Algoritmi & Strutture Dati

Andrea Comar

October 2024

Contents

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)

1.1 Notazione O-grande

$O(g(n)) = \{f(n)|\ \exists c > 0\ \exists \bar{n}\ \forall n \geq \bar{n}\ f(n) \leq c \cdot g(n)\}$
oppure data $f, g : \mathbb{N} \to \mathbb{R}^+$ monotone crescenti
$f(n) = O(g(n))$ se $\exists c > 0\ \exists \bar{n}\ \forall n \geq \bar{n}\ f(n) \leq c \cdot g(n)$

La notazione O-grande rappresenta un limite superiore alla complessità di un algoritmo.

1.2 Notazione Omega

1.3 Notazione Theta

1.4 Notazione o-piccolo

1.5 Notazione omega-piccolo

1.6 Notazione theta-piccolo

2 Stime di somma

2.1 Somma aritmetica

$T(n) = \sum_{i=1}^{h(n)} f(i)$

2.2 Somma di potenze o di Gauss

$\sum_{i=1}^{h} i^k = \theta(h^{k+1})$, $k \in \mathbb{N}, k \geq 1$

2.3 Somma parziale della serie geometrica

3 Logaritmi

4 Equazioni ricorsive di complessità

4.1 Metodo di sostituzione

4.2 albero delle chiamate ricorsive

5 Induzione

Part II
Algoritmi

6 Algoritmi di ordinamento

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

6.1 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 :

6.2 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;
```

6.3 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.

6.4 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 4: 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 :

6.5 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.heapsize ← H.length;
for i ← ⌊H.length/2⌋ downto 1 do
  | Heapify(H,i);
```

Part III

Strutture Dati

7 strutture dati lineari

7.1 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 $\theta(1)$

7.2 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 $\theta(n)$.

7.3 Pila

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

7.4 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 $n-1$.

7.5 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

7.6 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⌋
    }                    }                    }
```