ALGORITMI E STRUTTURE DATI

Dr. Manuela Montangero

A.A. 2022/23

ALGORITMI DI ORDINAMENTO: QuickSort

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

QuickSort

USIAMO la TECNICA DIVIDE&IMPERA

… di nuovo….

… ma con un’altra idea….

QuickSort

- **DIVIDE**: USIAMO la TECNICA DIVIDE&IMPERA
  - Scegliere un indice `t` tra `0` e `n-1`
  - Mettere a sinistra tutti gli elementi ≤ `A[t]`
  - Mettere a destra tutti gli elementi > `A[t]`
  - Mettere `A[t]` al posto giusto (tra gli elementi di sx e quelli di dx), sia la posizione `q`
- **IMPERA**: ordina `A[0..q-1]` e `A[q+1..n-1]` ricorsivamente
- **COMBINA**: niente da fare

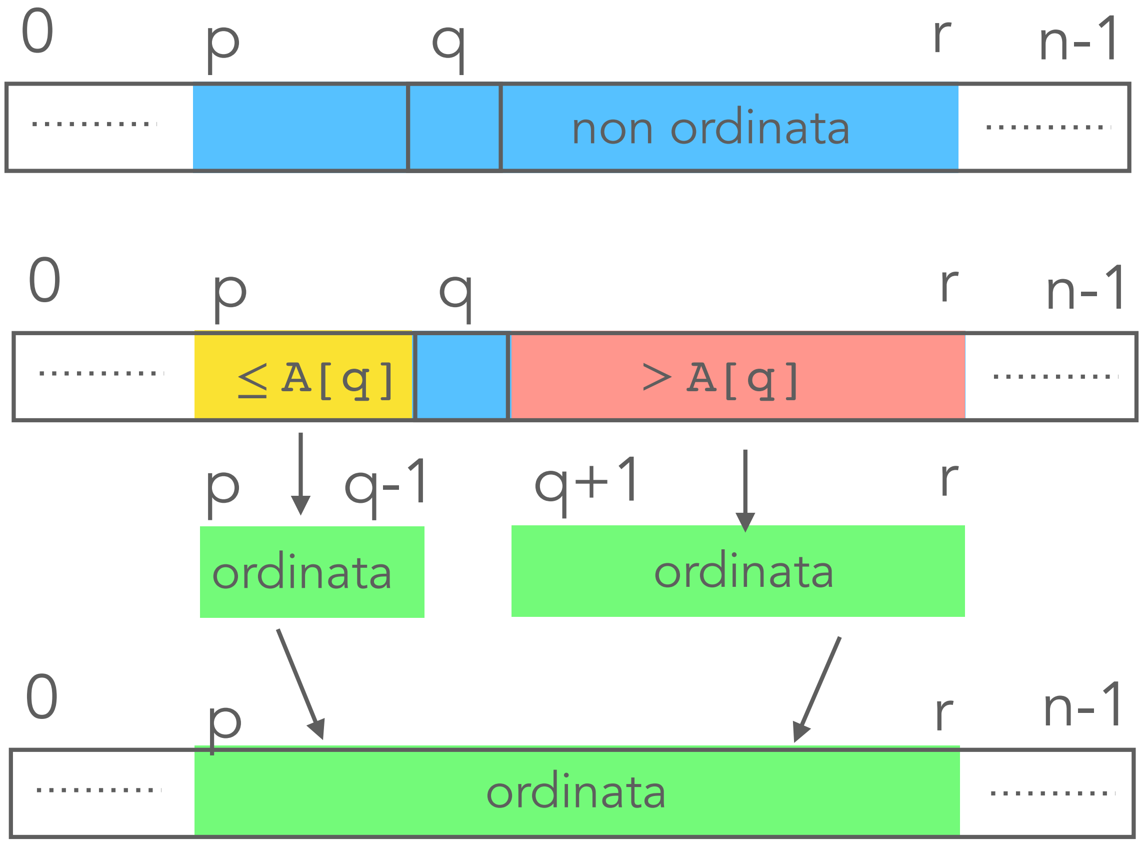
CASO BASE: in quali condizioni il problema è facile da risolvere?

Risposta: quando la sequenza ha un solo elemento (è già ordinata)

QuickSort

Scriviamo una procedura ricorsiva che ordini una porzione di un array `A`, compresa tra gli indici `p` (a sinistra) e `r` (a destra), estremi inclusi

```
QUICKSORT(A,p,r)
 if p < r
  then
   q := PARTITION(A,p,r)
   QUICKSORT(A,p,q-1)
   QUICKSORT(A,q+1,r)
```



Chiamata principale

QUICKSORT(A,0,n-1)

UNIMORE

QuickSort

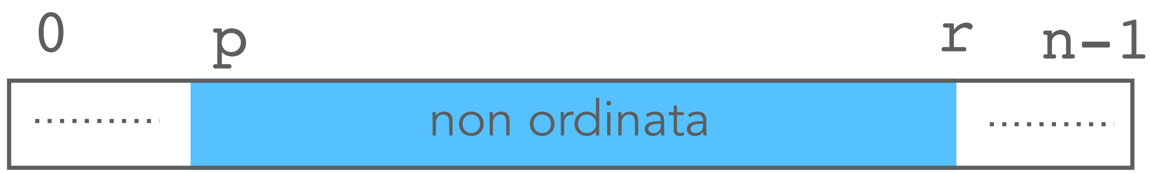
Scriviamo una funzione (`Partition`) che risolve il seguente problema:

INPUT: Array `A` e due indici $0 \leq p \leq r \leq n-1$.

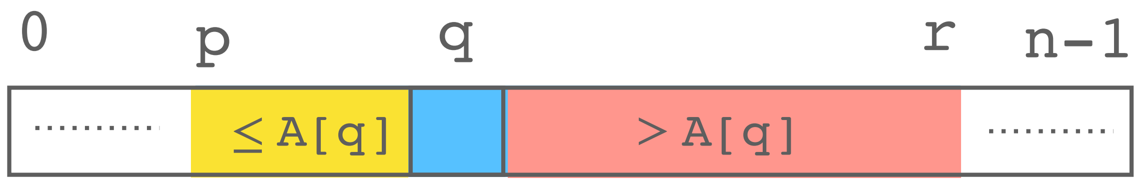
OUTPUT: Indice `q`, con $p \leq q \leq r$, e tale che:

- tutti gli elementi a sinistra di `A[q]` siano $\leq$ di `A[q]` E
- tutti gli elementi a destra di `A[q]` siano $>$ di `A[q]`

INPUT



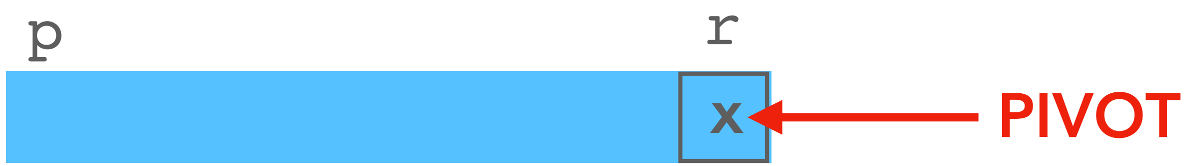
OUTPUT



QuickSort

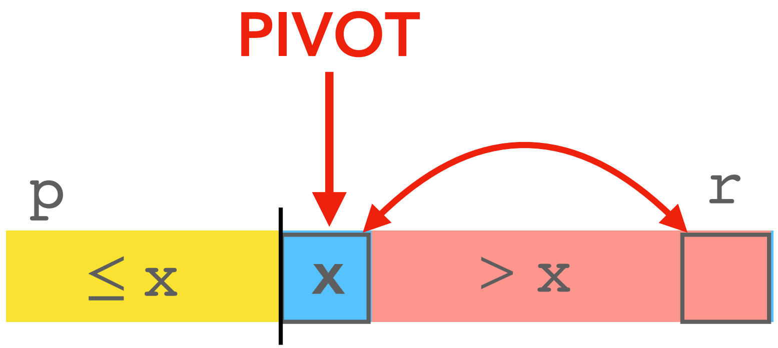
`Partition` - una delle tante versioni

- Scegliamo `t = r` (mettiamo al suo posto l'ultimo elemento di `A[p..r]`)



- Confrontiamo `x` con tutti gli altri elementi di `A[p..r-1]` e mettiamo a sinistra quelli ≤ `x`, a destra quelli > `x`

- Mettiamo il PIVOT al suo posto

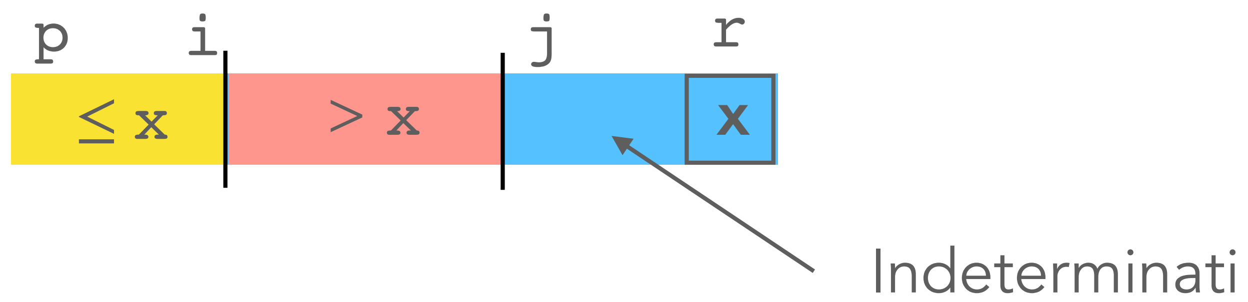


QuickSort

`Partition`

- Confrontiamo `x` con tutti gli altri elementi di `A[p..r-1]` e mettiamo a sinistra quelli ≤ `x`, a destra quelli > `x`

Ad una generica iterazione `j`



- `i` = ultimo indice dei valori ≤ `x`
- `j` = primo indice dei valori indeterminati
- `[i+1..j-1]` intervallo dei valori > `x`

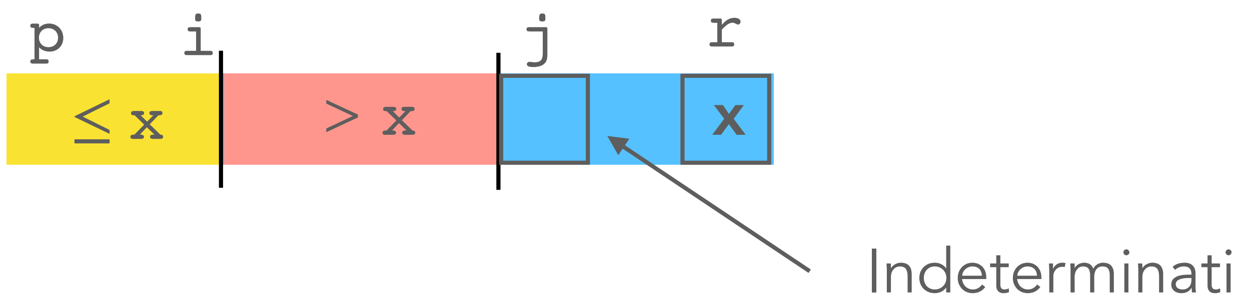
Confrontiamo `A[j]` con il Pivot `x`

QuickSort

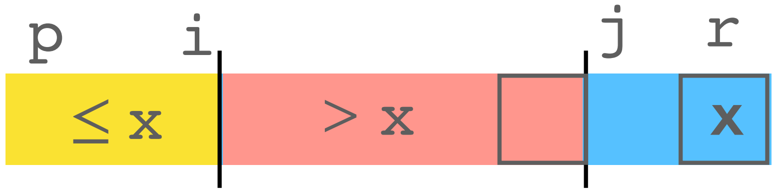
`Partition`

- Confrontiamo `x` con tutti gli altri elementi di `A[p..r-1]` e mettiamo a sinistra quelli $\leq$ `x`, a destra quelli > `x`

Ad una generica iterazione `j`



SE `A[j] > x`



`i` rimane invariato, `j` viene incrementato di 1

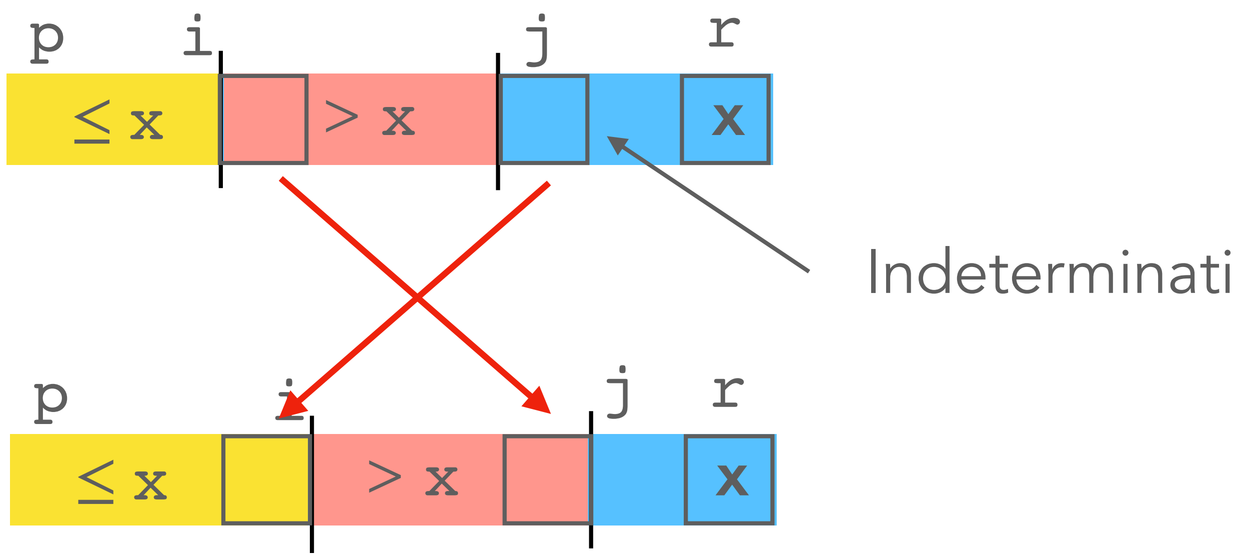
QuickSort

`Partition`

- Confrontiamo `x` con tutti gli altri elementi di `A[p..r-1]` e mettiamo a sinistra quelli ≤ `x`, a destra quelli > `x`

Ad una generica iterazione `j`

SE `A[j]` ≤ `x`



`A[j]` viene scambiato con `A[i+1]`

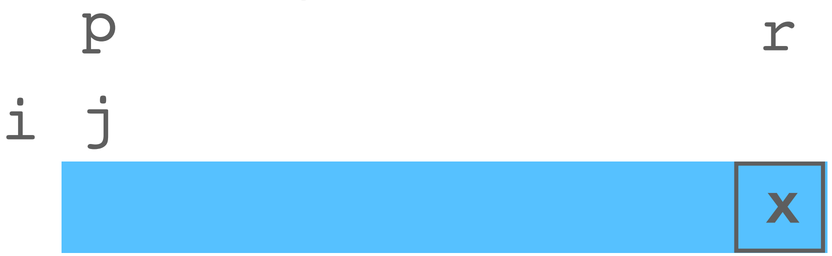
`i` e `j` vengono incrementati di 1

QuickSort

Partition

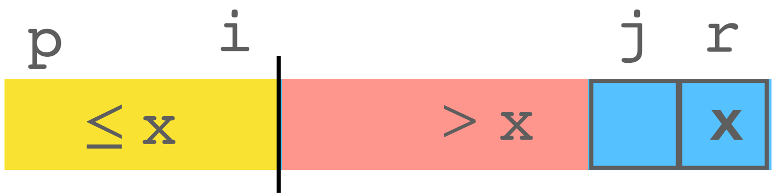
- Confrontiamo `x` con tutti gli altri elementi di `A[p..r-1]` e mettiamo a sinistra quelli $\leq$ `x`, a destra quelli $>$ `x`

Prima della prima iterazione:



`i = p - 1`
`j = p`

L'ultima iterazione:



`j = r - 1`

QuickSort

`Partition`

```
PARTITION(A,p,r)
 x := A[r]
 i := p-1
 for j = p to r-1
  if A[j] ≤ x
   then
    i := i+1
    scambia A[i] e A[j]
 scambia A[r] e A[i+1]
 return i+1
```

Un confronto per ogni iterazione del ciclo `for`

esattamente
$r - p$
confronti

E' una funzione

QuickSort

COSTO COMPUTAZIONALE

```
QUICKSORT(A,p,r)
 if p < r
  then
   q := PARTITION(A,p,r)
   QUICKSORT(A,p,q-1)
   QUICKSORT(A,q+1,r)
```

$$T(n) = \begin{cases} 0 & \text{se } n = 0,1 \\ T(q) + T(n-q-1) + n - 1 & \text{altrimenti} \end{cases}$$

Costo di **Partition**

Partizioni "estreme":

- Completamente sbilanciata:

  `q = 0` e `n-q-1 = n-1`

- Perfettamente bilanciata:

  `q = n-q-1`

Può essere rcorsivamente perfettamente bilanciata solo se `n` è il predecessore di una potenza di 2 (ovvero n = 2^k -1 per qualche k≥0).
Attenzione: il viceversa non è vero: non basta che n = 2^k -1 affinché sia ricorsivamente perfettamente bilanciata

QuickSort

COSTO COMPUTAZIONALE

```
QUICKSORT(A,p,r)
 if p < r
  then
   q := PARTITION(A,p,r)
   QUICKSORT(A,p,q-1)
   QUICKSORT(A,q+1,r)
```

$$T(n) = \begin{cases} 0 & \text{se } n = 0,1 \\ T(q) + T(n-q-1) + n - 1 & \text{altrimenti} \end{cases}$$

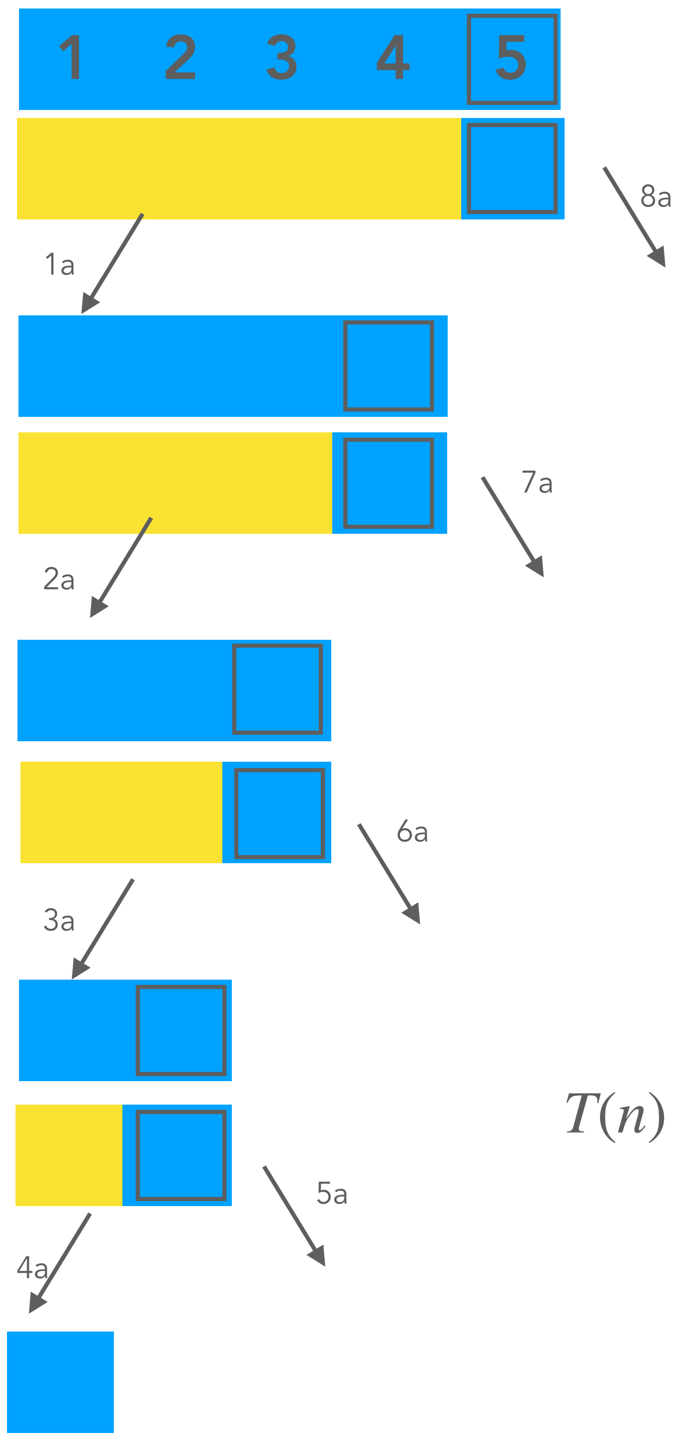
Costo di **`Partition`**

Partizione sbilanciata `q = n-1` `n-q-1 = 0`

Il PIVOT rimane sempre in ultima posizione —> già ordinato

QuickSort

COSTO COMPUTAZIONALE



Partizione sbilanciata

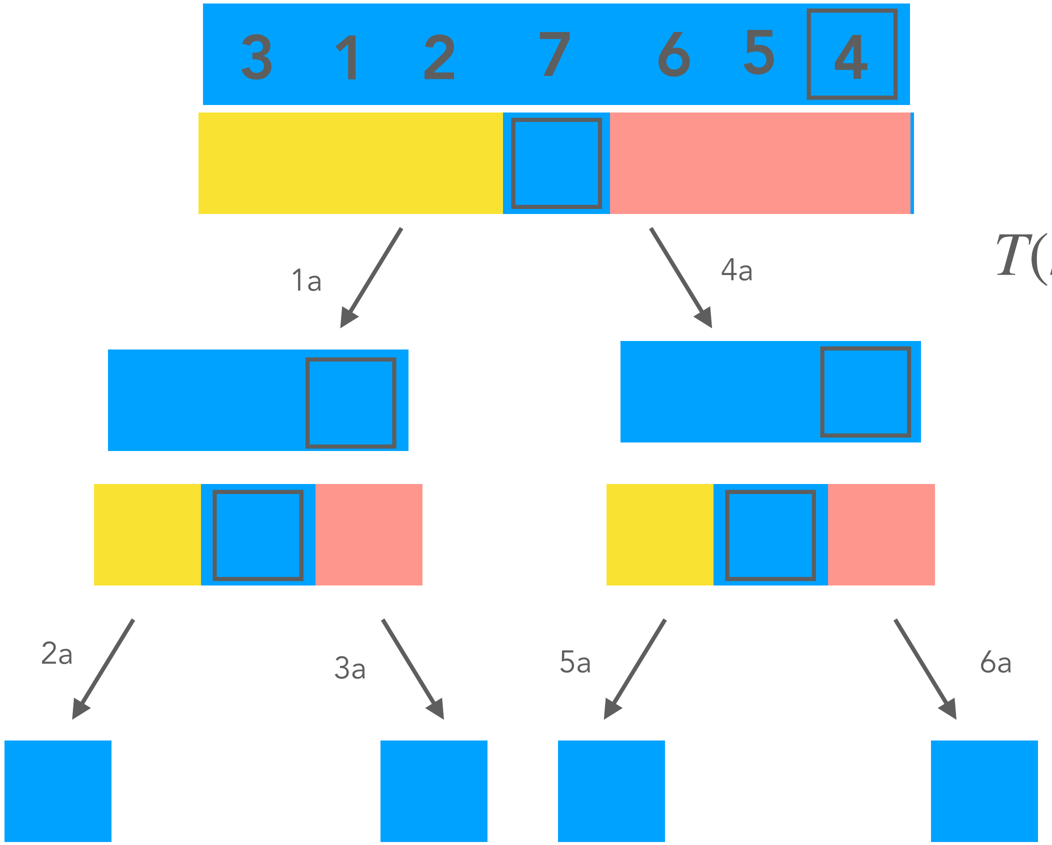
```
q = n-1
n-q-1 = 0
```

$$T(n) = \begin{cases} 0 & \text{se } n = 0,1 \\ T(n-1) + n - 1 & \text{altrimenti} \end{cases}$$

$$T(n) = (n-1) + (n-2) + (n-3) + \ldots \ldots + 1 \in \Theta(n^2)$$

QuickSort

COSTO COMPUTAZIONALE



Partizione perfettamente bilanciata

$$T(n) = \begin{cases} 0 & \text{se } n = 0{,}1 \\ 2T(n/2) + n - 1 & \text{altrimenti} \end{cases}$$

Usiamo il Master Theorem

$$a = 2$$
$$b = 2$$
$$d = 1$$

$$\log_2 2 = 1 = d \Rightarrow T(n) \in O(n \log n)$$

QuickSort

COSTO COMPUTAZIONALE

Partizioni "estreme":

- Completamente sbilanciata:

  `q = n-1` e `n-q-1 = 0` ⇨ $T(n) \in O(n^2)$

- Perfettamente bilanciata:

  `q = n-q-1` ⇨ $T(n) \in O(n \log n)$

Come possiamo fare per avvicinarci al caso migliore?

Scegliamo il pivot a caso!

QuickSort

```
RANDOMIZED-QUICKSORT(A,p,r)
 if p < r
  then
   q := RANDOMIZED-PARTITION(A,p,r)
   RANDOMIZED-QUICKSORT(A,p,q-1)
   RANDOMIZED-QUICKSORT(A,q+1,r)
```

Sceglie a caso un indice nell'intervallo [p,r]

Chiamata principale

RANDOMIZED-QUICKSORT(A,0,n-1)

```
RANDOMIZED-PARTITION(A,p,r)
 i := RANDOM(p,r)
 scambia A[i] e A[r]
 return PARTITION(A,p,r)
```

Mette il pivot scelto a caso in ultima posizione

Chiamata a PARTITION (quella di prima)

QuickSort

COSTO COMPUTAZIONALE

Partizioni "estreme":

- Completamente sbilanciata:

  `q = n-1` e `n-q-1 = 0` $\Rightarrow$ $T(n) \in O(n^2)$ 

- Perfettamente bilanciata:

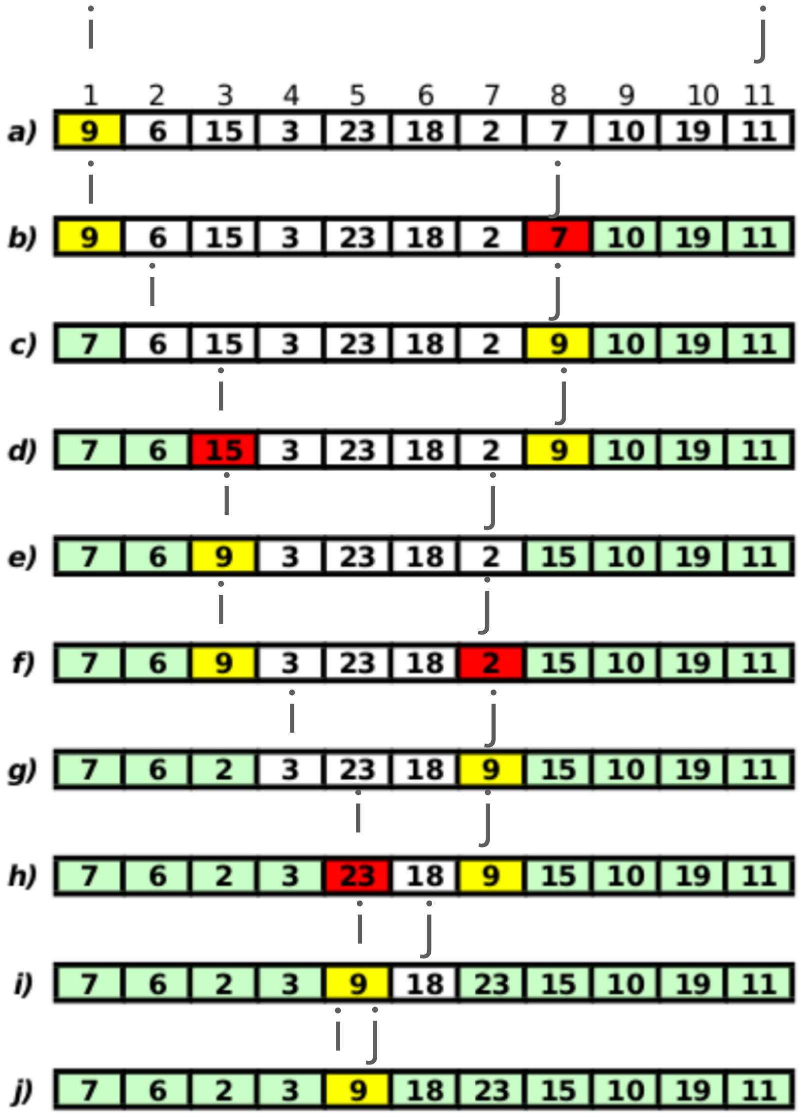
  `q = n-q-1` $\Rightarrow$ $T(n) \in O(n \log n)$ 

TEOREMA

Il numero atteso di confronti tra elementi di A eseguito dall'algoritmo `QuickSort` con pivot scelto random e' $\Theta(n \log n)$

QuickSort

`Partition` - altra versione (idea)



Usiamo due cursori i e j che,
partendo dagli estremi,
si avvicinano al centro
in modo che:

- a destra di j ci siano solo elementi $\geq$ del pivot
- a sinistra di i ci siano solo elementi $\leq$ del pivot

si fermano quando si trovano
su due elementi "fuori posto"
che vengono invertiti.
Quando gli indici si incontrano
(o si invertono) l'algoritmo termina.

QuickSort

- `QuickSort` ordina i valori IN-PLACE (non ha bisogno di spazio aggiuntivo), a differenza di `MergeSort`
- La versione randomizzata di `QuickSort` ha lo stesso costo computazionale (atteso) di `MergeSort`
- In pratica `QuickSort` è più veloce di MergeSort

QuickSort

- `QuickSort` ordina i valori IN-PLACE (non ha bisogno di spazio aggiuntivo), a differenza di `MergeSort`
- La versione randomizzata di `QuickSort` ha lo stesso costo computazionale (atteso) di `MergeSort`
- Nella pratica `QuickSort` è più veloce di MergeSort