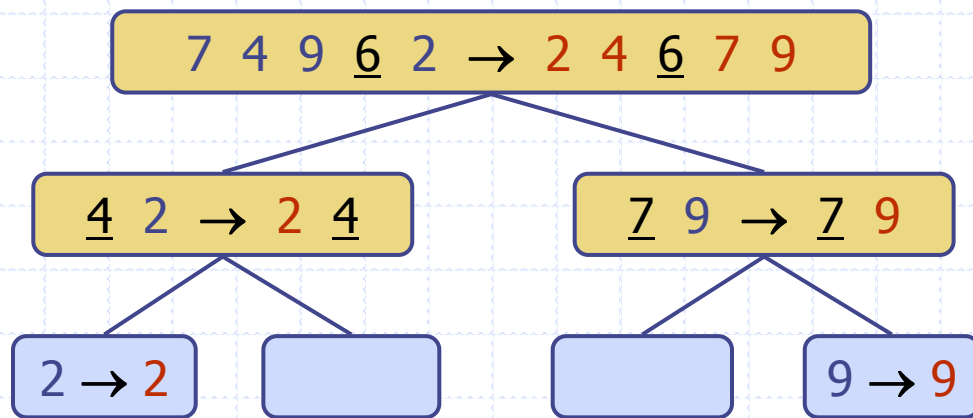


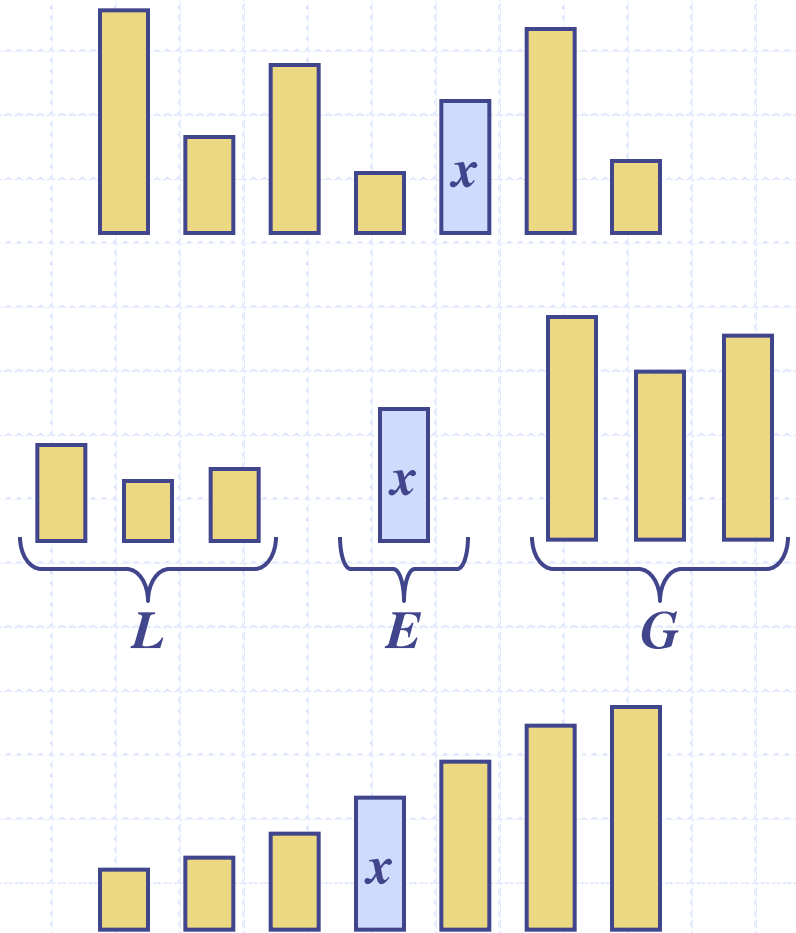
Quick-Sort



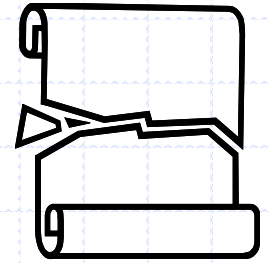
Quick-Sort (§ 11.2)

◆ **Quick-sort** è un algoritmo di ordinamento randomizzato basato sul paradigma divide-et-impera:

- **Divide**: scegli un elemento casuale x (chiamato **pivot**) e partiziona S in
 - ◆ L : elementi minori di x
 - ◆ E : elementi uguali a x
 - ◆ G : elementi maggiori di x
- **Ricorri**: ordina L e G
- **Impera**: unisci L , E e G



Partizione



- ◆ Partizioniamo la sequenza di input come segue:
 - rimuoviamo ogni elemento y da S e
 - inseriamo y in L , E or G , sulla base del risultato del confronto con il pivot x
- ◆ Ogni inserimento e rimozione è all'inizio o alla fine della sequenza e quindi richiede tempo $O(1)$
- ◆ Quindi, il passo di partizione del quick-sort richiede tempo $O(n)$

Algorithm *partition*(S, p)

Input sequence S , position p of pivot

Output subsequences L , E , G of the elements of S less than, equal to, or greater than the pivot, resp.

$L, E, G \leftarrow$ empty sequences

$x \leftarrow S.remove(p)$

$E.insertLast(x)$

while $\neg S.isEmpty()$

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

if $y < x$

$L.insertLast(y)$

else if $y = x$

$E.insertLast(y)$

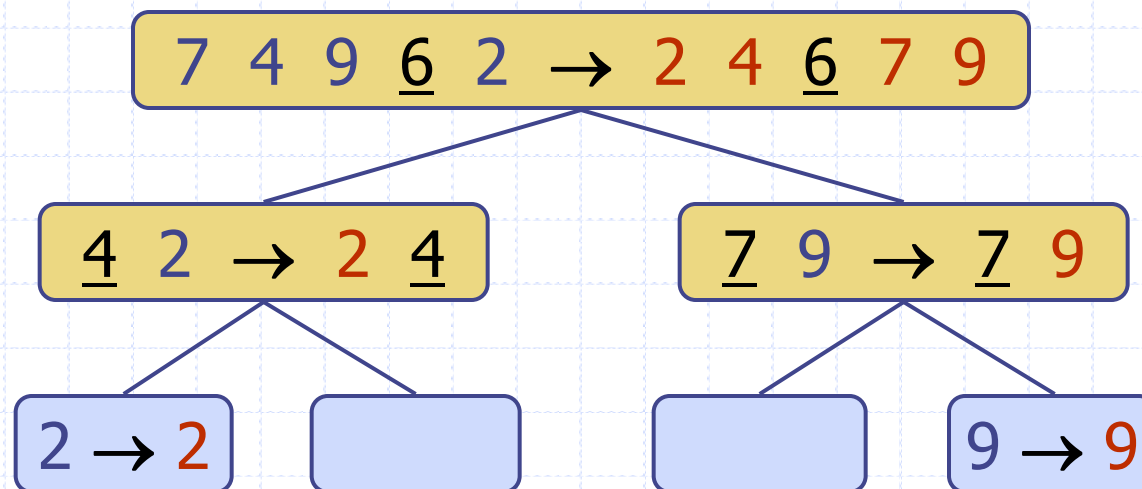
else $\{ y > x \}$

$G.insertLast(y)$

return L, E, G

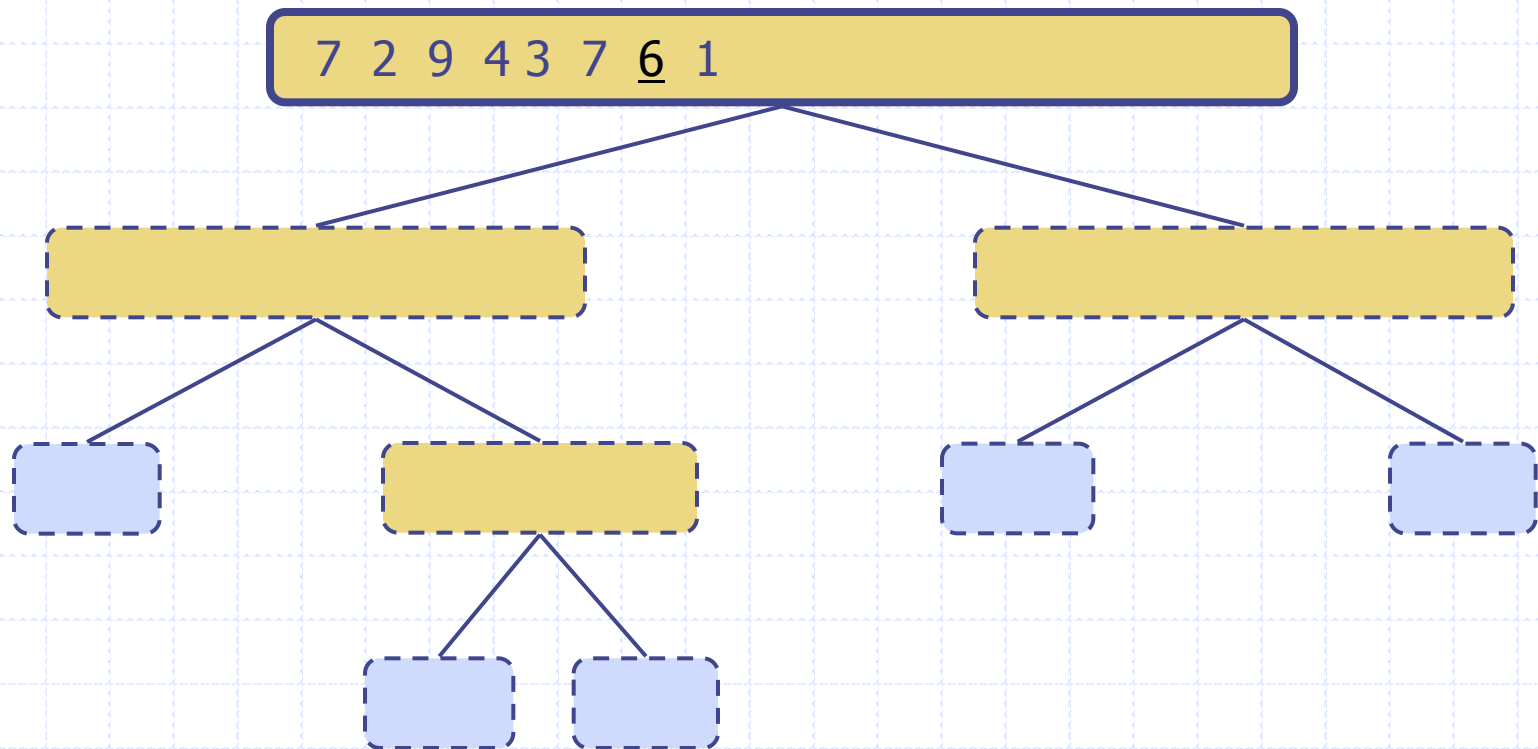
Albero di Quick-Sort

- ◆ Un'esecuzione del quick-sort è descritta da un albero binario
 - Ogni nodo rappresenta una chiamata ricorsiva di quick-sort e memorizza
 - ◆ sequenza non ordinata prima dell'esecuzione e il suo pivot
 - ◆ sequenza ordinata alla fine dell'esecuzione
 - La radice è la chiamata iniziale
 - Le foglie sono chiamate su sottosequenze di dimensione 0 o 1



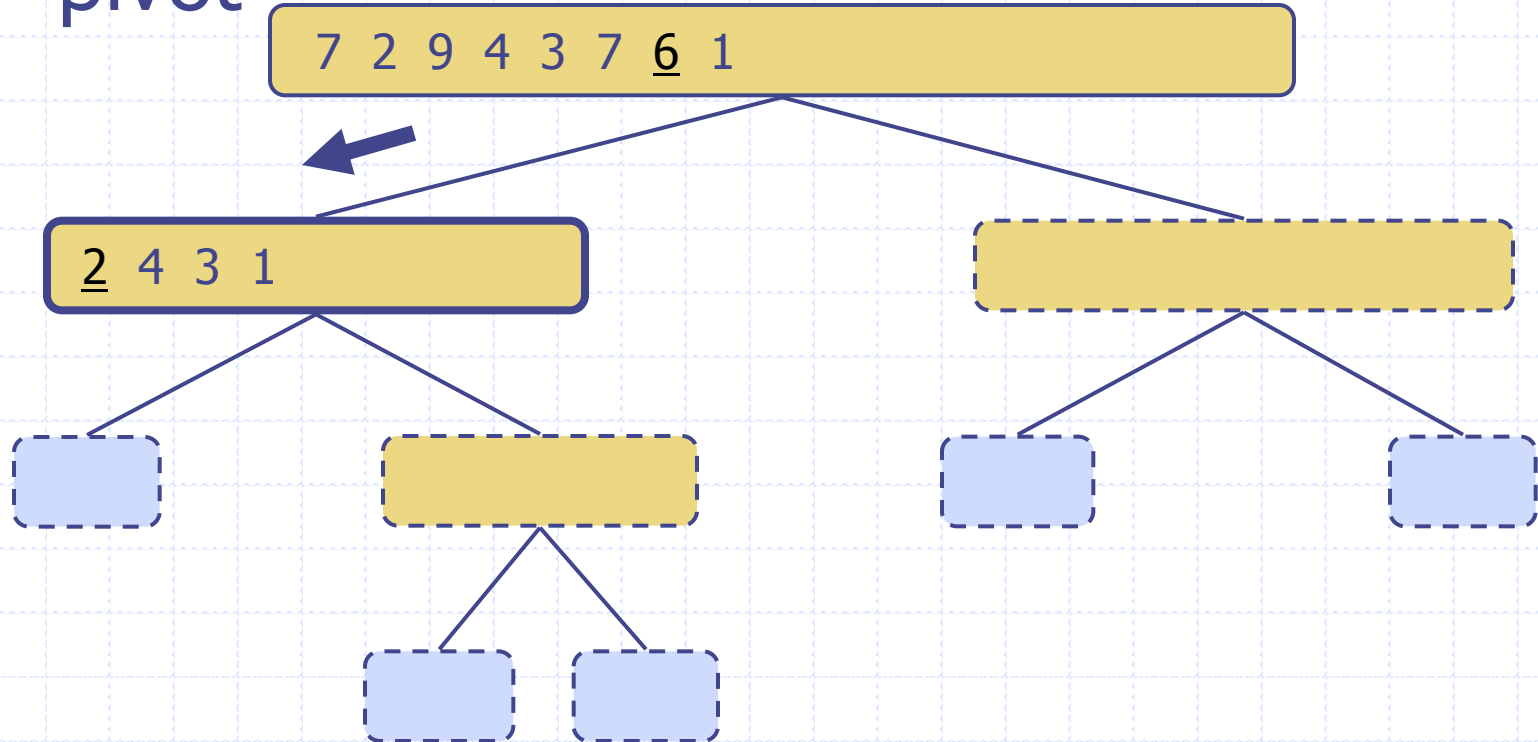
Esempio di esecuzione

◆ Selezione del pivot



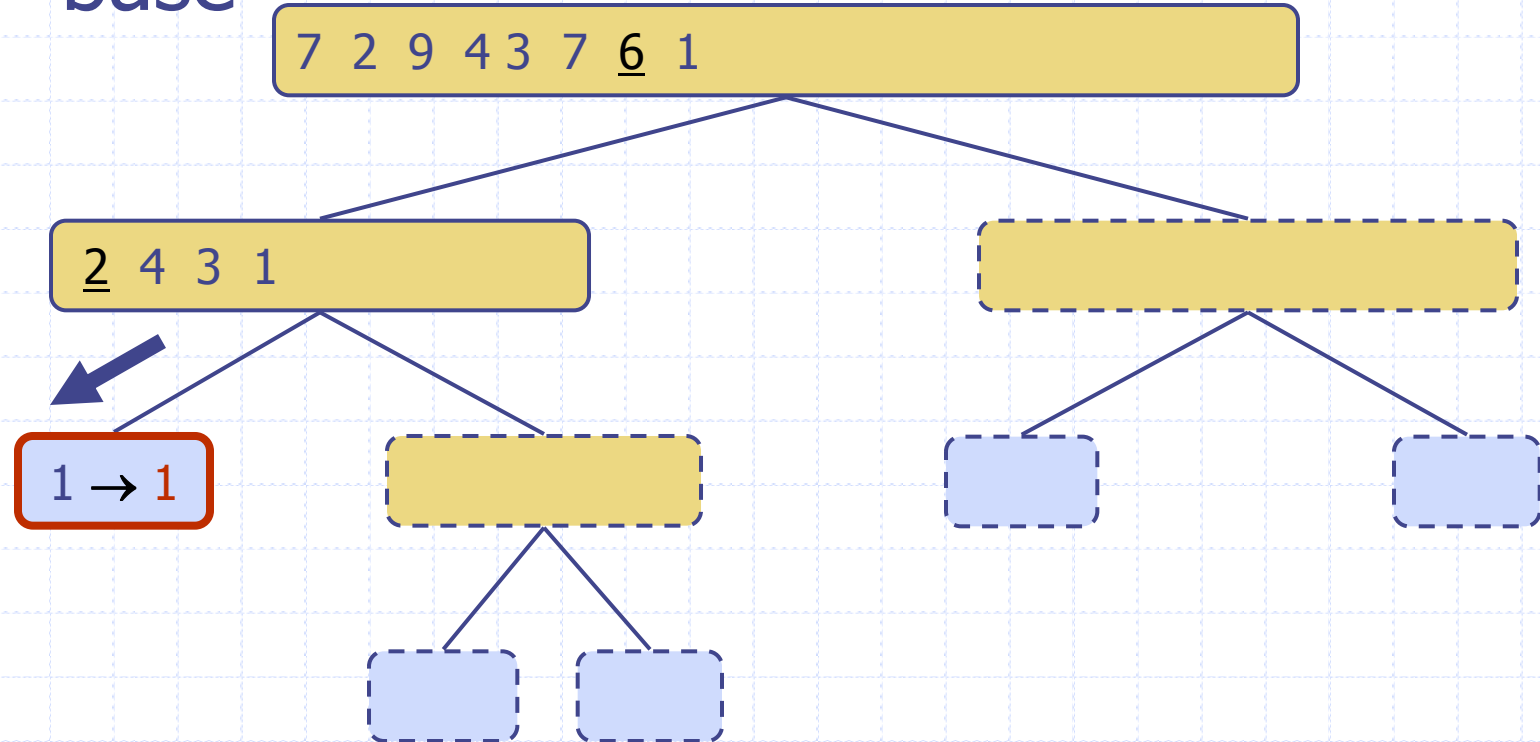
Esempio di esecuzione (cont.)

◆ Partizione, chiamata ricorsiva, selez. pivot



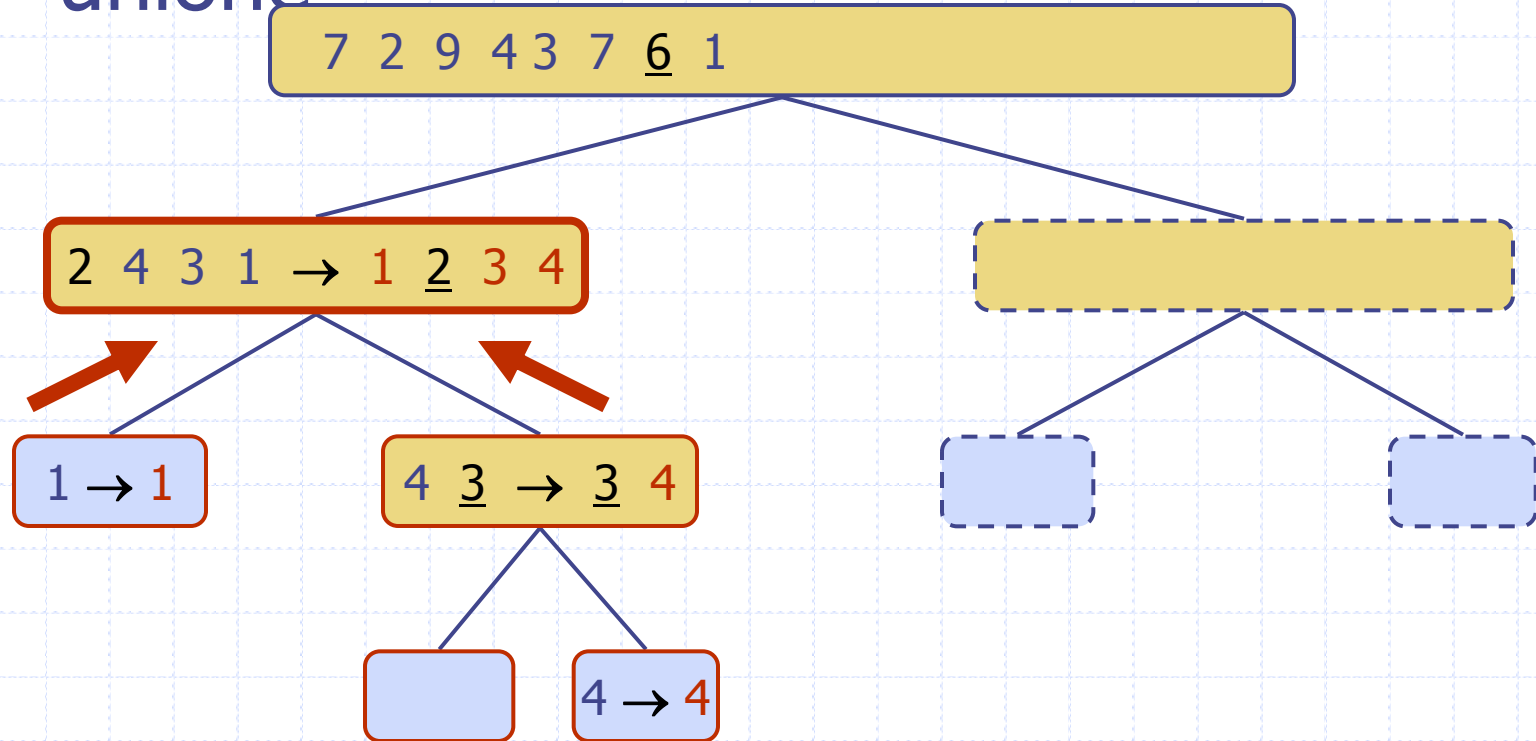
Esempio di esecuzione (cont.)

◆ Partizione, chiamata ricorsiva, caso base



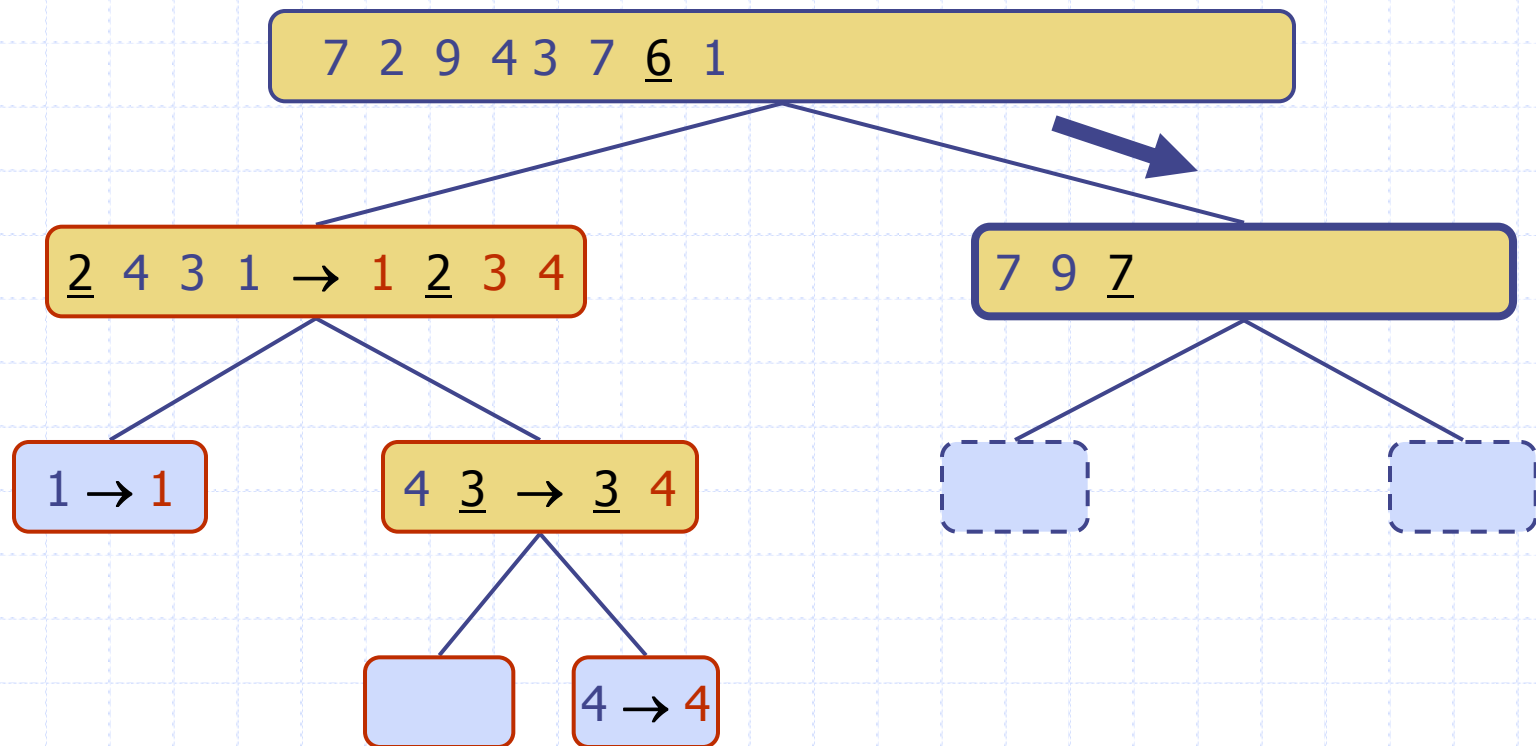
Esempio di esecuzione (cont.)

◆ Chiamata ricorsiva, ..., caso base, unione



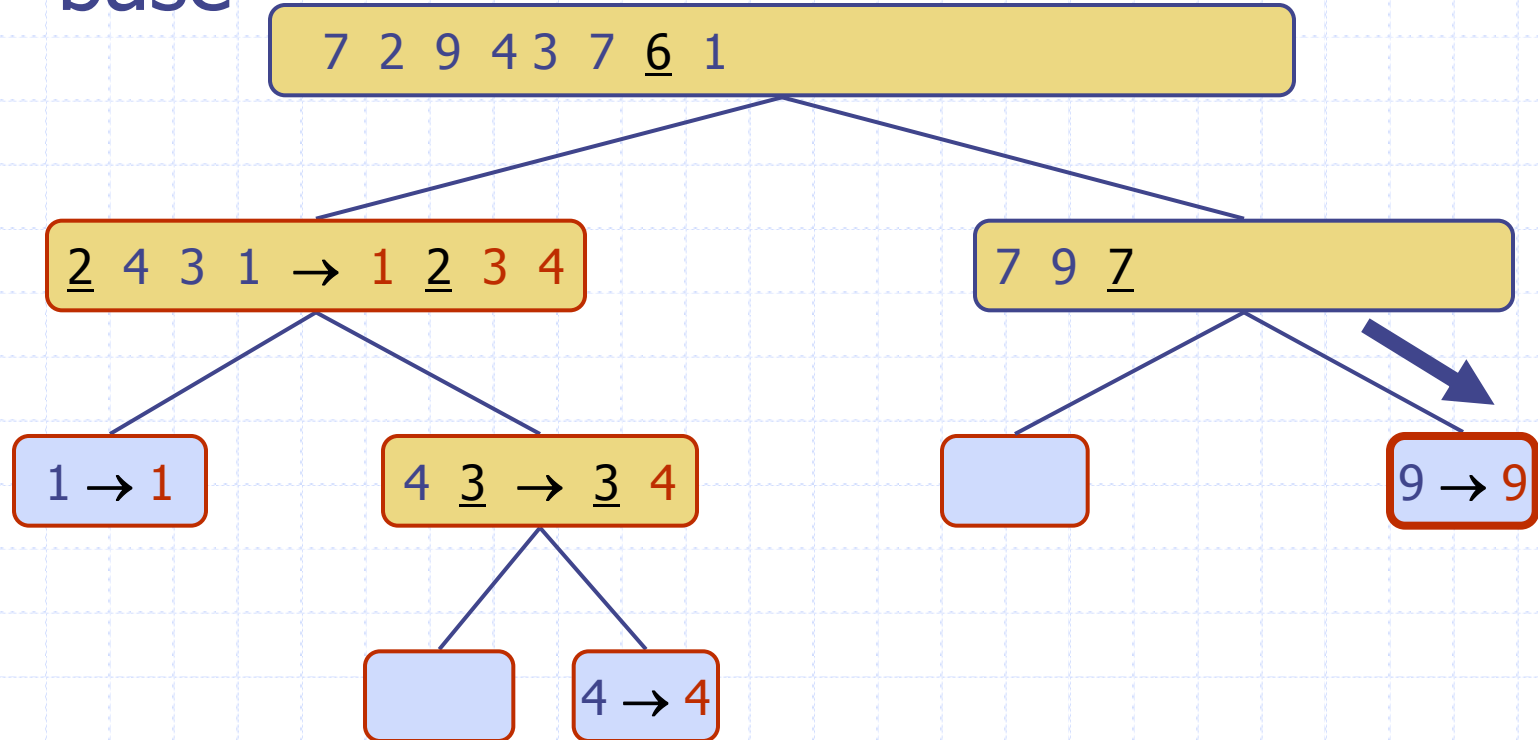
Esempio di esecuzione (cont.)

◆ Chiamata ricorsiva, selez. pivot



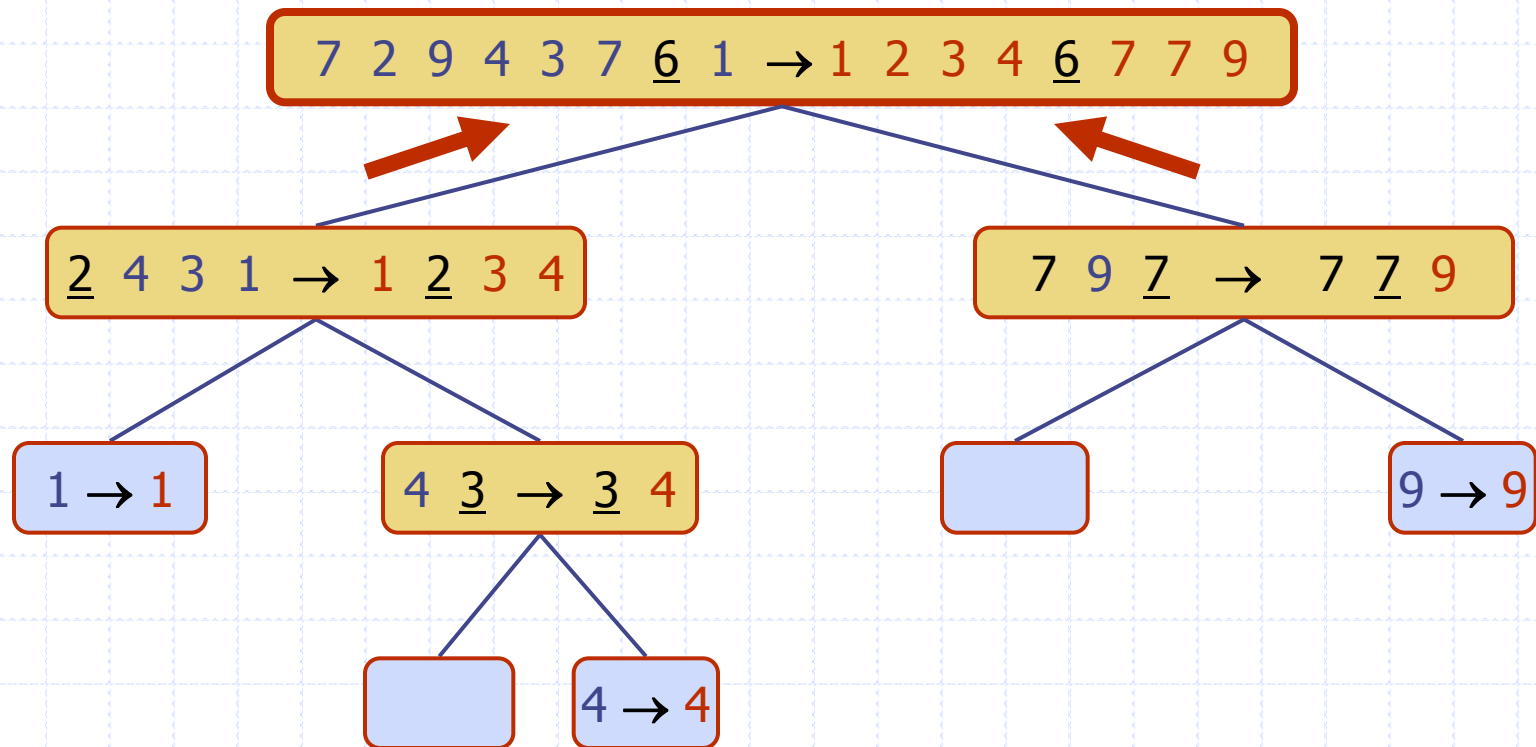
Esempio di esecuzione (cont.)

◆ Partizione, ..., chiamata ricorsiva, caso base



Esempio di esecuzione (cont.)

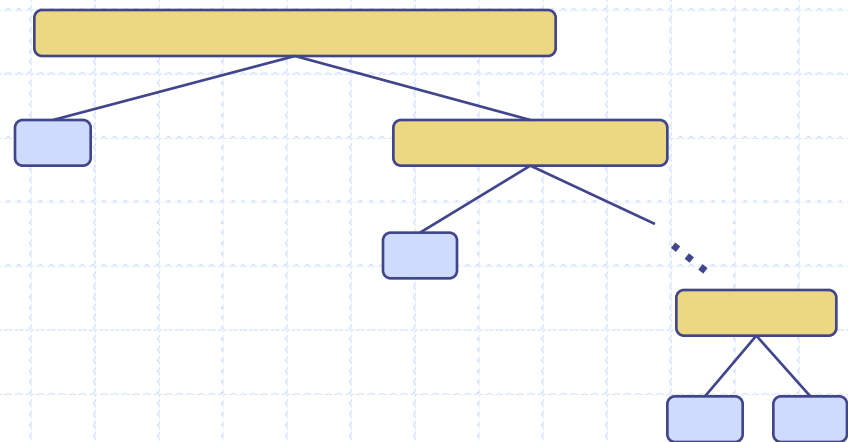
◆ Unione, unione



Tempo di esecuzione nel caso peggiore

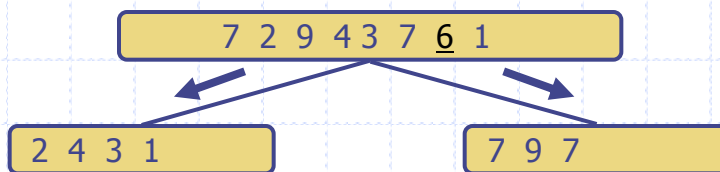
- ◆ Il caso peggiore per il quick-sort si presenta quando il pivot è l'unico minimo o massimo elemento
- ◆ Uno di L e G ha dimensione $n - 1$ e l'altro ha dimensione 0
- ◆ Il tempo di esecuzione è proporzionale alla somma
$$n + (n - 1) + \dots + 2 + 1$$
- ◆ Quindi, il tempo di esecuzione nel caso peggiore è $O(n^2)$

profondità	tempo
0	n
1	$n - 1$
...	...
$n - 1$	1

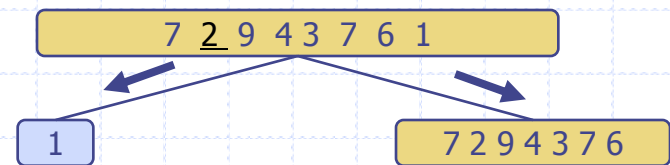


Tempo medio di esecuzione

- ◆ Considera una chiamata ricorsiva di quick-sort su una sequenza di dimensione s
 - **Buona chiamata:** le dimensioni di L e G sono ognuna meno di $3s/4$
 - **Cattiva chiamata:** uno di L e G ha maggiore di $3s/4$



Buona chiamata



Cattiva chiamata

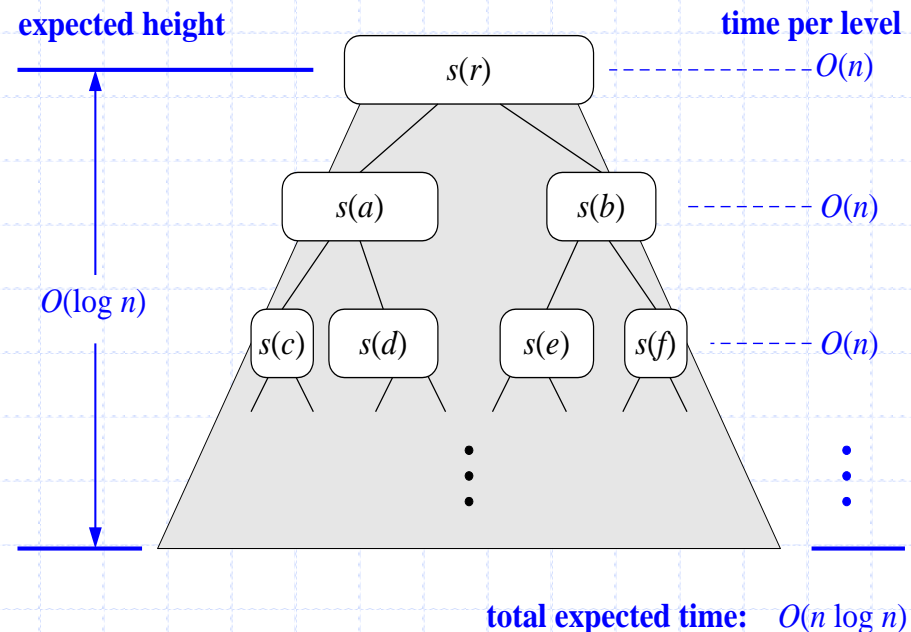
- ◆ Una chiamata è buona con prob. $1/2$
 - $1/2$ dei possibili pivots determina buone chiamate:



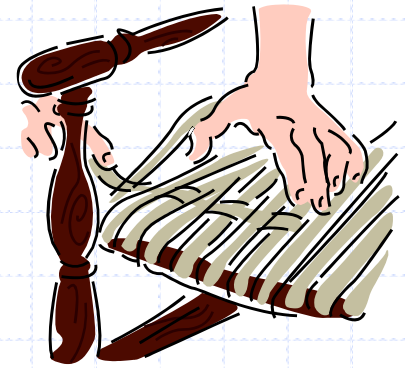
Tempo medio di esecuzione, parte 2

- ◆ **Fatto probabilistico:** Il numero atteso di lanci di monete necessari per ottenere k volte testa è $2k$
- ◆ Per un nodo di profondità i , attendiamo
 - $i/2$ antenati con chiamate buone
 - La dimensione della sequenza di input per la chiamata corrente al più' $(3/4)^{i/2}n$

- ◆ Quindi, abbiamo
 - Per un nodo di profondità $2\log_{4/3}n$, la dimensione attesa dell'input è 1
 - Il valore atteso dell'altezza dell'albero di quick-sort tree è $O(\log n)$
- ◆ La quantità di lavoro ai nodi di uguale profondità è $O(n)$
- ◆ Il tempo di esecuzione atteso del quick-sort è $O(n \log n)$



Quick-Sort sul posto



- ◆ Quick-sort può essere implementato sul posto
- ◆ Nella fase di partizione, usiamo operazioni di riordino degli elementi della sequenza di input in modo tale che
 - gli elementi minori del pivot hanno rank minore di h
 - gli elementi uguali al pivot hanno rank tra h e k
 - gli elementi maggiori del pivot hanno rank maggiore di k
- ◆ Le chiamate ricorsive considerano
 - elementi con rank minori di h
 - elementi con rank maggiore di k

Algorithm *inPlaceQuickSort*(S, l, r)

Input sequence S , ranks l and r

Output sequence S with the elements of rank between l and r rearranged in increasing order

if $l \geq r$

return

$i \leftarrow$ a random integer between l and r

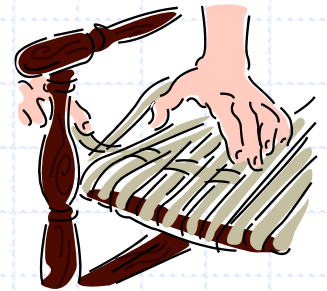
$x \leftarrow S.\text{elemAtRank}(i)$

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

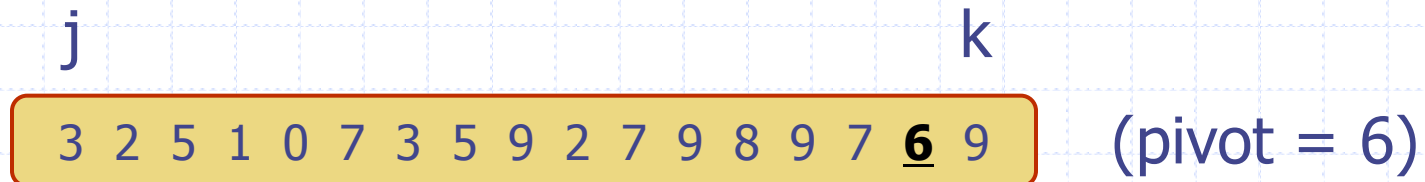
inPlaceQuickSort($S, l, h - 1$)

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

Partizionamento sul posto

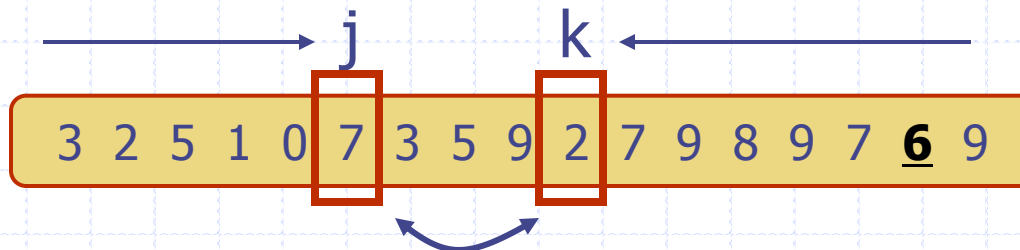


- Realizza la partizione usando due indici per dividere S in L e $E \cup G$ (un metodo simile può dividere $E \cup G$ in E e G).



- Ripeti finché j e k si incontrano:

- Muovi j alla destra finché si trova un elemento $\geq x$.
- Muovi k alla sinistra finché si trova un elemento $< x$.
- Scambia gli elementi con indici j e k

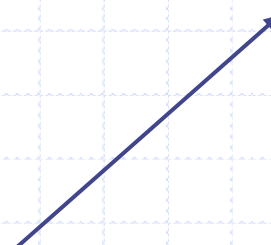


Riepilogo degli Algoritmi di Ordinamento

Algoritmo	Tempo	Proprietà
selection-sort	$O(n^2)$	<ul style="list-style-type: none">◆ sul posto◆ lento (buono per input piccoli)
insertion-sort	$O(n^2)$	<ul style="list-style-type: none">◆ sul posto◆ lento (buono per input piccoli)
quick-sort	$O(n \log n)$ atteso	<ul style="list-style-type: none">◆ sul posto, randomizzato◆ più veloce (buono per input grandi)
heap-sort	$O(n \log n)$	<ul style="list-style-type: none">◆ sul posto◆ veloce (buono per input grandi)
merge-sort	$O(n \log n)$	<ul style="list-style-type: none">◆ accesso sequenziale ai dati◆ veloce (buono per input enormi)

Java Implementation

funziona solo
per elementi
distinti



```
public static void quickSort (Object[] S, Comparator c) {  
    if (S.length < 2) return; // the array is already sorted in this case  
    quickSortStep(S, c, 0, S.length-1); // recursive sort method  
}  
  
private static void quickSortStep (Object[] S, Comparator c,  
                                   int leftBound, int rightBound) {  
    if (leftBound >= rightBound) return; // the indices have crossed  
    Object temp; // temp object used for swapping  
    Object pivot = S[rightBound];  
    int leftIndex = leftBound; // will scan rightward  
    int rightIndex = rightBound-1; // will scan leftward  
    while (leftIndex <= rightIndex) { // scan right until larger than the pivot  
        while ( (leftIndex <= rightIndex) && (c.compare(S[leftIndex], pivot)<=0) )  
            leftIndex++;  
        // scan leftward to find an element smaller than the pivot  
        while ( (rightIndex >= leftIndex) && (c.compare(S[rightIndex], pivot)>=0) )  
            rightIndex--;  
        if (leftIndex < rightIndex) { // both elements were found  
            temp = S[rightIndex];  
            S[rightIndex] = S[leftIndex]; // swap these elements  
            S[leftIndex] = temp;  
        }  
    } // the loop continues until the indices cross  
    temp = S[rightBound]; // swap pivot with the element at leftIndex  
    S[rightBound] = S[leftIndex];  
    S[leftIndex] = temp; // the pivot is now at leftIndex, so recurse  
    quickSortStep(S, c, leftBound, leftIndex-1);  
    quickSortStep(S, c, leftIndex+1, rightBound);  
}
```