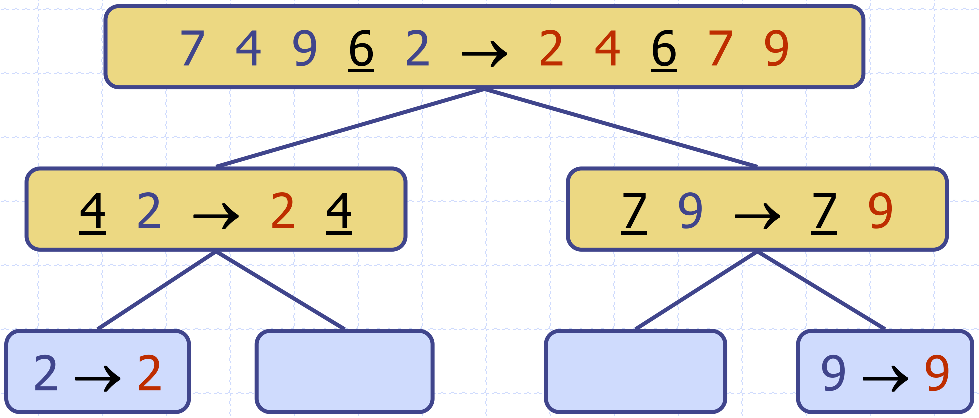
# Quick-Sort

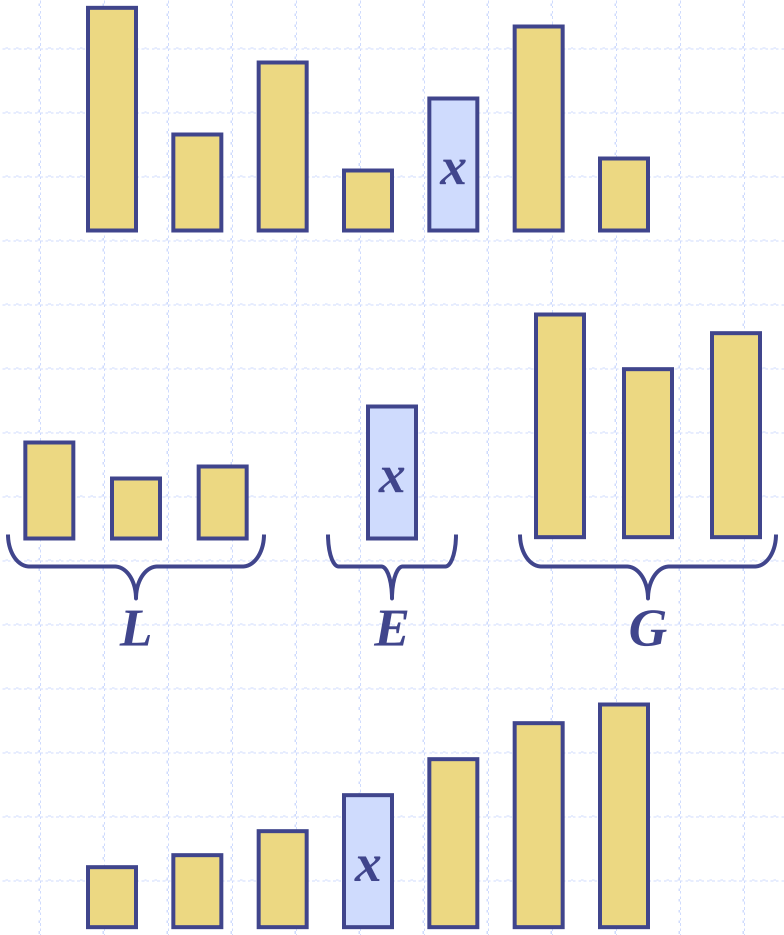


## Outline and Reading

- 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 $G$



## Isn’t that Merge-Sort?

- Quick-Sort is similar to Merge-Sort but with several key differences – details later

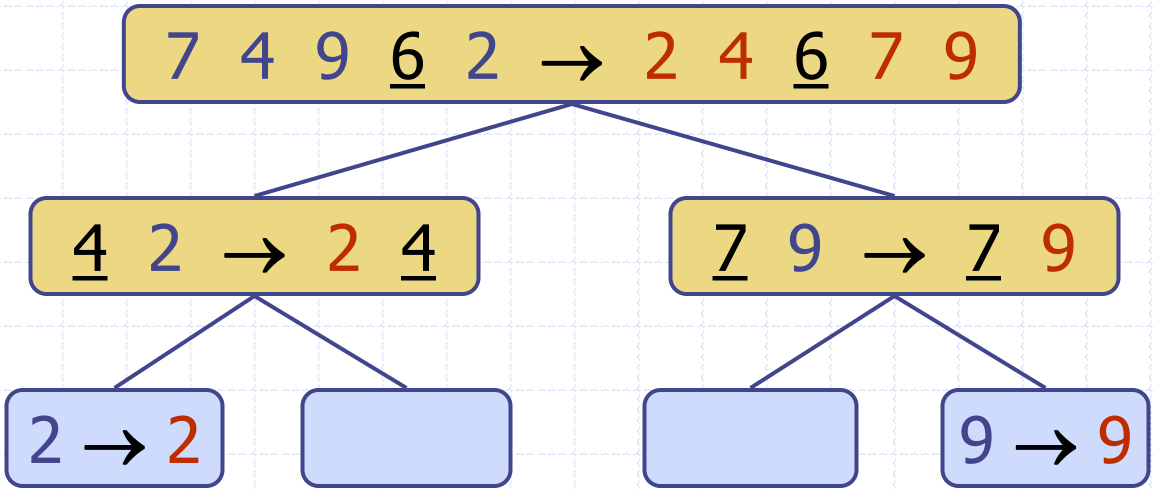
## Partition

- 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.
    $L, E, G \leftarrow$ empty sequences
    $x \leftarrow S.remove(p)$
    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 $L, E, G$
```

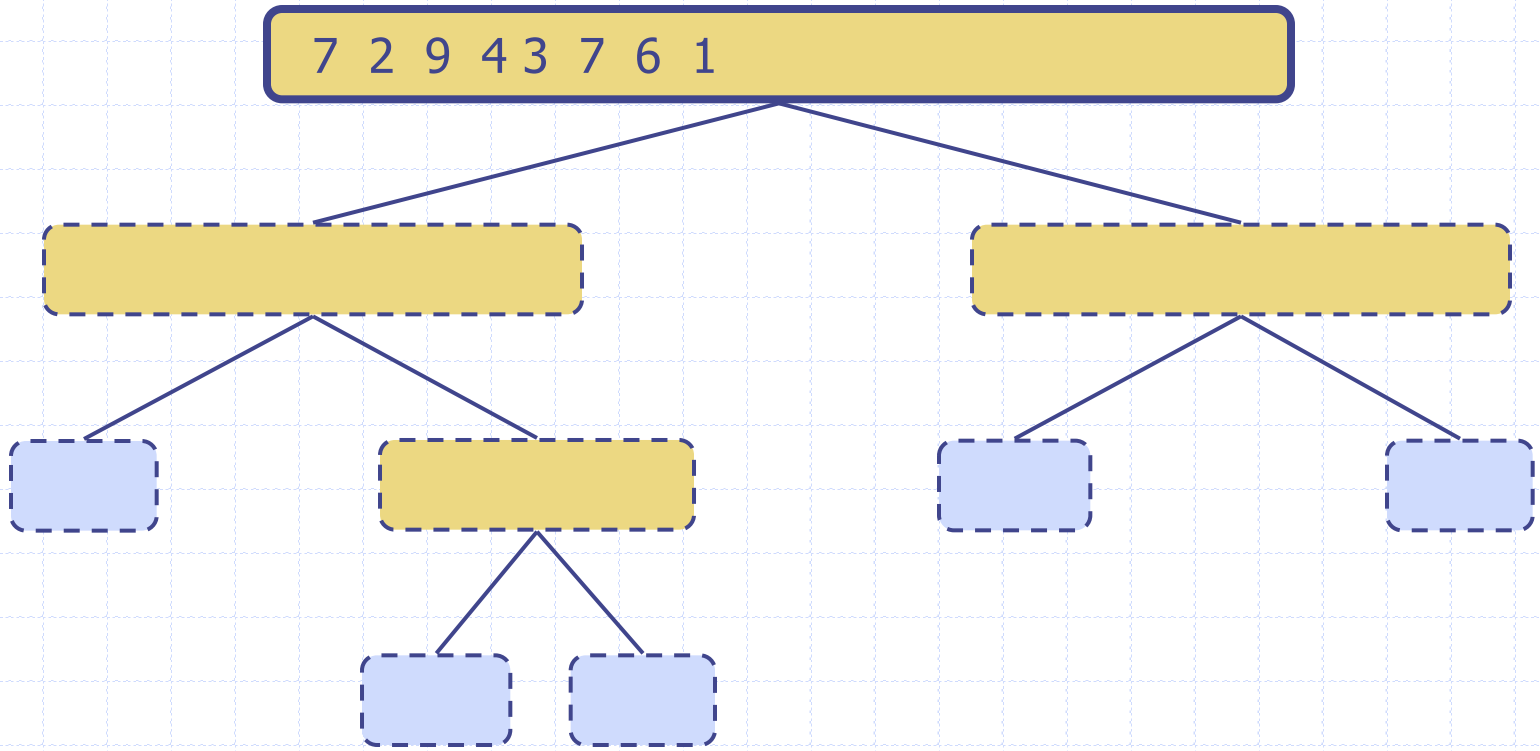
# Quick-Sort Tree

- 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



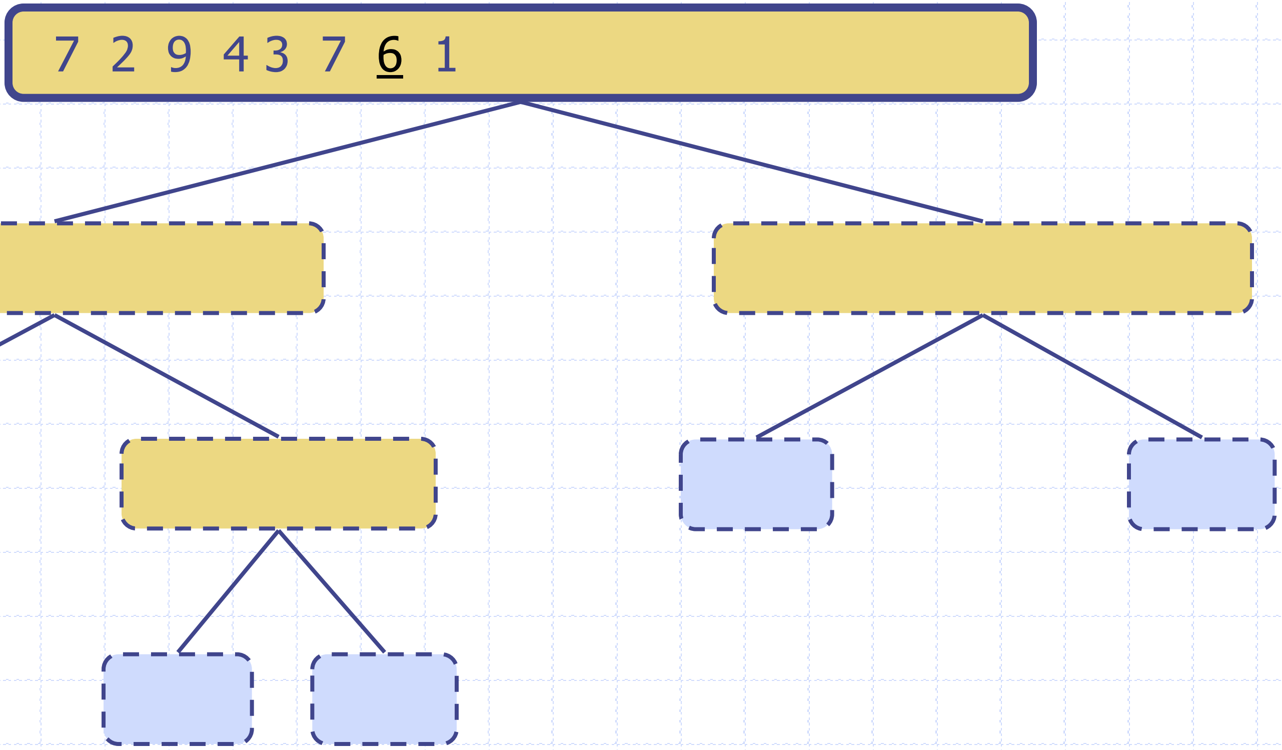
# Execution Example

- Pivot selection



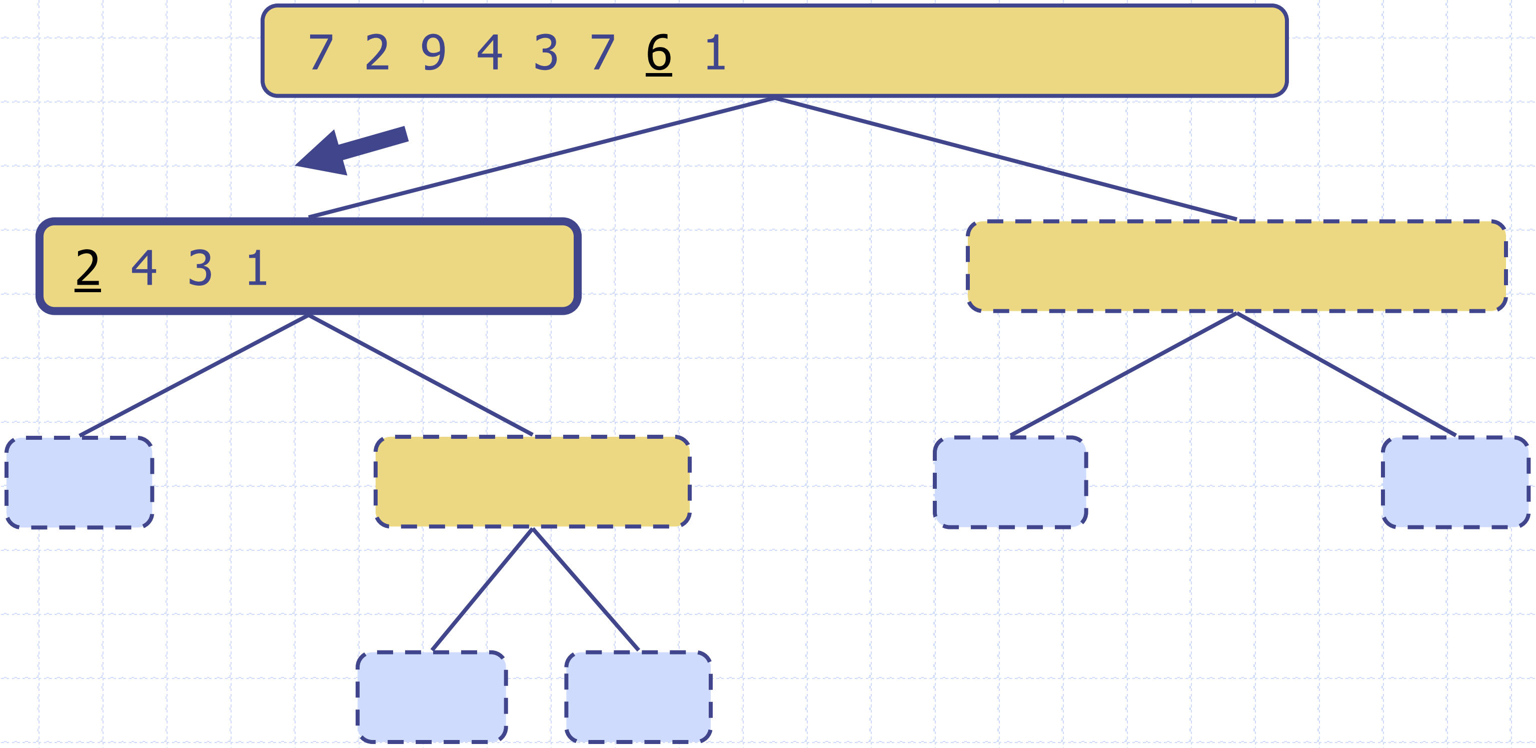
# Execution Example

- Pivot selection, e.g. "6"



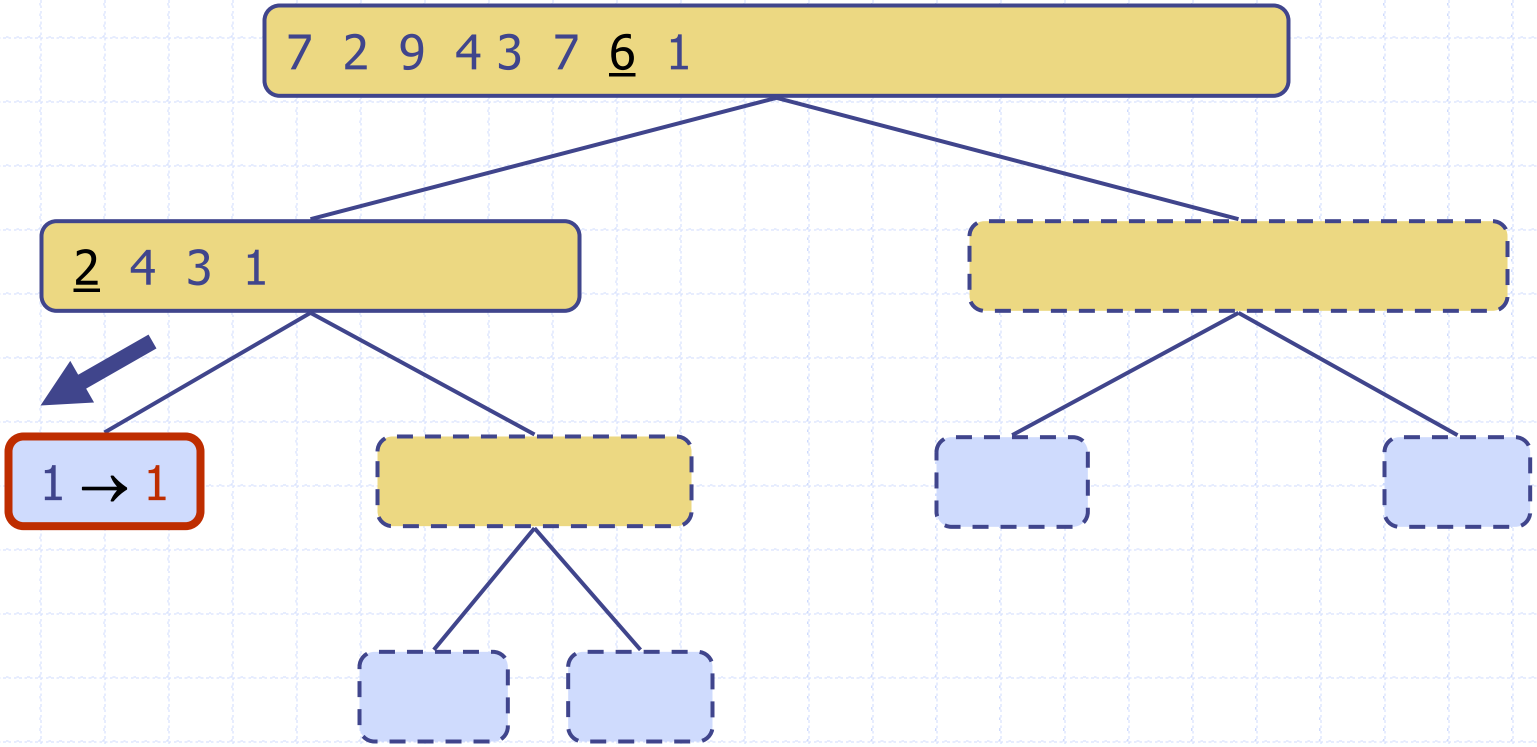
## Execution Example (cont.)

- Partition, recursive call, pivot selection



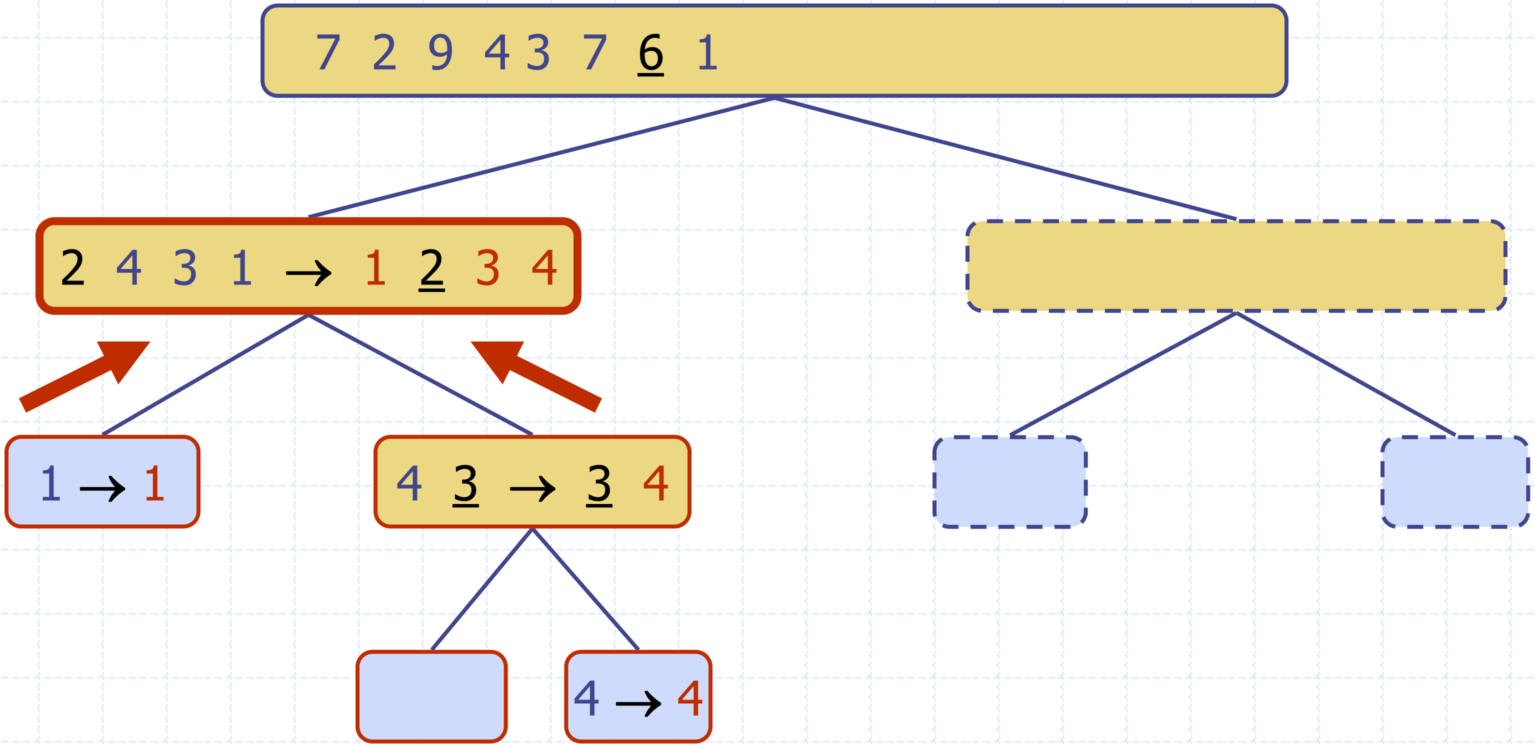
# Execution Example (cont.)

- Partition, recursive call, base case



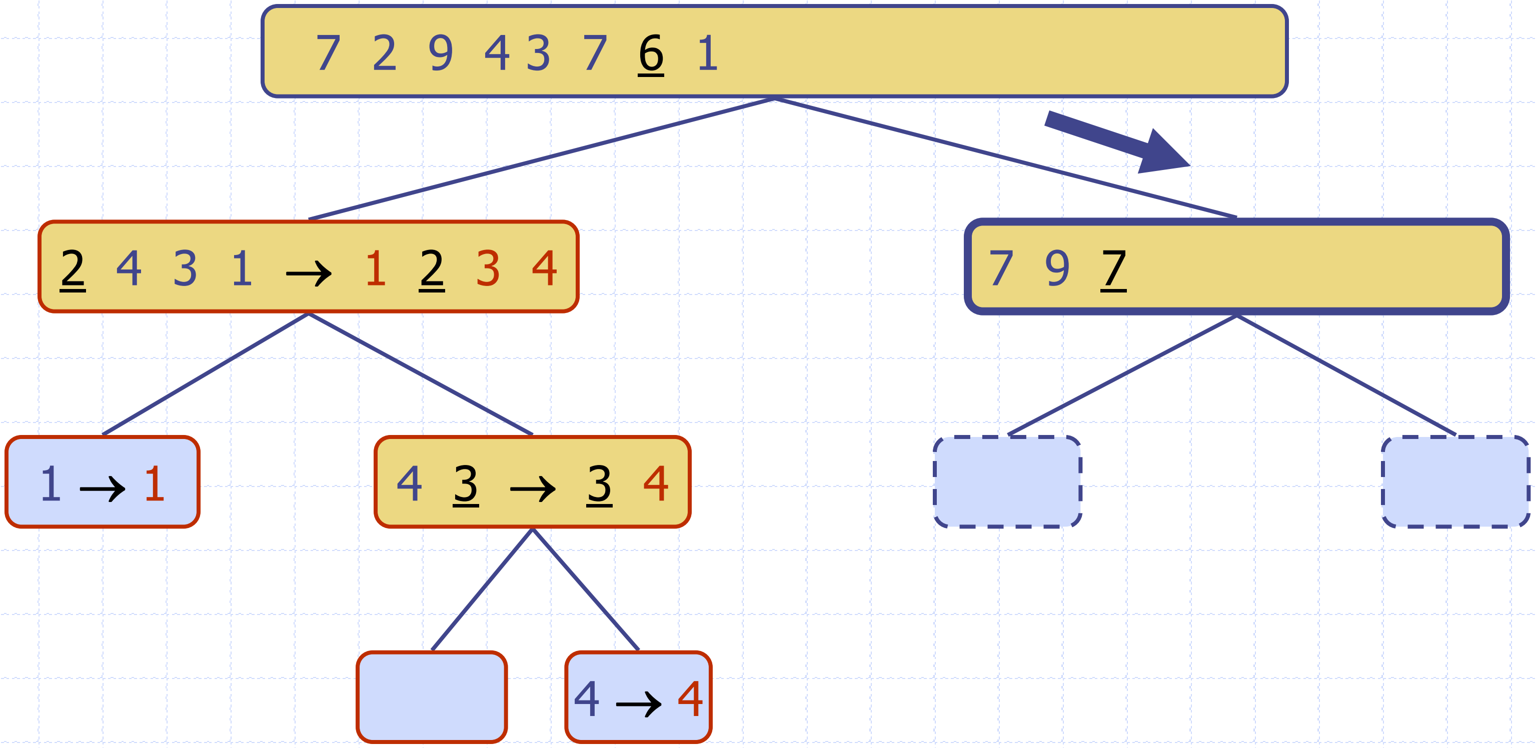
## Execution Example (cont.)

- Recursive call, …, base case, join



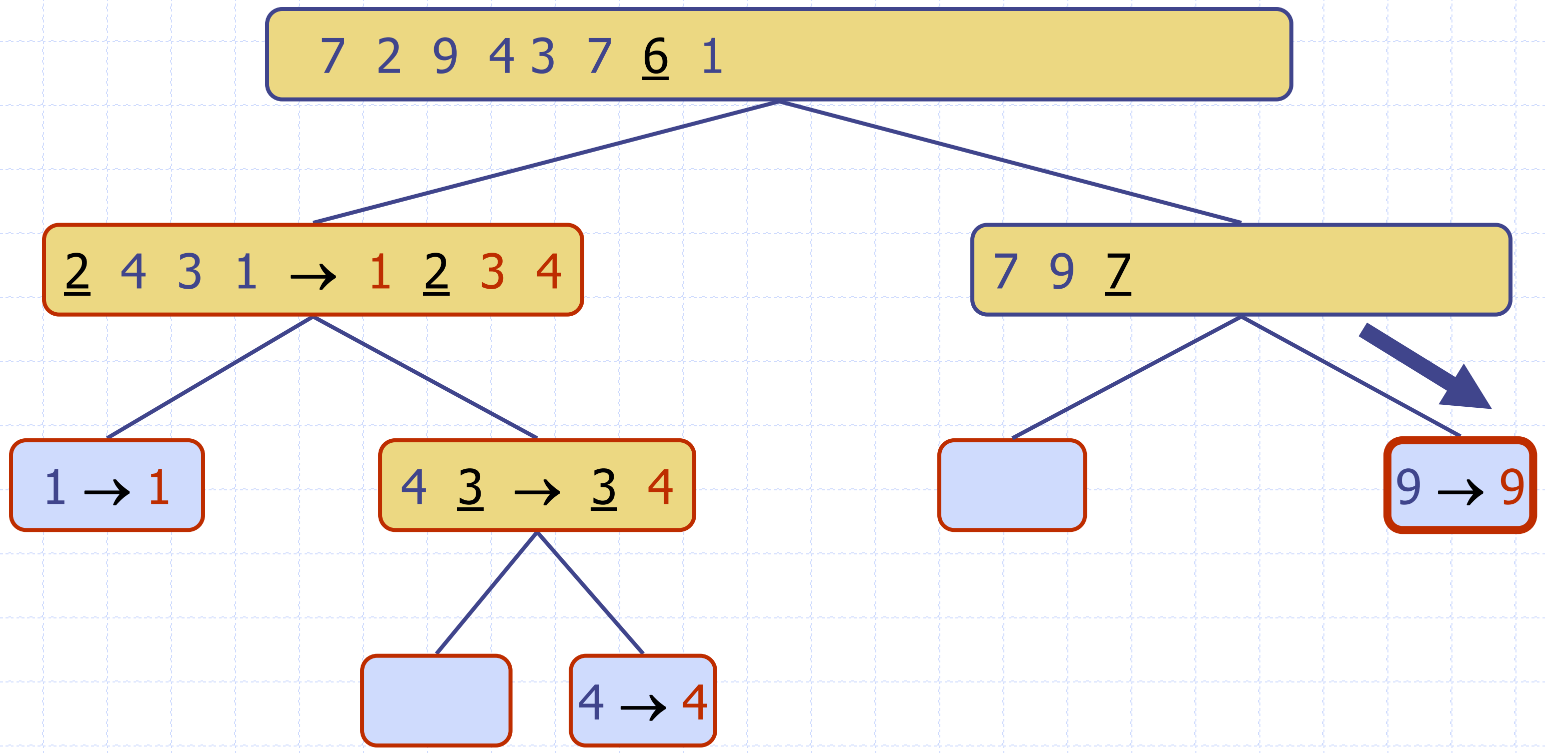
# Execution Example (cont.)

- Recursive call, pivot selection



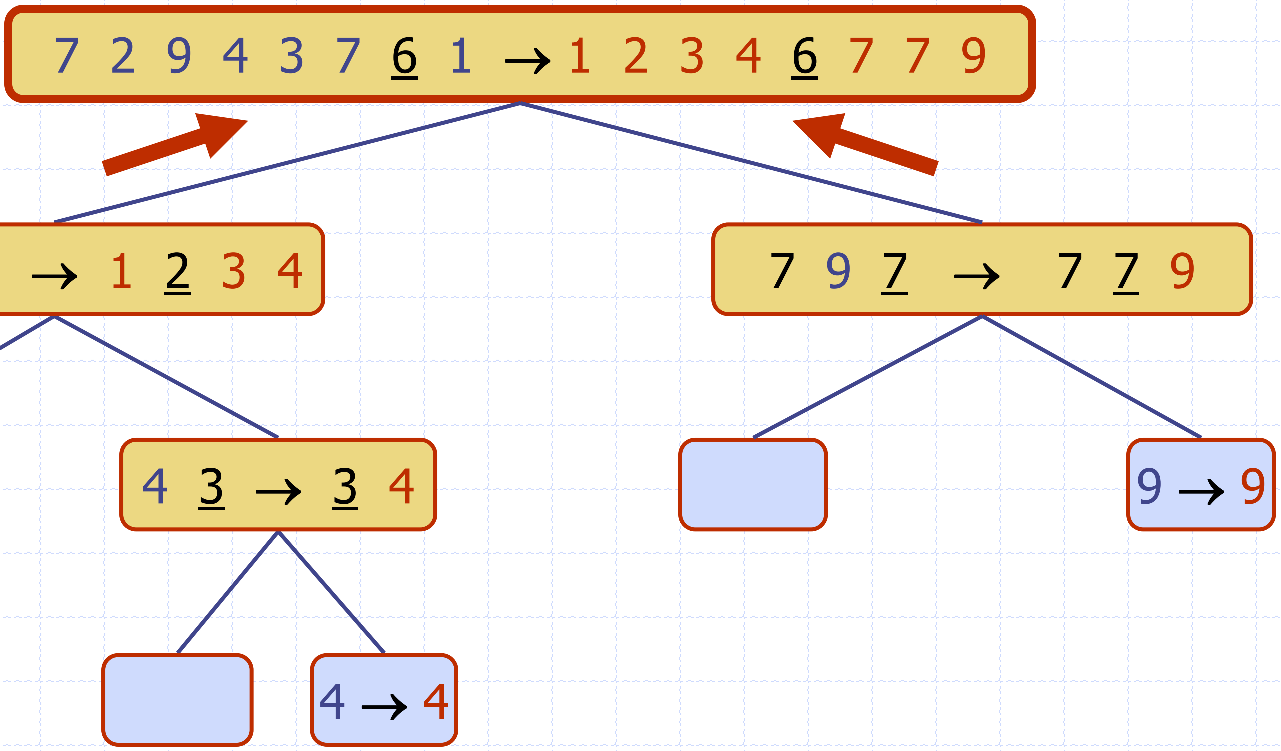
# Execution Example (cont.)

- Partition, …, recursive call, base case



## Execution Example (cont.)

- Join, join



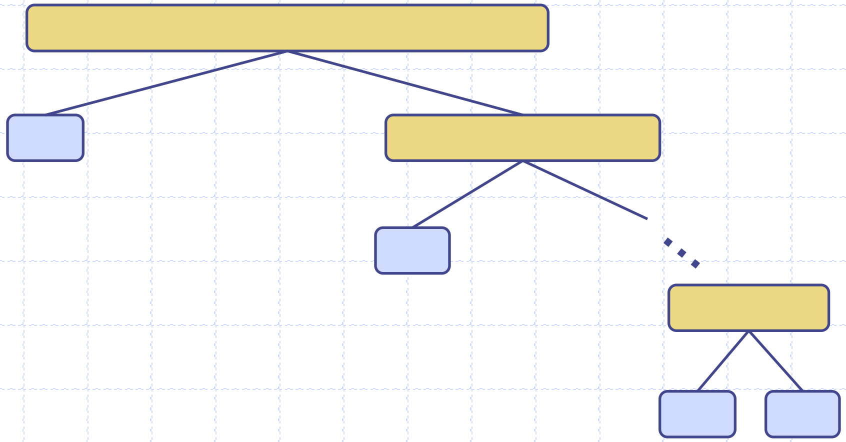
# Worst-case Running Time

- When does the worst-case running time occur?
  - When the pivot is the unique minimum or maximum element
- In such cases, one of $L$ and $G$ has size $n - 1$ and the other size $0$
- The running time is proportional to the sum

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

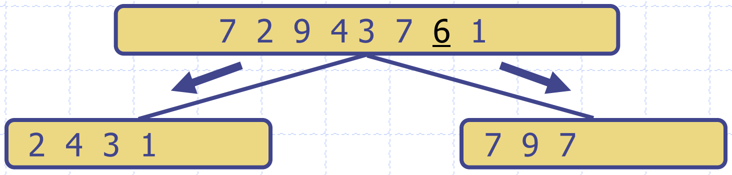
- Thus, the worst-case running time of quick-sort is
  - $O(n^2)$

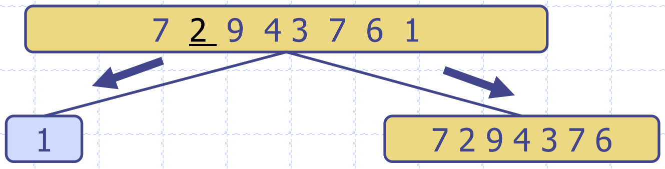
| depth | time |
|---|---|
| $0$ | $n$ |
| $1$ | $n - 1$ |
| … | … |
| $n - 1$ | $1$ |



# Expected Running Time

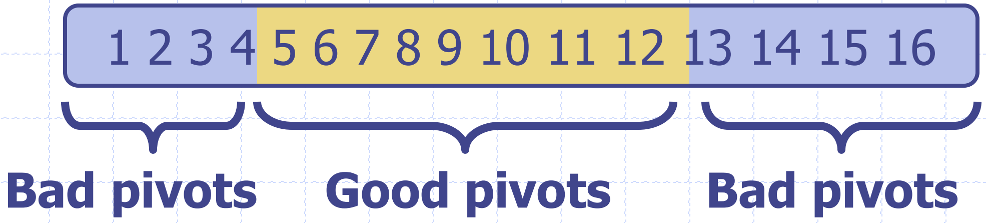
- Consider a recursive call of quick-sort on a sequence of size $s$
  - **Good call:** the sizes of $L$ and $G$ are each less than $3s/4$
  - **Bad call:** one of $L$ and $G$ has size greater than $3s/4$





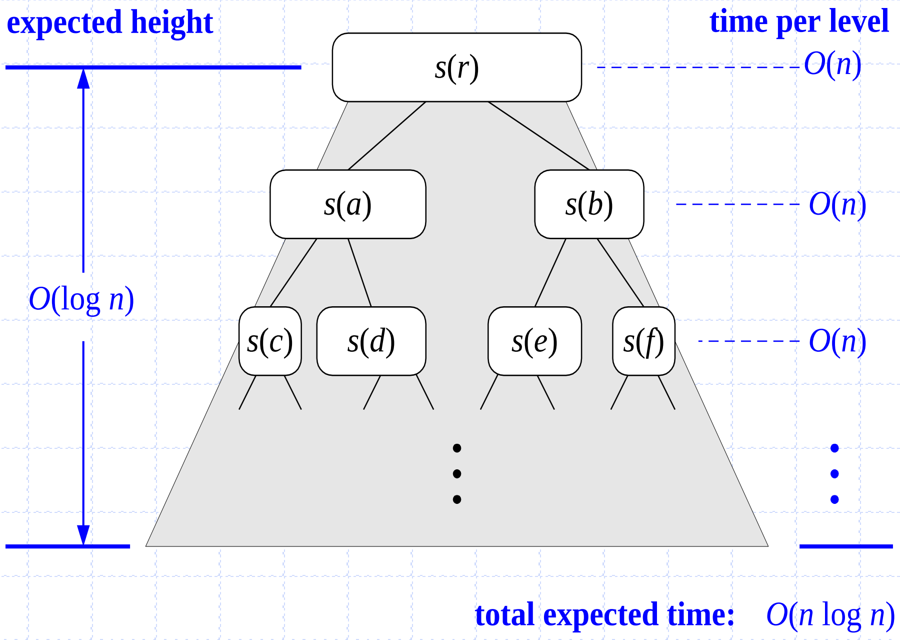
**Good call** **Bad call**

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

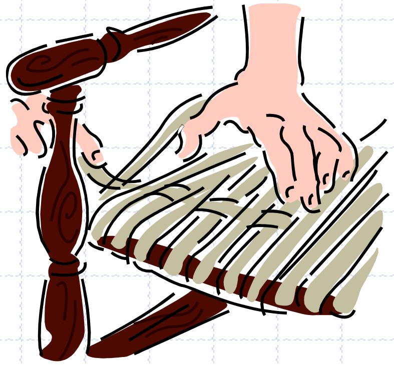


# Expected Running Time, Part 2

- (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
  - How many of the ancestor calls to be good?
    - $i/2$ ancestors
  - 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
  - Thus, 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 $O(n)$
- Hence, the expected running time of quick-sort is $O(n \log n)$



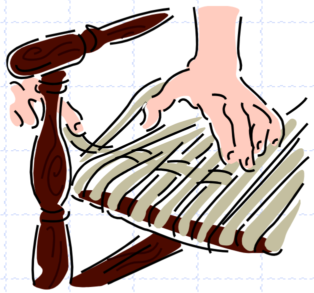
# In-Place Quick-Sort



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

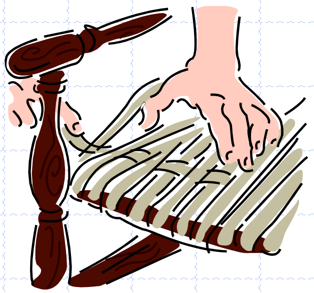
# In-Place Partitioning



- Perform the partition with pivot Y using two indices to split S into L and EYG (a similar method can split EYG into E and G)

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

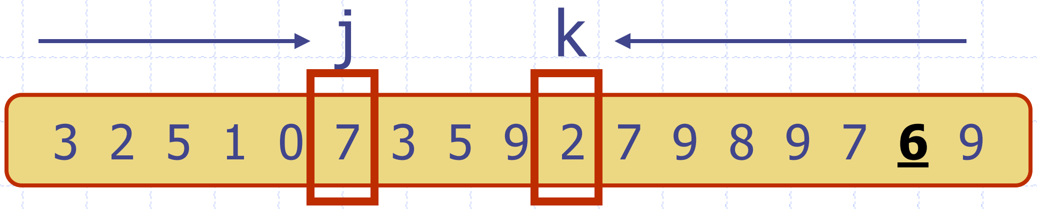
# In-Place Partitioning



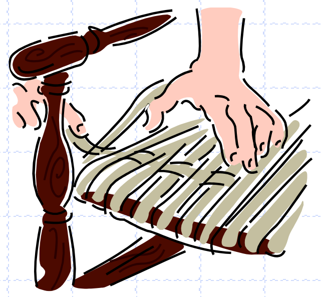
- Perform the partition with pivot Y using two indices to split S into L and E∪G (a similar method can split E∪G into E and G)

```
  j                                             k
  3  2  5  1  0  7  3  5  9  2  7  9  8  9  7  6  9    ("Y" pivot = 6)
```

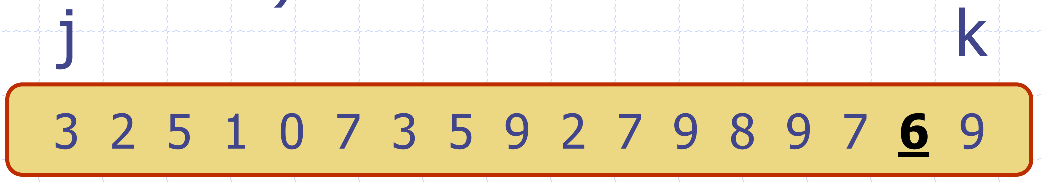
- Repeat
  - Scan j to the right until finding an element ≥ x
  - Scan k to the left until finding an element < x
  - Swap elements at indices j and k



# In-Place Partitioning

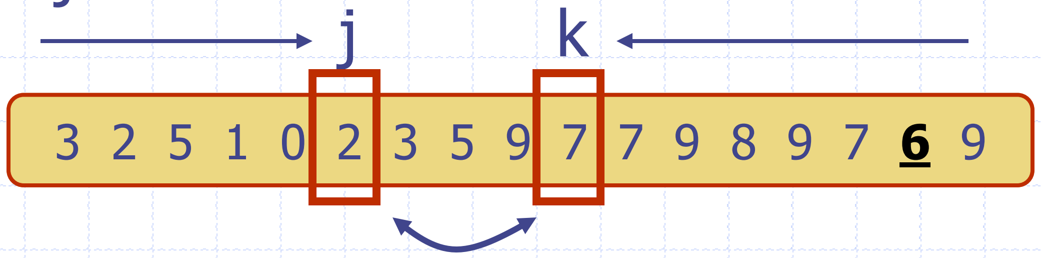


- Perform the partition with pivot Y using two indices to split S into L and EYG (a similar method can split EYG into E and G)

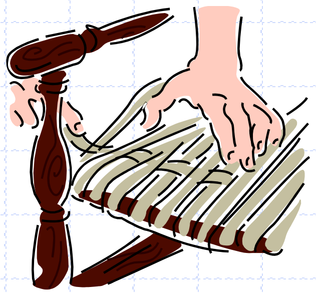


("Y" pivot = 6)

- Repeat
  - 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
- Until j and k meet



# In-Place Partitioning



- Perform the partition with pivot Y using two indices to split S into L and EYG (a similar method can split EYG into E and G)

```
  j                                k
  3  2  5  1  0  7  3  5  9  2  7  9  8  9  7  6  9
```

("Y" pivot = 6)

- Repeat
  - 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
- Until j and k meet

```
  ———————————→ jk ←———————————
  3  2  5  1  0  2  3  5  9  7  7  9  8  9  7  6  9
```

(done with all partitioning operations and eventually sorting the array)

# Isn't all this Merge-Sort?

- Quick-Sort is similar to Merge-Sort but with the following key differences
  - (In-place) Quick-Sort uses at most $O(\log n)$ space vs. Merge-Sort uses $O(n)$ space
  - Quick-Sort is $O(n^2)$ vs. Merge-Sort which is $O(n \log n)$
    - But a "good" Quick-Sort is $O(n \log n)$ or better
  - Quick-Sort implementation details are more friendly towards current computer architecture and thus in practice the "constant" is very small

## 3-Way Quicksort

- Instead of partitioning into 2 sets each time, partition into 3 sets:
  - Less than set
  - Equal than set
  - Greater than set

- Helps slightly with many repeated keys

# Summary of Sorting Algorithms

| Algorithm | Time | Notes |
|---|---|---|
| selection-sort | $O(n^2)$ | ◈ in-place<br>◈ slow (good for small inputs) |
| insertion-sort | $O(n^2)$ | ◈ in-place, O(n) for almost sorted<br>◈ slow (good for small inputs) |
| quick-sort | $O(n \log n)$ expected | ◈ in-place, randomized<br>◈ fastest (good for large inputs) |
| heap-sort | $O(n \log n)$ | ◈ in-place, O(n) first results<br>◈ fast (good for large inputs) |
| merge-sort | $O(n \log n)$ | ◈ sequential data access<br>◈ fast (good for huge inputs) |

## Demo

- https://www.youtube.com/watch?v=kPRA0W1kECg
- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html
- https://www.toptal.com/developers/sorting-algorithms