

Algoritmi e Strutture Dati

HeapSort II

Complessità di Heapify

$$T(n) = \max(O(1), \max(O(1), T(?) + O(1)))$$

```

H
  l = SINISTRO(i)
  r = DESTRO(i)
  IF l ≤ heapsize[A] AND A[l] > A[i]
    THEN maggiore = l
  ELSE maggiore = i
  IF r ≤ heapsize[A] AND A[r] > A[maggiore]
    THEN maggiore = r
  IF maggiore ≠ i } = O(1)
  THEN "scambia A[i] e A[maggiore]"
  T(?) = { Heapify(A, maggiore)
  
```

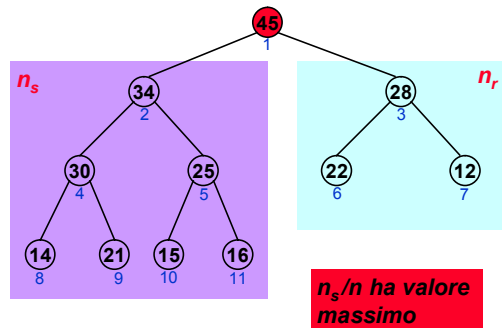
Complessità di Heapify: caso peggiore

$$T(n) = \max(O(1), \max(O(1), T(?) + O(1)))$$

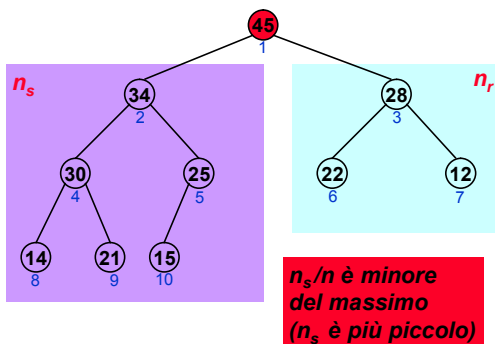
Nel **caso peggiore** Heapify ad ogni chiamata ricorsiva, viene eseguito su un numero di nodi che è minore dei $2/3$ del numero di nodi correnti n .

Cioè il numero di nodi n_s del sottoalbero su cui Heapify è chiamato ricorsivamente è al più $2/3 n$ (o $n_s \leq 2/3 n$)

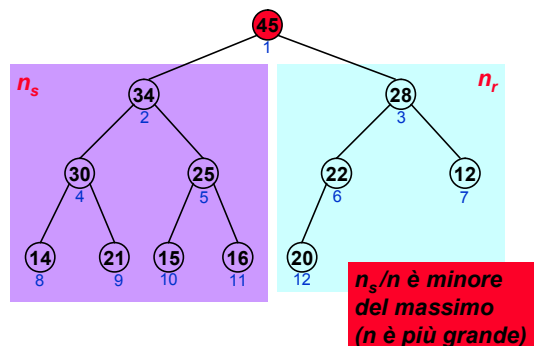
Complessità di Heapify: caso peggiore

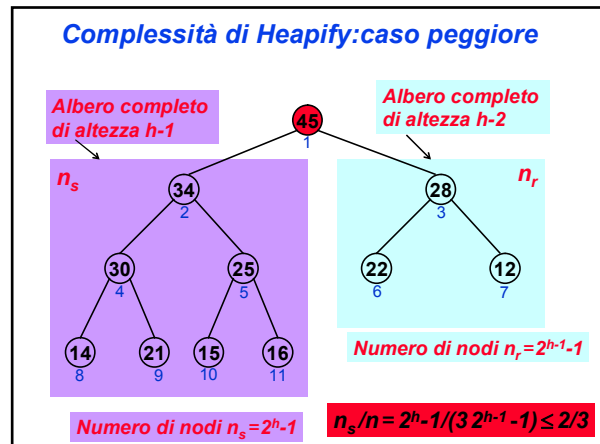
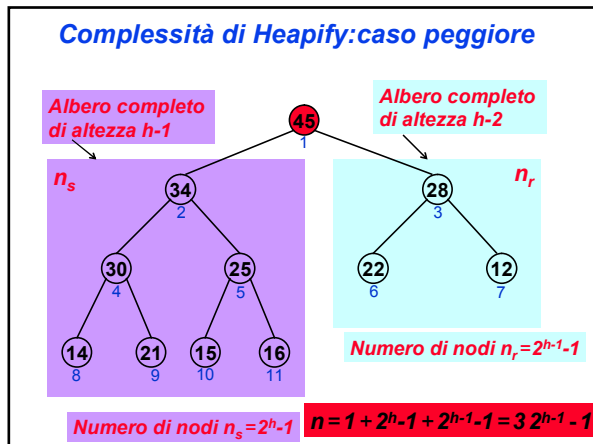


Complessità di Heapify: caso peggiore



Complessità di Heapify: caso peggiore





Complessità di Heapify: caso peggiore

$T(n) = \max(O(1), \max(O(1), T(?) + O(1))$
 $\leq \max(O(1), \max(O(1), T(2n/3) + O(1))$
 $\leq T(2n/3) + \Theta(1)$

$T'(n) = T'(2n/3) + \Theta(1)$

Proviamo ad applicare il Metodo Iterativo!

$T'(n) = \Theta(\log n)$

Complessità di Heapify: caso peggiore

$T(n) = \max(O(1), \max(O(1), T(?) + O(1))$
 $\leq \max(O(1), \max(O(1), T(2n/3) + O(1))$
 $\leq T(2n/3) + \Theta(1)$

Quindi

$T(n) = O(\log n)$

Heapify impiega tempo proporzionale all'altezza dell'albero su cui opera !

Complessità di Heapify: caso migliore

$T(n) = T(?) + O(1)$

Nel caso migliore Heapify ad ogni chiamata ricorsiva, viene eseguito su un numero di nodi che è maggiore di $1/3$ del numero di nodi correnti n .

Cioè il numero di nodi n_s del sottoalbero su cui Heapify è chiamato ricorsivamente è al più $1/3 n$ (o $n_s \geq 1/3 n$)

Complessità di Heapify: caso migliore

$T(n) = T(?) + O(1)$
 $\geq T(n/3) + \Theta(1)$

$T'(n) = T'(n/3) + \Theta(1)$

Applicando il Metodo Iterativo!

$T'(n) = \Theta(\log n)$ quindi $T(n) = \Omega(\log n)$

Costruisci Heap: intuizioni

Costruisci-Heap(A): utilizza l'algoritmo **Heapify**, per inserire ogni elemento dell'array in uno **Heap**, risistemando sul posto gli elementi:

- gli ultimi $\lceil n/2 \rceil$ elementi dell'array sono foglie, cioè radici di sottoalberi vuoti, quindi sono già degli **Heap**
- è sufficiente inserire nello **Heap** solo i primi $\lfloor n/2 \rfloor$ elementi, utilizzando **Heapify** per ripristinare la proprietà **Heap** sul sottoalbero del nuovo elemento.

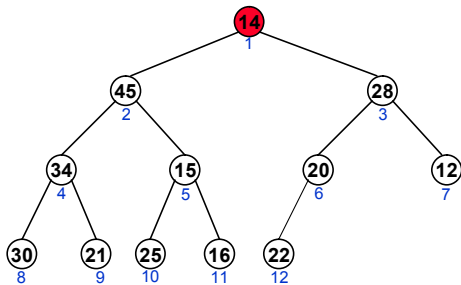
Costruisci Heap

```

Costruisci-Heap(A)
  heapsize[A] = length[A]
  FOR i =  $\lfloor \text{length}[A]/2 \rfloor$  DOWNTO 1
    DO Heapify(A, i)
    
```

Costruisci Heap

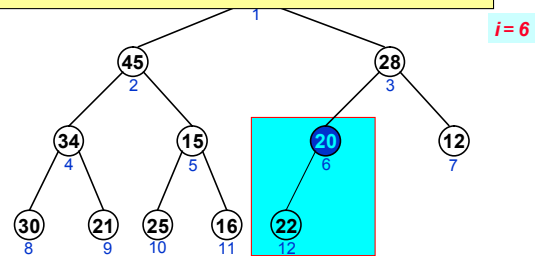
1 2 3 4 5 6 7 8 9 10 11 12
 14 45 28 34 15 20 12 30 21 25 16 22



Costruisci Heap

```

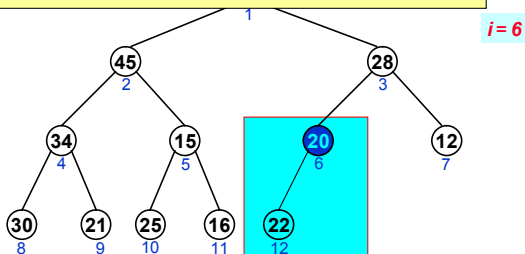
Costruisci-Heap(A)
  heapsize[A] = length[A]
  FOR i =  $\lfloor \text{length}[A]/2 \rfloor$  DOWNTO 1
    DO Heapify(A, i)
    
```



Costruisci Heap

```

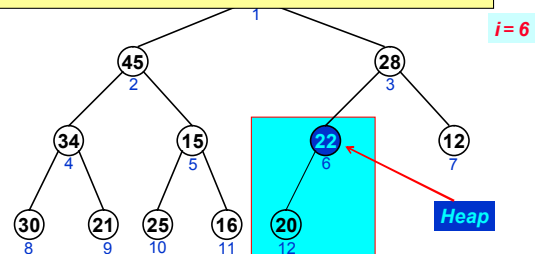
Costruisci-Heap(A)
  heapsize[A] = length[A]
  FOR i =  $\lfloor \text{length}[A]/2 \rfloor$  DOWNTO 1
    DO Heapify(A, i)
    
```



Costruisci Heap

```

Costruisci-Heap(A)
  heapsize[A] = length[A]
  FOR i =  $\lfloor \text{length}[A]/2 \rfloor$  DOWNTO 1
    DO Heapify(A, i)
    
```



Costruisci Heap

Costruisci-Heap (A)
 heapsize[A] = length[A]
 FOR i = $\lfloor \text{length}[A]/2 \rfloor$ DOWNTO 1
 DO Heapify(A, i)

i = 5

Costruisci Heap

Costruisci-Heap (A)
 heapsize[A] = length[A]
 FOR i = $\lfloor \text{length}[A]/2 \rfloor$ DOWNTO 1
 DO Heapify(A, i)

i = 5

Costruisci Heap

Costruisci-Heap (A)
 heapsize[A] = length[A]
 FOR i = $\lfloor \text{length}[A]/2 \rfloor$ DOWNTO 1
 DO Heapify(A, i)

i = 4

Costruisci Heap

Costruisci-Heap (A)
 heapsize[A] = length[A]
 FOR i = $\lfloor \text{length}[A]/2 \rfloor$ DOWNTO 1
 DO Heapify(A, i)

i = 4

Costruisci Heap

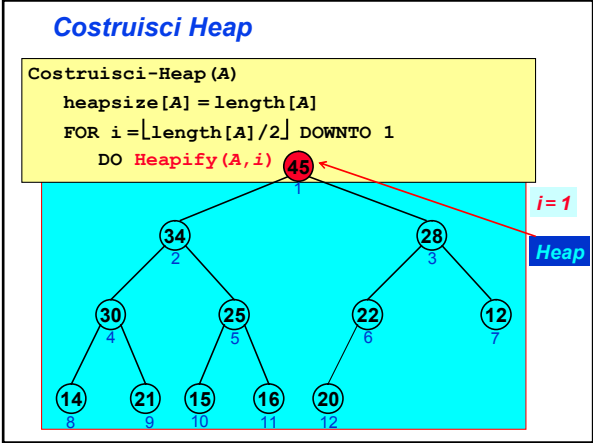
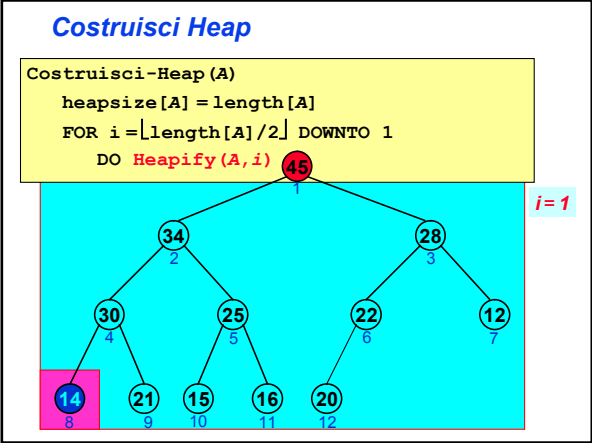
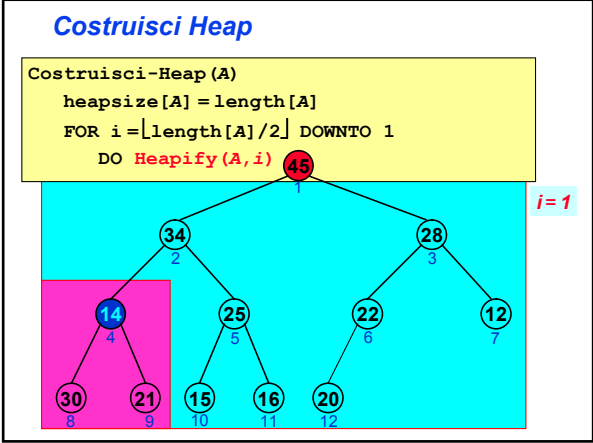
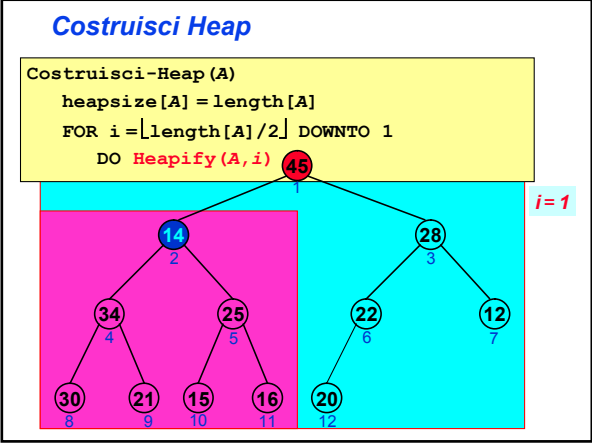
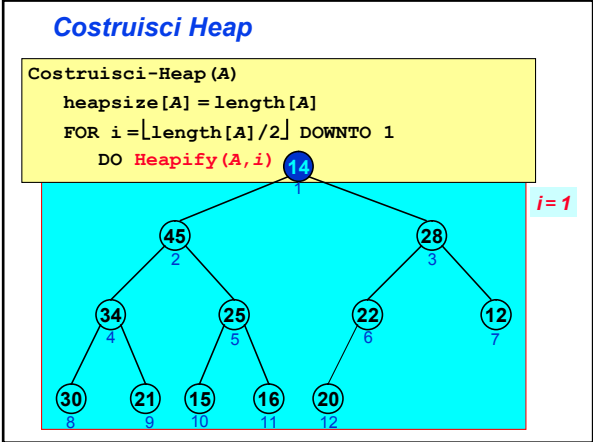
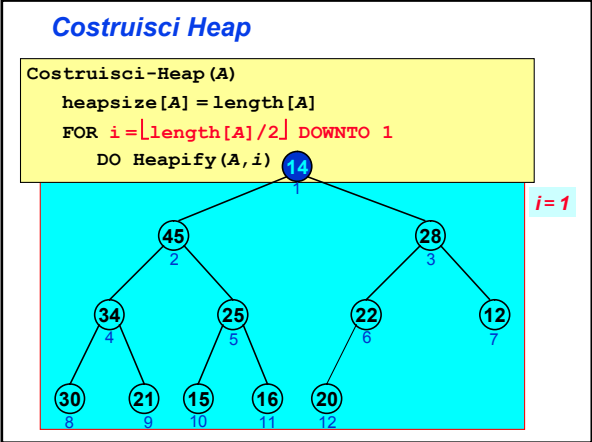
Costruisci-Heap (A)
 heapsize[A] = length[A]
 FOR i = $\lfloor \text{length}[A]/2 \rfloor$ DOWNTO 1
 DO Heapify(A, i)

i = 3

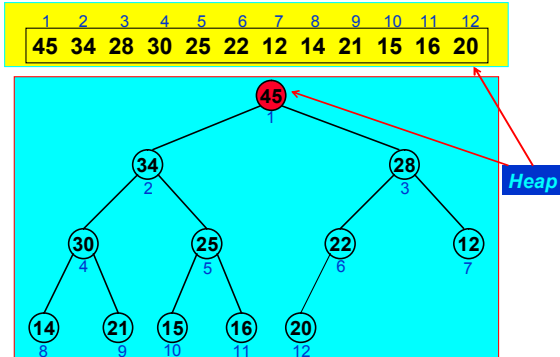
Costruisci Heap

Costruisci-Heap (A)
 heapsize[A] = length[A]
 FOR i = $\lfloor \text{length}[A]/2 \rfloor$ DOWNTO 1
 DO Heapify(A, i)

i = 2



Costruisci Heap



Complessità di Costruisci Heap

```

Costruisci-Heap (A)
  heapsize[A] = length[A] } = O(1)
  FOR i = [length[A]/2] DOWNTO 1 } = O(?)
    DO Heapify (A, i)
  
```

Complessità di Costruisci Heap

$$T(n) = \max(O(1), O(?)) = \max(O(1), O(f(n)))$$

Poiché *Heapify* viene chiamata $n/2$ volte si potrebbe ipotizzare

$$f(n) = O(n \log n)$$

e quindi

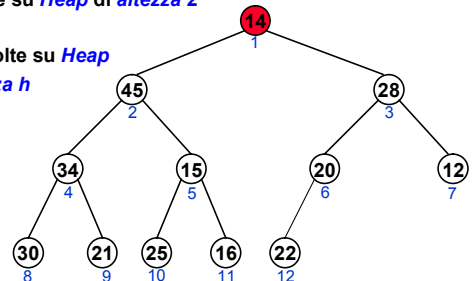
$$T(n) = \max(O(1), O(n \log n)) = O(n \log n)$$

ma....

Complessità di Costruisci Heap

Costruisci-Heap chiama *Heapify*

- $n/2$ volte su *Heap* di altezza 0 (non eseguito)
- $n/4$ volte su *Heap* di altezza 1
- $n/8$ volte su *Heap* di altezza 2
- ...
- $n/2^{h+1}$ volte su *Heap* di altezza h



Complessità di Costruisci Heap

$$T(n) = \max(O(1), O(?)) = \max(O(1), O(f(n)))$$

$$f(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

Costruisci-Heap chiama *Heapify*

- $n/2$ volte su *Heap* di altezza 0 (in realtà non eseguito)
- $n/4$ volte su *Heap* di altezza 1
- $n/8$ volte su *Heap* di altezza 2
- ...
- $n/2^{h+1}$ volte su *Heap* di altezza h

Complessità di Costruisci Heap

$$T(n) = \max(O(1), O(?)) = \max(O(1), O(f(n)))$$

$$f(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$$

Complessità di Costruisci Heap

$$T(n) = \max(O(1), O(?)) = \max(O(1), O(f(n)))$$

$$\begin{aligned} f(n) &= \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \\ &= O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} h / 2^h\right) \\ &= O\left(\frac{n}{2} \sum_{h=0}^{\infty} h / 2^h\right) \end{aligned}$$

Complessità di Costruisci Heap

$$T(n) = \max(O(1), O(?)) = \max(O(1), O(f(n)))$$

$$\begin{aligned} f(n) &= \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \\ &= O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} h / 2^h\right) \\ &= O\left(\frac{n}{2} \sum_{h=0}^{\infty} h / 2^h\right) \\ &= O(2n/2) \end{aligned}$$

$\sum_{h=0}^{\infty} hx^h = \frac{x}{(1-x)^2}$ $\frac{x}{(1-x)^2} = 2$
 $x = 1/2 \leq 1$

Complessità di Costruisci Heap

$$T(n) = \max(O(1), O(?)) = \max(O(1), O(f(n)))$$

$$\begin{aligned} f(n) &= \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \\ &= O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} h / 2^h\right) \\ &= O\left(\frac{n}{2} \sum_{h=0}^{\infty} h / 2^h\right) \\ &= O(n) \end{aligned}$$

Complessità di Costruisci Heap

$$T(n) = \max(O(1), O(?)) = \max(O(1), O(f(n)))$$

$$f(n) = O(n)$$

$$T(n) = \max(O(1), O(n)) = O(n)$$

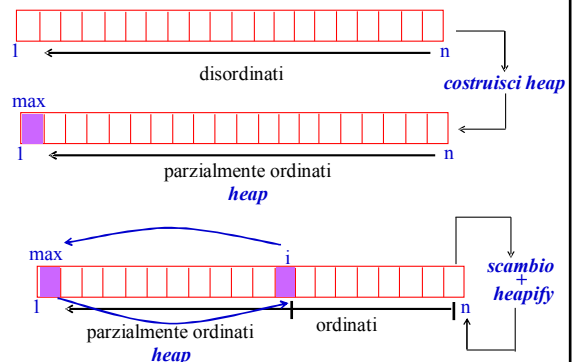
Costruire uno Heap di n elementi è poco costoso, al più costa O(n) !

Heap Sort: intuizioni

Heap-Sort: è una variazione di **Select-sort** in cui la ricerca dell'elemento massimo è facilitata dal mantenimento della sequenza in uno heap:

- si costruisce uno **Heap** a partire dall'array non ordinato in input.
- viene sfruttata la proprietà degli **Heap** per cui la radice **A[1]** dello **Heap** è sempre il massimo:
 - scandisce tutti gli elementi dell'array a partire dall'ultimo e ad ogni iterazione
 - la radice **A[1]** viene scambiata con l'elemento nell'ultima posizione corrente dello **Heap**
 - viene ridotta la dimensione dello **Heap** e
 - ripristinato lo **Heap** con **Heapify**

Heap Sort: intuizioni



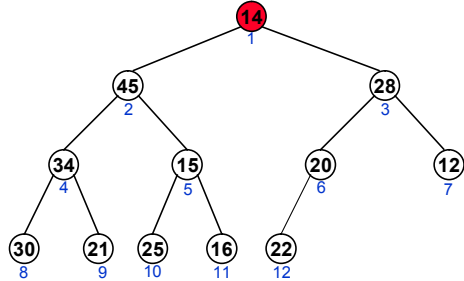
Heap Sort

```
Select-Sort(A)
FOR i = length[A] DOWNTO 2
  DO max = Findmax(A,i)
     "scambia A[max] e A[i]"
```

```
Heap-Sort(A)
  Costruisci-Heap(A)
FOR i=length[A] DOWNTO 2
  DO /* elemento massimo in A[1] */
     "scambia A[1] e A[i]"
     /* ripristina lo heap */
     heapsize[A] = heapsize[A]-1
     Heapify(A,1)
```

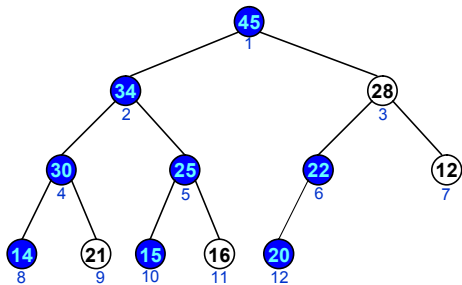
Heap Sort

```
Heap-Sort(A)
  Costruisci-Heap(A)
  ...
```



Heap Sort

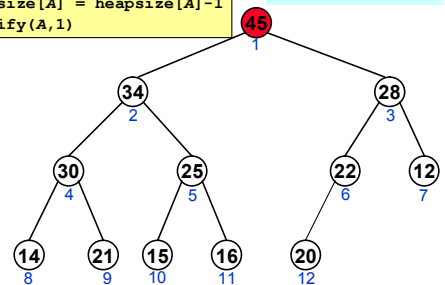
```
Heap-Sort(A)
  Costruisci-Heap(A)
  ...
```



Heap Sort

```
Heap-Sort(A)
  ...
  FOR i = length[A] DOWNTO 2
    DO "scambia A[1] e A[i]"
       heapsize[A] = heapsize[A]-1
       Heapify(A,1)
```

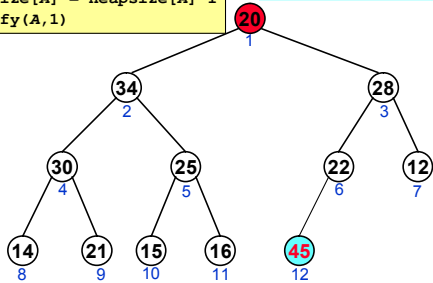
i = 12
heapsize[A] = 12



Heap Sort

```
Heap-Sort(A)
  ...
  FOR i = length[A] DOWNTO 2
    DO "scambia A[1] e A[i]"
       heapsize[A] = heapsize[A]-1
       Heapify(A,1)
```

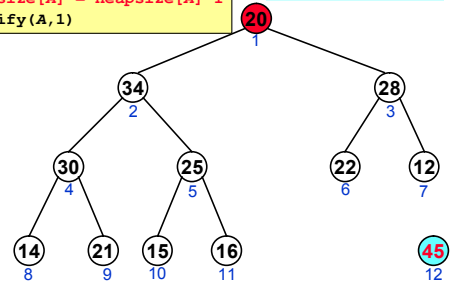
i = 12
heapsize[A] = 12

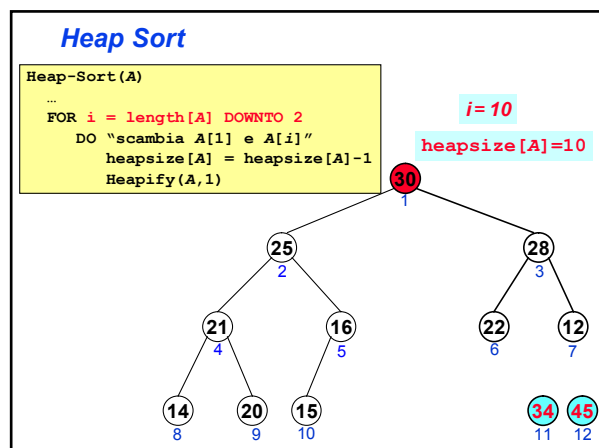
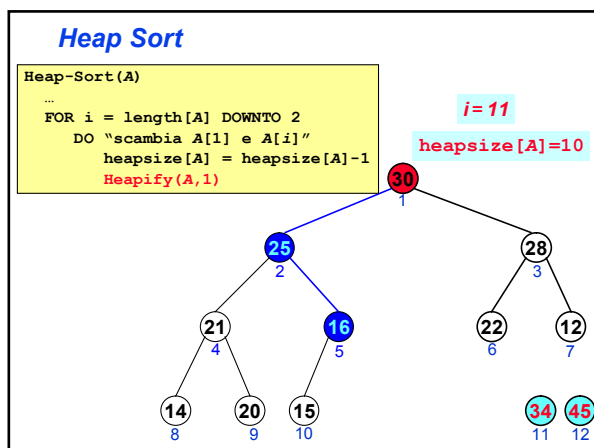
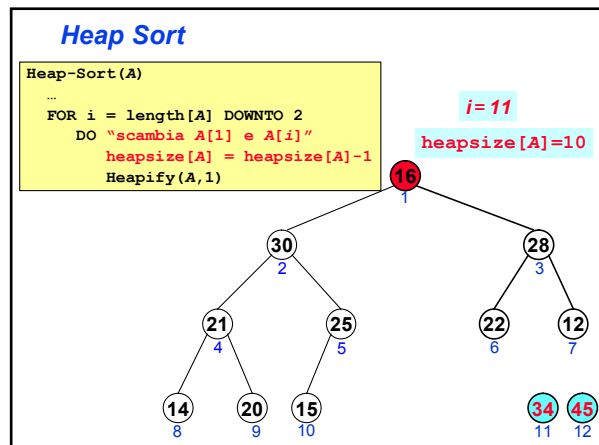
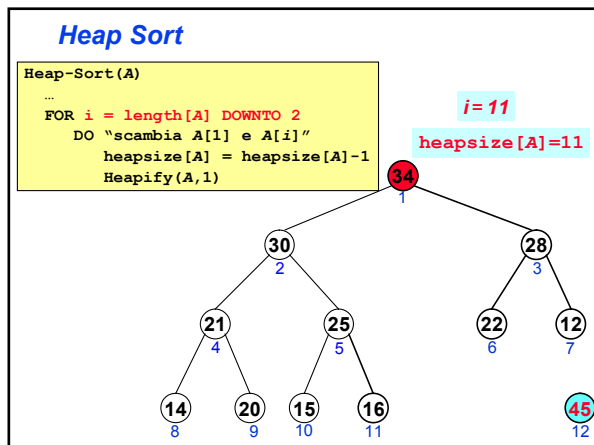
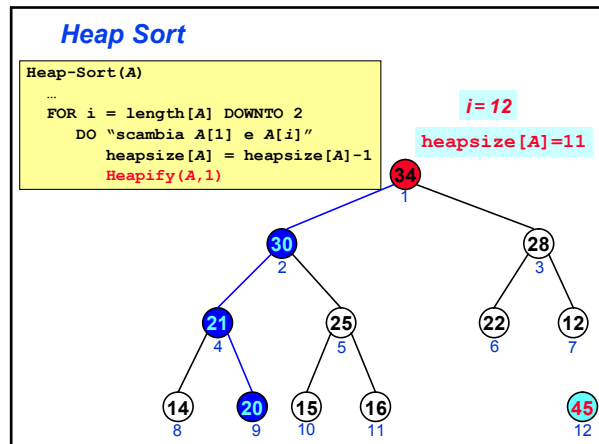
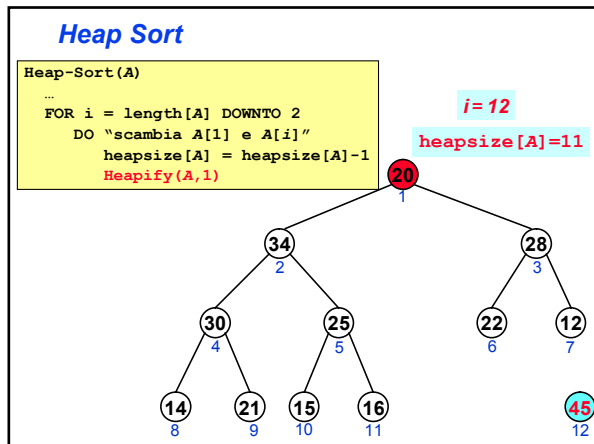


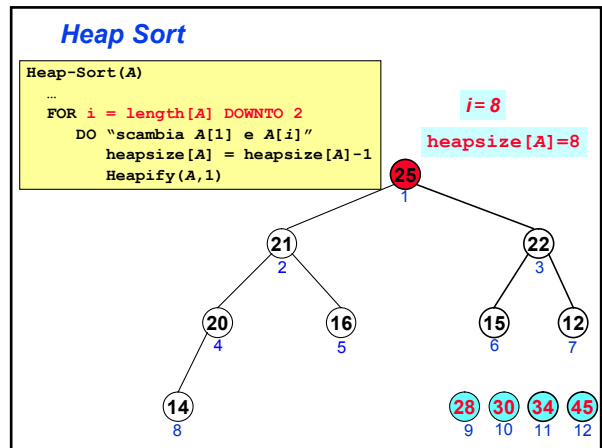
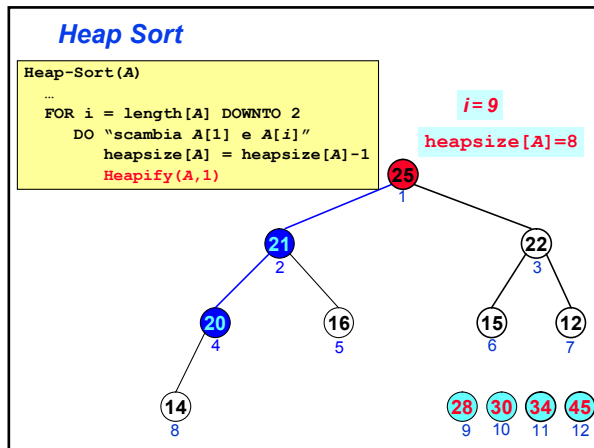
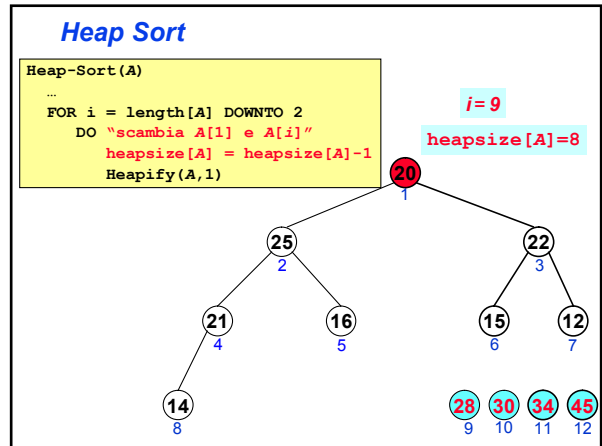
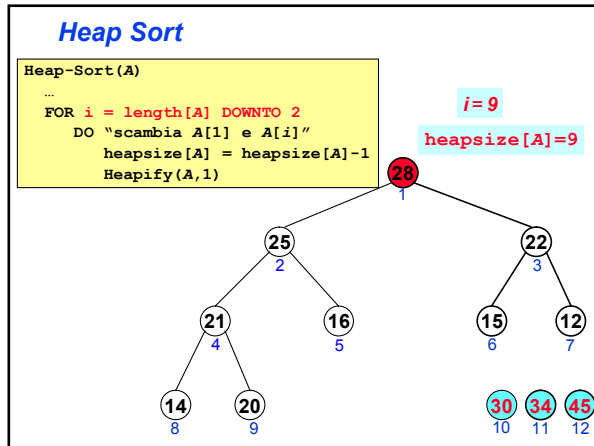
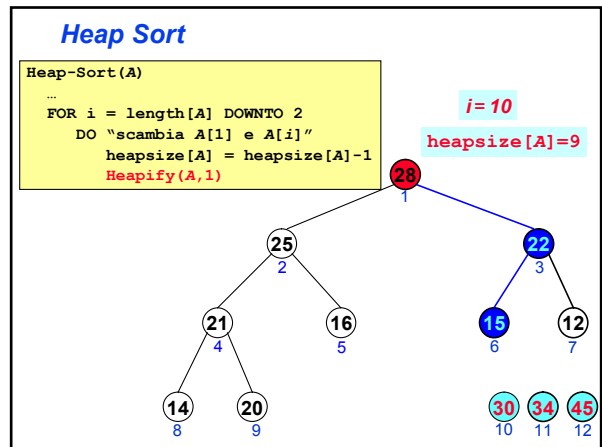
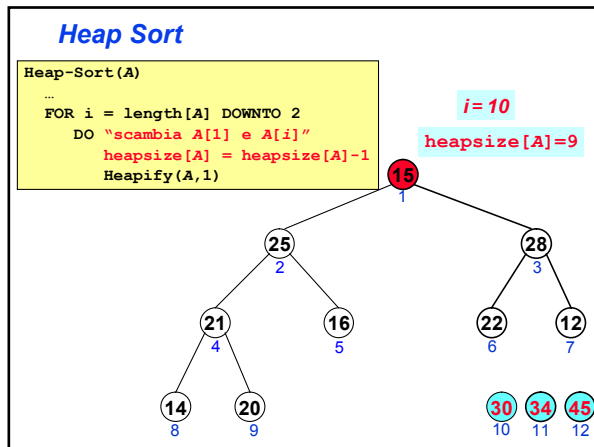
Heap Sort

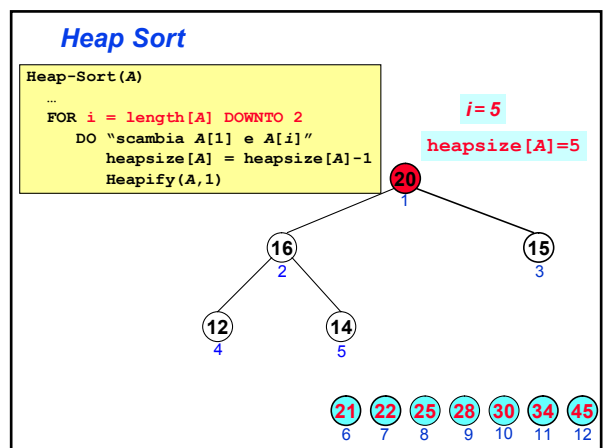
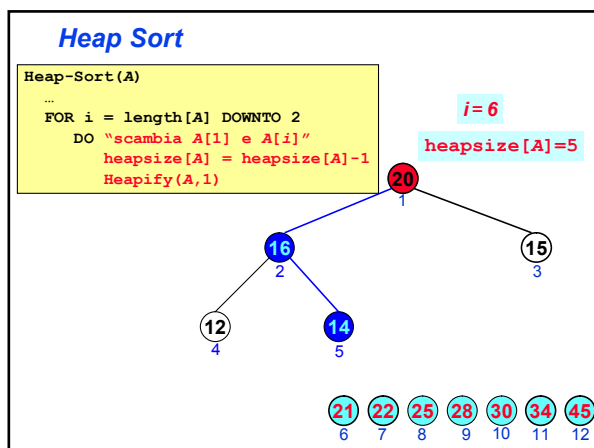
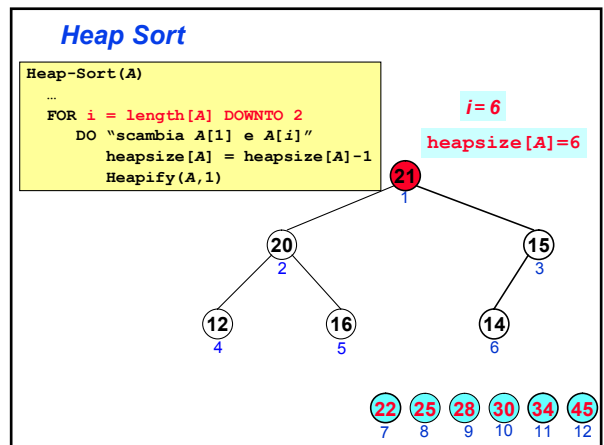
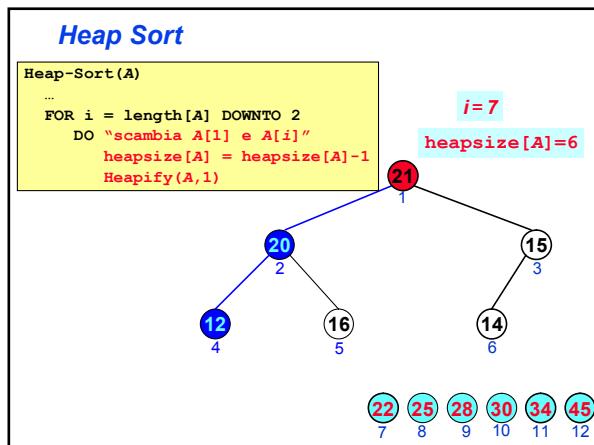
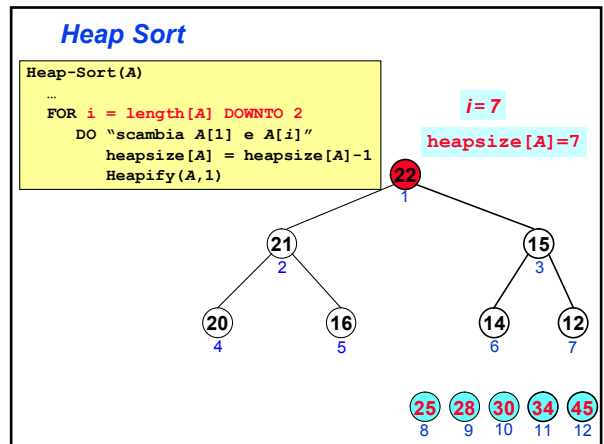
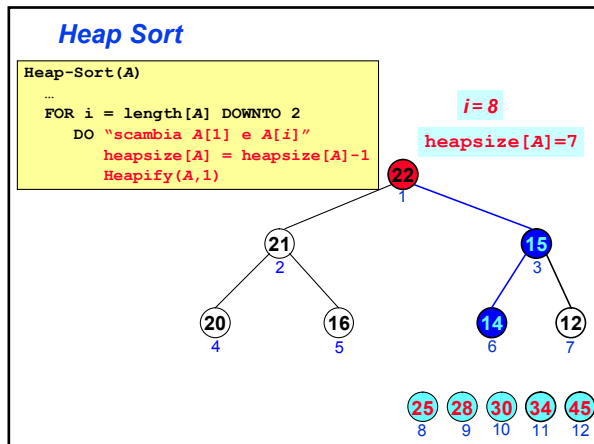
```
Heap-Sort(A)
  ...
  FOR i = length[A] DOWNTO 2
    DO "scambia A[1] e A[i]"
       heapsize[A] = heapsize[A]-1
       Heapify(A,1)
```

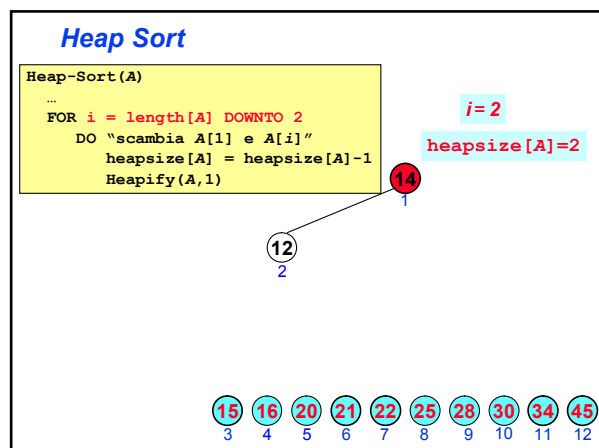
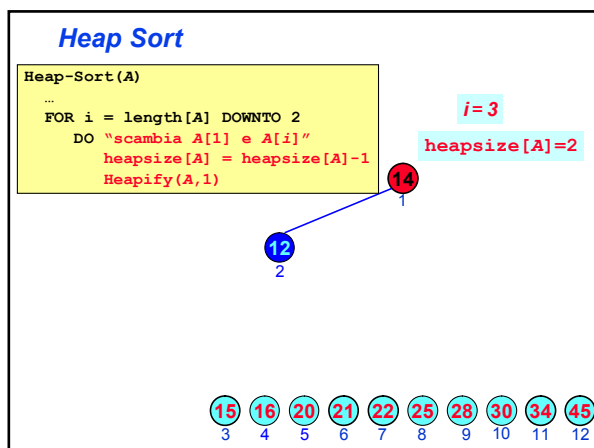
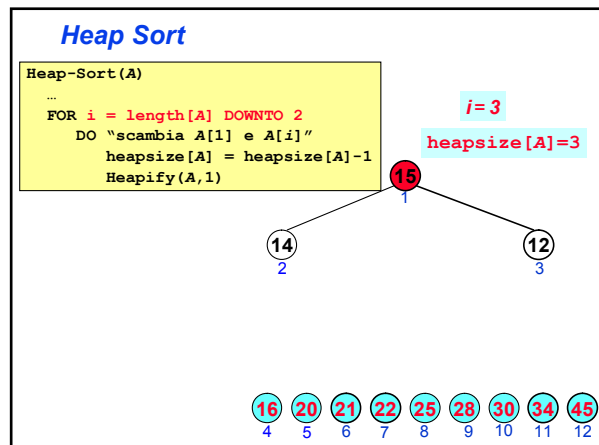
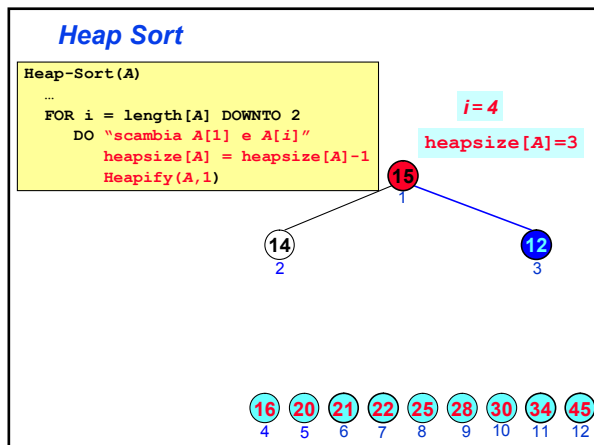
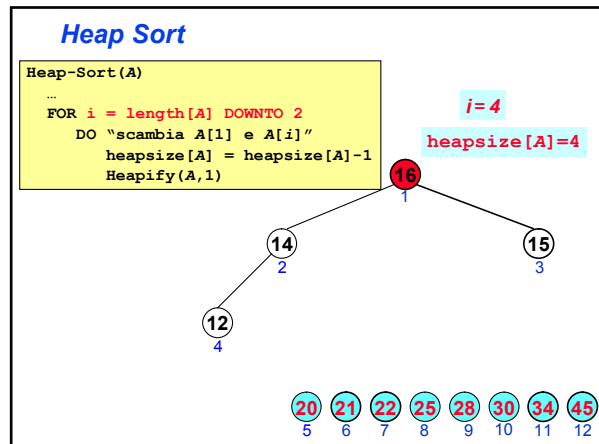
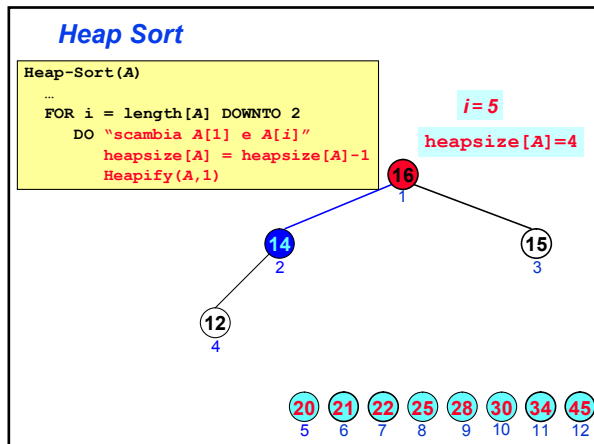
i = 12
heapsize[A] = 11











Heap Sort

```

Heap-Sort (A)
...
FOR i = length[A] DOWNT0 2
  DO "scambia A[1] e A[i]"
      heapsize[A] = heapsize[A]-1
      Heapify(A,1)

```

$i = 2$
 $heapsize[A] = 1$

14 15 16 20 21 22 25 28 30 34 45
2 3 4 5 6 7 8 9 10 11 12

Heap Sort

```

Heap-Sort (A)
...
FOR i = length[A] DOWNT0 2
  DO "scambia A[1] e A[i]"
      heapsize[A] = heapsize[A]-1
      Heapify(A,1)

```

$i = 1$
 $heapsize[A] = 1$

12 14 15 16 20 21 22 25 28 30 34 45
1 2 3 4 5 6 7 8 9 10 11 12

Heap Sort

```

Heap-Sort (A)
...
FOR i = length[A] DOWNT0 2
  DO "scambia A[1] e A[i]"
      heapsize[A] = heapsize[A]-1
      Heapify(A,1)

```

$i = 1$
 $heapsize[A] = 1$

L'array A ora è ordinato!

12 14 15 16 20 21 22 25 28 30 34 45
1 2 3 4 5 6 7 8 9 10 11 12

Heap Sort

```

Heap-Sort (A)
  Costruisci-Heap (A) } = O(n)
  FOR i = length[A] DOWNT0 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify (A,1) } = O(1)

```

$\} = O(\log n)$

Complessità di Heap Sort

Nel caso peggiore *Heap-Sort* chiama

- una volta *Costruisci-Heap*;
- $n-1$ volte *Heapify* sullo *Heap* corrente

$$T(n) = \max(O(n), (n-1) \times \max(O(1), T(\text{Heapify})))$$

Complessità di Heap Sort

Nel caso peggiore *Heap-Sort* chiama

- una volta *Costruisci-Heap*;
- $n-1$ volte *Heapify* sull'intero *Heap* .

$$T(n) = \max(O(n), (n-1) \times \max(O(1), T(\text{Heapify})))$$

$$= \max(O(n), \max(O(n), O(n \log n)))$$

$T(n) = O(n \log n)$

HeapSort: conclusioni

HeapSort

- Algoritmo di ordinamento *sul posto per confronto* che impiega tempo $O(n \log n)$.
- Algoritmo non immediato nè ovvio.
- Sfrutta le proprietà della struttura dati astratta *Heap*.

HeapSort: conclusioni

HeapSort dimostra che:

- scegliere una buona rappresentazione per i dati spesso facilita la progettazione di buoni algoritmi;
- importante pensare a quale può essere una buona rappresentazione dei dati prima di implementare una soluzione.