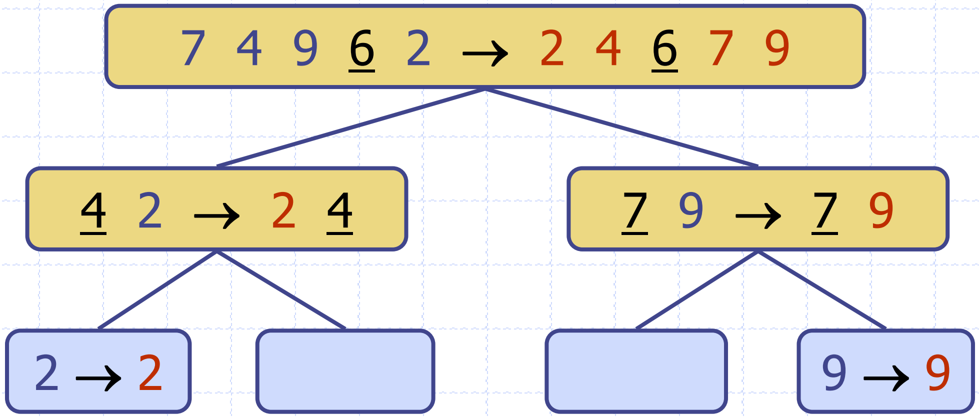
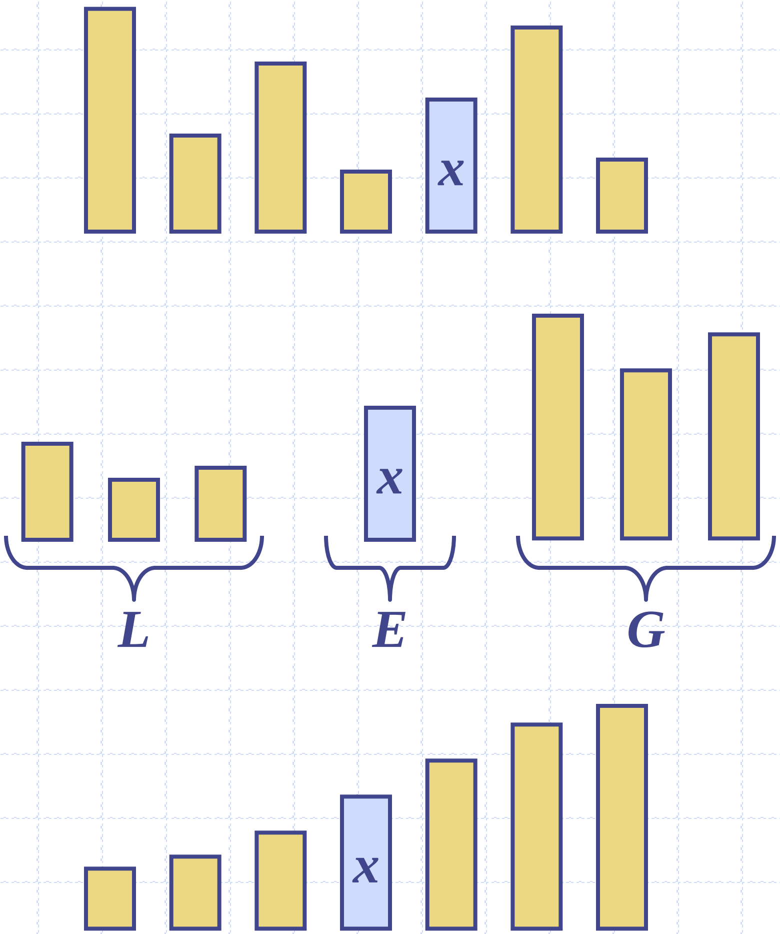
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Quick-Sort



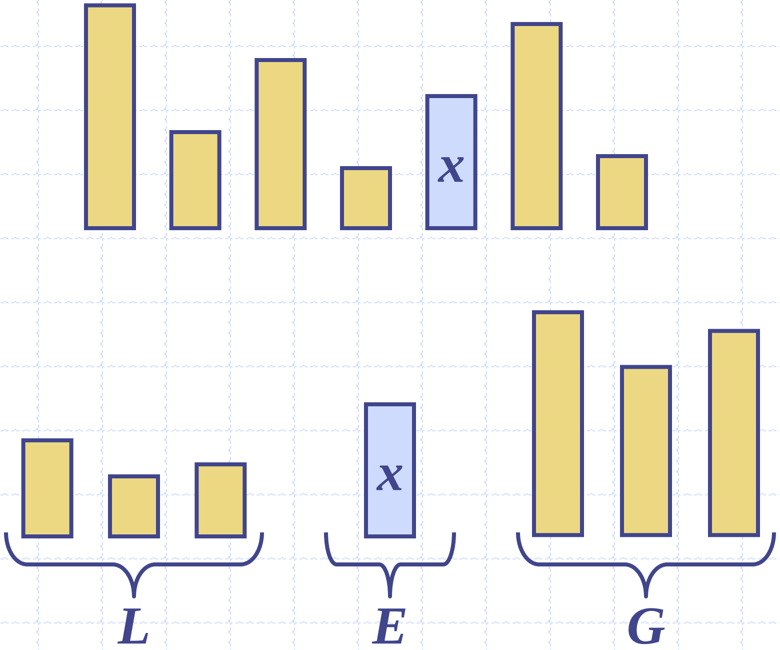
Quick-Sort

- 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



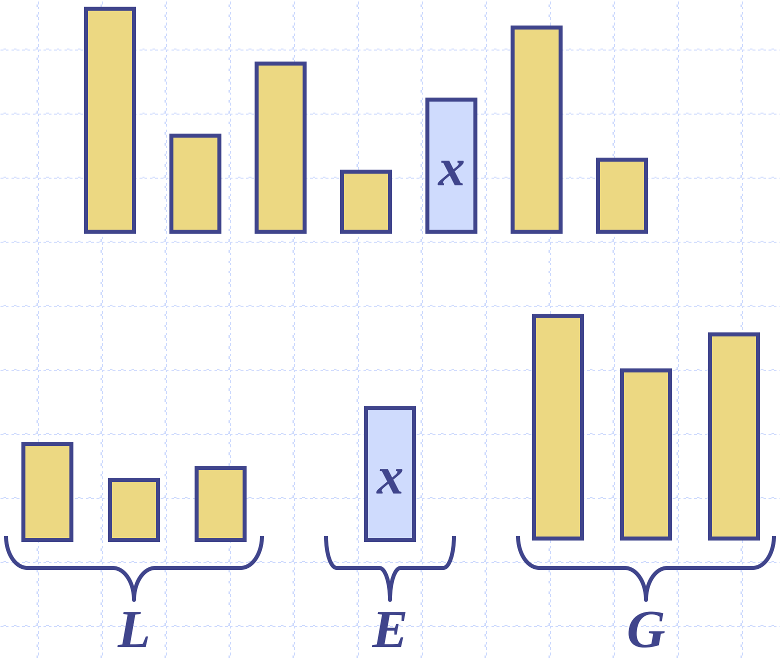
Importance of Partitioning

- After partitioning
  - What can you say about the position of the pivot?

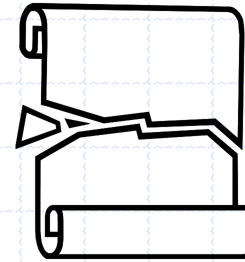


Importance of Partitioning

- After partitioning
  - What can you say about the position of the pivot?
    - The pivot is at the correct spot
  - Also, two smaller subproblems
    - Not including the pivot



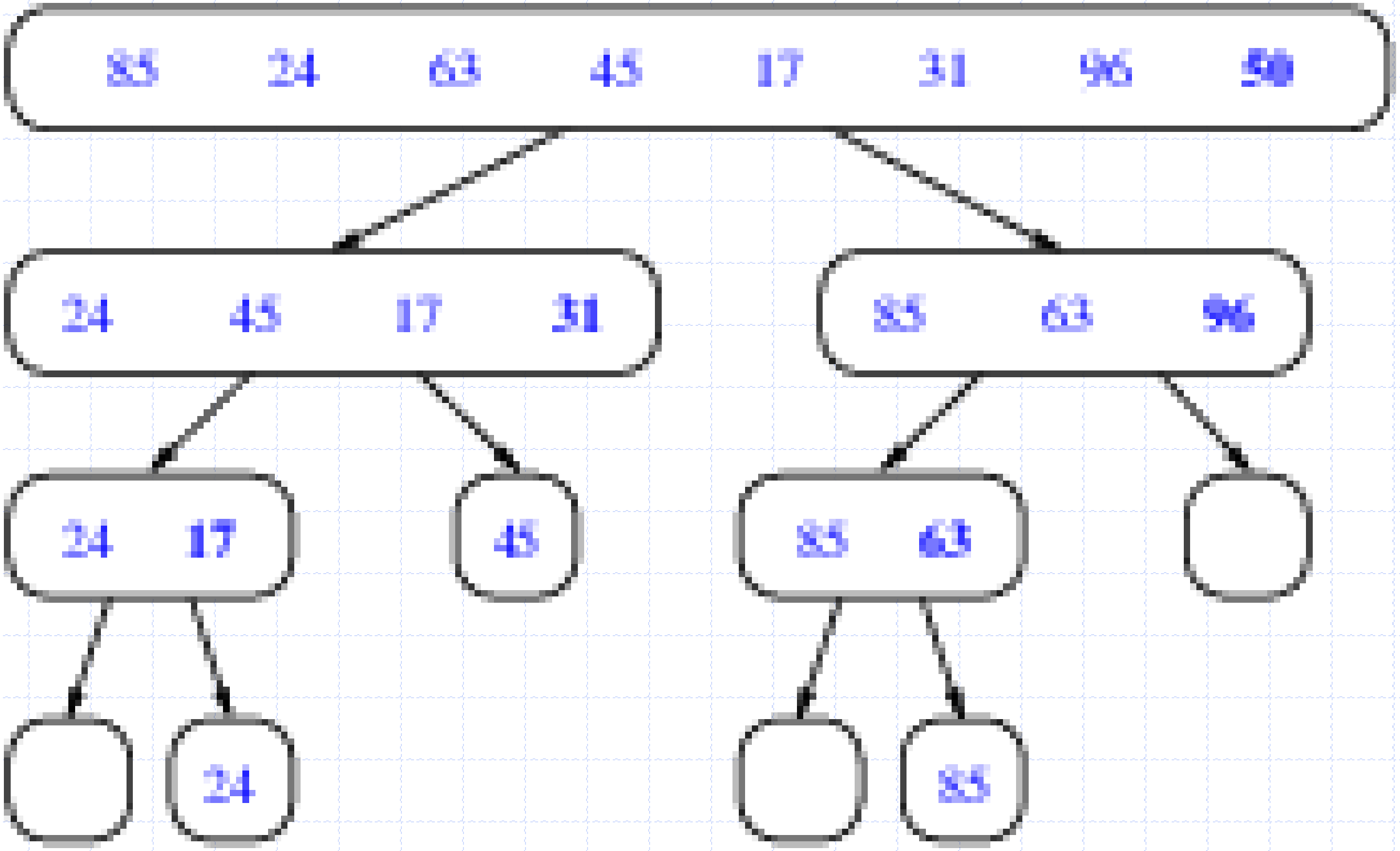
Partition



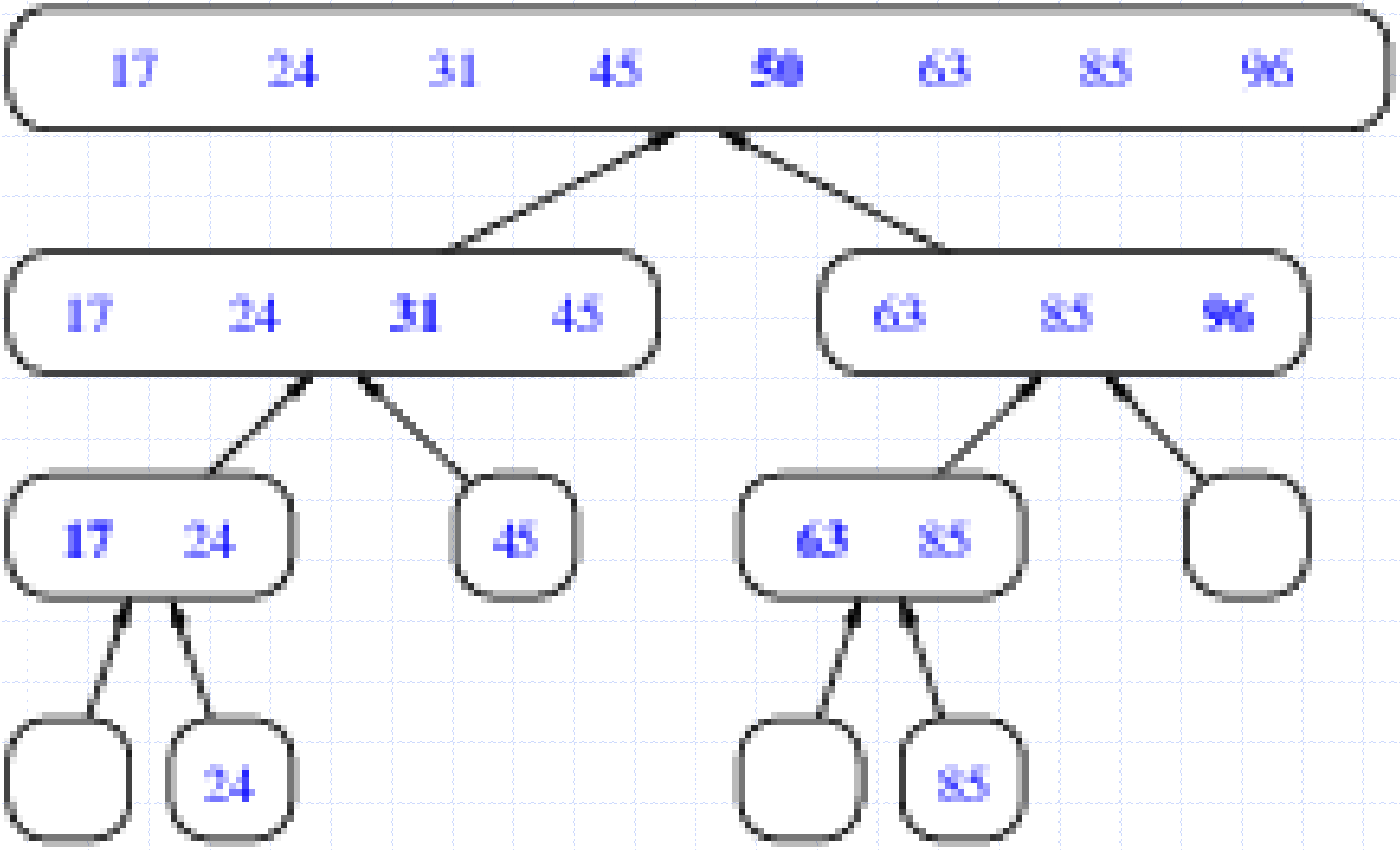
- partition an input sequence:
  - remove each element y from S and
  - insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- partition step of quick-sort takes $O(n)$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

Partition the list recursively



Merge the lists and the pivot



In-place Quick Sort

- O(1) extra space
- Same basic algorithm
  - Partition based on a pivot
  - Quick Sort on the two partitions
- Partitioning uses O(1) extra space
  - Left and right indices to scan for elements on the "wrong side":
    - Smaller elements that are on the right side
    - Larger element that are on the left side

left					right	pivot
34	67	87	23	98	43	56

left					right	pivot
34	67	87	23	98	43	56

	left				right	pivot
34	67	87	23	98	43	56

left					right	pivot
34	67	87	23	98	43	56

	left				right	pivot
34	67	87	23	98	43	56

	left				right	pivot
34	43	87	23	98	67	56

left					right	pivot
34	67	87	23	98	43	56

	left				right	pivot
34	67	87	23	98	43	56

	left				right	pivot
34	43	87	23	98	67	56

		left	right			pivot
34	43	87	23	98	67	56

left					right	pivot
34	67	87	23	98	43	56

	left				right	pivot
34	67	87	23	98	43	56

	left				right	pivot
34	43	87	23	98	67	56

		left	right			pivot
34	43	87	23	98	67	56

		left	right			pivot
34	43	23	87	98	67	56

left					right	pivot
34	67	87	23	98	43	56

	left				right	pivot
34	67	87	23	98	43	56

	left				right	pivot
34	43	87	23	98	67	56

		left	right			pivot
34	43	87	23	98	67	56

		left	right			pivot
34	43	23	87	98	67	56

		right	Left			pivot
34	43	23	87	98	67	56

left					right	pivot
34	67	87	23	98	43	56

	left				right	pivot
34	67	87	23	98	43	56

	left				right	pivot
34	43	87	23	98	67	56

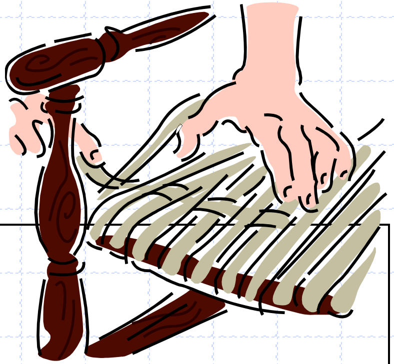
		left	right			pivot
34	43	87	23	98	67	56

		left	right			pivot
34	43	23	87	98	67	56

		right	left			pivot
34	43	23	87	98	67	56

		right	left			pivot
34	43	23	56	98	67	87

In-Place Quick-Sort



Algorithm ***inPlaceQuickSort*(*S, start, end*)**

Input sequence ***S***, ***start*** and ***end*** indices

Output sequence ***S*** sorted between ***start*** and ***end***

```
if start ≥ end  return
left ← start
right ← end – 1          // before pivot
pivot ← S[end]           // pivot is the last element
while left <= right      // still have elements
    while (left <= right & S[left] < pivot) // find element larger than pivot
        left++
    while (left <= right & S[right] >pivot) // find element smaller than pivot
        right--
    if (left <= right)              // put the two elements in the correct partitions
        swap S[left] and S[right]; left++; right—
Swap S[end] and S[left]         // put pivot at the correct spot
inPlaceQuickSort(S, start, left – 1)
inPlaceQuickSort(S, left + 1, end)
```

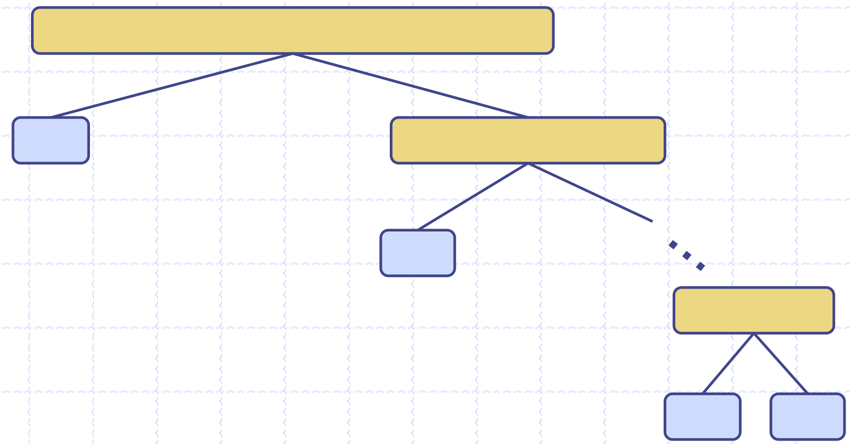
Worst-case Time Complexity

- 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



Expected Time Complexity

- O(n log n)
- Proof in the book
  - And skipped slides at the end

Selection of Pivots

- Last element (or first element)
  - If the list is partially sorted
    - might be the smallest/largest element
      - the worst-case scenario
- Ideas?

Selection of Pivots

- Last element (or first element)
  - If the list is partially sorted
    - might be the smallest/largest element
      - the worst-case scenario
- Random element
  - But calling random() has time overhead
- Median-of-three
  - Median of first, last, and middle elements

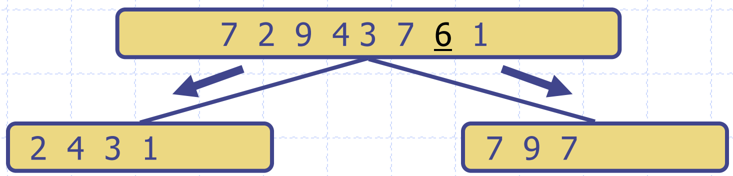
Summary of Sorting Algorithms

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)

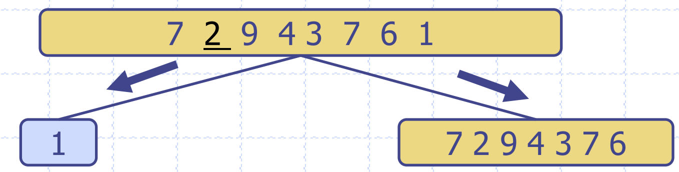
Skipping the rest

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 $3s/4$

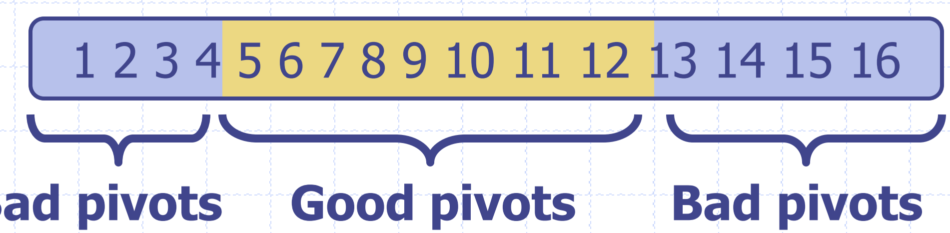


Good call



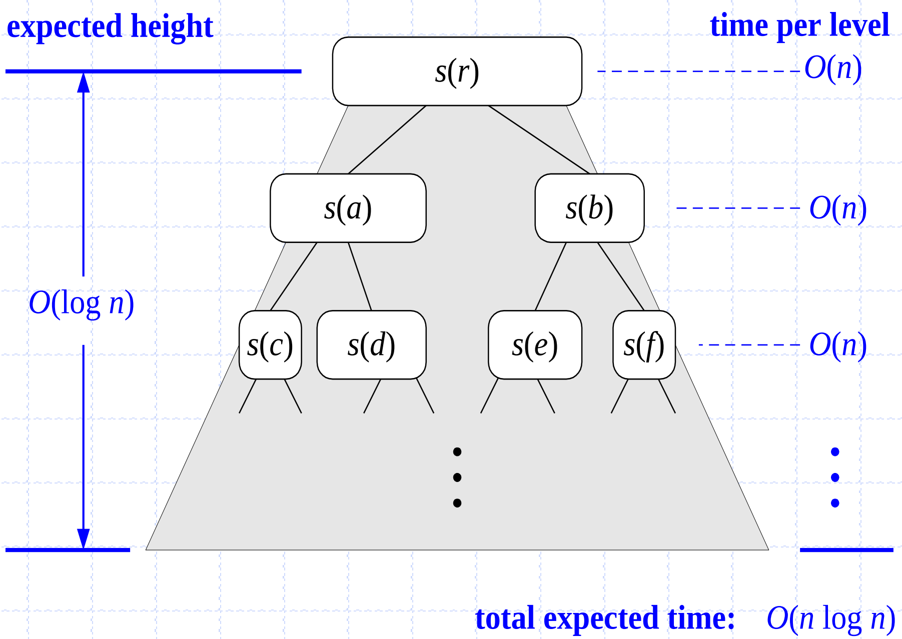
Bad call

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



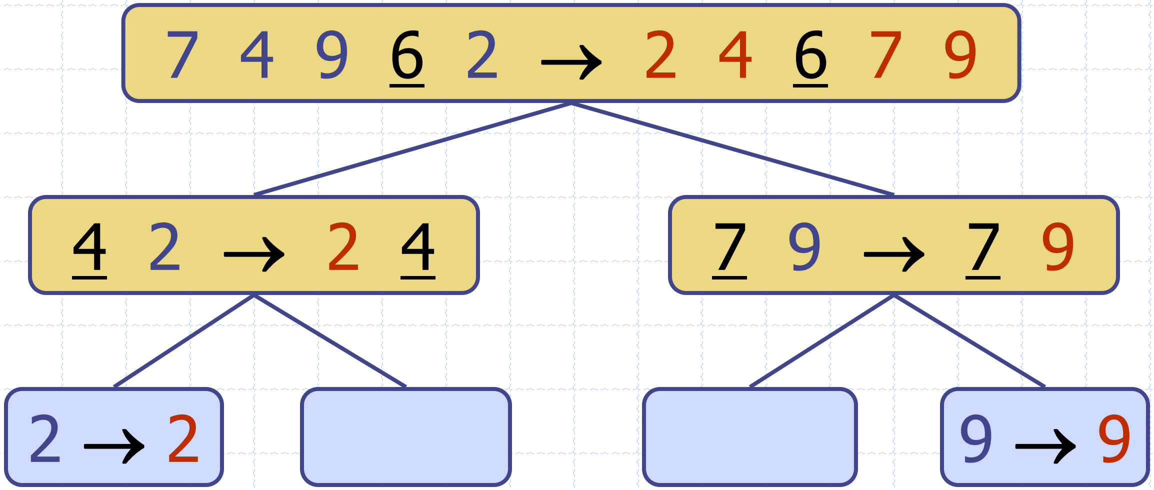
Expected Running Time, Part 2

- 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
  - The expected height of the quick-sort tree is $O(\log n)$
- The amount or work done at the nodes of the same depth is $O(n)$
- Thus, the expected running time of quick-sort is $O(n \log n)$



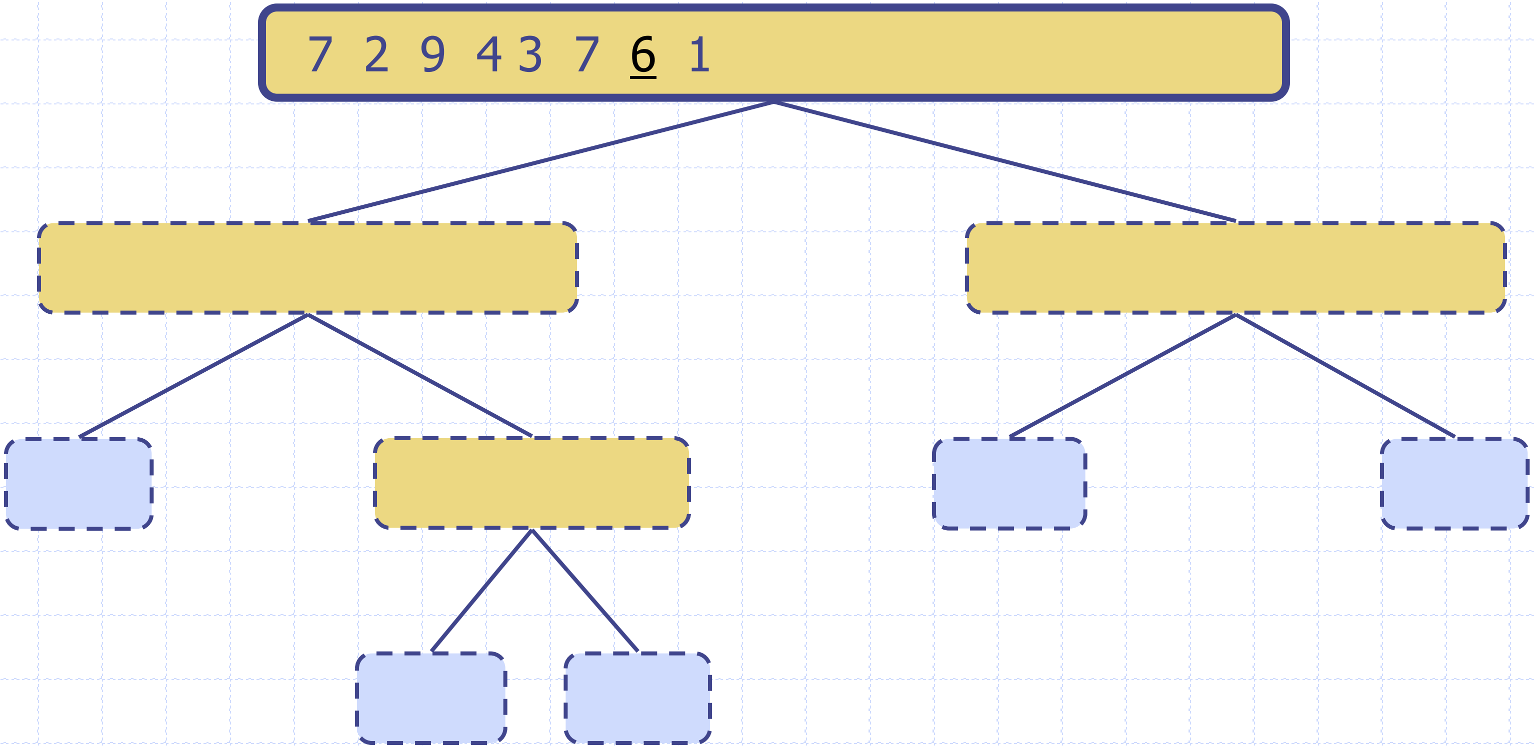
Quick-Sort Tree

- An execution 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



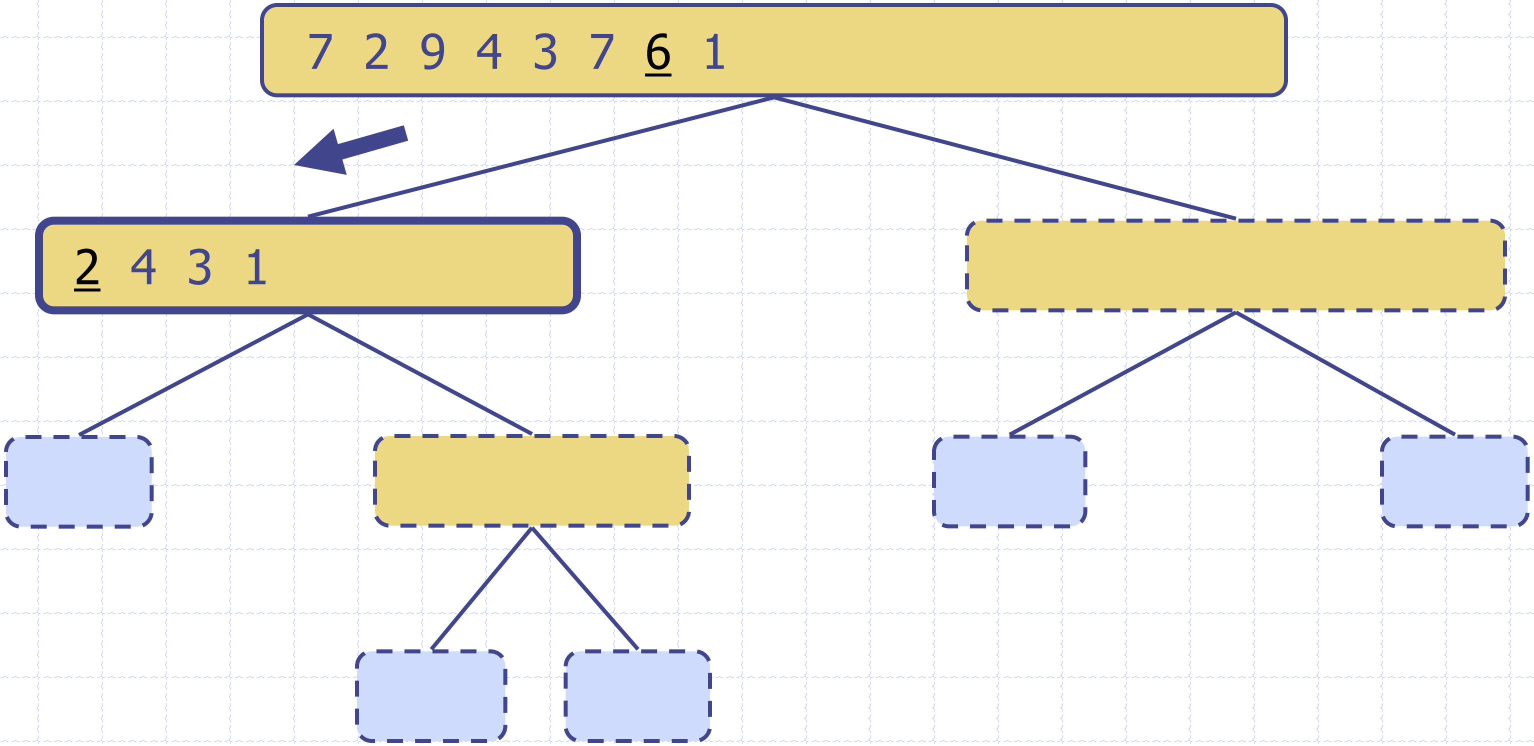
Execution Example

- Pivot selection



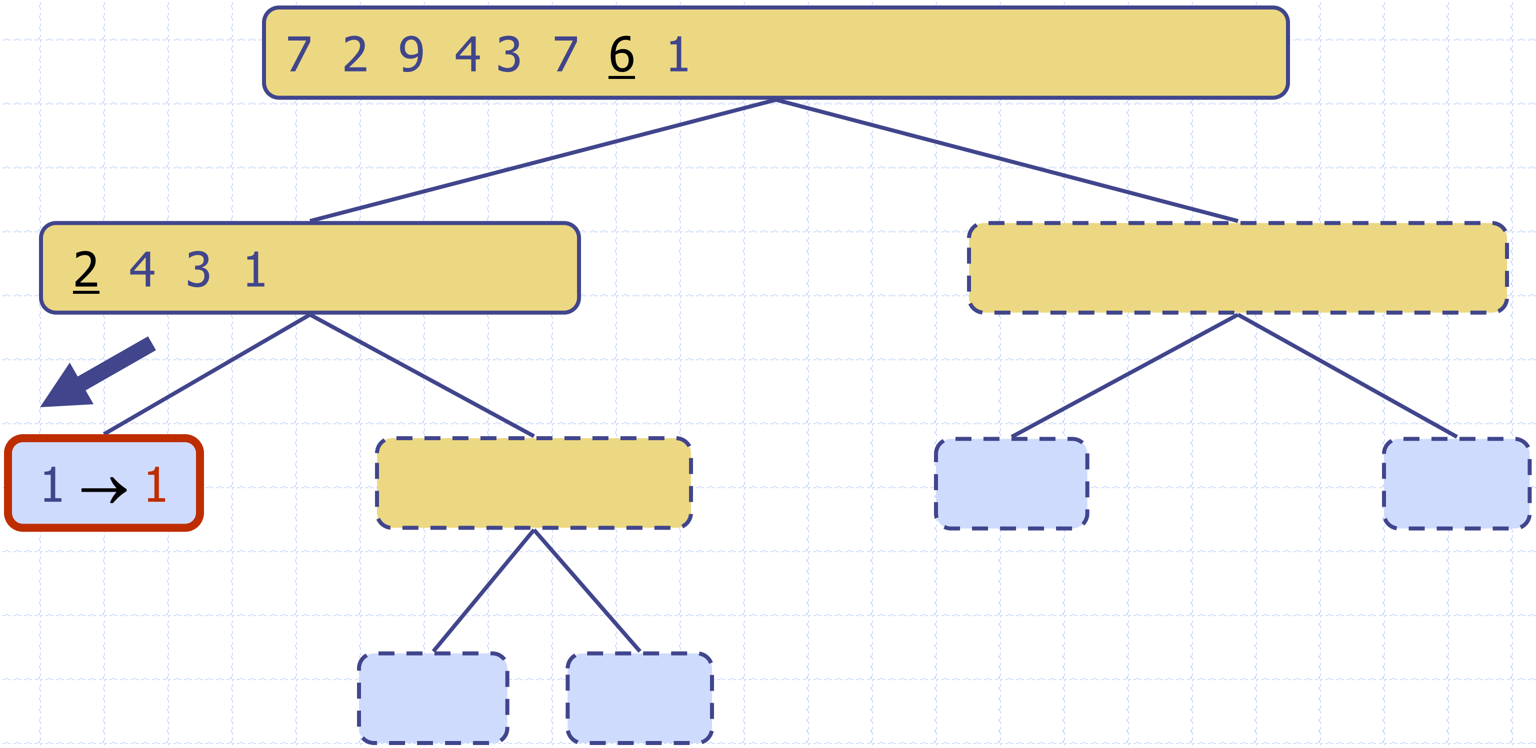
Execution Example (cont.)

- Partition, recursive call, pivot selection



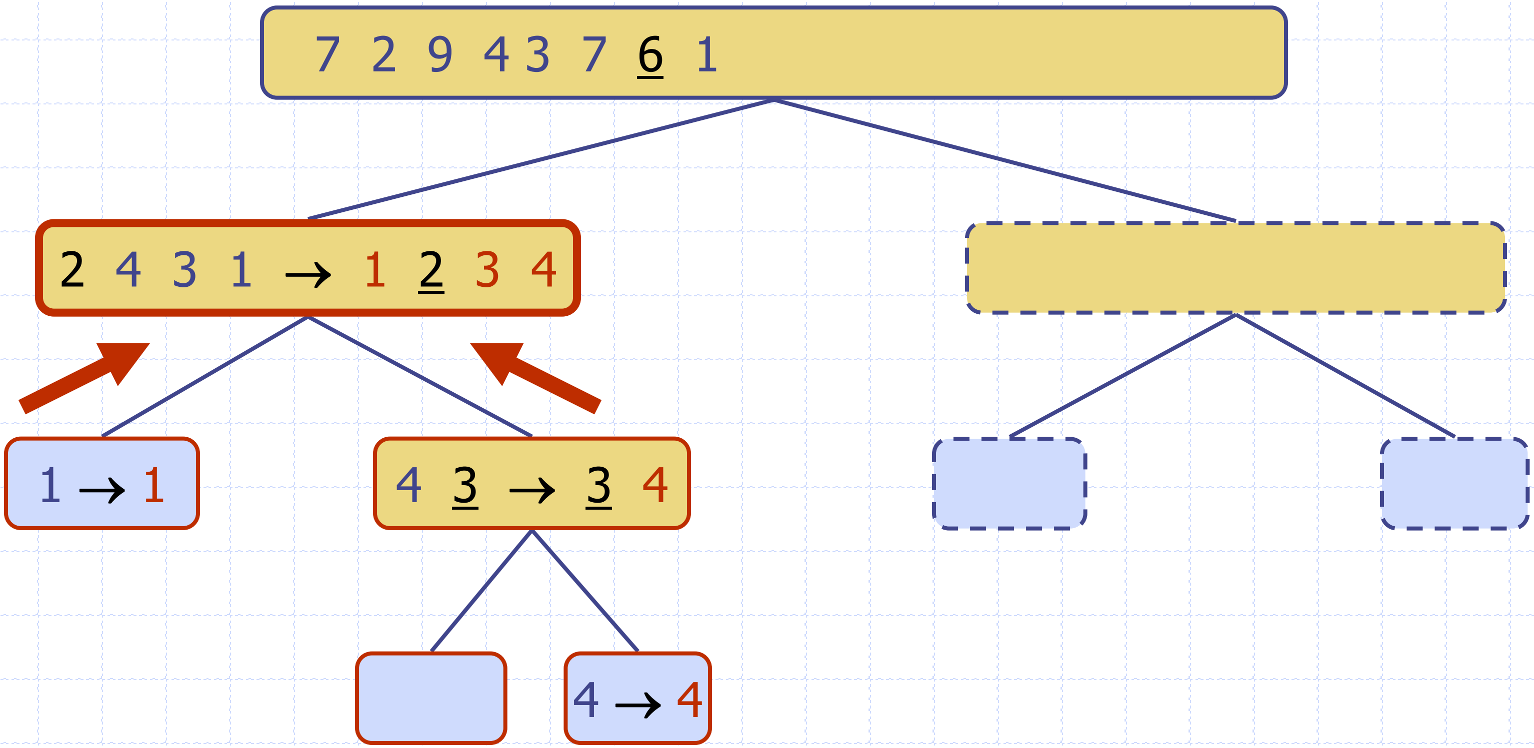
Execution Example (cont.)

- Partition, recursive call, base case



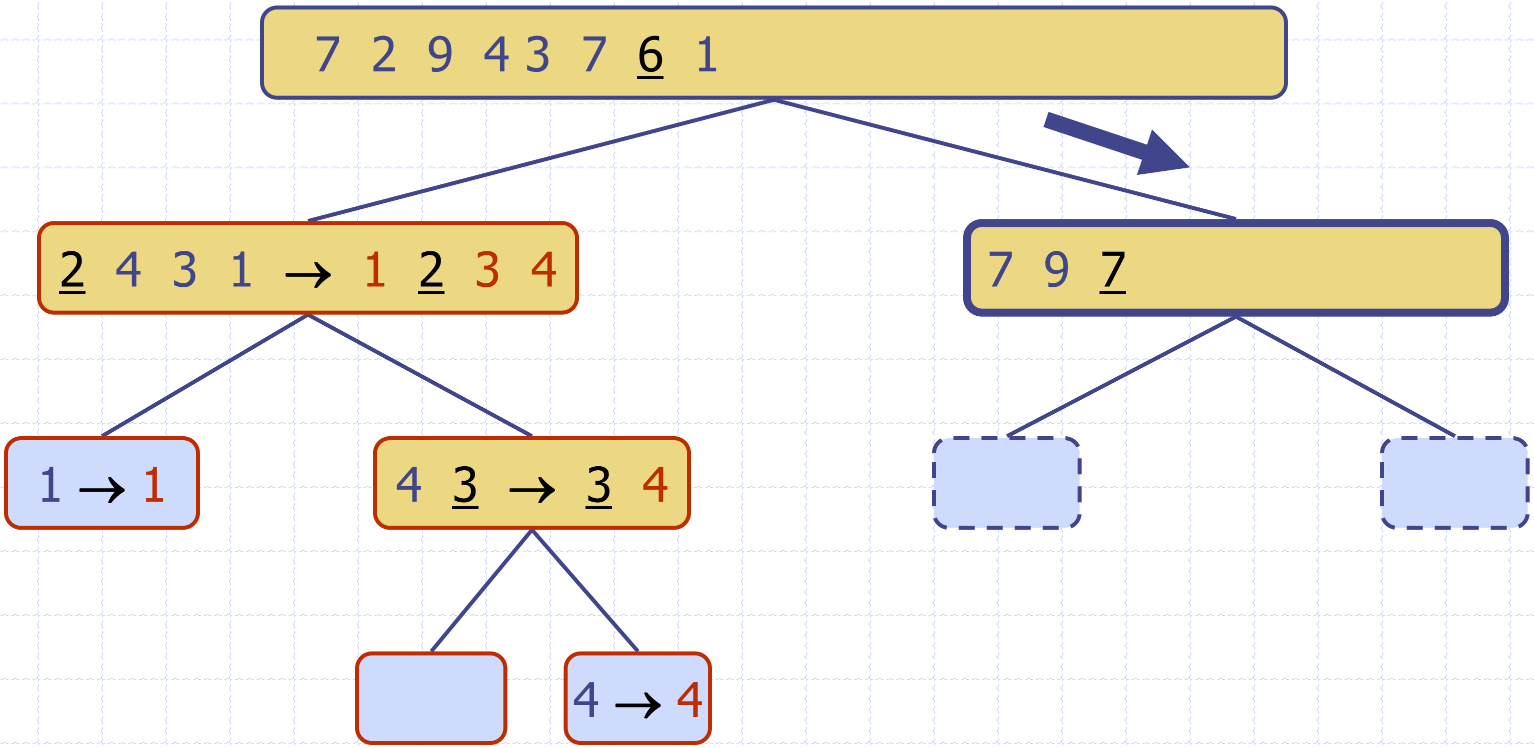
Execution Example (cont.)

- Recursive call, …, base case, join



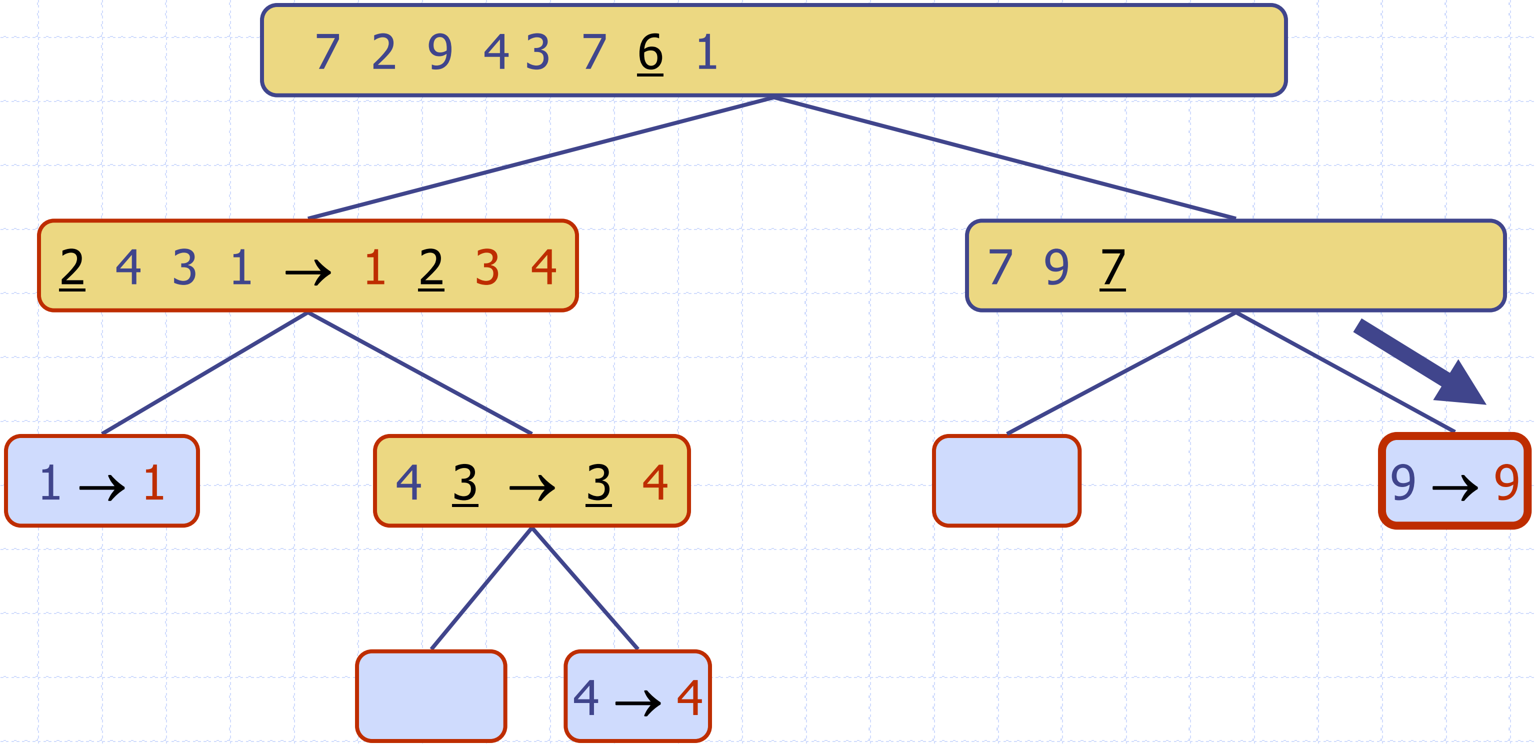
Execution Example (cont.)

- Recursive call, pivot selection



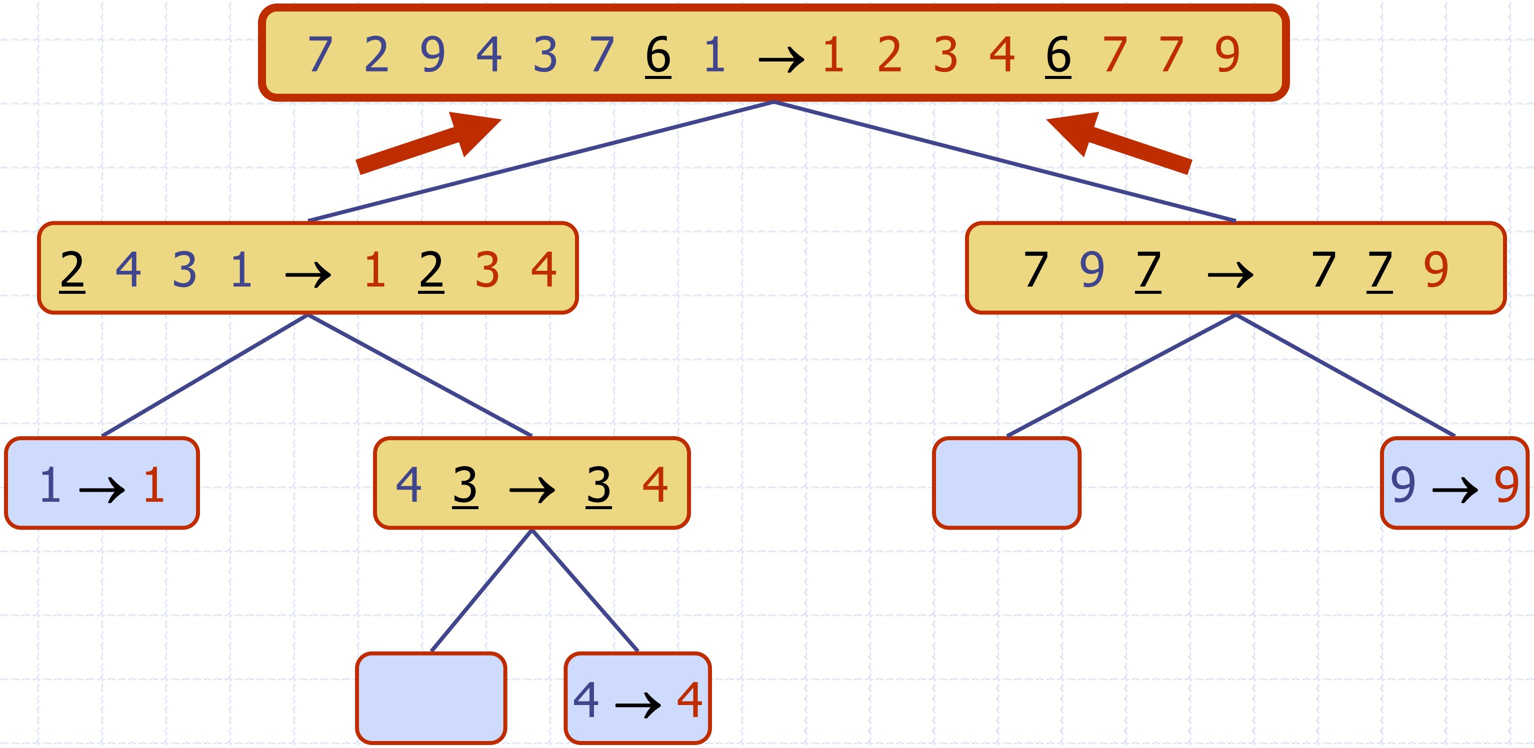
Execution Example (cont.)

- Partition, …, recursive call, base case

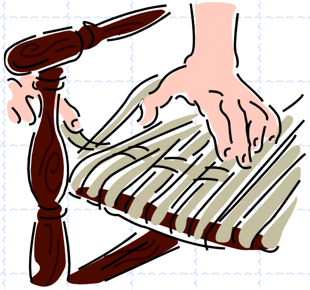


Execution Example (cont.)

- Join, join



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