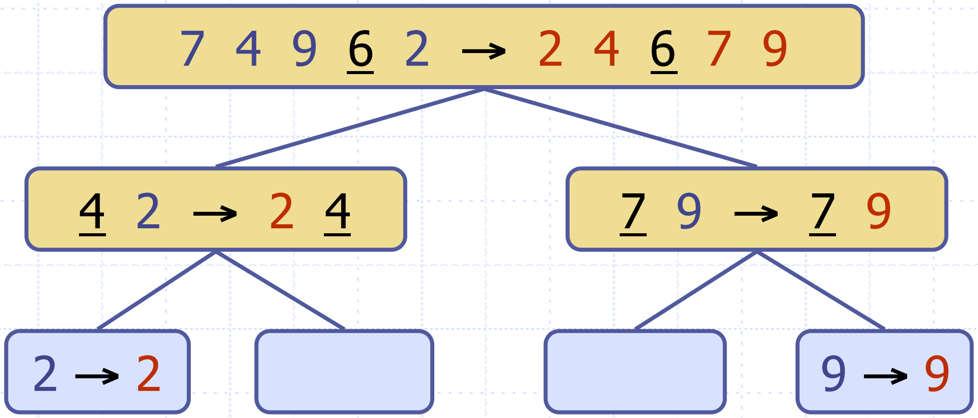
QuickSort



QuickSort

- QuickSort on an input sequence S with n elements consists of three steps:
    - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
    - Recurse: recursively sort S_1 and S_2
    - Conquer?

```
QuickSort(S)
    if S.size() <= 1
        return

    last = last item in S
    (S_1, S_2) = partition(S, last)
    QuickSort(S_1)
    QuickSort(S_2)
```

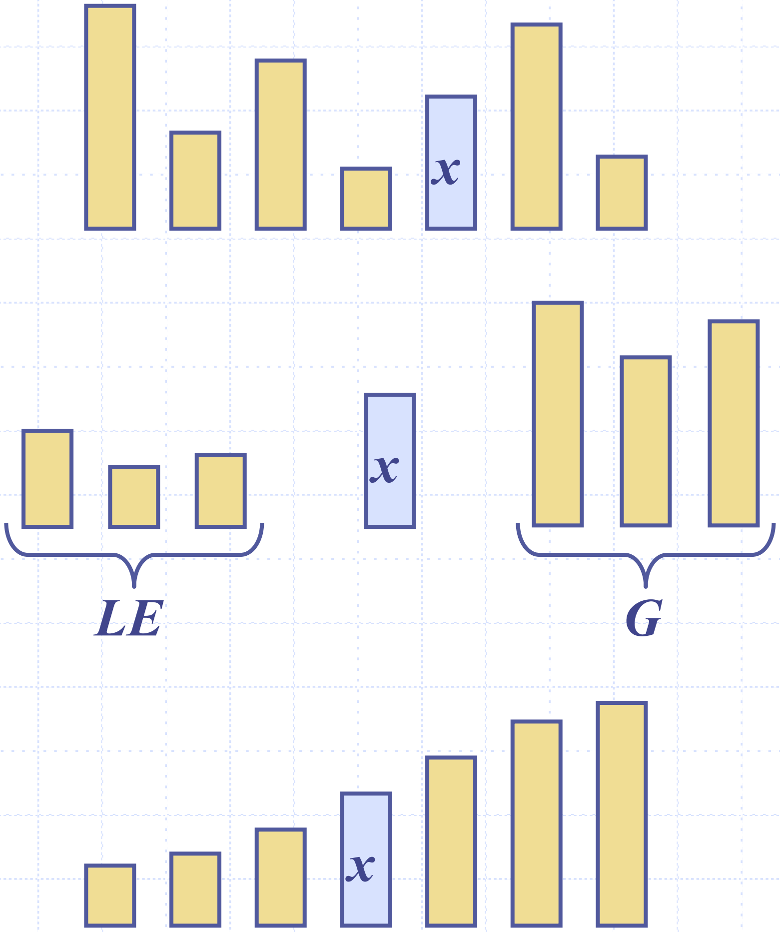
Partition

- We partition by removing, in turn, each element *y* from *S* and inserting insert *y* into ***LE*** (less than or equal to the ***pivot***) or ***G***, (greater than the ***pivot***)
- Each insertion and removal takes constant time, so partitioning takes *O*(*n*) time

```
partition(S, pivot)
    LE = empty list
    G = empty list
    while S.isEmpty == false
        y = S.get(0)
        S.remove(0)
        if y <= pivot
            LE.add(y)
        else   // y > pivot
            G.add(y)
    return LE and G
```

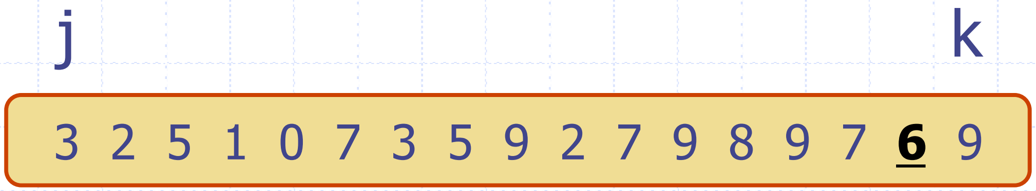
QuickSort

- Divide: take the last element x as the *pivot* and partition the list into
    - LE, elements <= x
    - G, elements > x
- Recurse: sort LE and G
- Conquer: Nothing to do!
- Issue: In-Place?
    - Was MergeSort?



In-Place Partitioning

- Perform the partition using two indices to split S into LE and G.

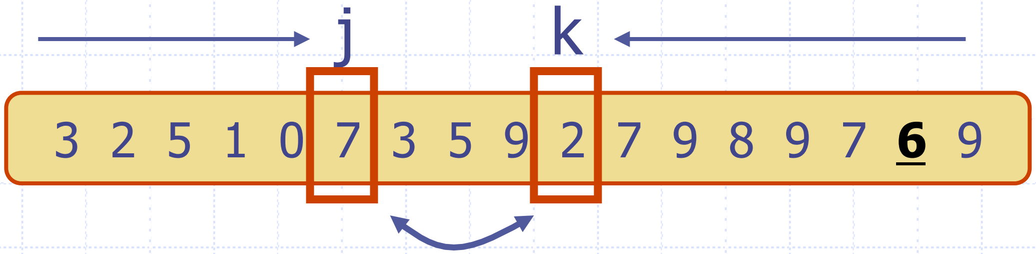


j k

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

(pivot x = 6)

- Repeat until j and k cross:
  - 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
- Then swap the element at index j with the pivot.



j k

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

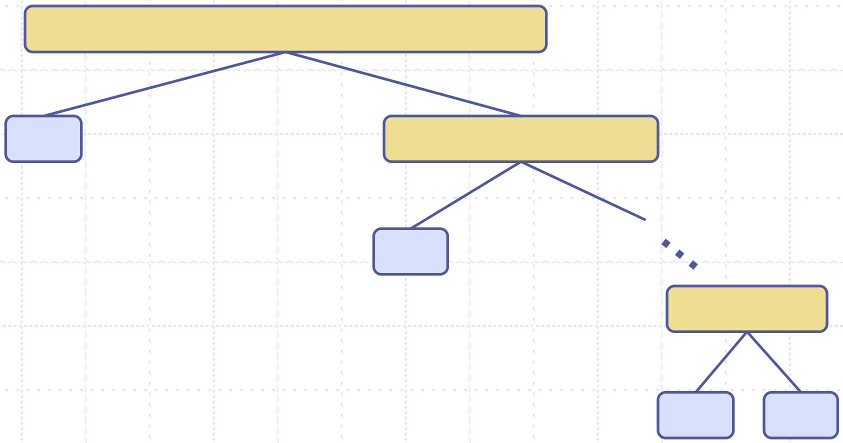
Worst-Case Running Time?

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

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

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

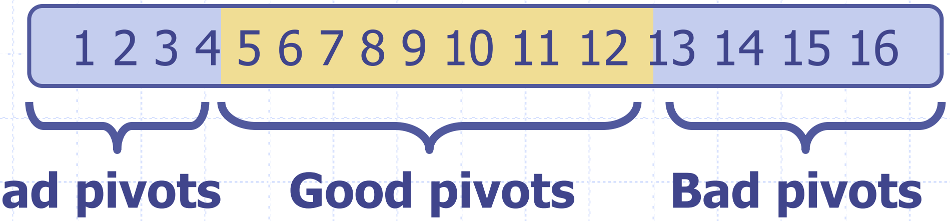
depth	time
0	n
1	$n - 1$
…	…
$n - 1$	1



Expected Running Time, Part 1

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

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



1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

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

Expected Running Time, Part 2

- What is the most number of levels at which we need to get "good" splits to get down to an input size of 1?
- The **worst** "good" split is an n/4, 3n/4 split
- How many worst "good" splits do we need to get down to size 1?

$$\left(\frac{3}{4}\right)^i n = 1 \quad \text{which means that} \quad i = \frac{\lg n}{\lg(4/3)}$$

- Probability Fact: The expected number of coin tosses required in order to get k heads is $2k$
- "Good" splits happen half the time, but they might all be worst "good" splits, so we will need i of them, and the expected number of splits needed to get i worst "good" splits is 2i or:

$$\frac{2\lg n}{\lg(4/3)} \approx 4.8 \lg n$$

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

QuickSort: Random is Better

- Choosing the last element as the pivot can lead to worst-cast behavior
- Choosing a pivot randomly can still lead to worst-case behavior, but it's much less likely
- Random pivot is standard

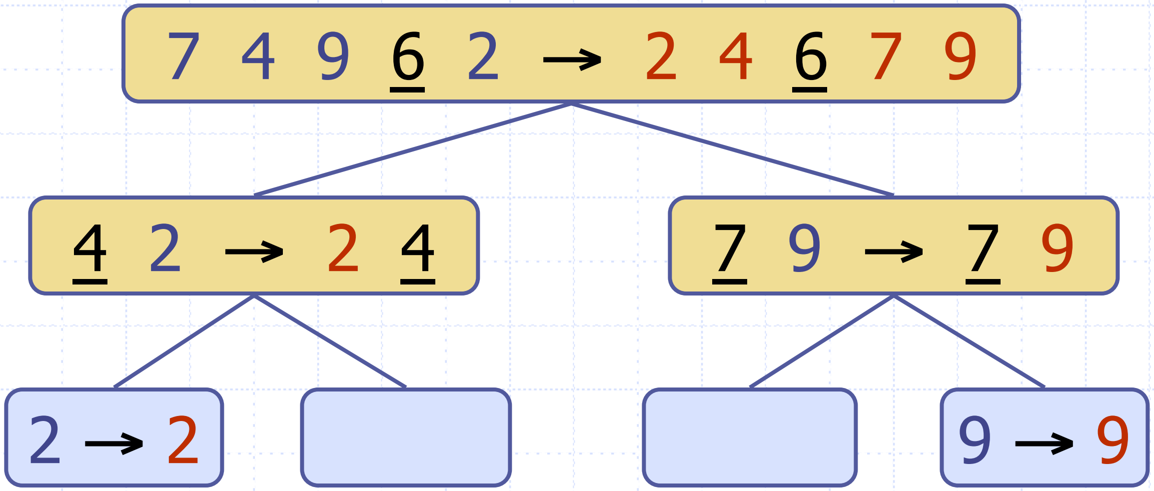
```
QuickSort(S)
    if S.size() <= 1
        return

    rItem= random item in S
    (S_1, S_2) = partition(S, rItem)
    QuickSort(S_1)
    QuickSort(S_2)
```

Power of Randomization

- Can show that randomized QuickSort runs in $O(n \log n)$ with high probability
- What if we didn't choose the pivot randomly?
  - Not first or last element
  - Median of 3
- What would be the best possible pivot?
- Why not use that?

QuickSort Tree

- An execution of QuickSort 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



7 4 9 $\underline{6}$ 2 → 2 4 $\underline{6}$ 7 9

$\underline{4}$ 2 → 2 $\underline{4}$

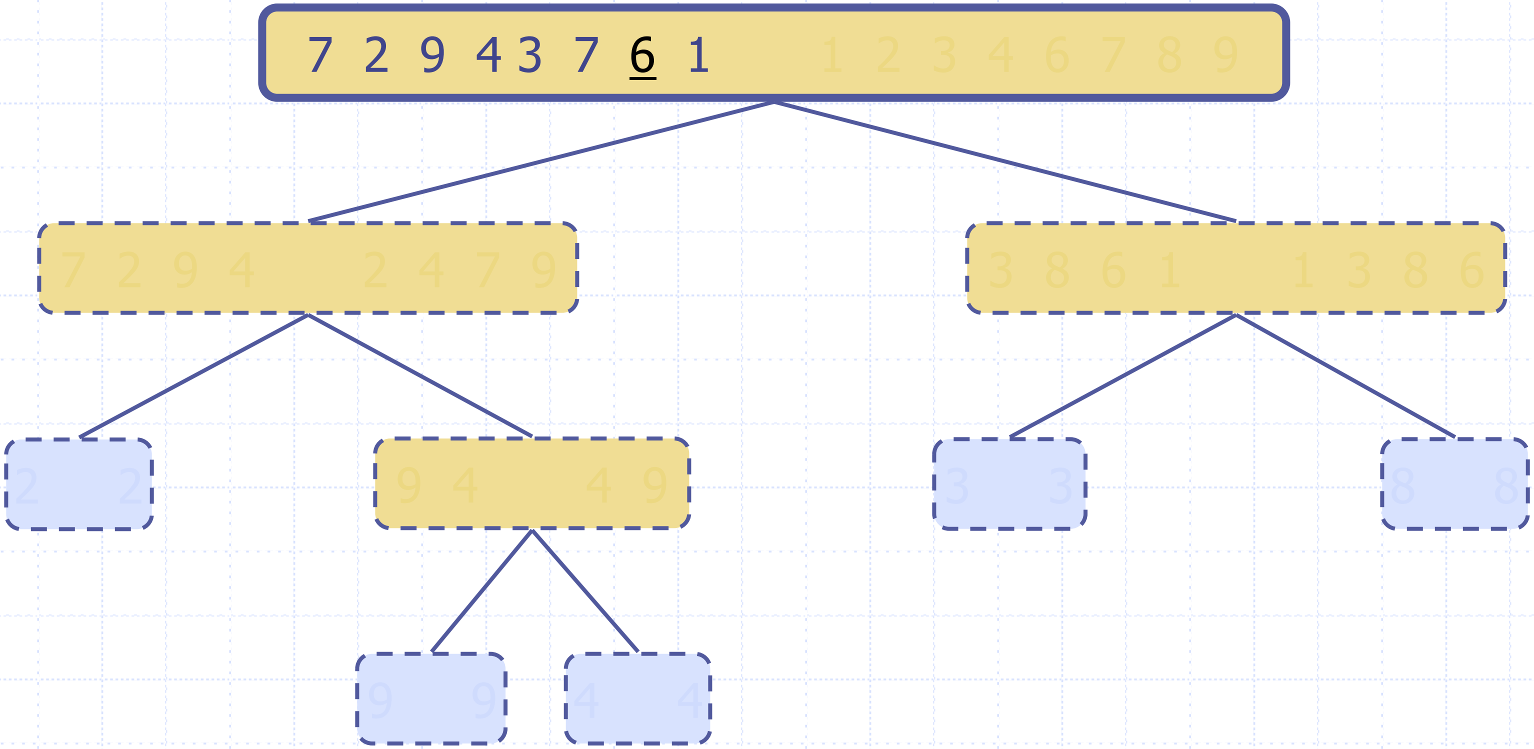
$\underline{7}$ 9 → $\underline{7}$ 9

2 → 2

9 → 9

Execution Example

- Pivot selection



7 2 9 4 3 7 6 1 1 2 3 4 6 7 8 9

7 2 9 4 2 4 7 9

3 8 6 1 1 3 8 6

2 2

9 4 4 9

3 3

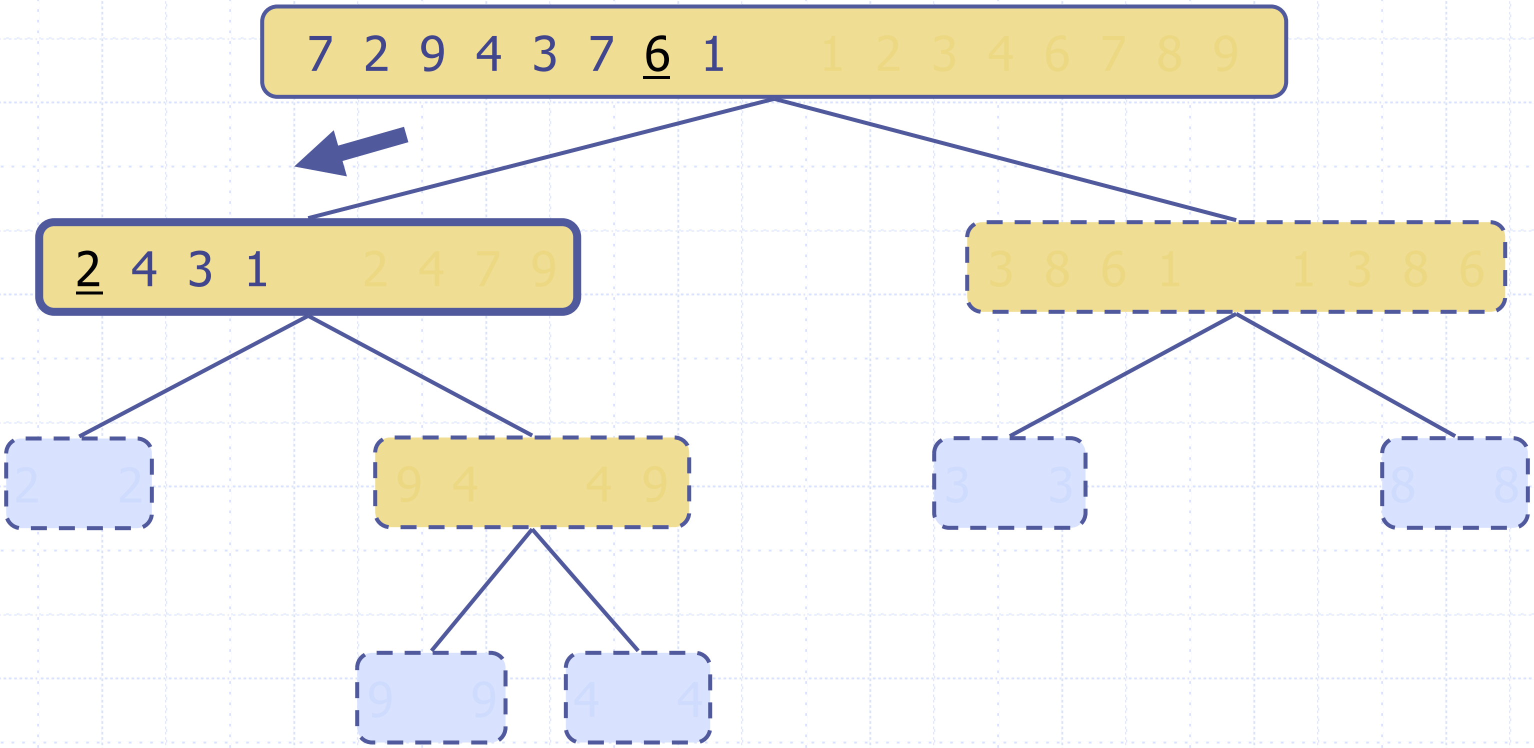
8 8

9 9

4 4

Execution Example (cont.)

- Partition, recursive call, pivot selection



7 2 9 4 3 7 6 1 1 2 3 4 6 7 8 9

2 4 3 1 2 4 7 9

3 8 6 1 1 3 8 6

2 2

9 4 4 9

3 3

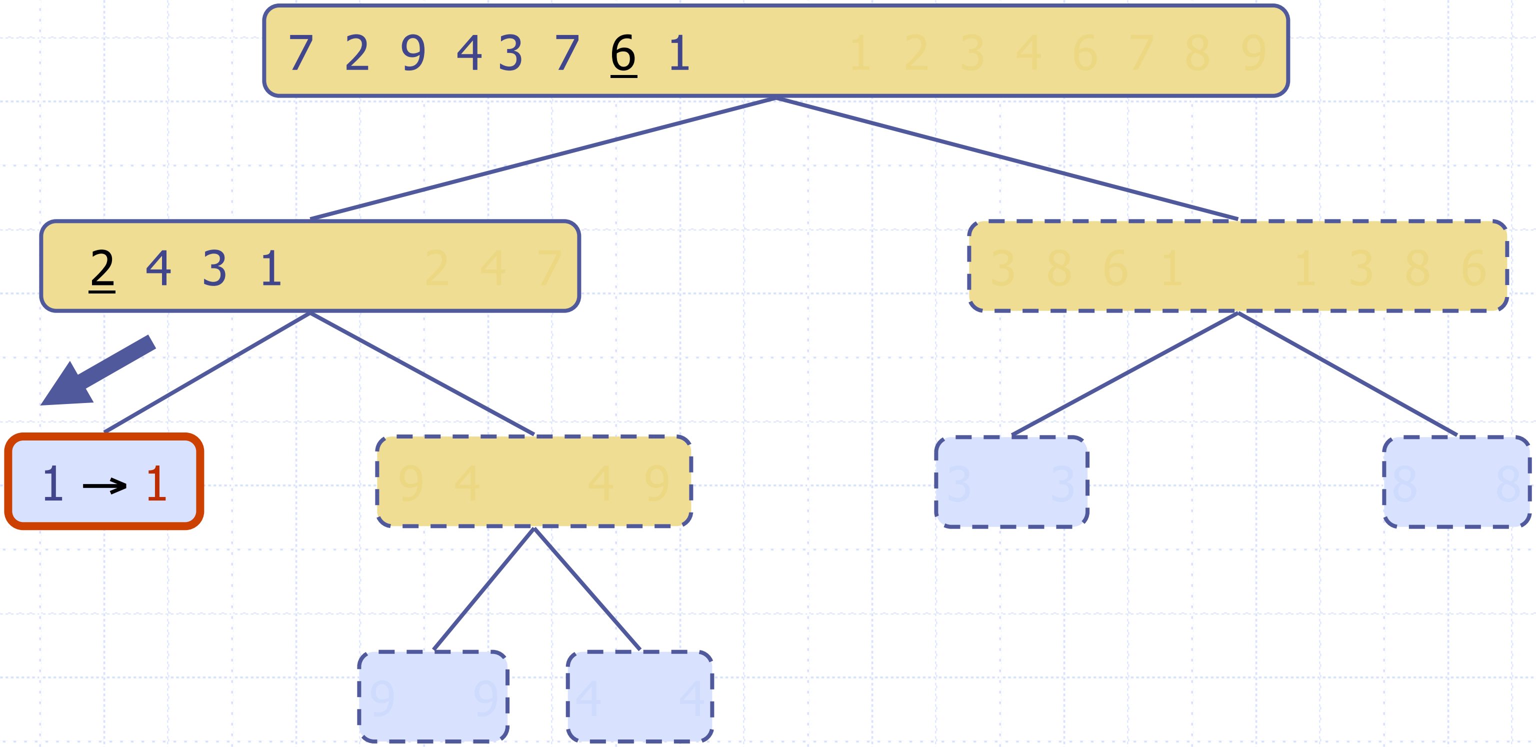
8 8

9 9

4 4

Execution Example (cont.)

- Partition, recursive call, base case



7 2 9 4 3 7 6 1 1 2 3 4 6 7 8 9

2 4 3 1 2 4 7

3 8 6 1 1 3 8 6

1 → 1

9 4 4 9

3 3

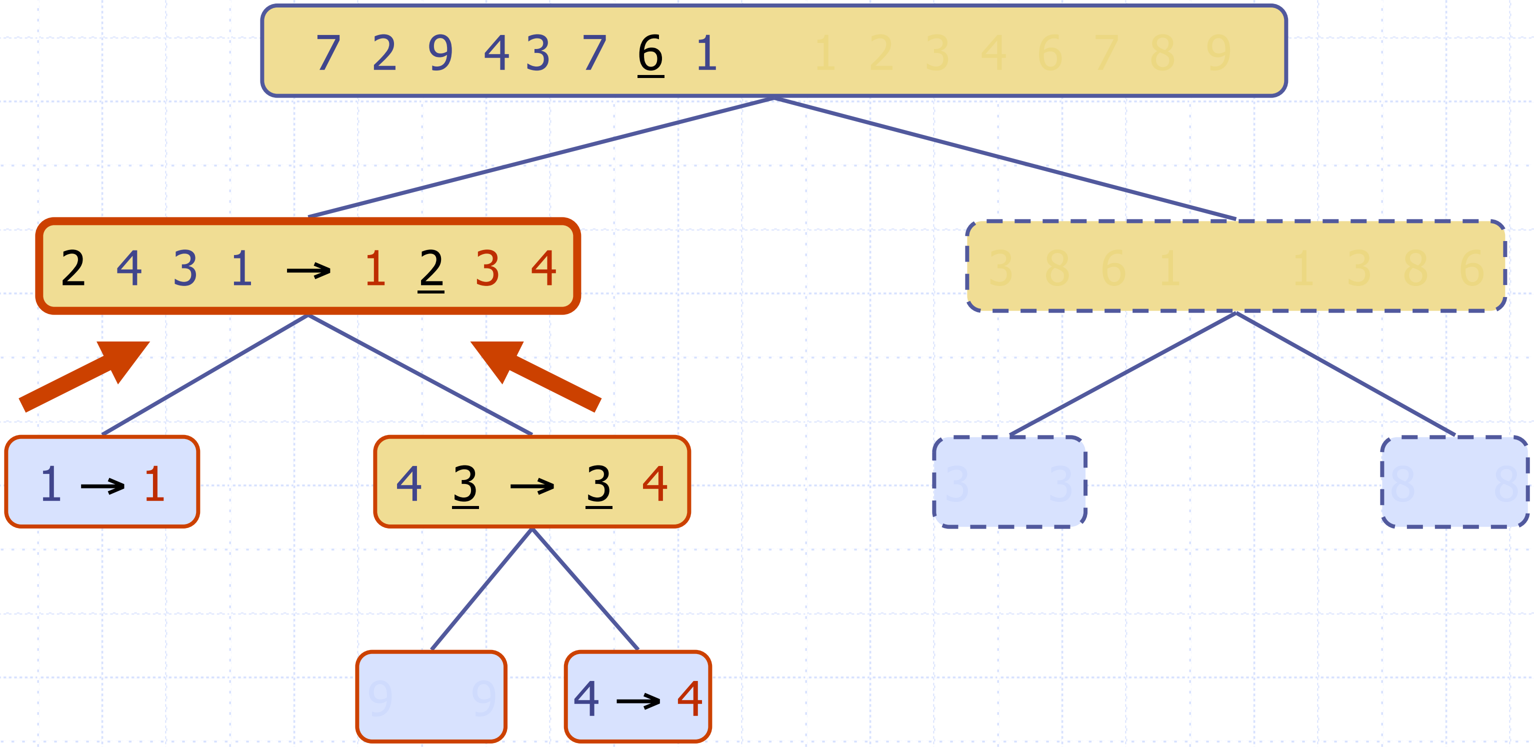
8 8

9 9

4 4

Execution Example (cont.)

- Recursive call, …, base case, join



7 2 9 4 3 7 6 1 1 2 3 4 6 7 8 9

2 4 3 1 → 1 2 3 4

3 8 6 1 1 3 8 6

1 → 1

4 3 → 3 4

3 3

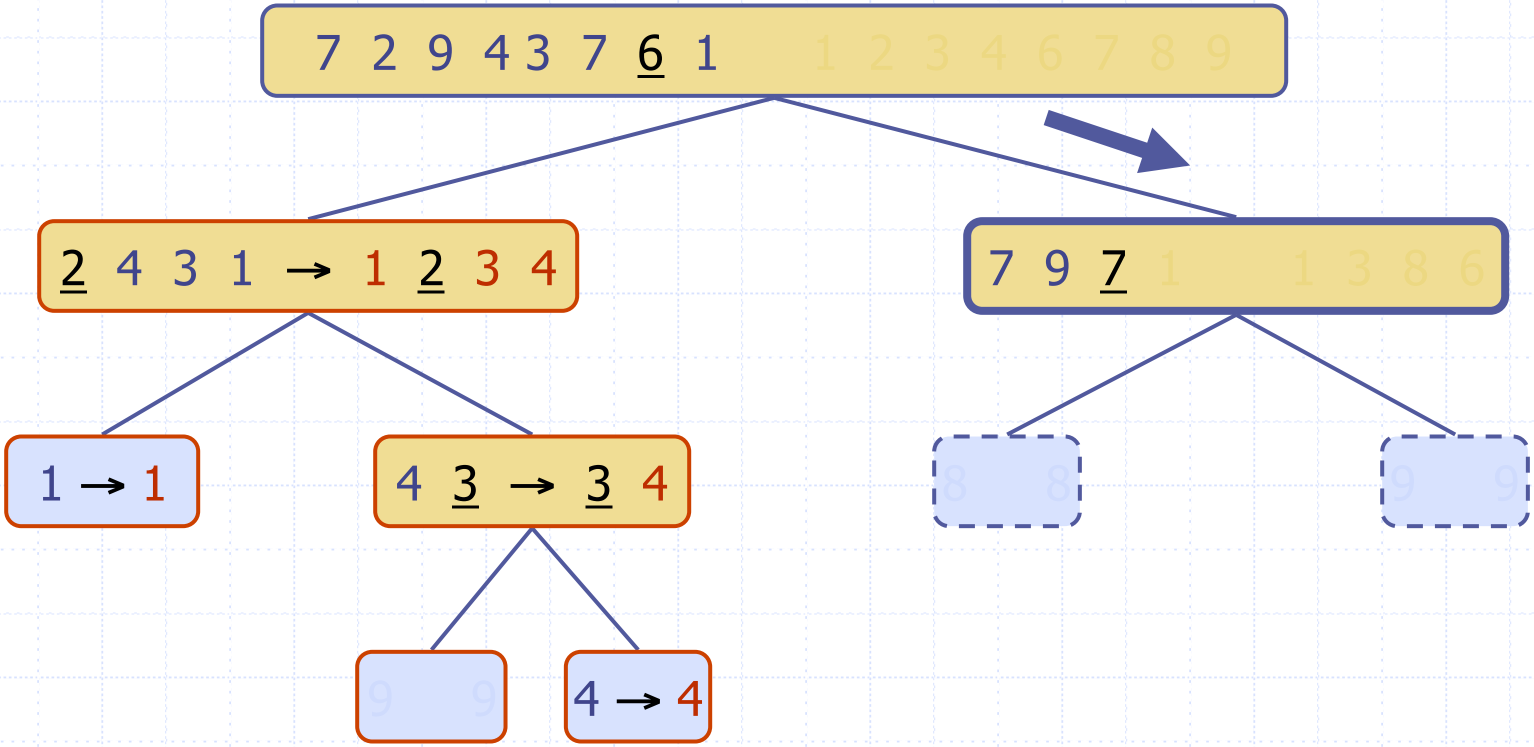
8 8

9 9

4 → 4

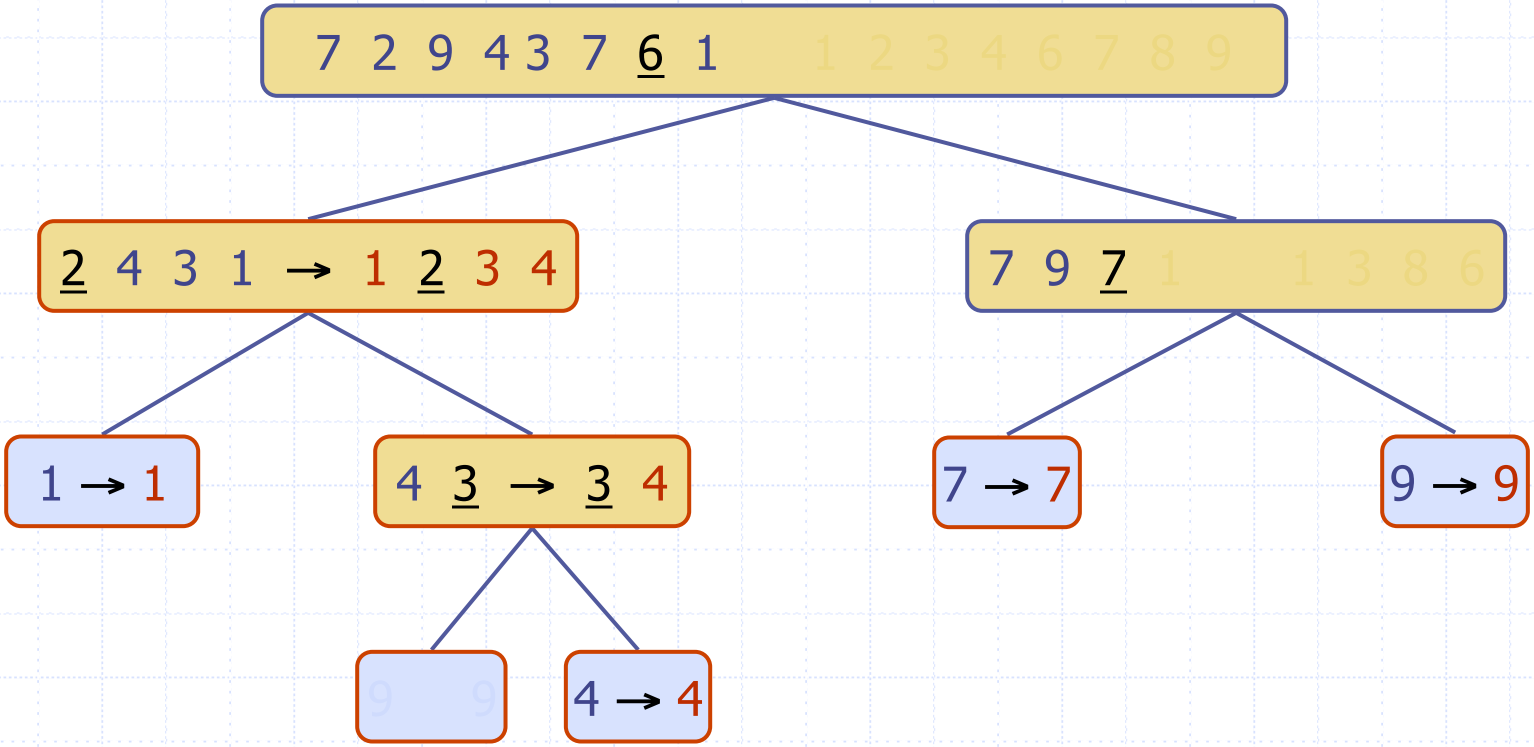
Execution Example (cont.)

- Recursive call, pivot selection



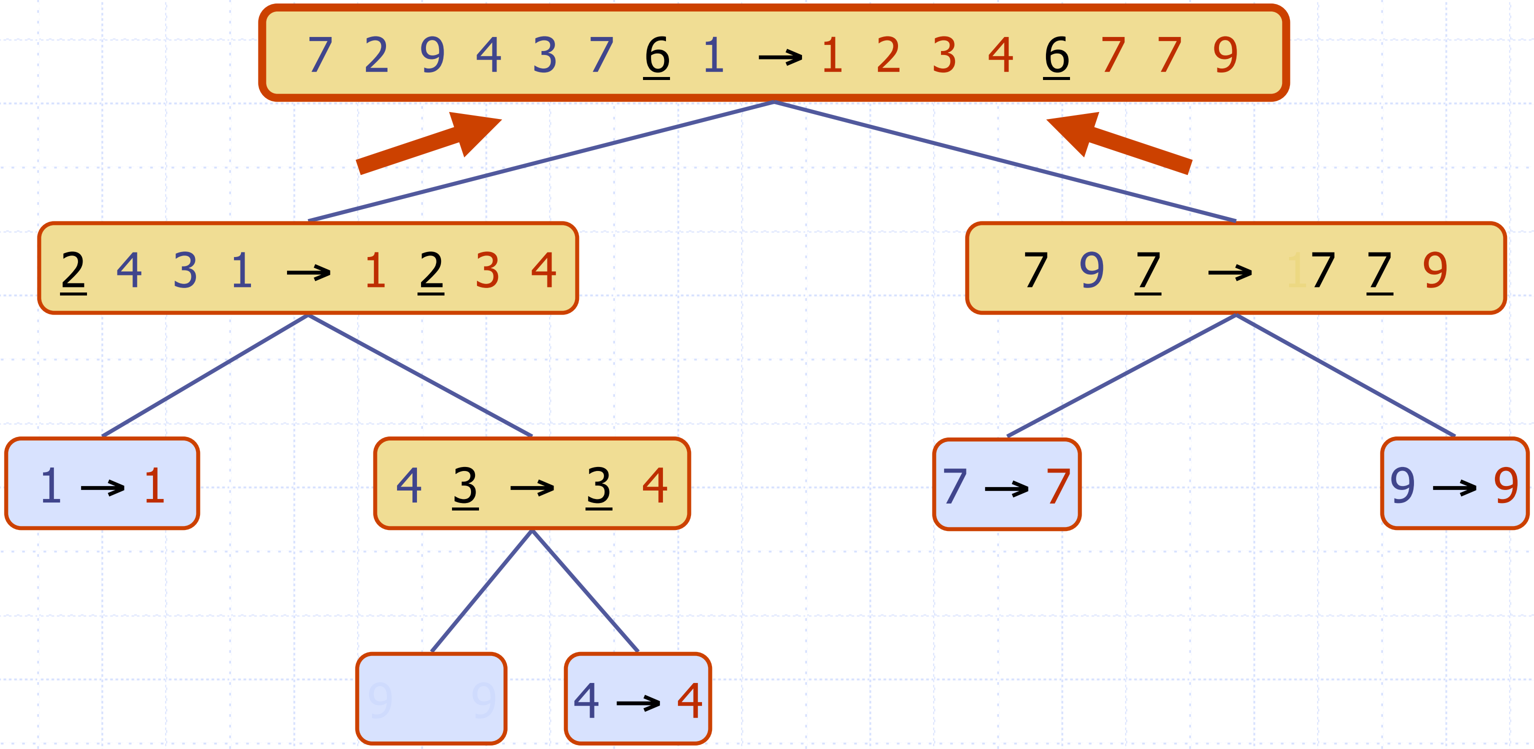
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



7 2 9 4 3 7 6 1 → 1 2 3 4 6 7 7 9

2 4 3 1 → 1 2 3 4

7 9 7 → 7 7 9

1 → 1

4 3 → 3 4

7 → 7

9 → 9

4 → 4

QuickSort Visualization

[Sorting Algorithms]