Heapsort

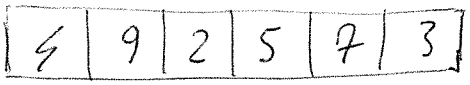
Una coda a priorità fornisce un semplice algoritmo di ordinamento:

- gli n elementi da ordinare vengono inseriti nella coda, uno per uno, con n operazioni di ENQUEUE
- per $n-1$ volte, si scambia la radice con l'ultima foglia, si riduce l'heap di un elemento, e si ristabilisce la proprietà di heap con RIORGANIZZAZIONE.

L'algoritmo risultante prende il nome di HeapSort.

```
HeapSort (heapArray)
heapSize = 0;
FOR (i = 0; i < n; i = i+1)
{   ENQUEUE (heapArray[i]);
}
WHILE (heapSize > 0){
   Scambia (heapSize-1, 0);
   heapSize = heapSize - 1;
   RIORGANIZZAZIONE (0);
}
```

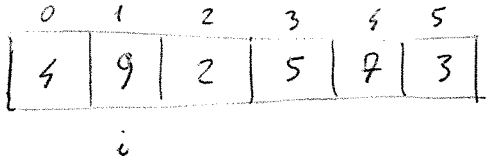
Esempio:



← Array da ordinare

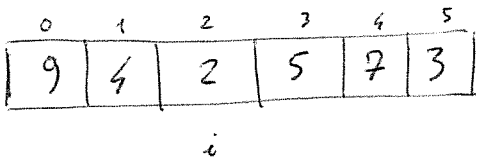
Nella prima fase, usando lo stesso array, viene costruita la coda a priorità. In particolare, durante l'i-esimo passo l'array è diviso in "due parti": i primi i elementi appartengono alla coda, gli altri sono ancora da inserire.

Prima iterazione:



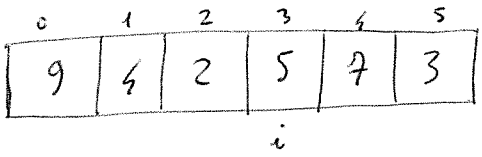
heapSize = 1

... seconda



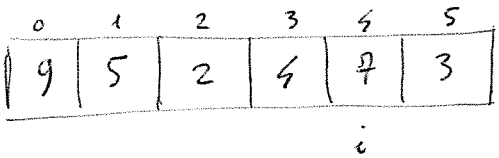
heapSize = 2

... terza



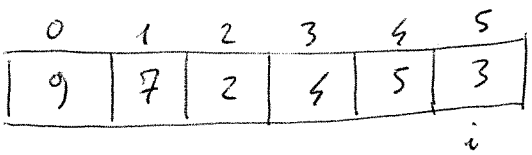
heapSize = 3

... quarta



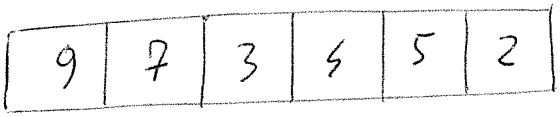
heapSize = 4

... quinta



heapSize = 5

... sesta

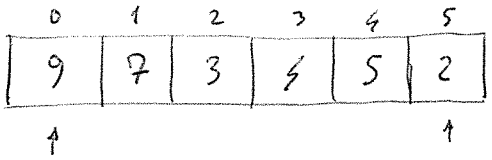


heapSize = 6

... heap costruito!

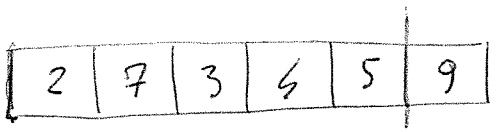
Nel secondo ciclo, per i che va da heapSize a 1, si considera ad ogni passo l'heap racchiuso tra gli indici 0 e i-1:
la radice (il massimo di questo heap) si scambia con l'ultima foglia.

Configurazione iniziale



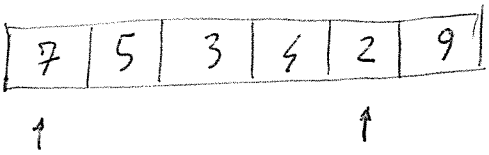
heapSize = 6

Prima iterazione: scambio e riorganizzare



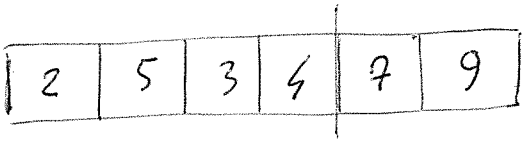
heapSize = 5

↓ RIORGANIZZAZIONE



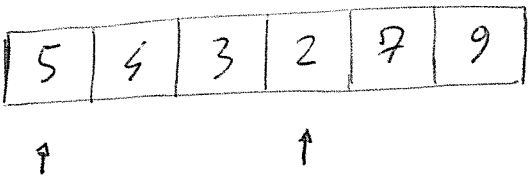
heapSize = 5

... seconda:



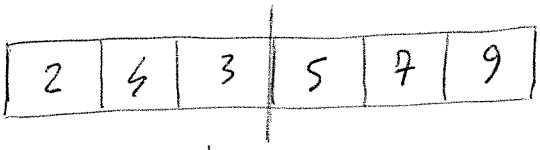
heapSize = 4

↓ RIORGANIZZAZIONE



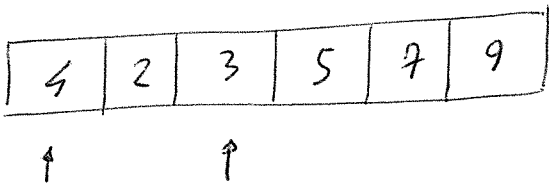
heapSize = 4

... terza:



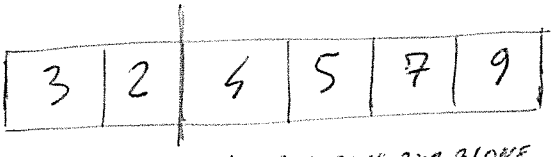
heapSize = 3

↓ RIORGANIZZAZIONE



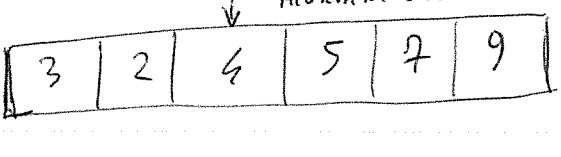
heapSize = 3

... quarta:



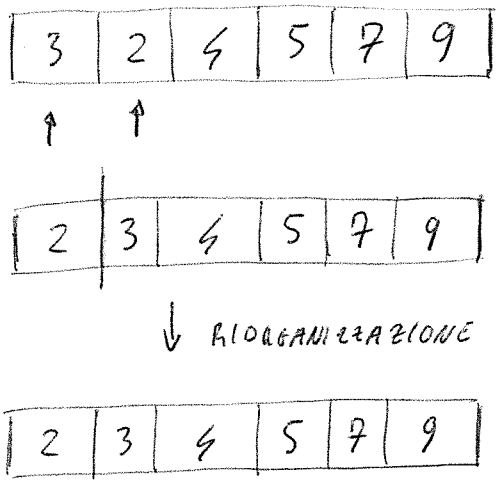
heapSize = 2

↓ RIORGANIZZAZIONE



heapSize = 2

... quinta :



... resta : non cambia nulla, porta solo all'uscita del ciclo.

(si può evitare arrestando il ciclo prima... WHILE (heapSize > 1))

Osservazione: HeapSort ordina IN LOCO.

Teorema. HeapSort ordina un array di n elementi in tempo $O(n \log n)$.

Dimostrazione. Proviamo prima di tutto che HeapSort è corretto. Dal codice si evince che, una volta costruito l'heap iniziale, al termine della j-esima iterazione del ciclo WHILE, l'heap restante ha $n-j$ elementi, rappresentati da heapArray$[0..n-j-1]$, e gli elementi in heapArray$[n-j, n-1]$ sono già ordinati.

Ogni iterazione riduce la taglia dell'heap di una unità ed estende di una unità la taglia della parte dell'array ordinata.

Pertanto, al termine della n-esima iterazione, l'array risulta ordinato.

Circa la complessità:

- Il costo per la costruzione dell'heap iniziale è $O(n \log n)$. Inseriamo n elementi e ogni inserimento costa $O(\log n)$

- Il costo del ciclo WHILE è $O(n \log n)$. Effettuiamo n scambi ed ogni scambio comporta una riorganizzazione dell'heap restante, che costa $O(\log n)$

Discende che, in totale, l'HeapSort ha un costo $O(n \log n)$.

Come evidenziato in lezioni precedenti, per grosse taglie dell'input, HeapSort è drasticamente più efficiente di SelectionSort e InsertionSort che hanno complessità quadratica.

Lower bound per algoritmi di ordinamento basati su confronti.

Teorema. Ogni algoritmo di ordinamento basato su confronti di elementi richiede $\Omega(n \log n)$ confronti nel caso pessimo.

Dimostrazione. Usiamo un argomento "combinatorio".

Supponiamo che gli elementi da ordinare siano distinti. Ogni confronto dà luogo a due possibili esiti: $a < b$ oppure $a > b$. Pertanto, con un confronto posso discernere tra due possibili "situazioni" ($a < b$ oppure $a > b$). Con due confronti similmente posso discernere tra $4 = 2^2$ possibili situazioni.

Per esempio, dati a, b e c, posso discernere tra

$a < b$	$a < b$	$a > b$	$a > b$
$a < c$	$a > c$	$a < c$	$a > c$

4 situazioni

ed individuare le relazioni tra a e b e a e c.

In generale, disponendo di t confronti, posso discernere tra 2^t possibili situazioni.

Ordinare n numeri significa individuare uno tra i possibili $n!$ riarrangiamenti (permutazioni) degli elementi.

Un algoritmo di ordinamento basato su confronti usa, appunto, i confronti tra elementi per capire come spostare gli elementi per ottenere il ri-arrangiamento giusto.

Per esempio, ordinare 3 numeri significa individuare uno tra i possibili $3! = 6$ ri-arrangiamenti, cioè:

1. $a_1 < a_2 < a_3$ $\quad (a_1, a_2, a_3)$
2. $a_1 < a_3 < a_2$ $\quad (a_1, a_3, a_2)$
3. $a_2 < a_1 < a_3$ $\quad (a_2, a_1, a_3)$
4. $a_2 < a_3 < a_1$ $\quad (a_2, a_3, a_1)$
5. $a_3 < a_1 < a_2$ $\quad (a_3, a_1, a_2)$
6. $a_3 < a_2 < a_1$ $\quad (a_3, a_2, a_1)$

Con il test $a_1 \overset{?}{<} a_2$, l'algoritmo restringe l'attenzione o ai ri-arrangiamenti 1., 2., e 5; oppure 3., 4., e 6. Supponiamo che risulti $a_1 < a_2$.

Con il test successivo $a_1 \overset{?}{<} a_3$, che supponiamo soddisfatto, l'algoritmo restringe ulteriormente ai rearrangiamenti 1., e 2.

Con l'ultimo test $a_2 \overset{?}{<} a_3$, che supponiamo non soddisfatto, individua il ri-arrangiamento 2. (a_1, a_3, a_2)

Ovviamente ogni specifico algoritmo di ordinamento basato su confronti, ad ogni passo, sposta gli elementi secondo la propria strategia. SelectionSort, InsertionSort ed HeapSort come abbiamo visto usano strategie diverse.

Indipendentemente però dalla strategia specifica, il numero t di confronti necessari ad individuare il rearrangiamento giusto deve essere tale da poter discernere tra tutti i possibili rearrangiamenti, cioè

$$2^t \geq n!$$

D'altra parte è facile vedere che:

$$n! = n \cdot (n-1)(n-2) \cdots \left(\frac{n}{2}+1\right)\left(\frac{n}{2}\right) \cdots 1 > \underbrace{\frac{n}{2} \cdots \frac{n}{2}}_{n/2 \text{ volte}}$$

$$= \left(\frac{n}{2}\right)^{\frac{n}{2}}.$$

Pertanto

$$2^t \geq n! > \left(\frac{n}{2}\right)^{\frac{n}{2}},$$

da cui

$$t > \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \cdot \log \frac{n}{2},$$

ovvero

$$t = \Omega(n \log n)$$

□

Osservazioni. Poiché l'heapsort opera per confronti e ha complessità $O(n \log n)$, discende che l'algoritmo è ottimo.

Inoltre, la complessità dell'ordinamento tramite algoritmi basati su confronti ha complessità $\Theta(n \log n)$.

Esercizio. Provare che $\frac{n}{2}\log\frac{n}{2} = \Omega(n \log n)$.

Occorre dimostrare che $\exists\ c, n_0 : \frac{n}{2}\log\frac{n}{2} \geq c \cdot n \log n$, $\forall n \geq n_0$

Si noti che: $\frac{n}{2}\log\frac{n}{2} = \frac{n}{2}(\log n - \log 2) = \frac{n}{2}(\lg n - 1)$

$$\frac{n}{2}(\log n - 1) \geq c \cdot n \log n \iff$$

$$n \log n - n \geq 2c\, n \log n \iff$$

$$(1-2c)\, n \cdot \lg n \geq n \iff$$

$$\log n \geq \frac{1}{(1-2c)}.$$

Scegliendo $c = \frac{1}{4}$, la condizione risulta:

$$\log n \geq \frac{1}{(1-2\cdot\frac{1}{4})} = 2$$

Pertanto, fissando $n_0 = 4$, la disuguaglianza è soddisfatta.

Esercizio. Scrivere un frammento di codice che trasformi la rappresentazione esplicita di un albero completo a sinistra nella corrispondente rappresentazione implicita.

```
Implicita (u, nodo):
  ultimo = attuale = 0;
  nodo [attuale] = u;
  WHILE (attuale ≤ ultimo)
  { u = nodo [attuale];
    IF (u.sx != NULL) {
      nodo [ultimo + 1] = u.sx;
      ultimo = ultimo + 1;
    }
    IF (u.dx != NULL) {
      nodo [ultimo + 1] = u.dx;
      ultimo = ultimo + 1;
    }
    attuale = attuale + 1;
  }
```

Invocata sulla radice r dell'albero in forma esplicita e l'array nodo che rappresenterà implicitamente l'albero

/* se c'è, aggiunge il figlio sinistro di u

/* se c'è, aggiunge il figlio destro di u

/* passa al successivo

Si noti che il codice esamina l'albero per livelli, inserendo man mano in fondo all'array i nodi incontrati.