\frac{n}{2^i}$	b	$2^i \cdot b$
...	...	...	...
$2^x$	$\frac{n}{2^x} = 1$	a	$2^x \cdot a$

**costo:** somma del costo dei livelli

$$T(n) = b + 2 \cdot b + \dots 2^i \cdot b + \dots + 2^x \cdot a$$

$$\frac{n}{2^x} = 1 \Rightarrow x = \log_2 n$$

$$T(n) = \sum_{i=0}^{\log_2 n - 1} 2^i \cdot b + 2^{\log_2 n} \cdot a$$

ora opero sull'equazione

$$T(n) = b()$$

## 12 esercizi di induzione e correttezza

## 13 sfide

### 13.1 Possibility Majority Candidate

### 13.2 Liste Circolari

### 13.3 Matrice