

Algoritmi & Strutture Dati

Andrea Comar

October 2024

Contents

| | | |
|------------|--|----------|
| I | Concetti Matematici | 3 |
| 1 | Notazione asintotica | 3 |
| 1.1 | Notazione O-grande (e o-piccolo) | 3 |
| 1.2 | Notazione Omega-grande (e omega-piccolo) | 4 |
| 1.3 | Notazione Theta | 4 |
| 1.4 | Proprietà | 5 |
| 1.5 | Comportamento rispetto alle operazioni | 5 |
| 2 | Cenni utili sui limiti | 6 |
| 3 | Stime di Somme | 6 |
| II | Algoritmi | 6 |
| 4 | Algoritmi di ordinamento | 6 |
| 4.1 | Insertion sort | 6 |
| 4.2 | Merge | 7 |
| 4.3 | Merge sort | 7 |
| 4.4 | Heapify (array) | 8 |
| 4.5 | Build-Max-Heap (array) | 8 |
| III | Strutture Dati | 9 |
| 5 | strutture dati lineari | 9 |
| 5.1 | Array | 9 |
| 5.2 | Lista | 9 |
| 5.3 | Pila | 9 |
| 5.4 | Albero Binario | 9 |
| 5.5 | Code di priorità | 10 |
| 5.6 | Max-Heap | 10 |

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"
- Ω (Omega): rappresenta il caso migliore, ovvero il **limite asintotico inferiore**. "cresce al meno come"
- θ (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 $\exists c > 0$ 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 Ω 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 ω 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

- $\theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0 \exists \bar{n} \in \mathbb{N} \forall n \geq \bar{n} c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$
- 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 θ 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

Se $f(n)$ è O di $g(n)$ allora $g(n)$ è Ω 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.

$$\bullet f(n) \in \theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \text{ e } f(n) \in \Omega(g(n))$$

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

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

1.5 Comportamento rispetto alle operazioni

Le seguenti regole valgono indistintamente per O , Ω e θ , si ereditano dalle proprietà dei limiti.

Somma :

- $\bullet O(f(n)) + O(g(n)) = O(f(n) + g(n))$
- $\bullet \Omega(f(n)) + \Omega(g(n)) = \Omega(f(n) + g(n))$
- $\bullet \theta(f(n)) + \theta(g(n)) = \theta(f(n) + g(n))$

Prodotto :

- $\bullet O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$
- $\bullet \Omega(f(n)) \cdot \Omega(g(n)) = \Omega(f(n) \cdot g(n))$
- $\bullet \theta(f(n)) \cdot \theta(g(n)) = \theta(f(n) \cdot g(n))$

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

Costante :

- $\bullet O(c \cdot f(n)) = O(f(n)), \forall c \in \mathbb{R}_0$
- $\bullet \Omega(c \cdot f(n)) = \Omega(f(n))$
- $\bullet \theta(c \cdot f(n)) = \theta(f(n))$
- $\bullet O(f(n)) = O(g(n)) \Leftrightarrow f(n) \in \theta(g(n))$
- $\bullet O(f(n)) = O(g(n)) \Leftrightarrow f(n) \in \theta(g(n))$

2 Cenni utili sui limiti

3 Stime di Somme

Part II

Algoritmi

4 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:

4.1 Insertion sort

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

| |
|--|
| <p>Data: A array, i indice, j indice</p> <p>for $i \leftarrow 2$ to $A.length$ do</p> <p> $key \leftarrow A[i];$</p> <p> $j \leftarrow i - 1;$</p> <p> while $j > 0 \ \&\& \ A[j] > key$ do</p> <p> $A[j+1] \leftarrow A[j];$</p> <p> $j \leftarrow j - 1;$</p> <p> $A[j+1] \leftarrow key;$</p> |
|--|

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 :

4.2 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  
         $B[k] \leftarrow A[l]$ ;  
         $k \leftarrow k + 1$ ;
```

4.3 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.

4.4 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 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 :

4.5 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

5 strutture dati lineari

5.1 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 $\theta(1)$

5.2 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 **psuh** e **pull** hanno complessità $\theta(1)$ in quanto conta soltanto la cella individuata dall'indice e **max** ha complessità $\theta(n)$.

5.3 Pila

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

5.4 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 $n - 1$.

5.5 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

5.6 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 $\lfloor \frac{i}{2} \rfloor$ }</pre> |