



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Appunti di Algoritmi e Strutture Dati

a.a. 2017/2018

Autore:
Timoty Granziero

Repository:
<https://github.com/Vashy/ASD-Notes>

22 marzo 2018

Indice

1	Lezione del 28/02/2018	3
1.1	Problem Solving	3
1.2	Cosa analizzeremo nel corso	4
1.2.1	Approfondimento sul tempo di esecuzione $T(n)$	4
1.3	Problema dell'ordinamento (sorting)	4
1.4	Insertion Sort	5
1.4.1	Invarianti e correttezza	6
2	Lezione del 02/03/2018	7
2.1	Modello dei costi	7
2.2	Complessità di IS	8
2.2.1	Caso migliore	8
2.2.2	Caso peggiore	8
2.2.3	Caso medio	9
2.3	Divide et Impera	9
2.4	Merge Sort	9
2.4.1	Invarianti e correttezza	11
3	Lezione del 07/03/2018	13
3.1	Approfondimento sull'induzione	13
3.1.1	Induzione ordinaria	13
3.1.2	Induzione completa	13
3.2	Complessità di Merge Sort	13
3.3	Confronto tra IS e MS	15
4	Lezione dell'08/03/2018	16
4.1	Notazione asintotica	16
4.1.1	Limite asintotico superiore	17
4.1.2	Limite asintotico inferiore	19
4.1.3	Limite asintotico stretto	20
4.2	Metodo del limite	21
4.3	Alcune proprietà generali	21
5	Lezione del 09/03/2018	22
5.1	Complessità di un problema	22
5.2	Esempio: limite inferiore per l'ordinamento basato su scambi .	22
5.3	Soluzione di ricorrenze	23
5.3.1	Metodo di sostituzione	24
6	Lezione del 14/03/2018	26

7	Lezione del 15/03/2018	29
7.1	Master Theorem	29
7.1.1	Esercizi (Master Theorem)	30
8	Lezione del 21/03/2018	34
8.1	Heapsort	34
8.1.1	Max Heap	35
	Appendices	35
A	Raccolta algoritmi	35
A.1	Insertion Sort	35
A.2	Merge Sort	35
A.3	Insertion Sort ricorsivo	36
A.3.1	Correttezza di Insertion-Sort(A, j)	36
A.3.2	Correttezza di Insert(A, j)	37
A.4	CheckDup	37
A.4.1	Correttezza di DMerge(A, p, q, r)	38
A.5	SumKey	38
A.5.1	Correttezza di Sum(A, key)	39
B	Esercizi	41
B.1	Ricorrenze	41

8 Lezione del 21/03/2018

Ordinamento Finora abbiamo visto due algoritmi di ordinamento, in cui avevamo le seguenti premesse:

IN: $a_1 \dots a_n$;

OUT: permutazione $a'_1 \dots a'_n$ ordinata.

In particolare, abbiamo concluso che:

- **Insertion Sort**: $O(n^2)$, basato su scambi;
- **Merge Sort**: $\Theta(n \log n)$, ma con un costo in termini di *memoria*.

Memoria

- **Insertion Sort**:

$input + 1$ variabile \Rightarrow spazio *costante* $\Theta(1)$ (detto “in loco”)

- **Merge Sort**: spazio con costo lineare.

$$\begin{aligned} S_{MS}(n) &= \max \left\{ S\left(\left\lfloor \frac{n}{2} \right\rfloor\right), S\left(\left\lceil \frac{n}{2} \right\rceil\right), \Theta(n) \right\} \\ &= \Theta(n) \end{aligned}$$

8.1 Heapsort

L'**Heapsort**¹ è un algoritmo di ordinamento basato su una struttura chiamata **heap**, che prende le caratteristiche positive di **Insertion Sort** e **Merge Sort**:

- in “loco” (spazio $\Theta(1)$);
- complessità $\Theta(n \log n)$.

Cos'è un heap? Un **heap** è una struttura dati basata sugli alberi che soddisfa la “proprietà di heap”: se A è un genitore di B, allora la chiave di A è ordinata rispetto alla chiave di B conformemente alla relazione d'ordine applicata all'intero heap.

Seguono alcune definizioni.

¹Anche qui, si consiglia di dare un occhio ad altre fonti. In classe, sono stati viste molte rappresentazioni grafiche degli heap, e, come già detto, in \LaTeX non è per me facile rappresentarli.

Altezza: è la distanza dalla radice alla foglia più distante;

Albero completo: è un albero di altezza h con $\sum_{i=0}^h 2^i - 1$ nodi;

Albero quasi completo: è un albero completo a tutti i livelli eccetto l'ultimo, in cui possono mancare delle foglie.

Gli heap verranno rappresentati in array monodimensionali, nel modo descritto di seguito:

$$\forall i > 0$$

- $A[i]$ è il nodo genitore;
- $A[2i]$ è il figlio sx del nodo $A[i]$;
- $A[2i+1]$ è il figlio dx.

Inoltre, ogni array A sarà dinamico, e avrà:

- $A.length$ potenziale spazio, capacità massima dell'array;
- $A.heapsize$ celle effettive dell'array.

Vediamo alcune funzioni di utilità che verranno usate.

LEFT(i)

// restituisce il figlio sx del nodo i

1 **return** $2 * i$

RIGHT(i)

// restituisce il figlio dx del nodo i

1 **return** $2 * i + 1$

PARENT(i)

// restituisce il genitore del nodo i

1 **return** $\lfloor i/2 \rfloor$

8.1.1 Max Heap

Max Heap è uno heap che soddisfa la seguente proprietà:

$$\begin{aligned} &\forall \text{ nodo } A[i], \\ &A[i] \geq \text{discendenti} \\ &\Downarrow \\ &A[i] \geq A[\text{Left}(i)], A[\text{Right}(i)] \end{aligned}$$

Osservazioni

- Uno heap con un solo elemento è un *Max Heap*.
- Dati due Max Heap T_1 e T_2 e un nodo N , possiamo “combinarli” in uno heap con N come radice, T_1 come *left* e T_2 come *right*.

Ecco ora una procedura che, dato un nodo i , trasforma in un Max Heap il sotto-albero eradicato in esso (con radice i).

MAXHEAPIFY(A, i)

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if ( $l \leq A.\text{heapsize}$ ) and ( $A[l] > A[i]$ )
4  else
5       $max = i$ 
6  if ( $max \neq i$ )
7       $A[i] \leftrightarrow A[max]$ 
8      MAXHEAPIFY( $A, max$ )
```

L'algoritmo ha un costo di $O(h)$, con h altezza del sotto-albero radicato in i , con

$$O(h) \cong O(\log n) \quad (\text{Omessa la dimostrazione})$$

Ora vogliamo scrivere una procedura che costruisce un *Max Heap* da un array qualunque.

Quali sono i nodi foglia?

- Se $i \geq \lfloor \frac{n}{2} \rfloor + 1$

$$\begin{aligned} 2i &= 2\left(\frac{n}{2} + 1\right) \geq n + 2 - 1 = n + 1 \\ &\Rightarrow i \text{ foglia} \end{aligned}$$

- Se $i \leq \lfloor \frac{n}{2} \rfloor$

$$\begin{aligned} 2i &= 2\lfloor \frac{n}{2} \rfloor \leq n \\ &\Rightarrow i \text{ non foglia} \end{aligned}$$

BUILDMAXHEAP(A)

```
1   $A.\text{heapsize} = A.\text{length}$ 
2  for  $i = \lfloor A.\text{length}/2 \rfloor$  down to 1
3      MAXHEAPIFY( $A, i$ )
```