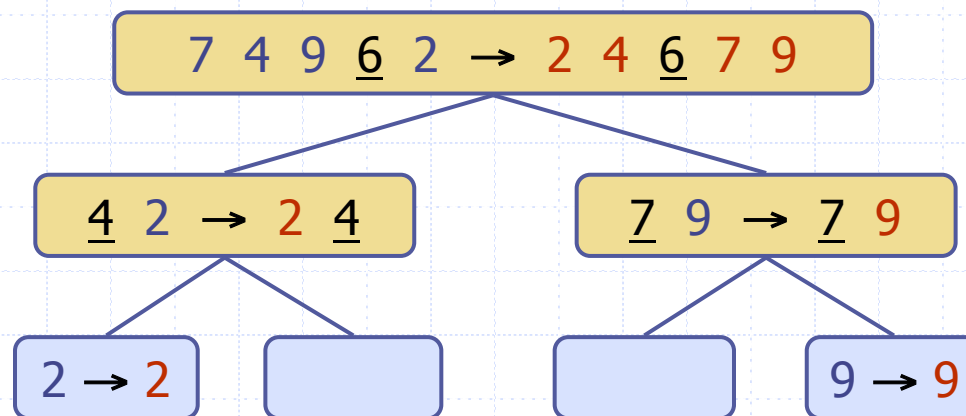
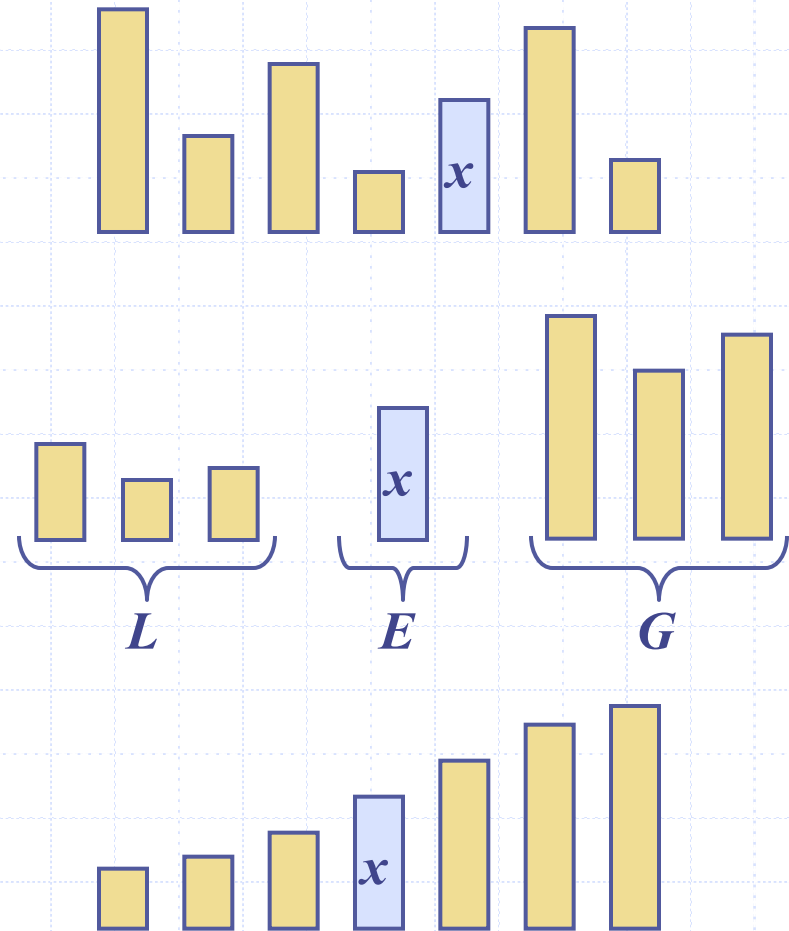


Quick-Sort

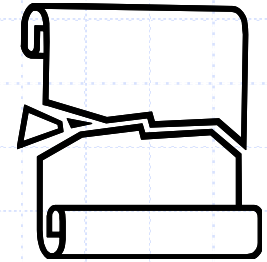


Quick-Sort

- ◆ Quick-sort è un algoritmo di ordinamento che **opera scelte casuali**
- ◆ Divide et impera
 - **Dividi**: scegli un elemento a caso x (detto **pivot**) e dividi S in tre insiemi
 - ◆ L elementi minori di x
 - ◆ E elementi uguali a x
 - ◆ G elementi maggiori di x
 - **Ricorsione**: ordina L e G
 - **Impera**: unisci L , E e G



Partizione



- ◆ Partiziona insieme dato nel seguente modo:
 - Togli ad uno ad uno elementi y da S e
 - Inserisci y in L , E or G , a seconda del risultato del confronto con il pivot x
- ◆ Ogni inserimento e rimozione avviene all'inizio o alla fine e quindi richiede tempo $O(1)$
- ◆ Pertanto, il partizionamento richiede tempo $O(n)$

Algorithm *partition*(S, p)

Input sequenza S , posizione p del pivot

Output sottosequenze L , E , G di elementi di S minori uguali o maggiori del pivot

$L, E, G \leftarrow$ sequenze vuote

$x \leftarrow S.remove(p)$

while $\neg S.isEmpty()$

$y \leftarrow S.remove(S.first())$

if $y < x$

$L.addLast(y)$

else if $y = x$

$E.addLast(y)$

else $\{ y > x \}$

$G.addLast(y)$

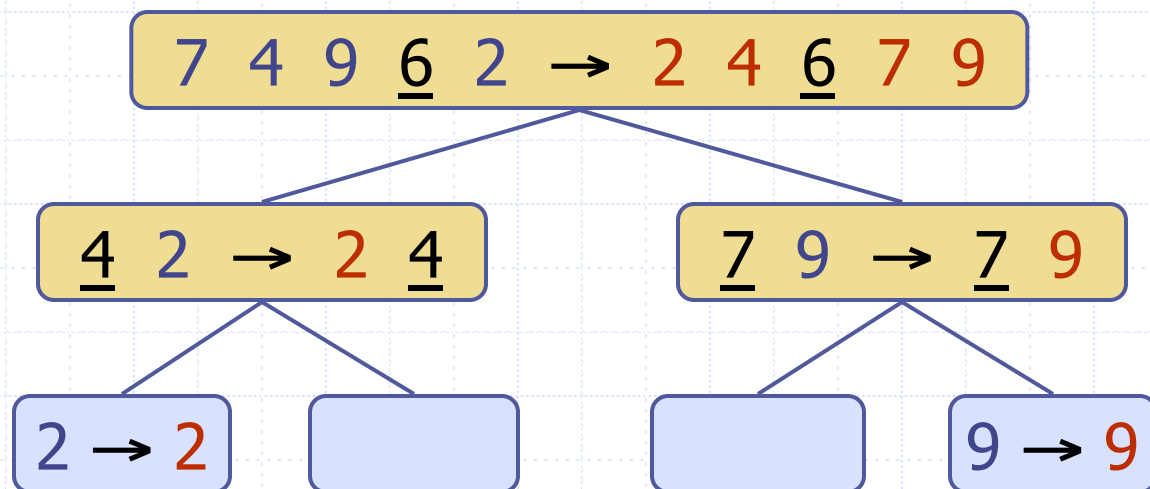
return L, E, G

Implementazione Java

```
1  /** Quick-sort contents of a queue. */
2  public static <K> void quickSort(Queue<K> S, Comparator<K> comp) {
3      int n = S.size();
4      if (n < 2) return;           // queue is trivially sorted
5      // divide
6      K pivot = S.first();         // using first as arbitrary pivot
7      Queue<K> L = new LinkedList<>();
8      Queue<K> E = new LinkedList<>();
9      Queue<K> G = new LinkedList<>();
10     while (!S.isEmpty()) {       // divide original into L, E, and G
11         K element = S.dequeue();
12         int c = comp.compare(element, pivot);
13         if (c < 0)                // element is less than pivot
14             L.enqueue(element);
15         else if (c == 0)          // element is equal to pivot
16             E.enqueue(element);
17         else                     // element is greater than pivot
18             G.enqueue(element);
19     }
20     // conquer
21     quickSort(L, comp);           // sort elements less than pivot
22     quickSort(G, comp);          // sort elements greater than pivot
23     // concatenate results
24     while (!L.isEmpty())
25         S.enqueue(L.dequeue());
26     while (!E.isEmpty())
27         S.enqueue(E.dequeue());
28     while (!G.isEmpty())
29         S.enqueue(G.dequeue());
30 }
```

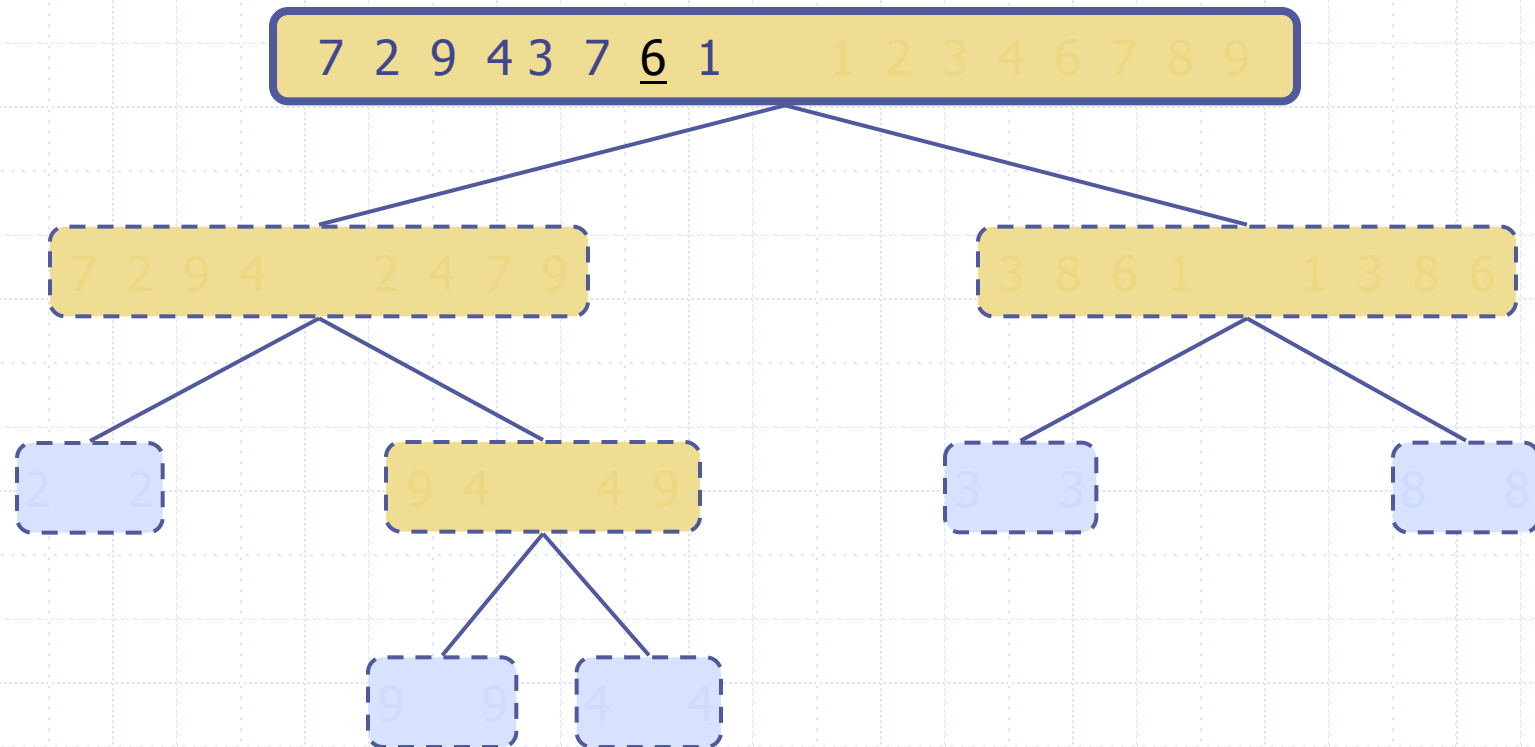
Albero chiamate Quick-Sort

- ◆ L'esecuzione è rappresentata da un albero binario
 - Ogni nodo rappresenta una attivazione ricorsiva e memorizza
 - ◆ Sequenze non ordinate prima dell'esecuzione
 - ◆ Sequenze ordinate alla fine dell'esecuzione
 - La radice rappresenta la prima chiamata
 - Le foglie rappresentano sequenze di 0 o 1 elemento



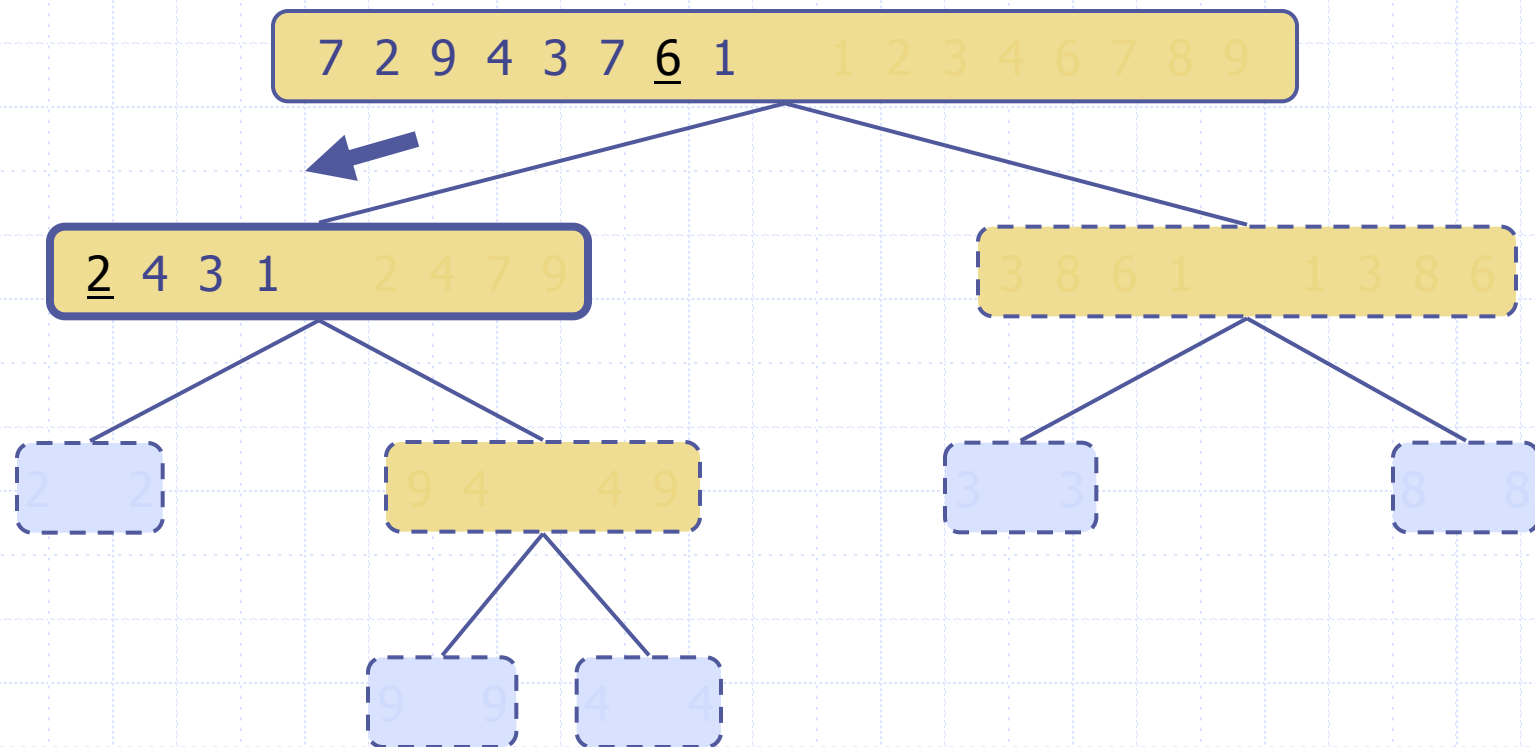
Esempio

◆ Selezione Pivot



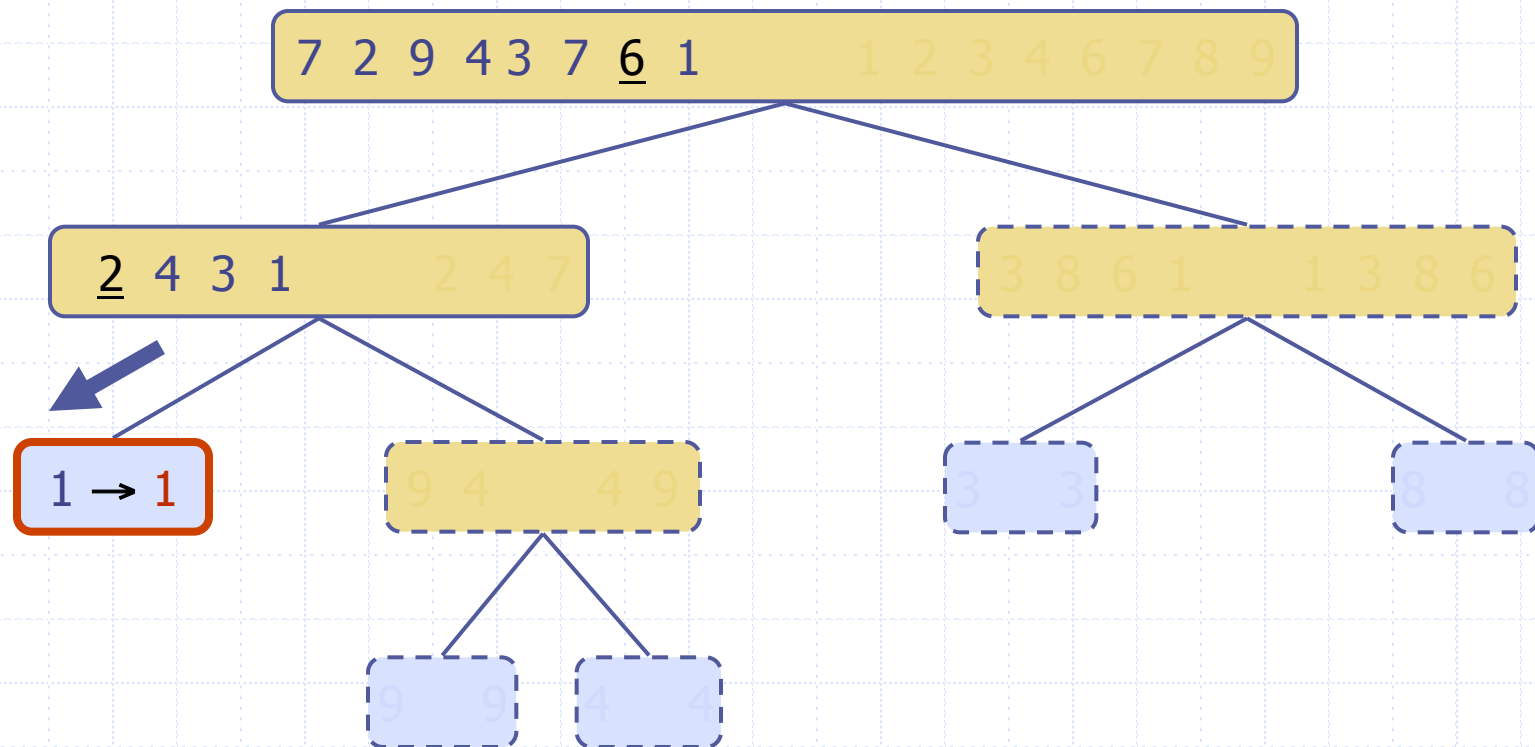
Esempio

◆ Partizione, ricorsione, selezione pivot



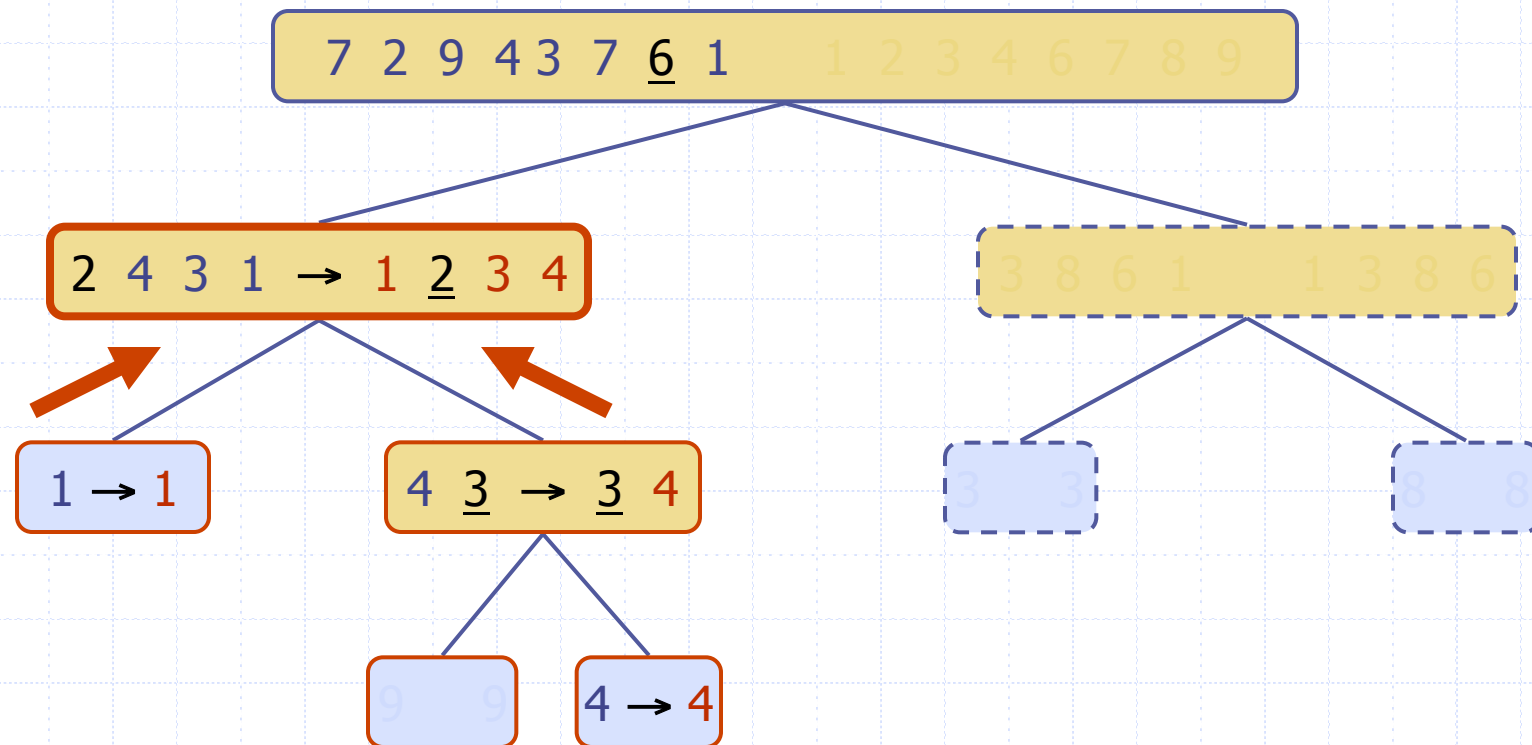
Esempio

◆ Partizione, ricorsione, caso base



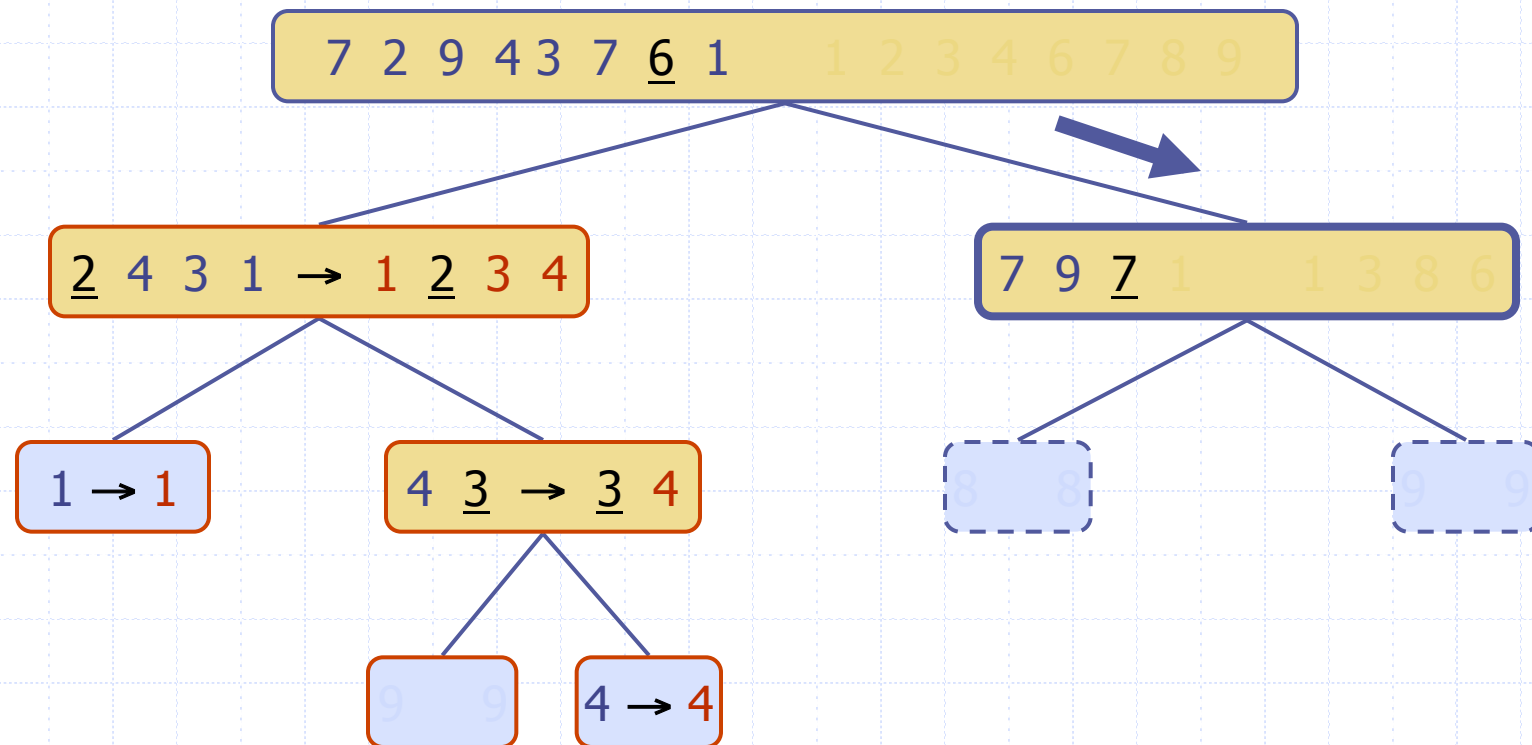
Esempio

◆ Partizione, ricorsione, unione



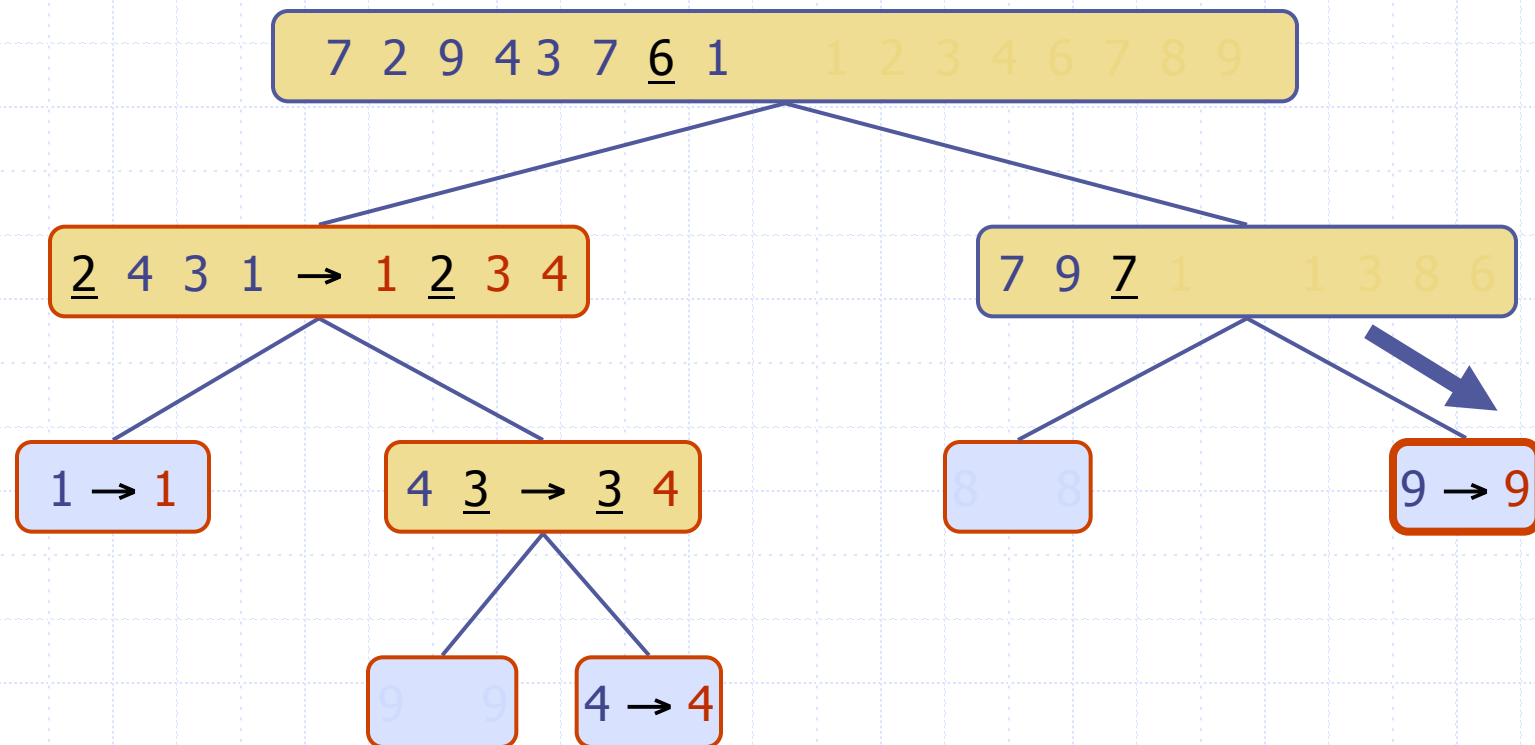
Esempio

◆ Partizione, ricorsione, selezione pivot



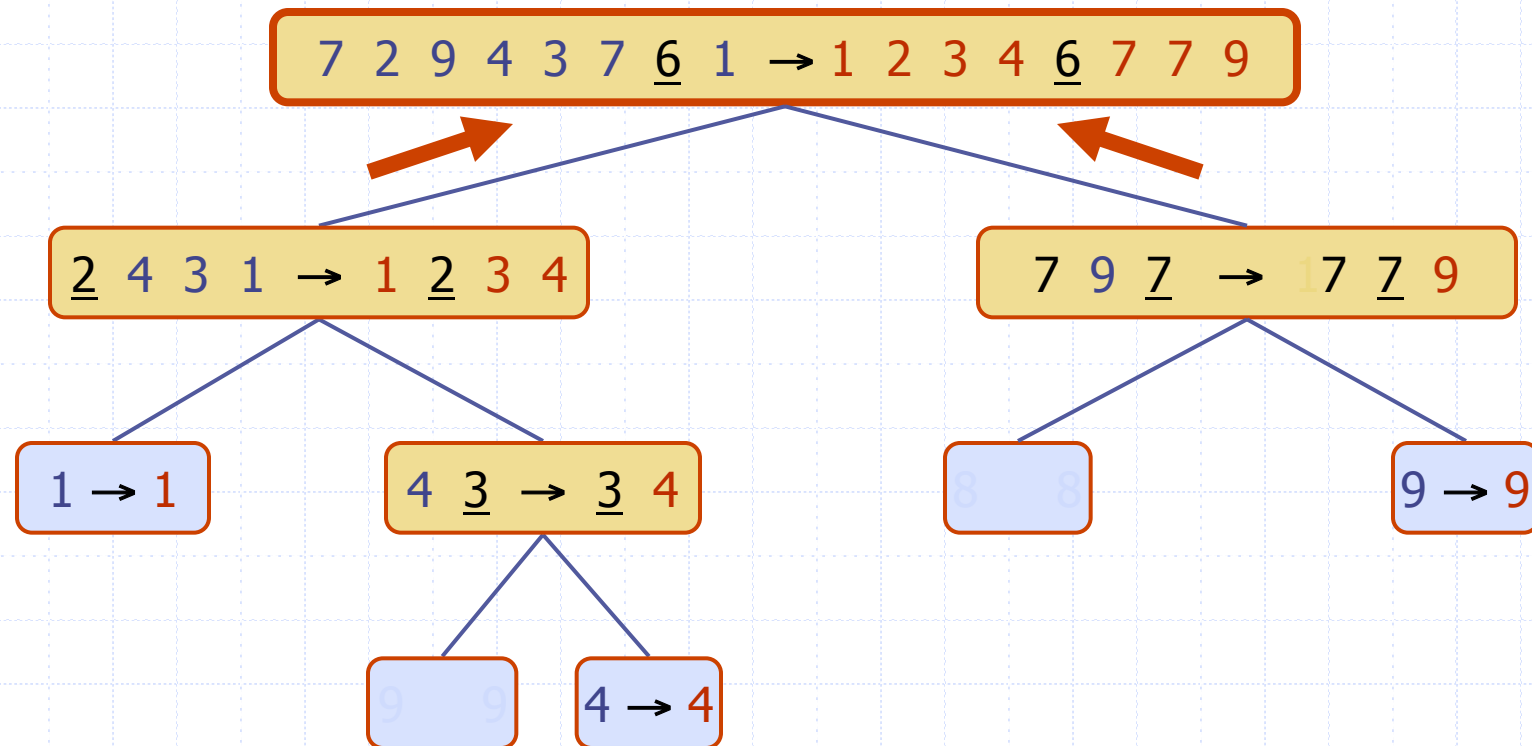
Esempio

◆ Partizione, ricorsione, caso base



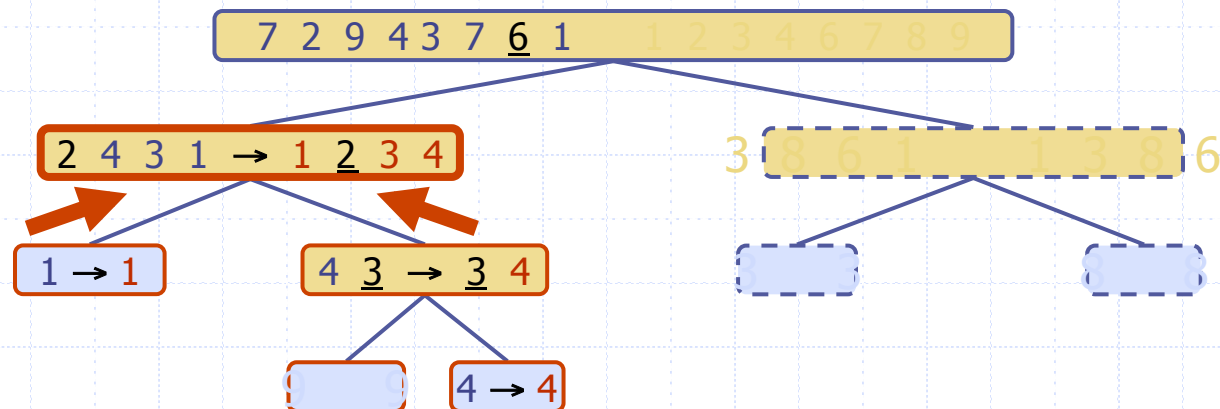
Esempio

◆ Unione, unione



Analisi costo

- ◆ Il costo dell'algoritmo può essere espresso dal numero di elementi presenti (contando duplicazioni) nei nodi dell'albero di quick-sort
- ◆ Consideriamo un nodo dell'albero con k elementi; questo nodo viene esaminato due volte; la prima volta si esaminano gli elementi per definire gli insiemi L , E e G (costo lineare)
- ◆ La seconda volta per fare la fusione dei sottoinsiemi ordinati; anche questa operazione ha costo lineare



Analisi costo caso peggiore

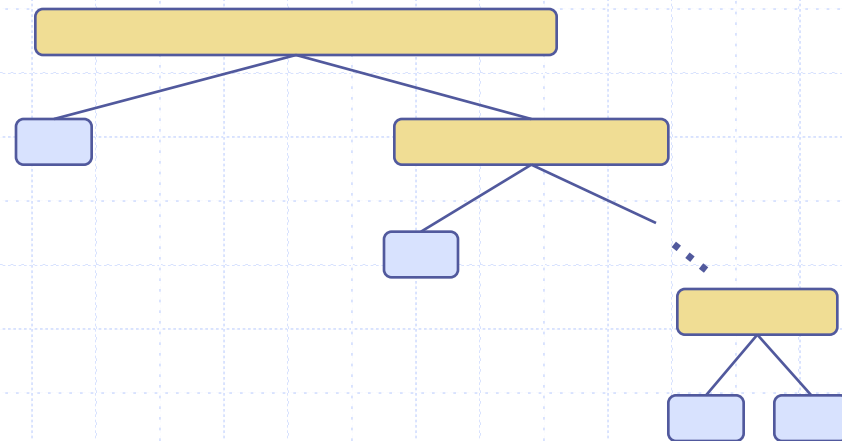
- ◆ Il caso peggiore per quick-sort si verifica quando il pivot è il minimo o il massimo elemento
- ◆ In questo caso uno fra L o G ha dimens. $n - 1$ e l'altro ha dim. 0
- ◆ Il costo in questo caso è proporzionale alla somma

$$n + (n - 1) + \dots + 2 + 1$$

- ◆ Quindi il costo nel caso peggiore di quick-sort è $O(n^2)$

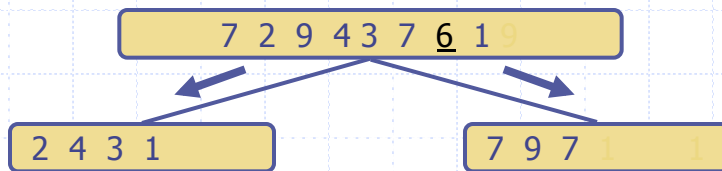
Profondità tempo

0	n
1	$n - 1$
...	...
$n - 1$	1

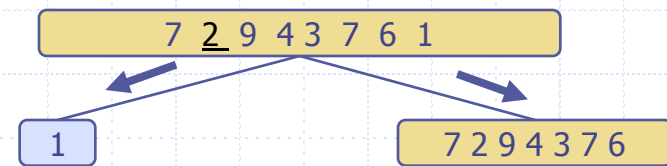


Tempo di esecuzione atteso

- ◆ Considera una attivazione ricorsiva su una sequenza of dimensione s
 - **Caso buono**: le dimens. di L e G sono ambedue minori di $3s/4$
 - **Caso cattivo**: uno fra L e G ha dimesnionne maggiore di $3s/4$

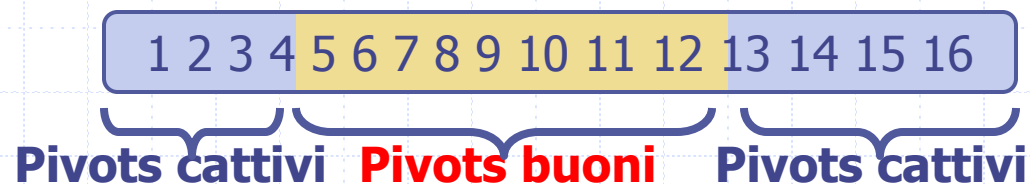


Caso buono



Caso cattivo

- ◆ Una attivazione è **buona con** probabilità $1/2$
 - $1/2$ di tutti i possibili pivot danno casi buoni:



Tempo di esecuzione atteso - 2

- ◆ **Fatto di teoria della probabilità: Il numero atteso di monete da lanciare per avere k volte Testa è $2k$**
- ◆ Nel nostro caso ci aspettiamo che, per un nodo di profondità i ,
 - $i/2$ antenati siano chiamate buone
 - dimensione attesa della sequenza da ordinare sia al più $(3/4)^{i/2}n$

◆ Pertanto, ci aspettiamo che

- Per un nodo a profondità $2\log_{4/3}n$ la dimensione attesa del nodo sia uno (se $i=2\log_{4/3}n$ sostituendo i in $(3/4)^{i/2}n$

otteniamo $(3/4)^{(2\log_{4/3}n)/2}n = 1$)

- Quindi la profondità attesa dell'albero è $O(\log n)$

◆ Per ogni profondità il lavoro totale fatto per tutti i nodi del livello è $O(n)$

◆ Quindi il tempo atteso di quick-sort è

$$O((\text{prof. attesa}) \times (\text{costo livello}) = O(n \log n))$$

expected height

$O(\log n)$

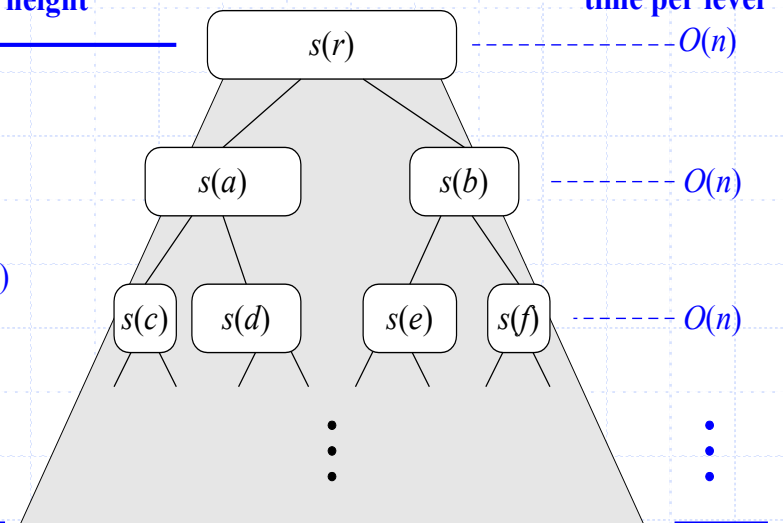
time per level

$O(n)$

$O(n)$

$O(n)$

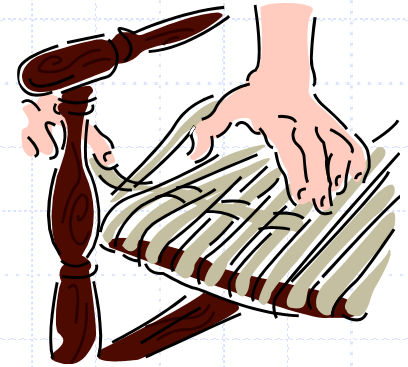
total expected time: $O(n \log n)$



Cosa sappiamo

- ◆ Caso peggiore $O(n^2)$, caso medio $O(n \log n)$
- ◆ Il caso peggiore si verifica quando scelgo come pivot un elemento molto grande (o molto piccolo)
- ◆ L'ideale sarebbe scegliere ad ogni iterazione il mediano (sta a metà nella sequenza ordinata) ma scegliere il mediano costa
- ◆ IDEA: scegli 3 (o 5) elementi a caso e prendi come pivot il mediano fra quelli scelti
- ◆ Costa poco farlo ed in pratica funziona bene!
- ◆ Nota: potrei applicare l'idea scegliendo 11 o 101 elementi a caso; esperimenti fatti mostrano che non ne vale la pena

Quick-Sort In-Place



- ◆ Quick-sort si può implementare con esecuzione in-place
- ◆ Nel passo di partizione usa operazioni di scambio per riordinare gli elementi della sequenza in modo tale che
 - Gli elementi minori del pivot abbiano **rango** minore di h
 - Gli elementi uguali al pivot abbiano **rango** fra h e k
 - Gli elementi maggiori del pivot abbiano rango maggiore di k
- ◆ Le chiamate ricorsive sono su
 - elementi con rango minore di h
 - elementi con rango maggiore di k

Algorithm *inPlaceQuickSort*(S, l, r)

Input sequenza S , ranghi l and r

Output sequenza S con gli elementi di rango fra l e r riordinati in ordine crescente

if $l \geq r$

return

$i \leftarrow$ un numero casuale fra l e r

$x \leftarrow S.\text{elemAtRank}(i)$

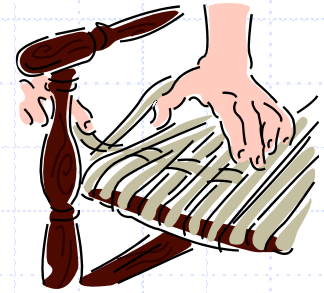
$(h, k) \leftarrow \text{inPlacePartition}(x)$

inPlaceQuickSort($S, l, h - 1$)

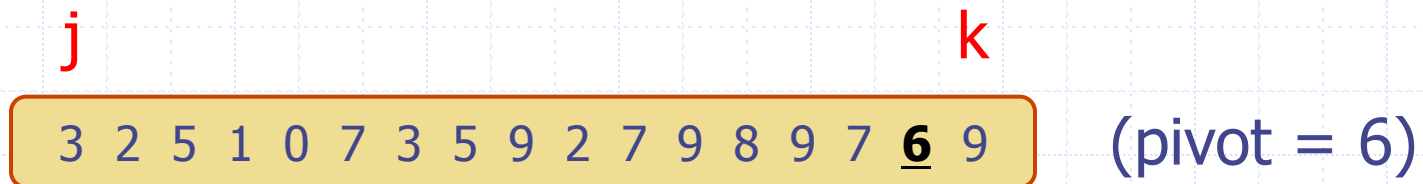
inPlaceQuickSort($S, k + 1, r$)

rango: posizione nell'ordinamento

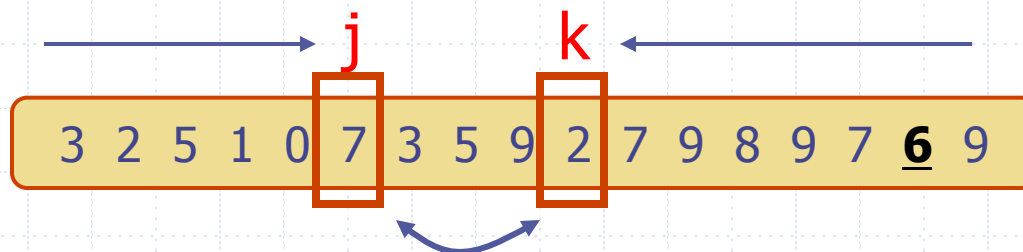
Partizionamento In-Place



- ◆ Esegui il partizionamento usando due indici per dividere S in L e $(E \cup G)$ (metodo simile divide $(E \cup G)$ in E e G)



- ◆ Ripeti fino a quando j e k si scambiano :
 - Scorri j sulla destra fino a quando trovi un elemento $\geq$ pivot.
 - Scorri k sulla sinistra fino a quando trovi un elemento $<$ pivot .
 - Scambia gli elementi con indici j e k



Implementazione Java

```
1  /** Sort the subarray S[a..b] inclusive. */
2  private static <K> void quickSortInPlace(K[ ] S, Comparator<K> comp,
3                                          int a, int b) {
4      if (a >= b) return;          // subarray is trivially sorted
5      int left = a;
6      int right = b-1;
7      K pivot = S[b];
8      K temp;                      // temp object used for swapping
9      while (left <= right) {
10         // scan until reaching value equal or larger than pivot (or right marker)
11         while (left <= right && comp.compare(S[left], pivot) < 0) left++;
12         // scan until reaching value equal or smaller than pivot (or left marker)
13         while (left <= right && comp.compare(S[right], pivot) > 0) right--;
14         if (left <= right) {      // indices did not strictly cross
15             // so swap values and shrink range
16             temp = S[left]; S[left] = S[right]; S[right] = temp;
17             left++; right--;
18         }
19     }
20     // put pivot into its final place (currently marked by left index)
21     temp = S[left]; S[left] = S[b]; S[b] = temp;
22     // make recursive calls
23     quickSortInPlace(S, comp, a, left - 1);
24     quickSortInPlace(S, comp, left + 1, b);
25 }
```

Algoritmi di Ordinamento

Algoritmo	Tempo	Note
selection-sort	$O(n^2)$	▪ in-place ▪ lento (solo per input piccoli)
insertion-sort	$O(n^2)$	▪ in-place ▪ lento (solo per input piccoli)
quick-sort	$O(n \log n)$ atteso	▪ in-place, fa scelte casuali ▪ il più veloce in pratica (la costante "nascosta" nella notaz. $O()$ è piccola rispetto agli altri
heap-sort	$O(n \log n)$	▪ in-place ▪ veloce (buono per grandi input)
merge-sort	$O(n \log n)$	▪ accesso sequenziale ai dati ▪ veloce (buono per grandi input)