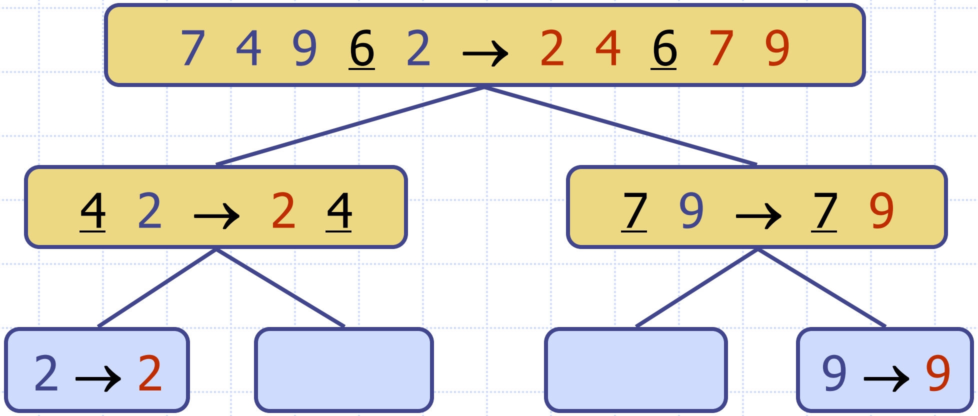
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Quick-Sort



Aim and Learning Objectives

- To be able to *understand* and *describe* the quick sort algorithm
- To be able to *analyze* the complexity of the quick sort algorithm
- To be able to *implement* the quick sort algorithm and *apply* it to solve problems

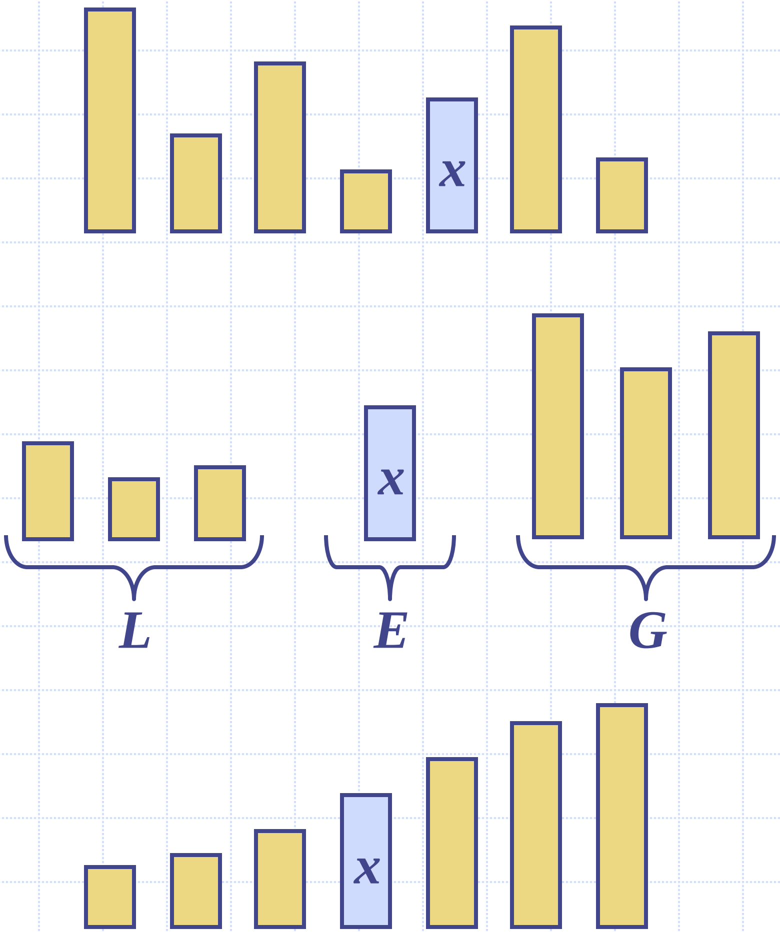
Reading

M. T. Goodrich, R. Tamassia and M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th Edition, 2014.

- **Chapter 13. Sorting and Selection**

Quick-Sort

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E and G



Partition

- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

```
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 ← empty sequences
    x ← S.remove(p)
    while ¬S.isEmpty()
        y ← 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
```

Quick Sort

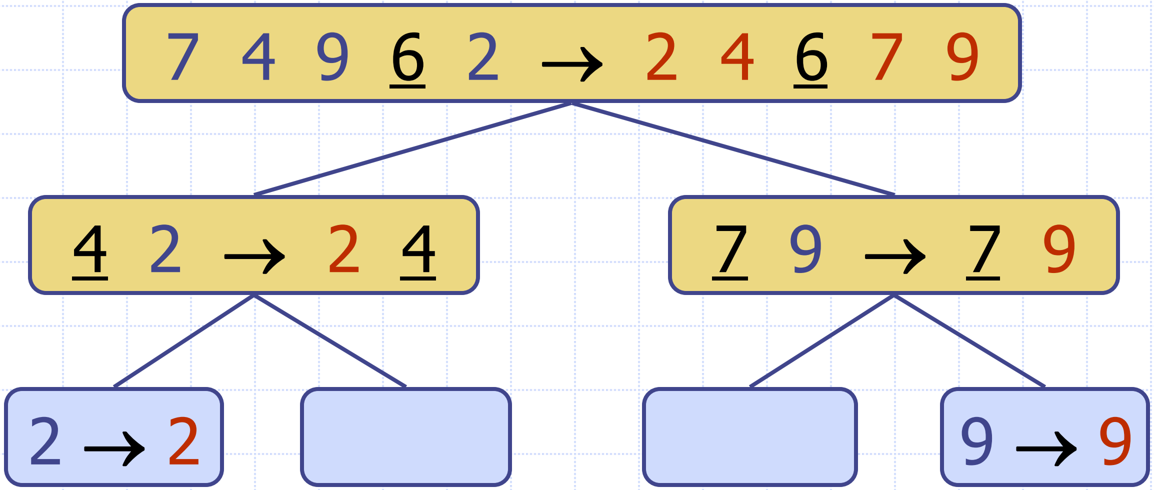
Algorithm *quickSort*(S)
  Input sequence S with n elements
  Output sequence S sorted according to C
  if $S.size() > 1$
    p ← getPivot(S)
    $(L, E, G) \leftarrow$ *partition*(S, p)
    quickSort(L)
    quickSort(G)
    $S \leftarrow$ *merge*(L, E, G)

How to define the algorithm merge(L, E, G)?

Write down the pseudocode by yourself.

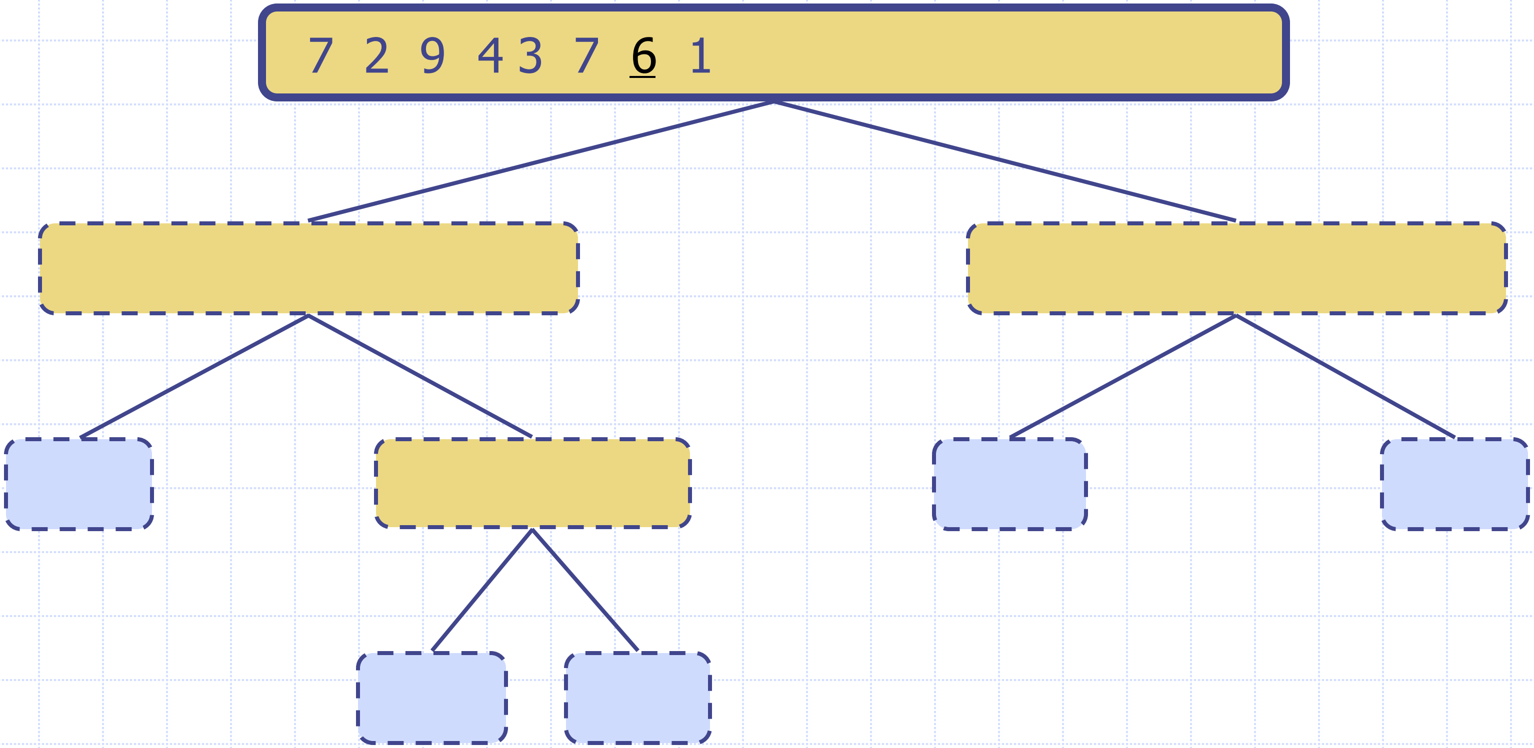
Quick-Sort Tree

- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1



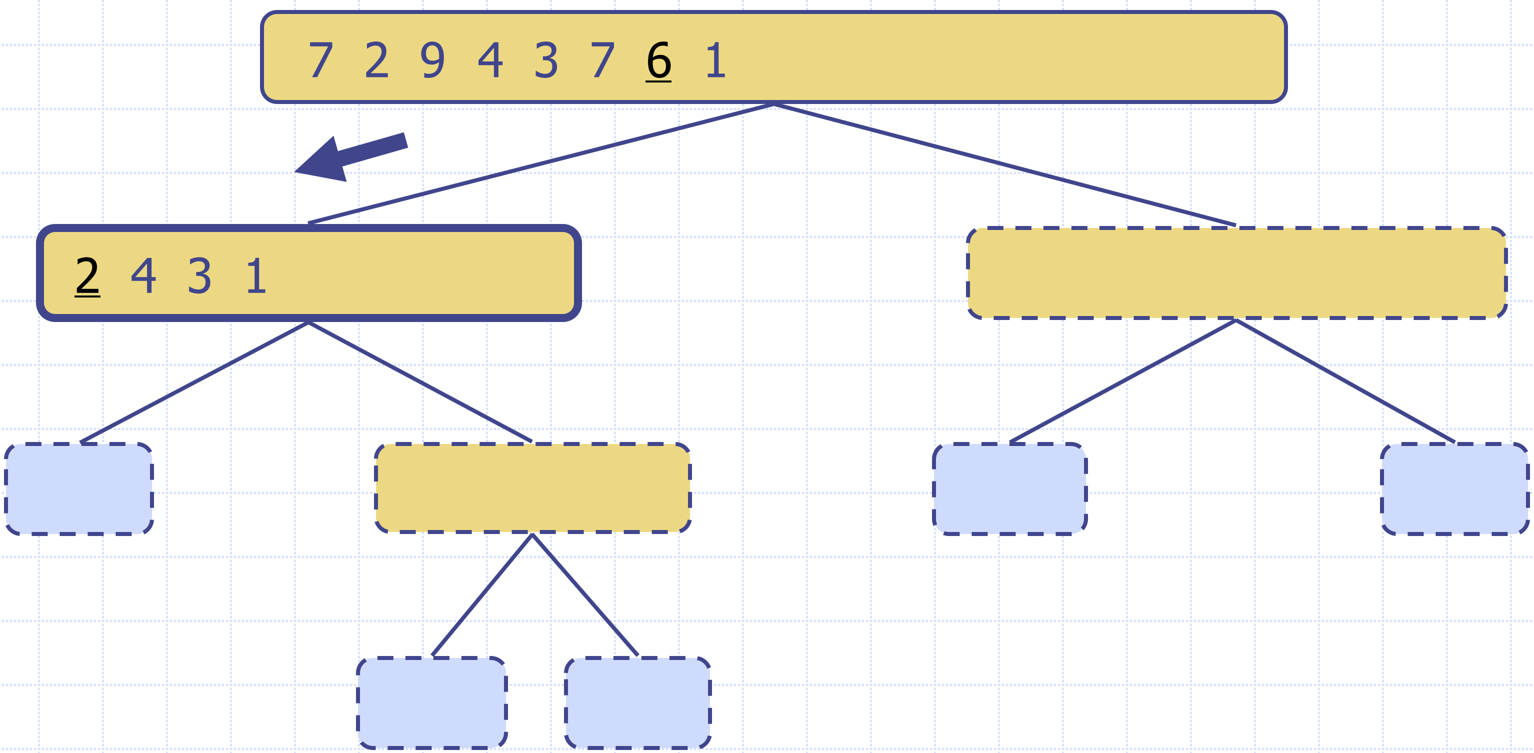
Execution Example

- Pivot selection



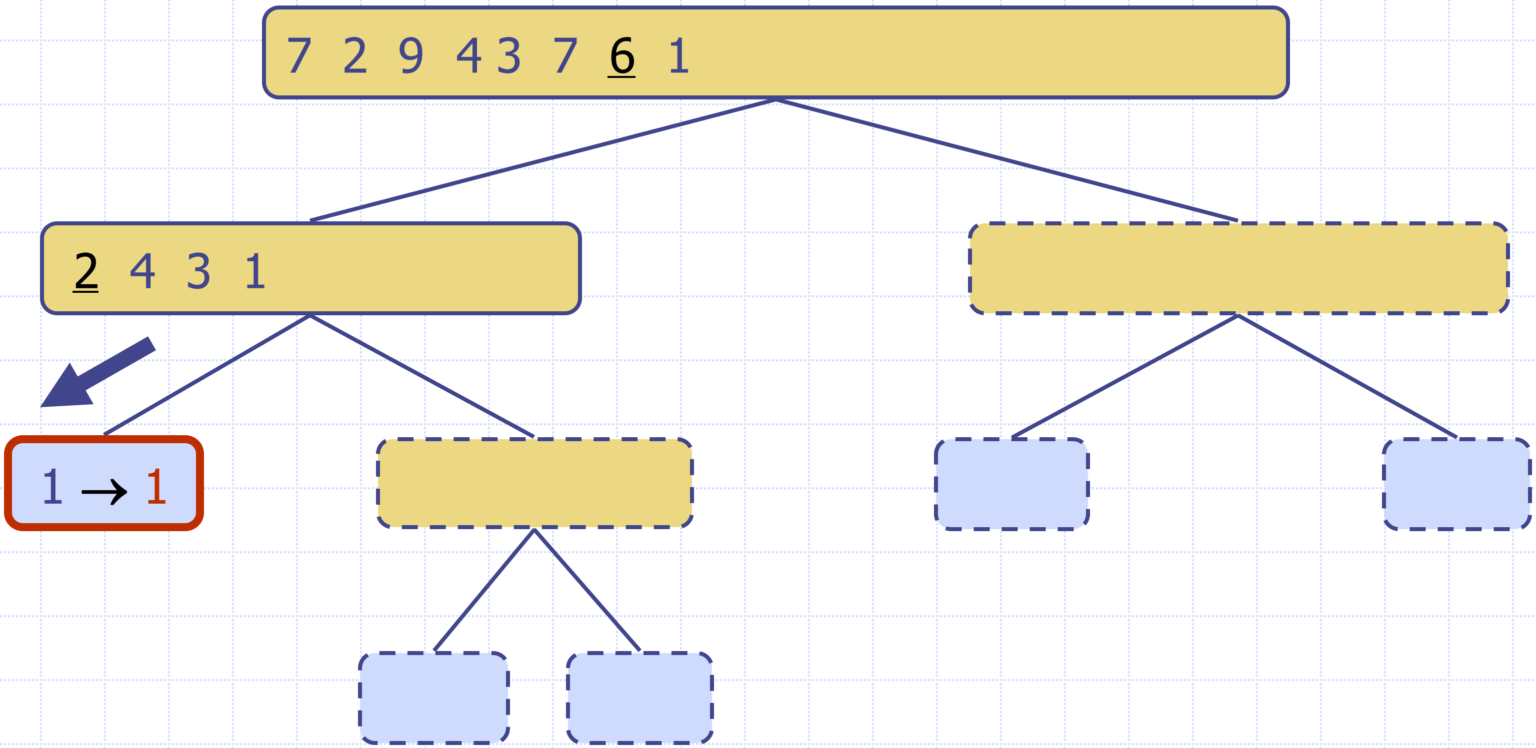
Execution Example (cont.)

- Partition, recursive call, pivot selection



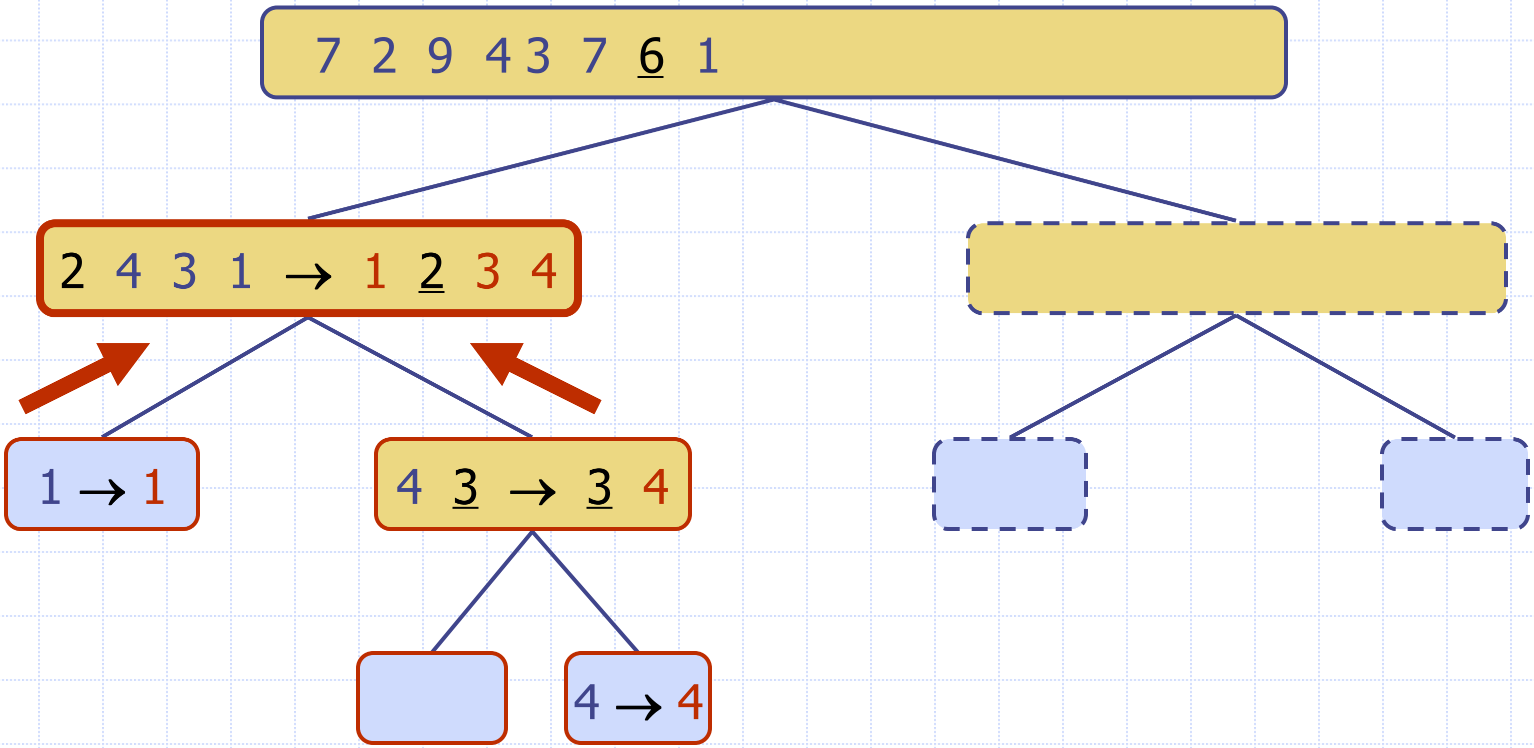
Execution Example (cont.)

- Partition, recursive call, base case



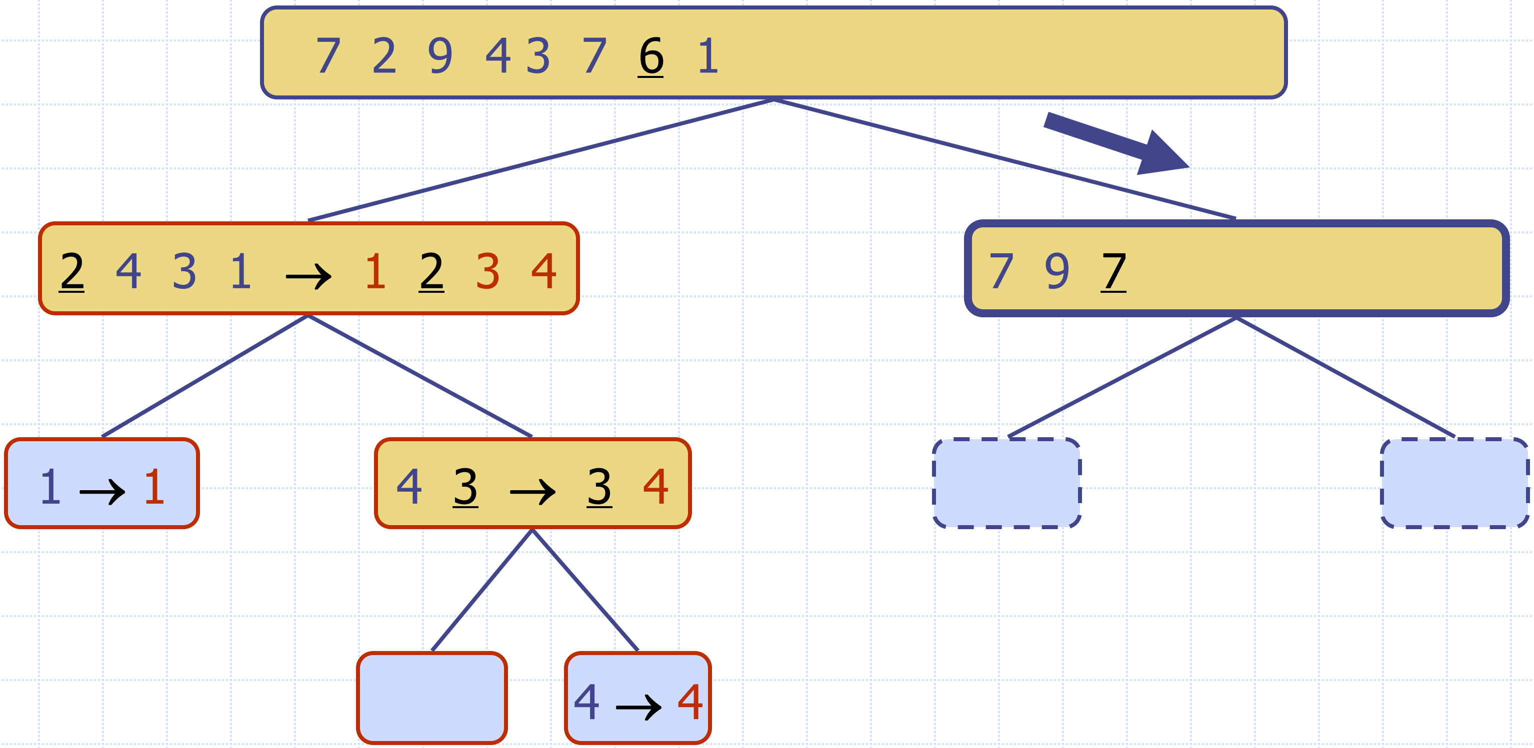
Execution Example (cont.)

- Recursive call, …, base case, join



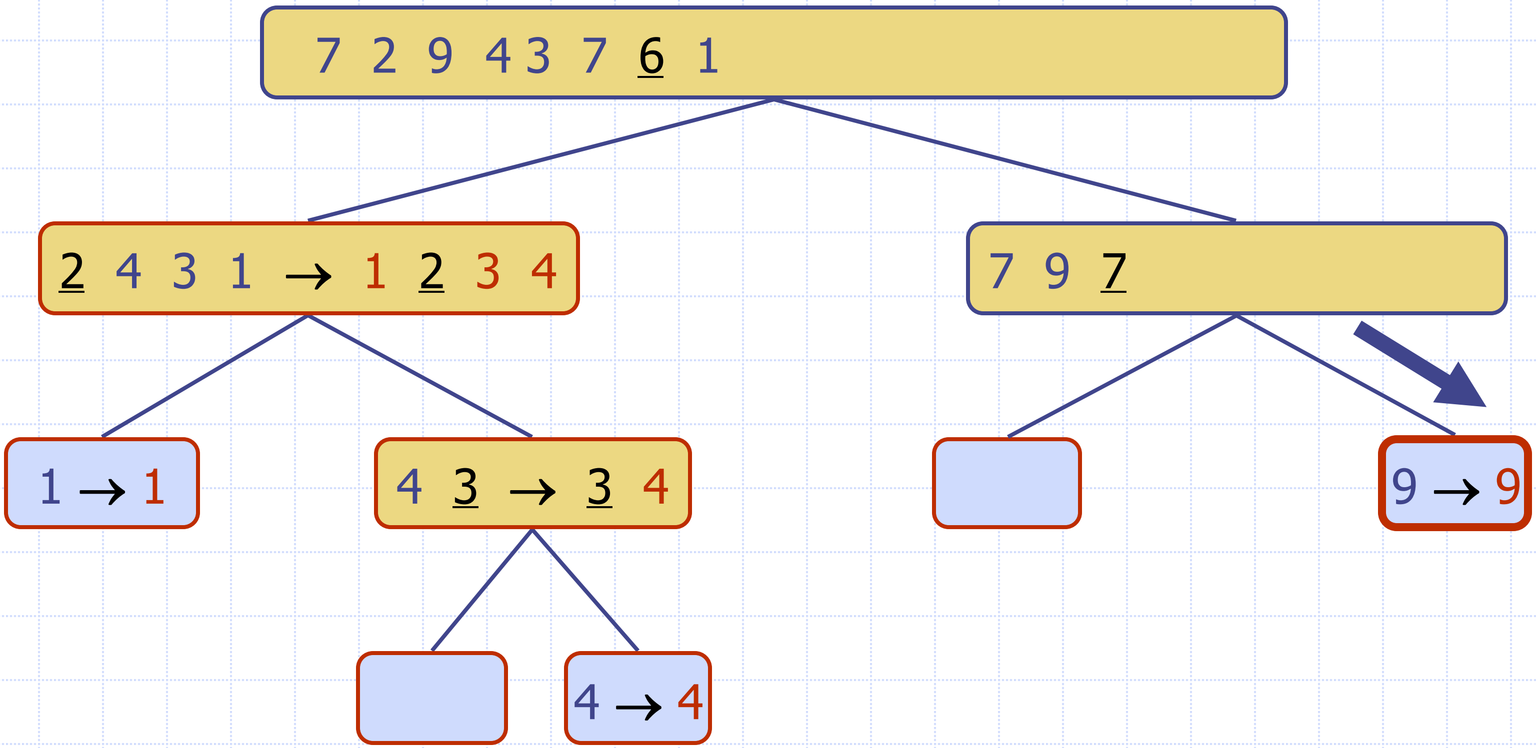
Execution Example (cont.)

- Recursive call, pivot selection



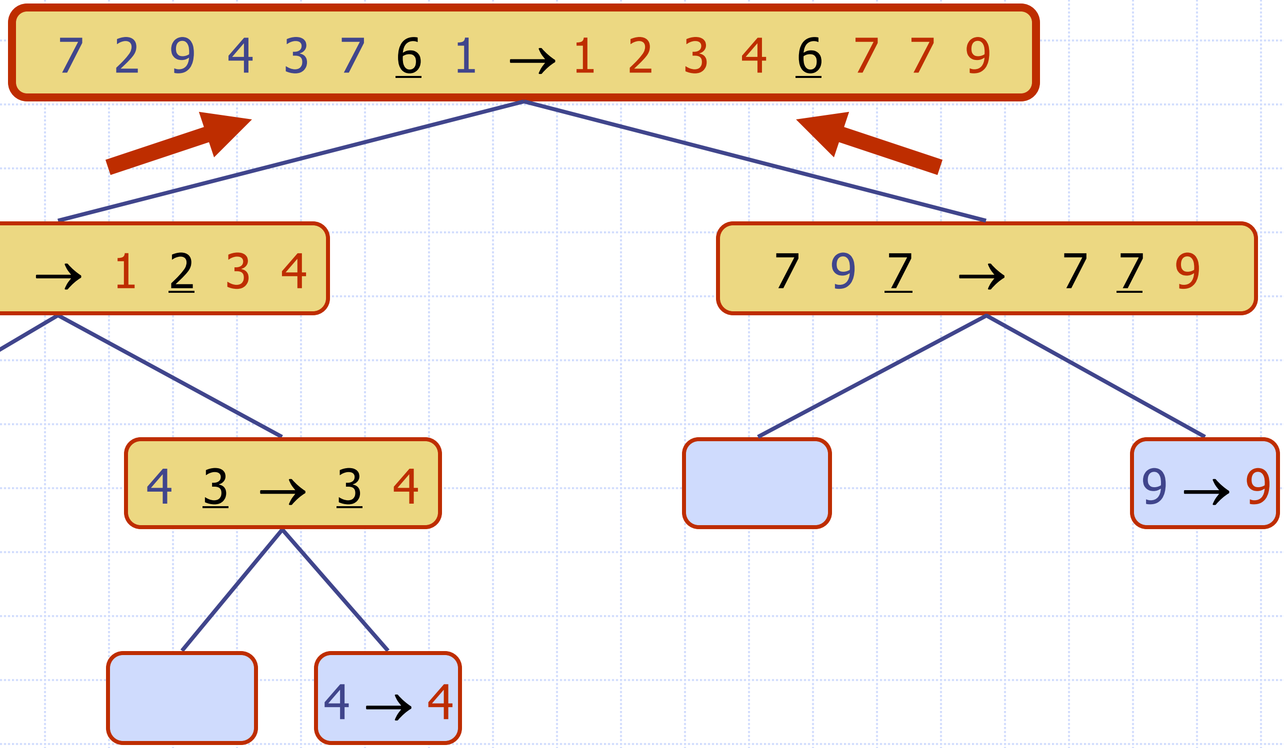
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



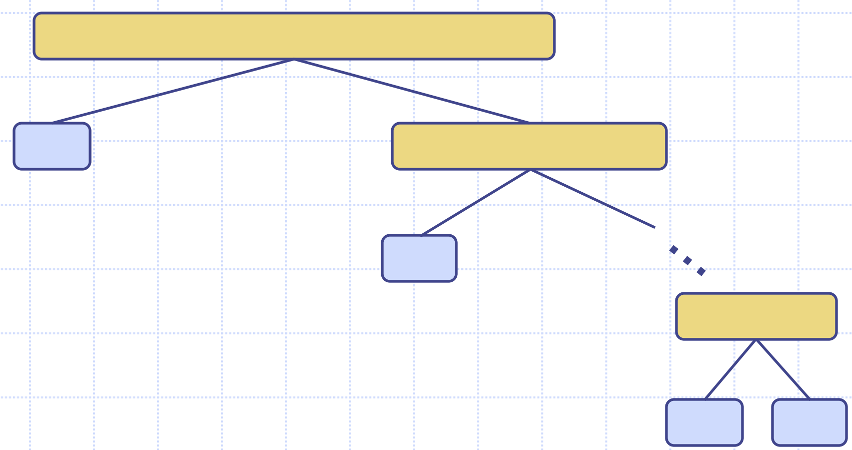
Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of L and G has size $n - 1$ and the other has size 0
- The running time is proportional to the sum

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

- Thus, the worst-case running time of quick-sort is $O(n^2)$

depth	time
0	n
1	$n - 1$
…	…
$n - 1$	1



Best-case Running Time

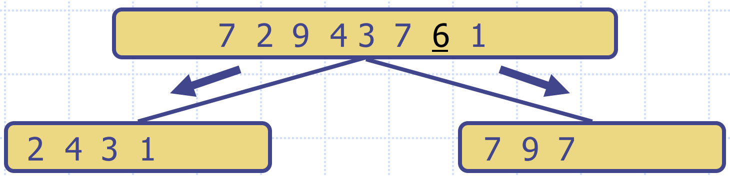
- The best case for quick-sort occurs when the pivot is the median element
- The L and G parts are equal – the sequence is split in halves, like in merge sort
- Thus, the best-case running time of quick-sort is $O(n \log n)$

Average-case Running Time

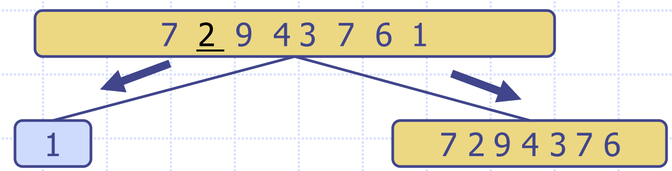
- The average case for quick-sort: half of the times, the pivot is roughly in the middle
- Thus, the average-case running time of quick-sort is $O(n \log n)$ again
- The detailed proof is in the textbook.

Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size s
  - **Good call:** the sizes of L and G are each less than $3s/4$
  - **Bad call:** one of L and G has size greater than $3s/4$

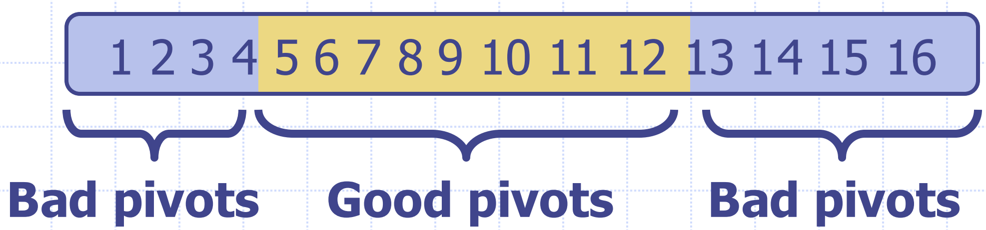


Good call



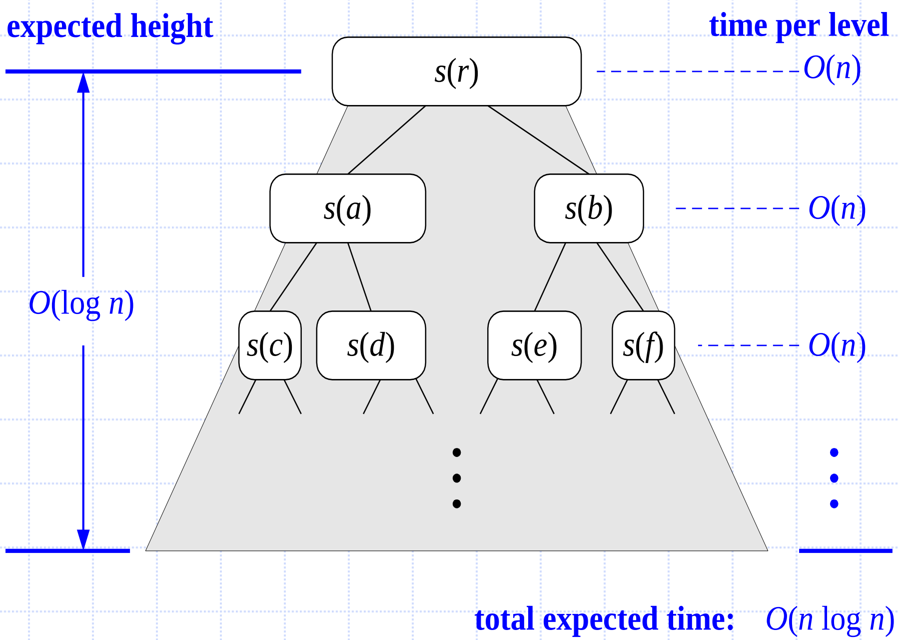
Bad call

- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:



Expected Running Time, Part 2

- Probabilistic Fact: The expected number of coin tosses required in order to get k heads is $2k$
- For a node of depth i, we expect
  - $i/2$ ancestors are good calls
  - The size of the input sequence for the current call is at most $(3/4)^{i/2}n$
- Therefore, we have
  - For a node of depth $2\log_{4/3} n$, the expected input size is one (use $\log_a b = \log_c b \,/\, \log_c a$)
  - The expected height of the quick-sort tree is $O(\log n)$ (prove $2\log_{4/3} n$ is in $O(\log n)$)
- The amount or work done at the nodes of the same depth is $O(n)$
- Thus, the expected running time of quick-sort is $O(n \log n)$

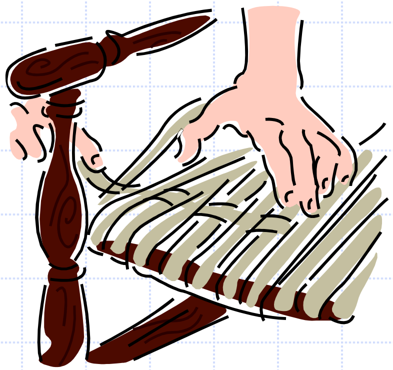


In-place

- An algorithm is ***in-place*** if it uses only a small amount of memory in addition to that needed for the original input.
- Among all the sorting algorithms we covered so far, which of them are in-place?

In-Place Quick-Sort

- Quick-sort can be implemented to run in-place
- In the partition step, we use replace operations to rearrange the elements of the input sequence such that
  - the elements less than the pivot have rank less than ***h***
  - the elements equal to the pivot have rank between ***h*** and ***k***
  - the elements greater than the pivot have rank greater than ***k***
- The recursive calls consider
  - elements with rank less than ***h***
  - elements with rank greater than ***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

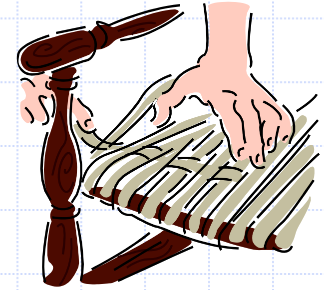
  $x \leftarrow S.elemAtRank(i)$

  $(h, k) \leftarrow inPlacePartition(x)$

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

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

In-Place Partitioning



- Perform the partition using two indices to split S into L and E U G (a similar method can split E U G into E and G).

j k

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 **6** 9 (pivot = 6)

- Repeat until j and k cross:
  - Scan j to the right until finding an element $\geq$ x.
  - Scan k to the left until finding an element < x.
  - Swap elements at indices j and k

j k

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 **6** 9

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
insertion-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	▪ in-place, randomized ▪ fastest (good for large inputs)
heap-sort	$O(n \log n)$	▪ in-place ▪ fast (good for large inputs)
merge-sort	$O(n \log n)$	▪ sequential data access ▪ fast (good for huge inputs)

Reading

M. T. Goodrich, R. Tamassia and M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th Edition, 2014.

- **Chapter 13. Sorting and Selection**