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 di base . . . 5
1.5 Comportamento rispetto alle operazioni . . . 5

2 Cenni utili sui limiti 7

3 Stime di Somme 7

II Algoritmi 8

4 Tabella riassuntiva costi temporali 8

5 Algoritmi di ordinamento e costruzione 9
5.1 Insertion sort . . . 9
5.2 Merge . . . 10
5.3 Merge sort . . . 10
5.4 Build-Max-Heap (array) . . . 11
5.5 Heapify (array) . . . 11
5.6 Heap Sort . . . 12
5.7 Quick Sort . . . 12
5.8 Partition . . . 12
5.9 Selection . . . 12

III Strutture Dati 13

6 strutture dati lineari 13
6.1 Array . . . 13
6.2 Lista . . . 13
6.3 Pila . . . 13
6.4 Albero Binario . . . 13
6.5 Code di priorità . . . 14
6.6 Max-Heap . . . 14

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:

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

1.1 Notazione O-grande (e o-piccolo)

Notazione O-grande O Abbiamo due definizioni equivalenti:

- $O(g(n)) = \{f(n) |\ \exists c > 0\ \exists \bar{n}\ \forall n \geq \bar{n}\ f(n) \leq c \cdot g(N)\}$
- Date $f, g : \mathbb{N} \to \mathbb{R}^+$ **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**

- $\limsup\limits_{n\to\infty} |\frac{f(n)}{g(n)}| < \infty \Leftrightarrow f(n) = O(g(n))$

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:

- $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Leftrightarrow f(n) = o(g(n))$

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:

- $\Omega(g(n)) = \{f(n)|\ \exists c > 0\ \exists \bar{n} \in \mathbb{N}\ \forall n \geq \bar{n}\ f(n) \geq c \cdot g(N)\}$
- Date $f, g : \mathbb{N} \to \mathbb{R}^+$ **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**

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

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:

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

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

1.3 Notazione Theta

Notazione Theta Θ Abbiamo due definizioni equivalenti:

- $\Theta(g(n)) = \{f(n)|\ \exists c_1, c_2 > 0\ \exists \bar{n} \in \mathbb{N}\ \forall n \geq \bar{n}\ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$
- Date $f, g : \mathbb{N} \to \mathbb{R}^+$ **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**

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

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

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

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 Ω di f(n). Significa che g(n) cresce di più asintoticamente.

- $f(n) \in O(g(n)) \;\Leftrightarrow\; g(n) \in \Omega(f(n))$

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

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

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

- $f(n) \in o(g(n)) \;\Leftrightarrow\; g(n) \in \omega(f(n))$

Infine due funzioni piuttosto basilari, ovvero:

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

1.5 Comportamento rispetto alle operazioni

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

Abuso di notazione

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

Transitività

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

Additività

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

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 moltiplicatica

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

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

Trascurare termini additivi di ordine inferiore

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

Ovver 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**
Insertion Sort	$\Theta(n^2)$	$\Theta(n)$	stabile	inplace
Merge	$\Theta(n)$	$\Theta(n)$	stabile	non inplace
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$		non inplace
Heapify	$O(\log n)$			
Build Max Heap	$\Theta(n)$			inplace
Heap Sort	$O(n \log n)$		non stabile	
Quick Sort	$\Theta(n^2)$	$\Theta(n \log n)$		inplace
Selection Sort	$\Theta(n^2)$			
Counting Sort	$\Theta(n^2)$	$\Theta(n+k)$, $k \in O(\text{n})$	stabile	non inplace
Radix Sort		$d \cdot \Theta(n)$	stabile	inplace
Bucket Sort	$\Theta(n^2)$	$\Theta(n)$	dipende	non inplace

5 Algoritmi di ordinamento e costruzione

PROBLEMA Problema data una sequenza a1, a2, ..., an di numeri, trovare una permutazione tale che $a1 \leq a2 \leq ... \leq an$.
Soluzioni:

5.1 Insertion sort

Algorithm 1: InsertionSort

```
Data: A array, i indice, j indice
for i ← 2 to A.length do
    key ← A[i];
    j ← j - 1;
    while j > 0 && A[j] > key do
        A[j+1] ← A[j] ;
        j ← j - 1 ;
    A[j+1] ← 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 :

5.2 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 ← p;
  j ← r + 1;
  B ← new array of size q − p + 1;
  k ← 1;
  while i < r + 1 and j < q + 1 do
      if A[i] ≤ A[j] then
          B[k] ← A[i];
          i ← i + 1;
      else
          B[k] ← A[j];
          j ← j + 1;
      k ← k + 1;
  if i > r then
      for l ← j to q do
          B[k] ← A[l];
          k ← k + 1;
  else
      for l ← i to r do
          B[k] ← A[l];
          k ← k + 1;
```

5.3 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 ← ⌊(p+q)/2⌋;
      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.

5.4 Build-Max-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
H.heapsize ← H.length;
for i ← ⌊H.length/2⌋ downto 1 do
    Heapify(H,i);
```

5.5 Heapify (array)

Procedura che serve a trasformare una heap in una max-heap.
pre-condizioni: H[left(i)] e H[right(i)] sono max-heap.

Algorithm 5: Heapify

```
Input: Heap H, indice i
Output: Max-heap H
l ← left(i);
r ← right(i);
if l ≤ H.heapsize && H[l] > H[i] then
    m ← l (m è max);
else
    m ← i;
if r ≤ H.heapsize && H[r] > H[m] then
    m ← r;
if m ≠ i then
    scambia (H,i,m) ;
    Heapify(H,m);
```

Complessità Spaziale :

Complessità Temporale :

Correttezza :

5.6 Heap Sort

5.7 Quick Sort

5.8 Partition

5.9 Selection

Part III
Strutture Dati

6 strutture dati lineari

6.1 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 $\Theta(1)$

6.2 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 $\Theta(n)$.

6.3 Pila

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

6.4 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 $n-1$.

6.5 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

6.6 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:

left(i){             right(i){            parent(i){
    return 2i            return 2i + 1        return ⌊i/2⌋
    }                    }                    }
```