Ordinamento Ottimo

Sommario

- Ordinamento
  - Heap Sort
  - Quick Sort

Code con priorita' e ordinamento

- C'e' una stretta relazione fra code con priorita' e algoritmi di ordinamento
- Possiamo usare una coda con priorita' per creare un algoritmo di ordinamento:
  - Inserire tutti gli elementi in una coda con priorita' (enqueue)
  - Prelevarli uno ad uno in ordine (dequeue)
- In modo duale possiamo usare gli algoritmi di ordinamento per realizzare delle code con priorita'

Selection Priority Queue/Sort

- **Enqueue:** gli elementi non sono mantenuti in ordine (non costoso)
- **Dequeue:** per l'estrazione dell'elemento piu' grande (o piu' piccolo) dobbiamo scandire quello che rimane della lista e determinare il massimo (minimo) (costoso)
- **Sort:** selezionare via via il massimo (minimo) elemento fra quelli che rimangono

Insertion Priority Queue/Sort

- **Enqueue:** gli elementi sono inseriti in modo da mantenere ad ogni passo una lista ordinata (costoso)
- **Dequeue:** si accede in sequenza agli elementi che sono ordinati (non costoso)
- **Sort:** inserisci in lista crescente di elementi ordinati

Heap Priority Queue/Sort

- **Enqueue:** inserisci in heap (costoso)
- **Dequeue:** estrai radice e ricostruisci heap (costoso)
- **Sort:** estrai via via il massimo da un heap.
- Il Selection Sort deve la sua lentezza al fatto che per selezionare l'elemento piu' piccolo si impiega un tempo O(n)
- Se miglioriamo la ricerca del massimo con un algoritmo in tempo O(ln n) abbiamo un algoritmo di complessita' totale O(n ln n)

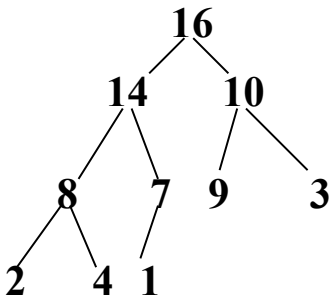
Heap Sort

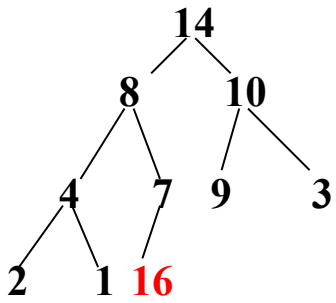
- Il metodo di ordinamento Heap Sort sfrutta la proprietà di ordinamento parziale dello Heap
- L'idea è di selezionare via via l'elemento più grande, eliminarlo dallo heap e poi utilizzare la procedura Heapify per ripristinare la proprietà di ordinamento parziale
- Invece di eliminare progressivamente gli elementi, si inseriscono gli elementi trovati via via oltre i limiti dello heap
- In questo modo si ottiene un vettore che contiene il massimo, l'elemento più grande dopo il massimo, e così via

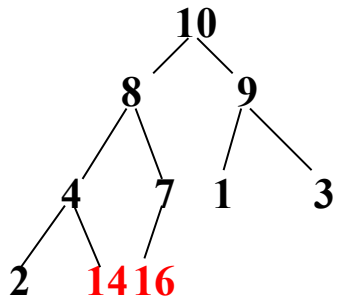
Pseudocodice per HeapSort

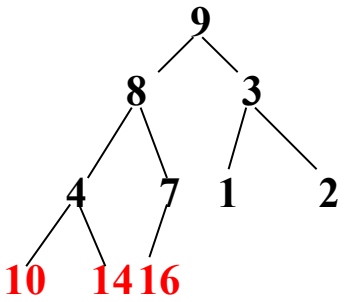
```
HeapSort(A)
1  BuildHep(A)
2  for i ← length[A] downto 2
3  do   swap A[1] ↔ A[i]
4       heap-size[A] ← heap-size[A]-1
5       Heapify(A,1)
```

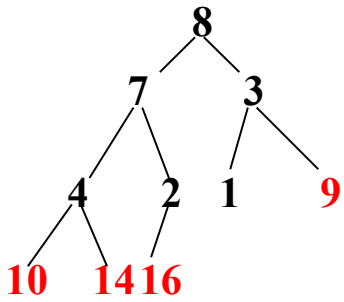
Visualizzazione HeapSort

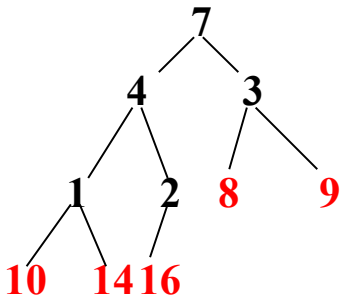


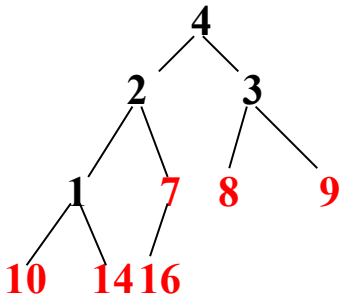


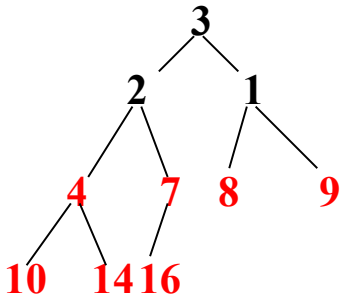


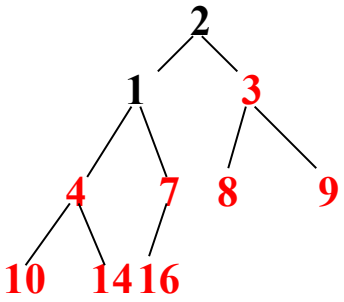












Tempo di Calcolo dell'HeapSort

- L'algoritmo chiama n-1 volte la procedura Heapify
- Si deve determinare il tempo di calcolo di Heapify
- Abbiamo visto che per Heapify si ha $T(n)= \Theta(\lg n)$
- Pertanto il tempo di calcolo per HeapSort è:
- $T(n)= \Theta(n \lg n)$

Ordinamento con partizionamento ricorsivo

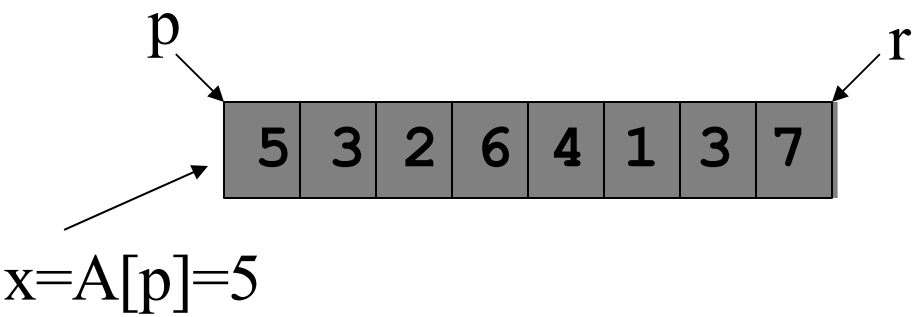
- Il QuickSort, come il MergeSort, è un algoritmo basato sul paradigma Divide et Impera
- Fasi:
  - Divide: il vettore è riorganizzato in modo da avere due sottosequenze di lunghezza diversa tali che qualsiasi elemento nella sottosequenza di sinistra è minore di un qualsiasi elemento nella sottosequenza di destra
  - Impera: le due sottosequenze sono ordinate ricorsivamente
  - Combina: non ce ne è bisogno. Infatti, le sottosequenze sono già ordinate internamente dato che per ogni indice gli elementi con indice inferiore sono minori degli elementi con indice superiori

PseudoCodice

```
QuickSort(A,p,r)
1  if p<r
2  then q ← Partition(A,p,r)
3       QuickSort(A,p,q)
4       QuickSort(A,q+1,r)
```

Spiegazione Intuitiva della Procedura Partition

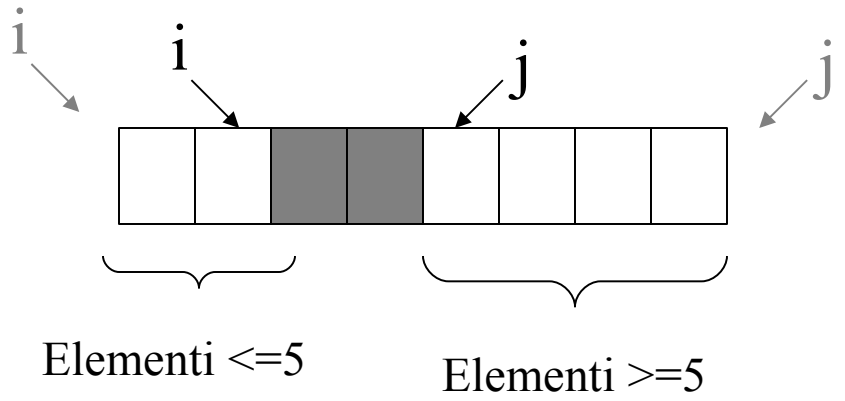
- Si prende un elemento x, ad es. il primo elemento della prima sottosequenza, come elemento *perno*



- si vuole dividere il vettore A in due sottosequenze:
  - nella prima devono esserci solo elementi <= 5
  - nella seconda solo elementi >=5

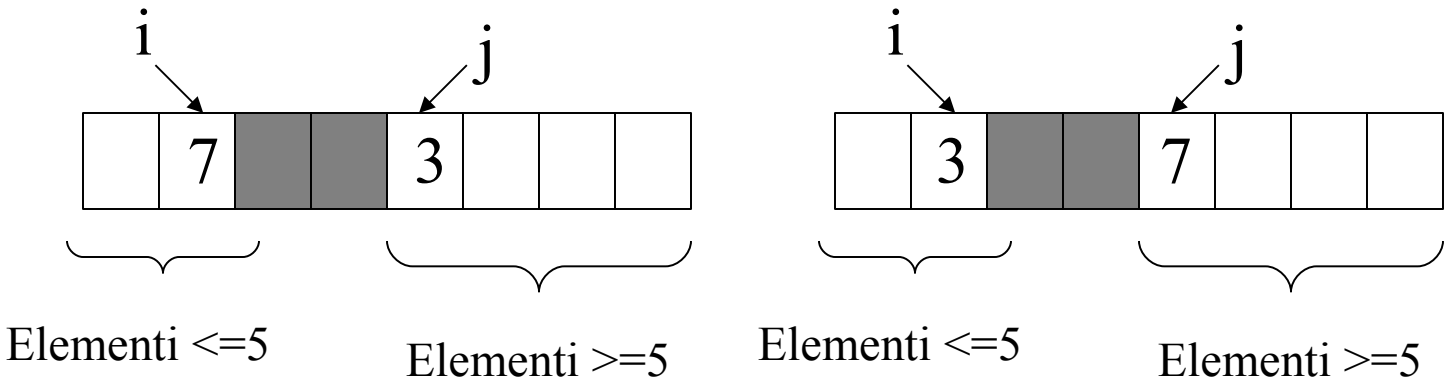
Spiegazione Intuitiva della Procedura Partition

- Si fanno crescere due regioni da entrambi gli estremi, utilizzando gli indici i,j a partire dagli estremi



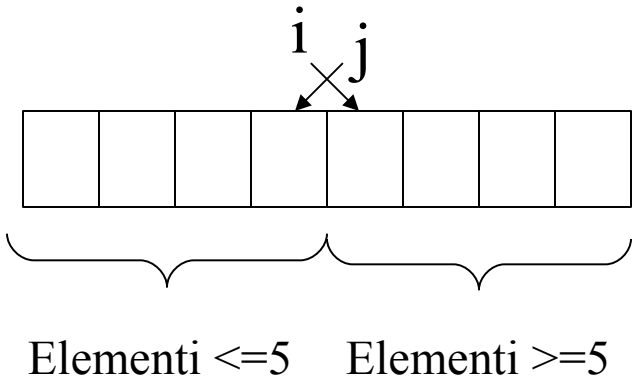
Spiegazione Intuitiva della Procedura Partition

- Mentre le due regioni crescono si verifica il valore degli elementi
- Se un elemento non deve appartenere alla regione in cui si trova (o se l'elemento ha un valore eguale al valore perno) si smette di far crescere la regione
- Quando non è possibile far crescere nessuna delle due regioni si scambiano gli elementi fra loro



Spiegazione Intuitiva della Procedura Partition

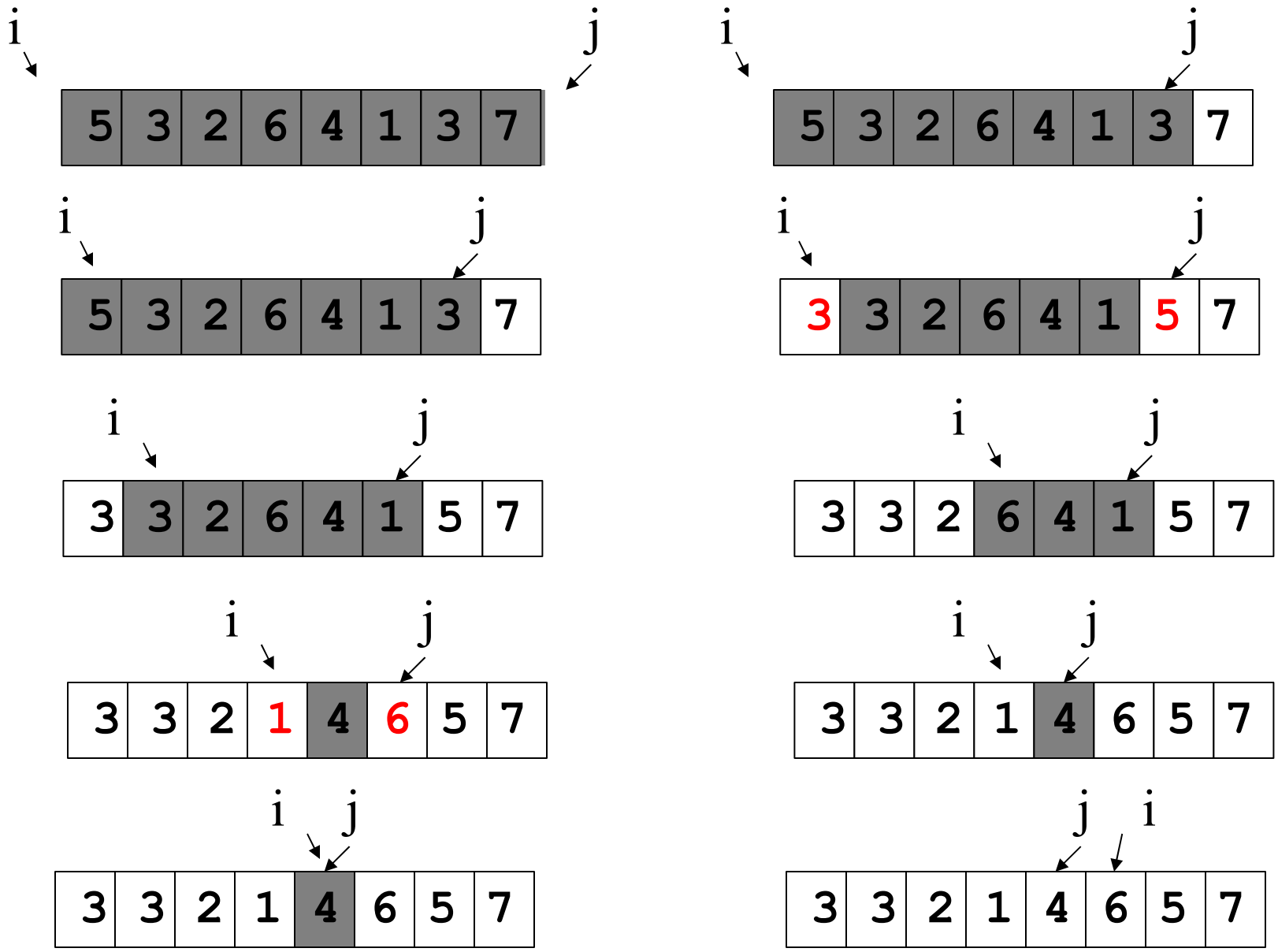
- Quando i diventa maggiore di j allora abbiamo completato le due regioni
- La procedura termina



PseudoCodice per Partition

```
Partition(A,p,r)
1  x ← A[p]
2  i ← p-1
3  j ← r+1
4  while TRUE
5       do repeat j ← j-1
6          until A[j] ≤ x
7          repeat i ← i+1
8          until A[i] ≥ x
9       if i < j
10      then   A[i] ↔ A[j]
11      else return j
```

Visualizzazione



Prestazioni del QuickSort

- Il tempo di esecuzione del QuickSort dipende dal fatto che il partizionamento sia più o meno bilanciato
- Il partizionamento dipende dagli elementi pivot.
- Se il partizionamento è bilanciato si hanno le stesse prestazioni del MergeSort
- Altrimenti può essere tanto lento quanto l'InsertionSort

Caso peggiore

- Il caso di peggior sbilanciamento si ha quando il partizionamento produce due sottosequenze di lunghezza 1 e n-1
- Il partizionamento richiede un tempo $\Theta(n)$ e il passo base della ricorsione richiede $T(1)=\Theta(1)$ pertanto:
- $T(n)=T(n-1)+\Theta(n)$
- Ad ogni passo si decrementa di 1 la dimensione dell'input, occorreranno pertanto n passi per completare la ricorsione
- $T(n)= \sum_{k=1..n}\Theta(k) = \Theta(\sum_{k=1..n}k) = \Theta(n^2)$

Caso migliore

- Il caso migliore si ha se ad ogni partizionamento si divide l'input in due sottosequenze di dimensione identica
- In questo caso si ha, come nel caso del MergeSort
  - $T(n)=2T(n/2)+\Theta(n)$
- Ovvero, per il Teorema Principale:
  - $f(n)=n$, $a=2$, $b=2$
  - $n^{\log_b a} = n^{\log_2 2} = n$
  - pertanto, dato che $f(n)=\Theta(n^{\log_b a})=\Theta(n)$ allora (caso 2)
  - $T(n)= \Theta(n \lg n)$