4 Ordinamento (cont.)

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:

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

Memoria

- `InsertionSort`:

$$\textit{input} \ + \ 1 \text{ variabile} \Rightarrow \text{spazio } \textbf{costante } \Theta(1) \text{ (detto ``in loco'')}$$

- `MergeSort`: spazio con costo lineare.

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

4.1 Heapsort

L'Heapsort[1] è un algoritmo di ordinamento basato su una struttura chiamata heap, che prende le caratteristiche positive di `InsertionSort` e `MergeSort`:

- 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.

[1] Anche 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 e le foglie presenti sono addossate a sinistra.

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
    return 2 * i
```

```
RIGHT(i)
    // restituisce il figlio dx del nodo i
    return 2 * i + 1
```

```
PARENT(i)
    // restituisce il genitore del nodo i
    return ⌊i/2⌋
```

4.1.1 Max Heap

Max Heap è uno heap che soddisfa la seguente proprietà:

$$\forall \text{ nodo } A[i],$$
$$A[i] \geq \text{discendenti}$$
$$\Downarrow$$
$$A[i] \geq A[\mathbf{Left(i)}],\ A[\mathbf{Right(i)}]$$

Equivalentemente

$$\forall \text{ nodo } A[i],$$
$$A[i] \leq \text{antenati}$$
$$\Downarrow$$
$$A[i] \leq A[\mathbf{Parent(i)}]$$

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).

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if (l ≤ A.heapsize) and (A[l] > A[i])
    max = l
else
    max = i
if (r ≤ A.heapsize) and (A[r] > A[max])
    max = r
if (max ≠ i)
    A[i] ↔ A[max]
    MAX-HEAPIFY(A, max)
```