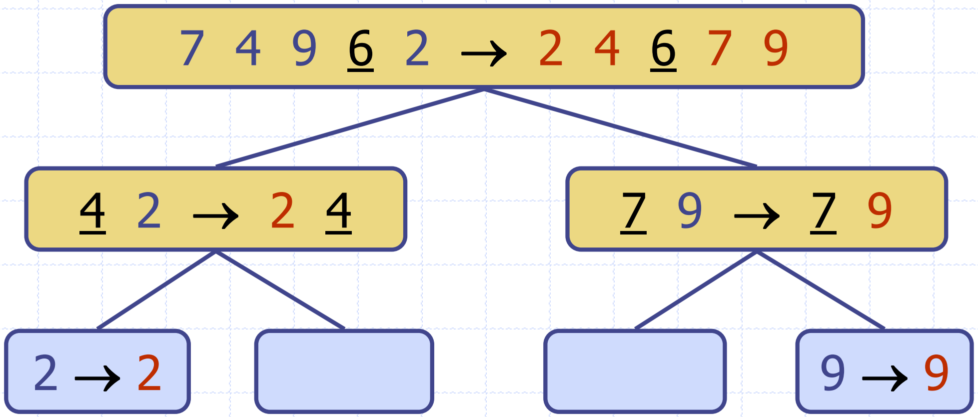
Quick-Sort

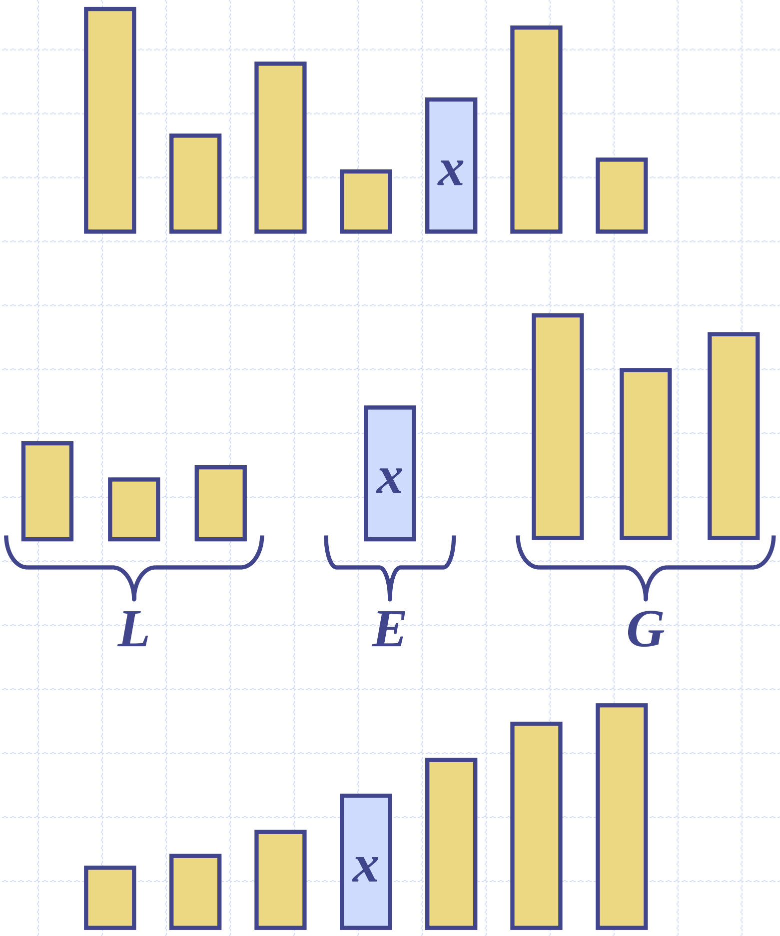


Outline and Reading

- Quick-sort
  - Algorithm
  - Partition step
  - Quick-sort tree
  - Execution example
- Analysis of quick-sort
- In-place quick-sort
- Summary of sorting algorithms

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



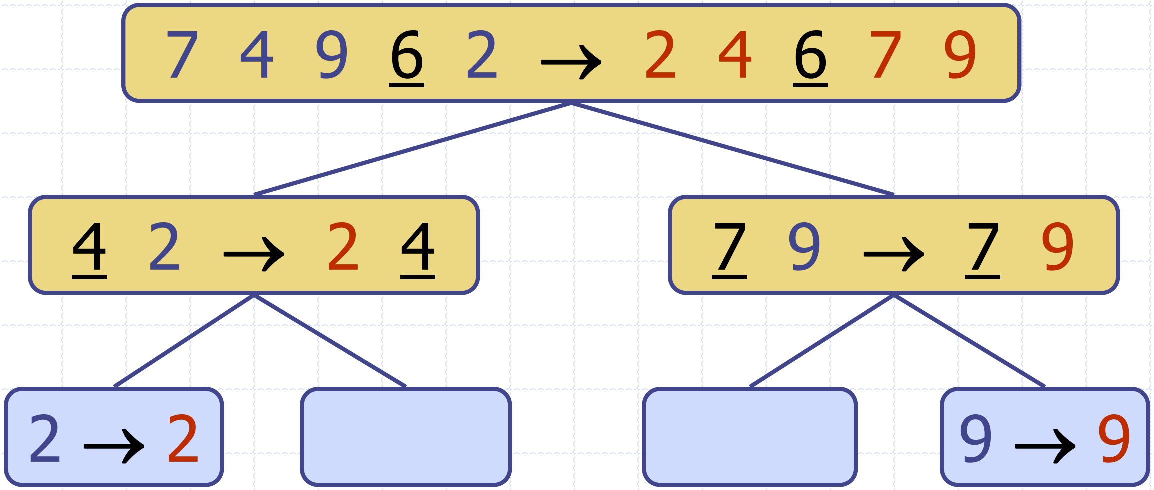
Partition

- 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.insertLast(y)
        else if y = x
            E.insertLast(y)
        else { y > x }
            G.insertLast(y)
    return L, E, G
```

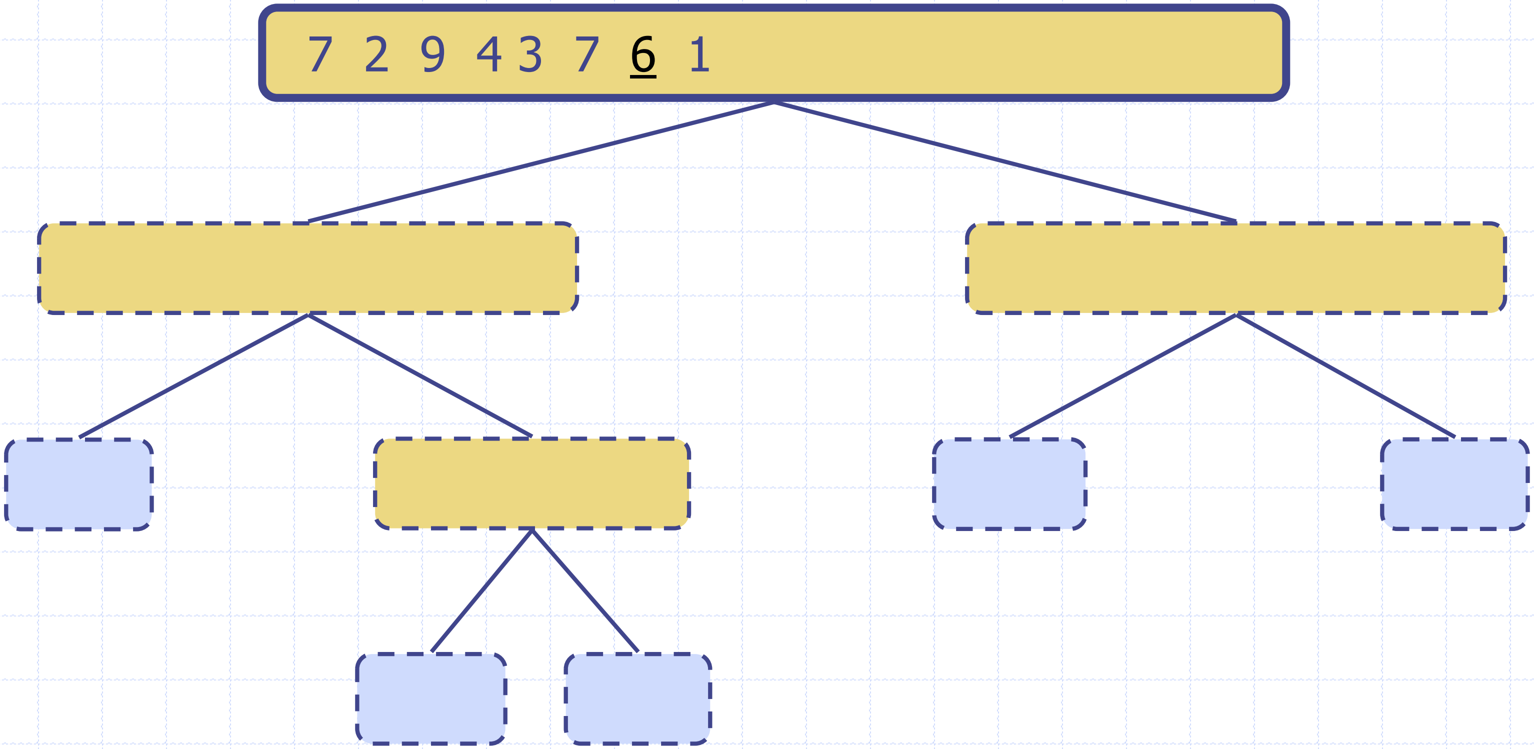
Quick-Sort Tree

- 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



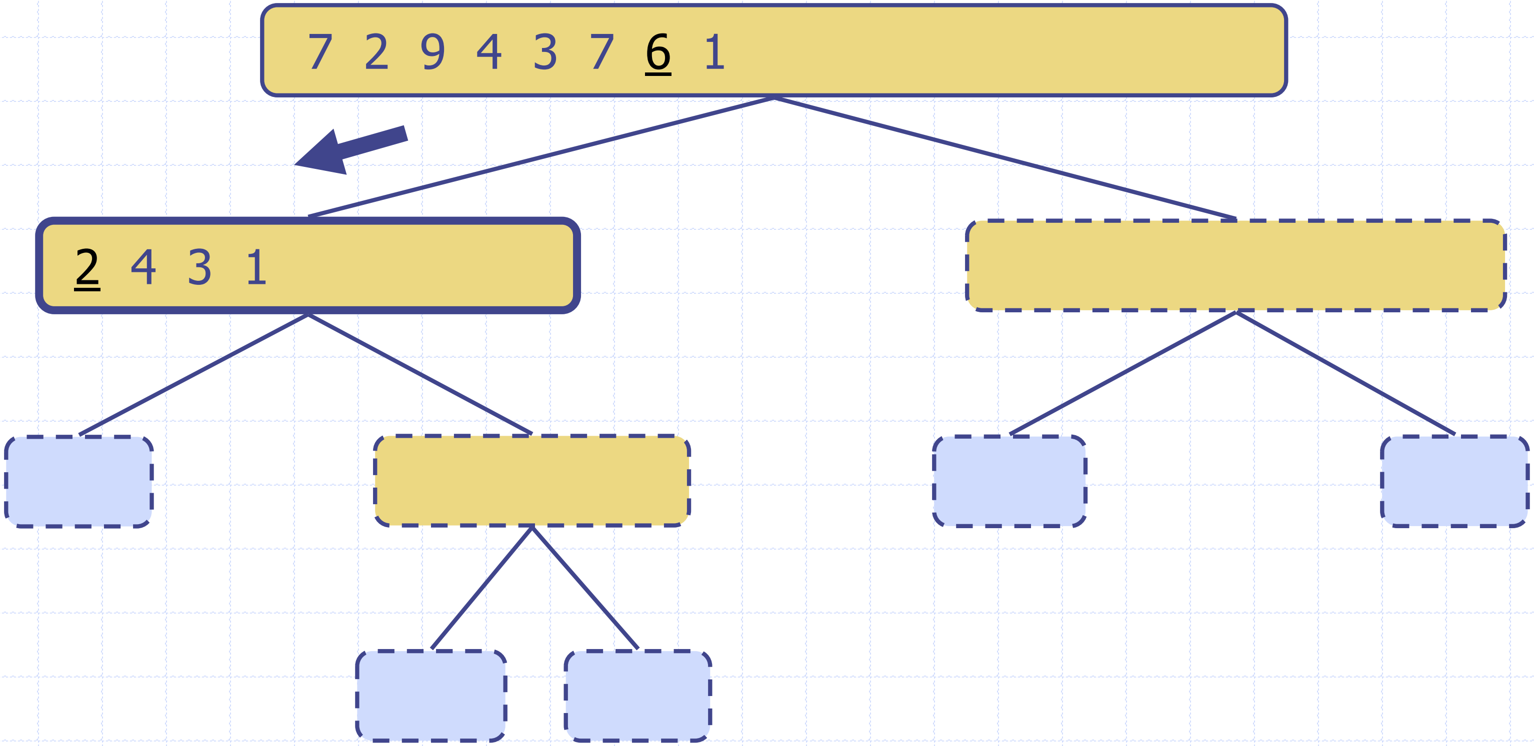
Execution Example

- Pivot selection



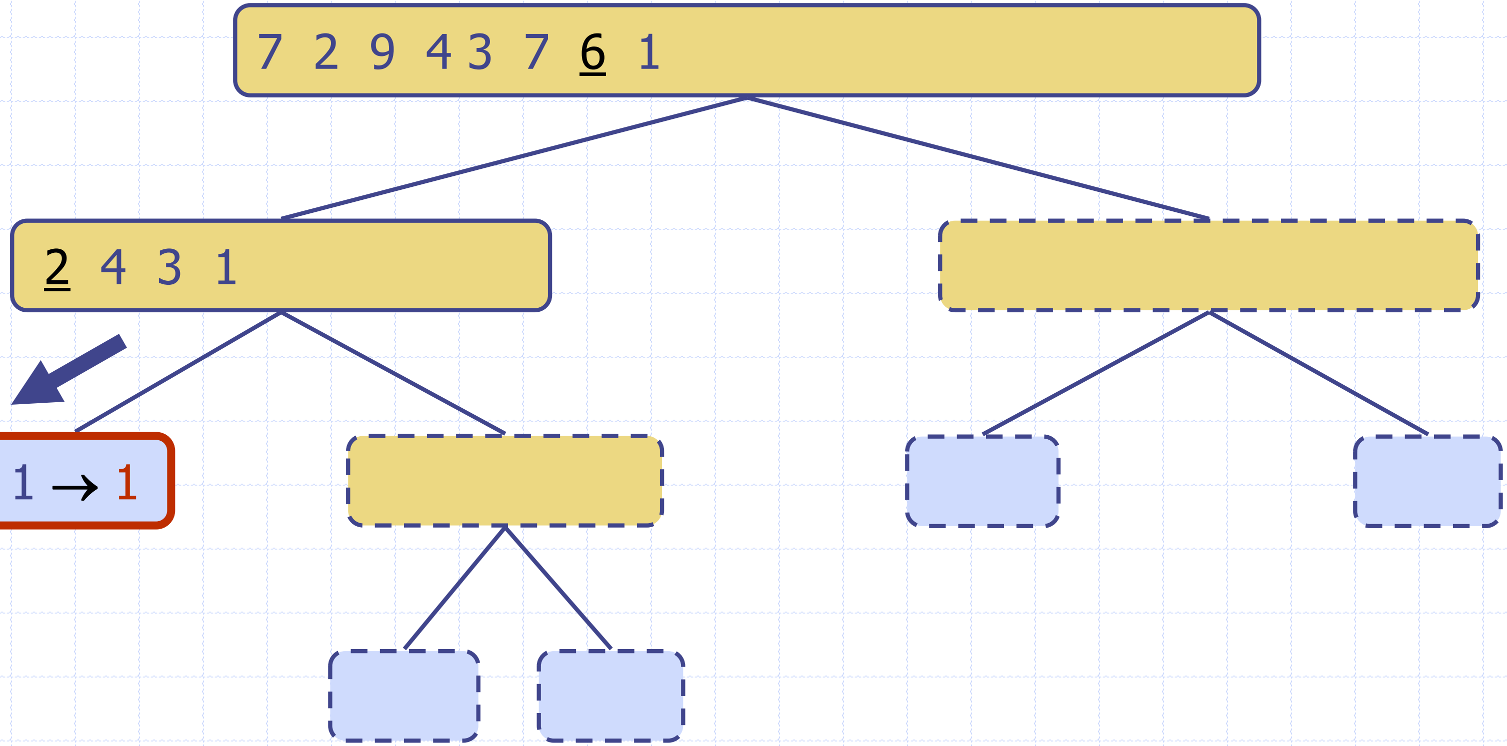
Execution Example (cont.)

- Partition, recursive call, pivot selection



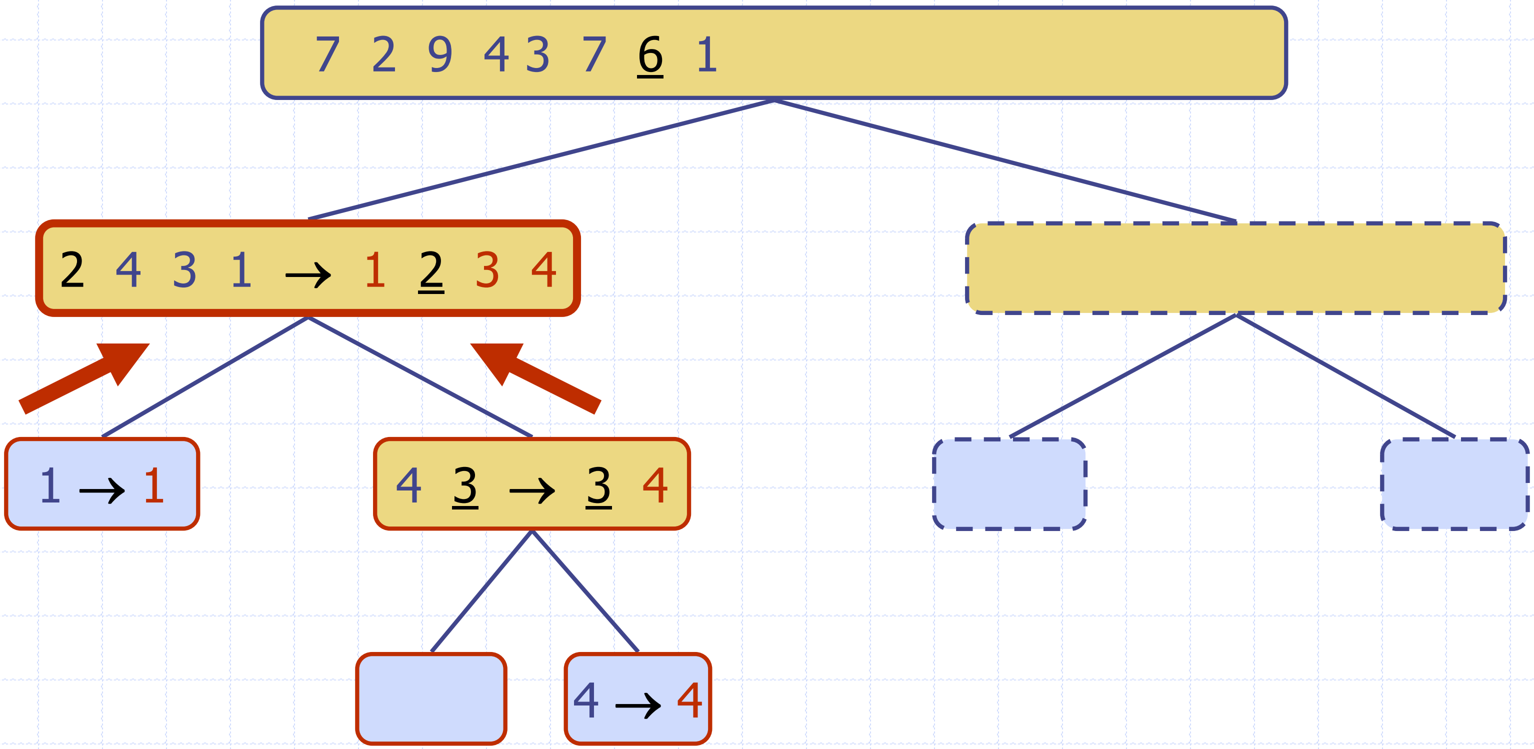
Execution Example (cont.)

- Partition, recursive call, base case



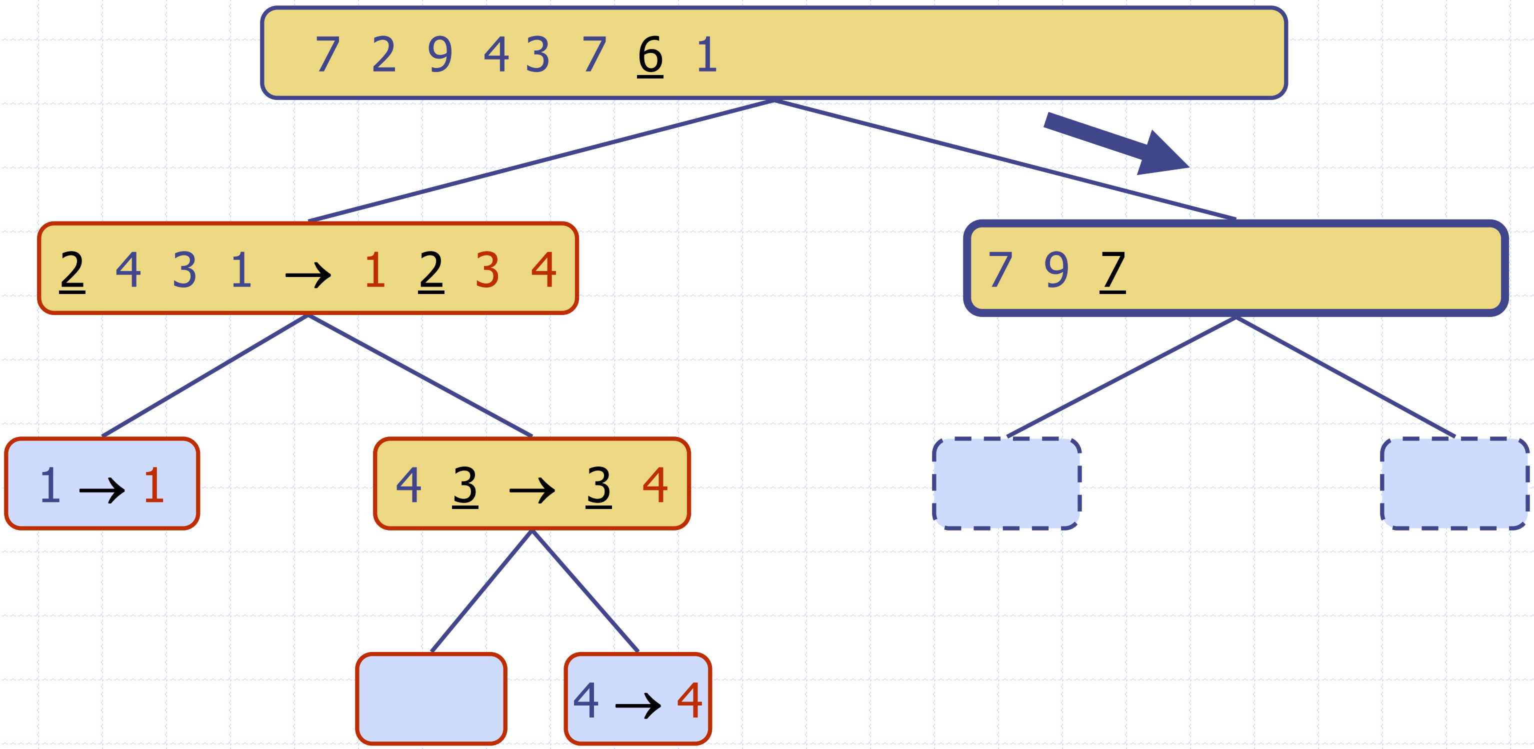
Execution Example (cont.)

- Recursive call, …, base case, join



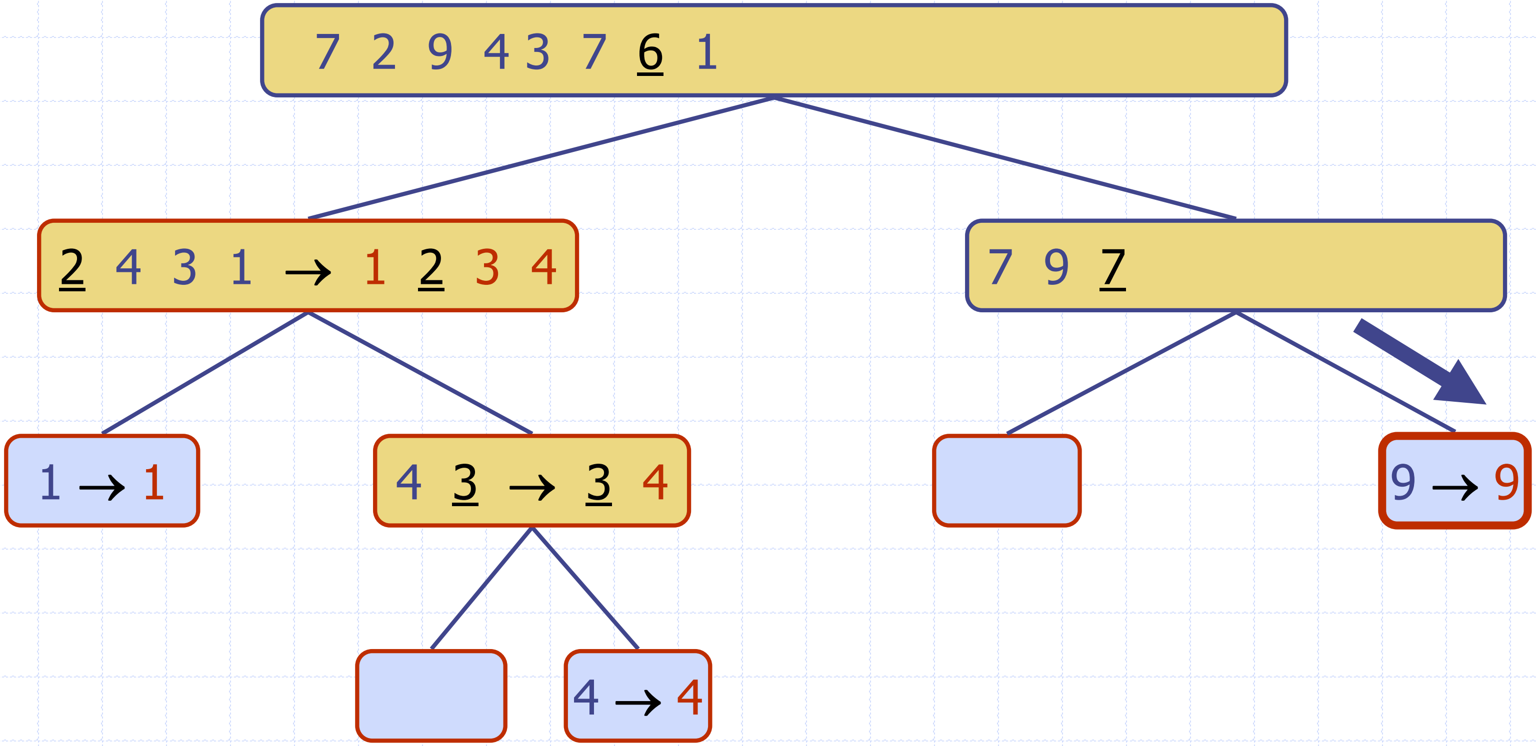
Execution Example (cont.)

- Recursive call, pivot selection



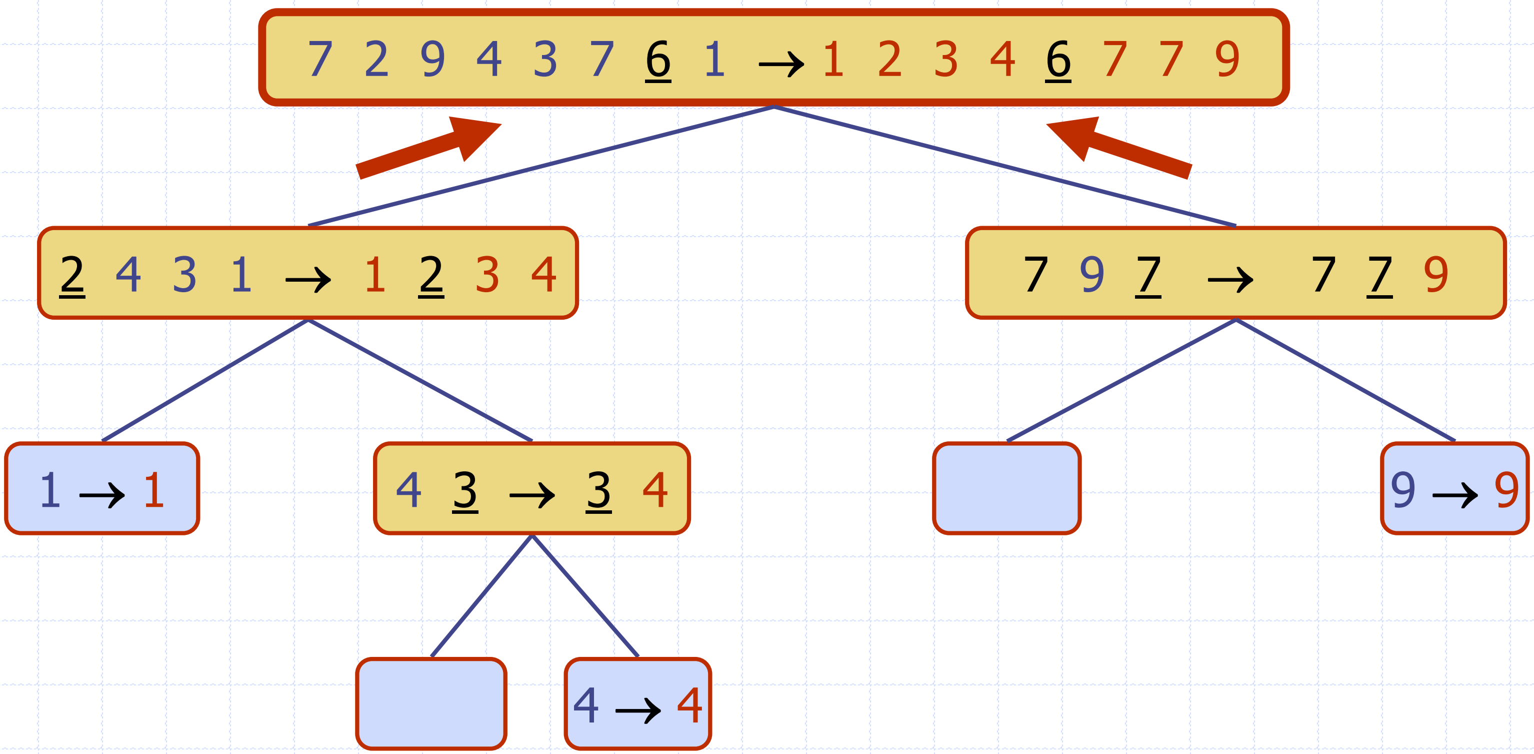
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



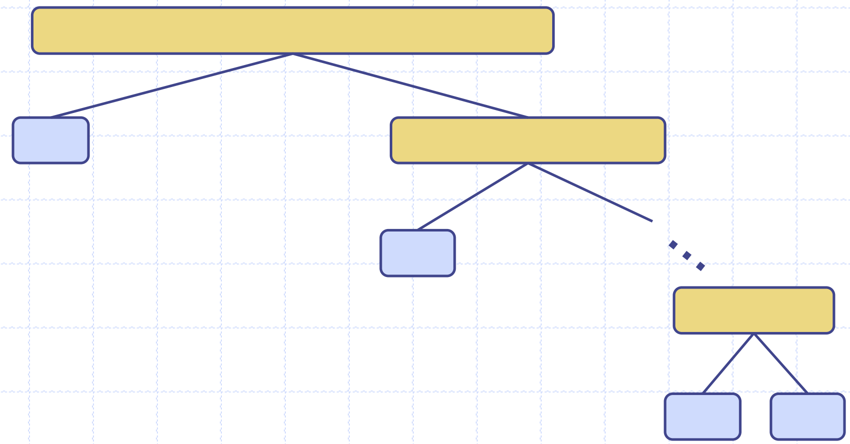
Worst-case Running Time

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

$$\boldsymbol{n}+(\boldsymbol{n}-1)+\ldots+2+1$$

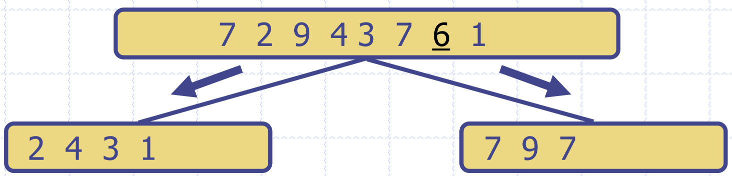
- Thus, the worst-case running time of quick-sort is $\boldsymbol{O}(\boldsymbol{n}^2)$

depth	time
0	$\boldsymbol{n}$
1	$\boldsymbol{n}-1$
…	…
$\boldsymbol{n}-1$	1

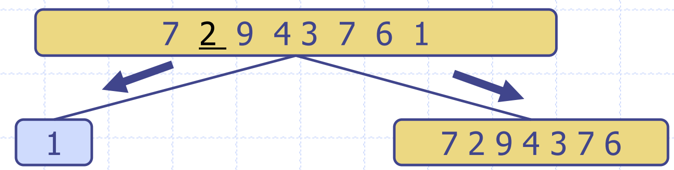


Expected Running Time

- 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

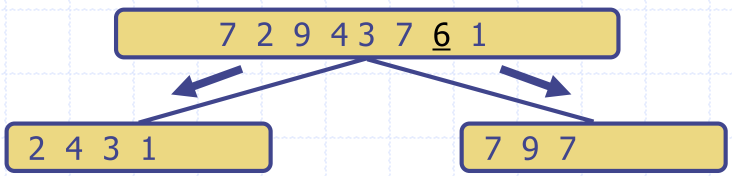


Bad call

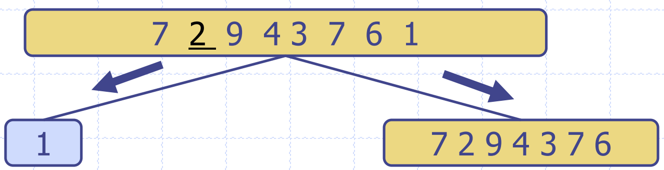
- A call is good with probability ?????

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 $3s/4$

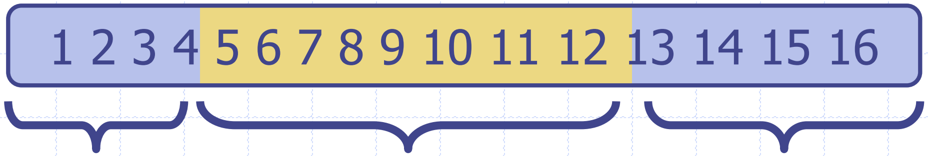


Good call



Bad call

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



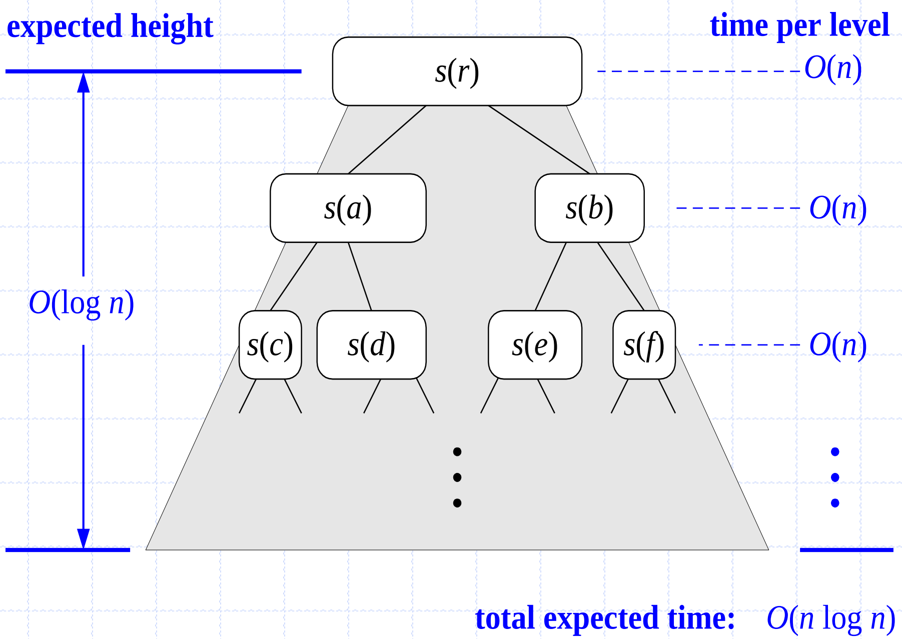
Bad pivots **Good pivots** **Bad pivots**

Expected Running Time, Part 2

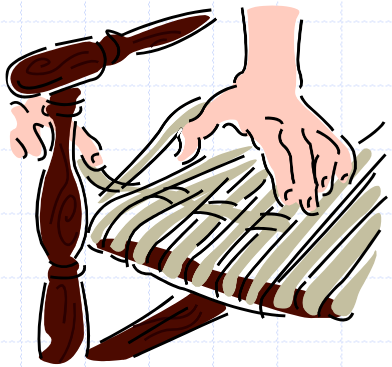
- For a node of depth $\boldsymbol{i}$,

  The size of the input sequence for the current call is at most $(3/4)^{i} \cdot n$

- **Probabilistic Fact:** The expected number of coin tosses required in order to get $\boldsymbol{k}$ heads is $2\boldsymbol{k}$
- Therefore, we have
  - The expected height of the quick-sort tree is $\boldsymbol{O}(\log \boldsymbol{n})$
- The amount or work done at the nodes of the same depth is $\boldsymbol{O}(\boldsymbol{n})$
- Thus, the expected running time of quick-sort is $\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$



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***
  $x \leftarrow S.elemAtRank(i)$
  $(h, k) \leftarrow inPlacePartition(x)$
  $inPlaceQuickSort(S, l, h-1)$
  $inPlaceQuickSort(S, k+1, r)$

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)