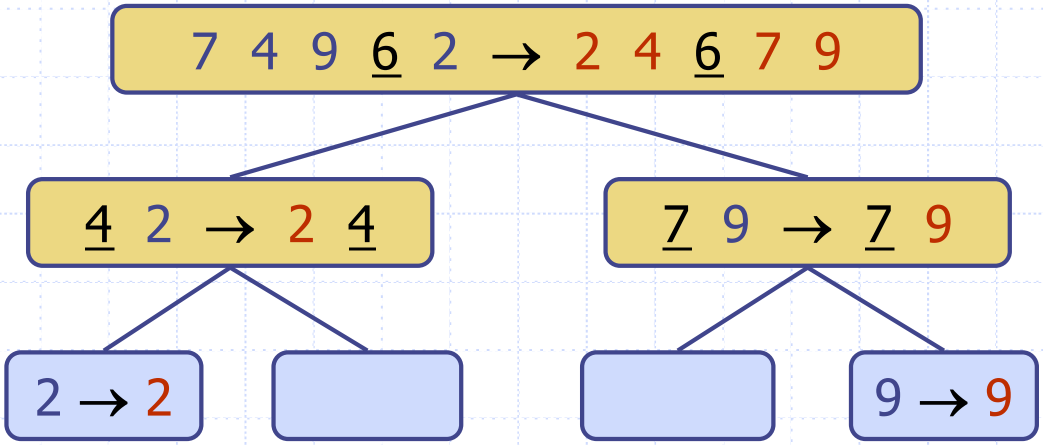
QuickSort



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: depends on what *partition* does.

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

    last = last item in S
    (S1, S2) = partition(S, last)
    QuickSort(S1)
    QuickSort(S2)
```

Partition

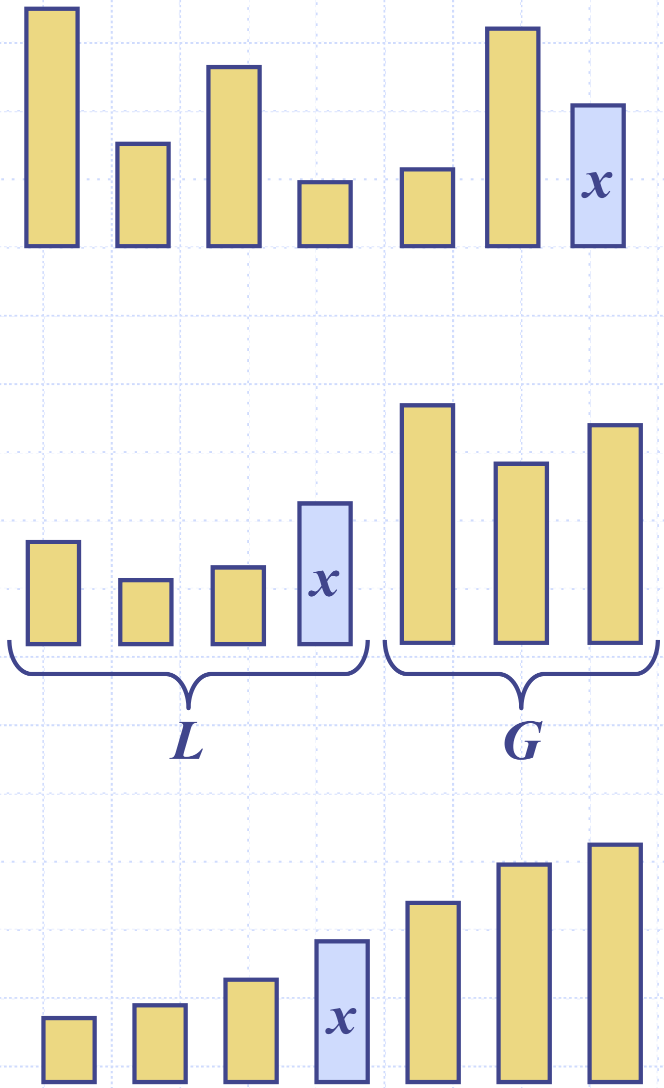
- We partition by removing, in turn, each element y from S and inserting y into L (less than the *pivot*) or G, (greater than the *pivot*)
- Each insertion and removal takes constant time, so partitioning takes $O(n)$ time

```
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
  - L, elements <= x
  - G, elements > x
- Recurse: sort L and G
- Conquer: Nothing to do!

- Issue: In-Place?



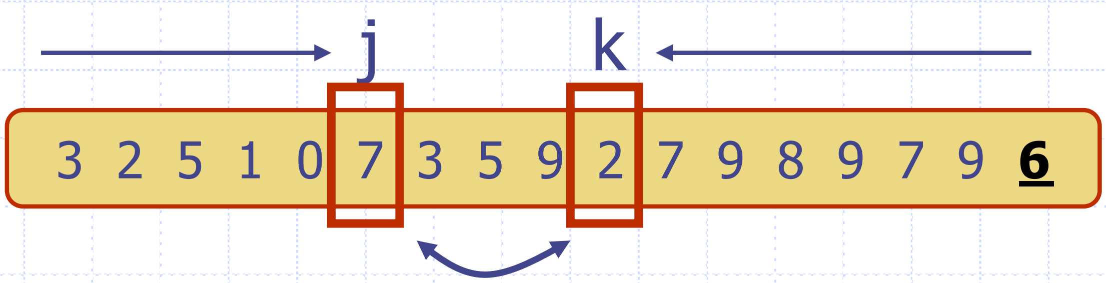
In-Place Partitioning (Hoare)

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

j k

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

- Repeat until j and k cross:
  - Scan j to the right until finding an element > pivot.
  - Scan k to the left until finding an element < pivot.
  - Swap elements at indices j and k
- Then swap the element at index j with the pivot.



In-Place Partitioning (Hoare)

```
HOARE-PARTITION(A, p, r)
x ← A[p]
i ← p − 1
j ← r + 1
while TRUE
    do repeat j ← j − 1
           until A[j] ≤ x
       repeat i ← i + 1
           until A[i] ≥ x
       if i < j
           then exchange A[i] ↔ A[j]
           else return j
```

In-Place Partitioning (Lomuto)

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1 DO
      if A[j] ≤ x
            i = i + 1
            swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

What's the Running Time?

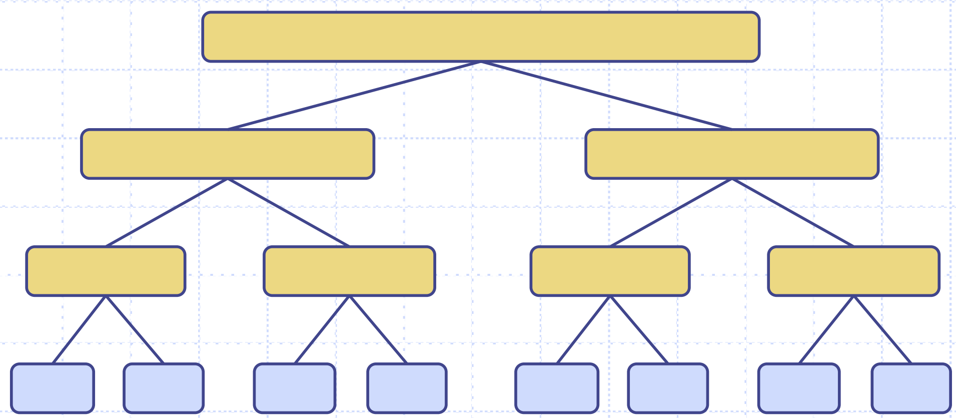
- It depends!
- On what?
- Best Case?
  - What's the recurrence?
  - What's the solution to the recurrence?
- Worst Case?
  - What's the recurrence?
  - What's the solution to the recurrence?

Best-Case Running Time

- The best case for quick-sort occurs when the pivot is the median
- Both sides of the partition have the same number of elements
- The running time is exactly like MergeSort:

$$T(n) = 2T(n/2) + n$$

T's	size	time
1	n	n
2	$n/2$	n
…	…	n
2^i	$n/2^i$	…



- So, the best-case running time of QuickSort is $O(n \lg n)$

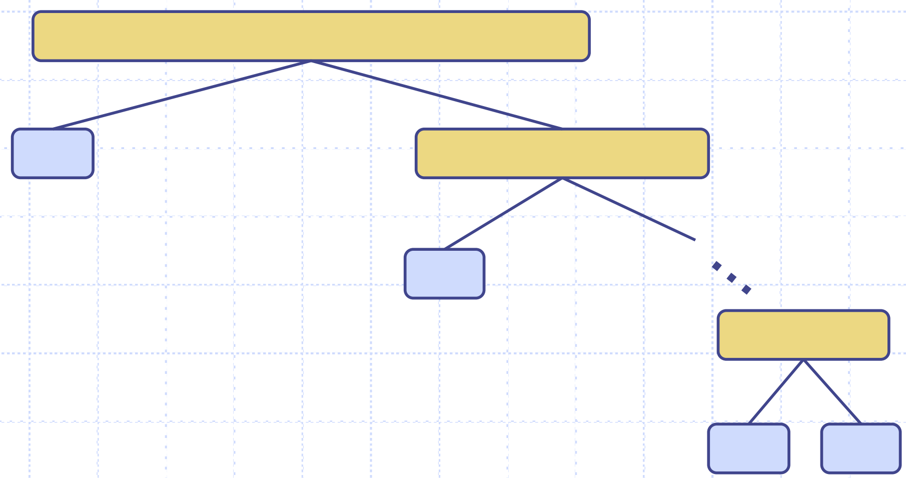
Worst-Case Running Time

- The worst case for quick-sort occurs when the pivot is the minimum or maximum element
- One side of the partition has $n - 1$ elements and the other has 0
- The running time is proportional to the sum of the partition times:

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

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

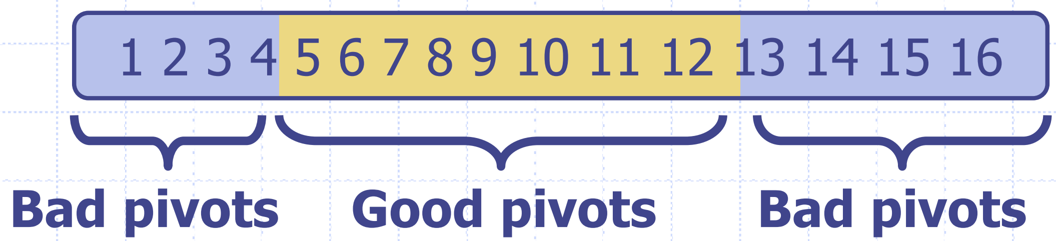
depth	time
0	n
1	$n - 1$
…	…
$n - 1$	1



Expected Running Time, Part 1

- Consider a recursive call of QuickSort on a sequence of size n
  - **Good split:** the sizes of L and G are each less than or equal to $3n/4$
  - **Bad split:** one of L and G has size greater than $3n/4$

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



- Use this to determine how many splits we need and, therefore, how many levels of recursion we will have

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 of these 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 $2k$.
- Since we need i "worst good" splits, and the probability of getting a "good" split is 1/2, the expected number of splits needed is 2i or:

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

- The amount of work done at all 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, especially if…
- Choosing a pivot randomly can still lead to worst-case behavior, but it's much less likely
- Random pivot is standard

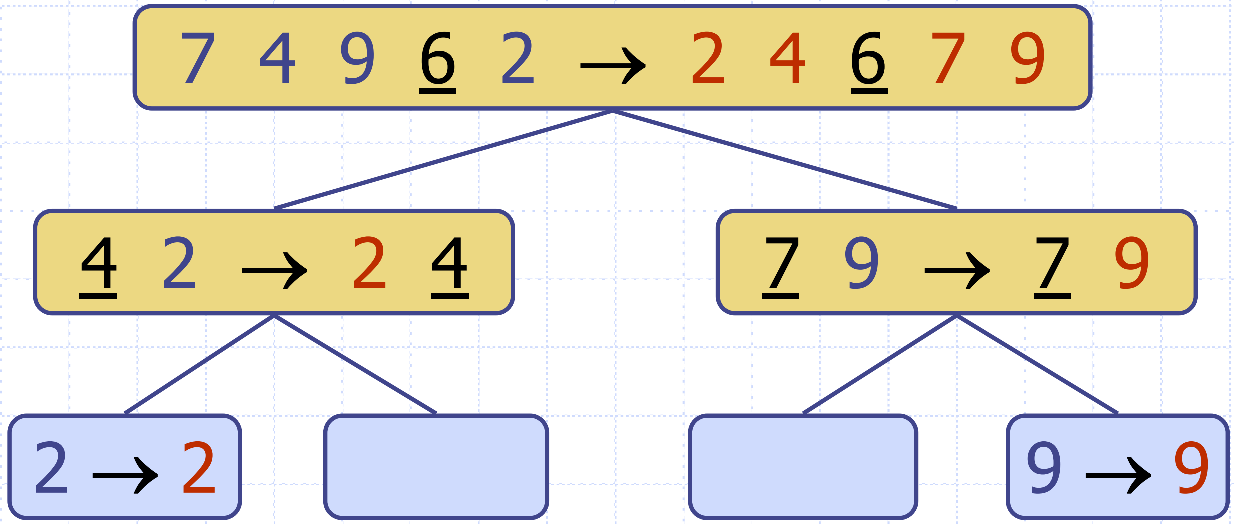
```
QuickSort(S)
    if S.size() <= 1
        return

    rItem= random item in S
    (S1, S2) = partition(S, rItem)
    QuickSort(S1)
    QuickSort(S2)
```

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?

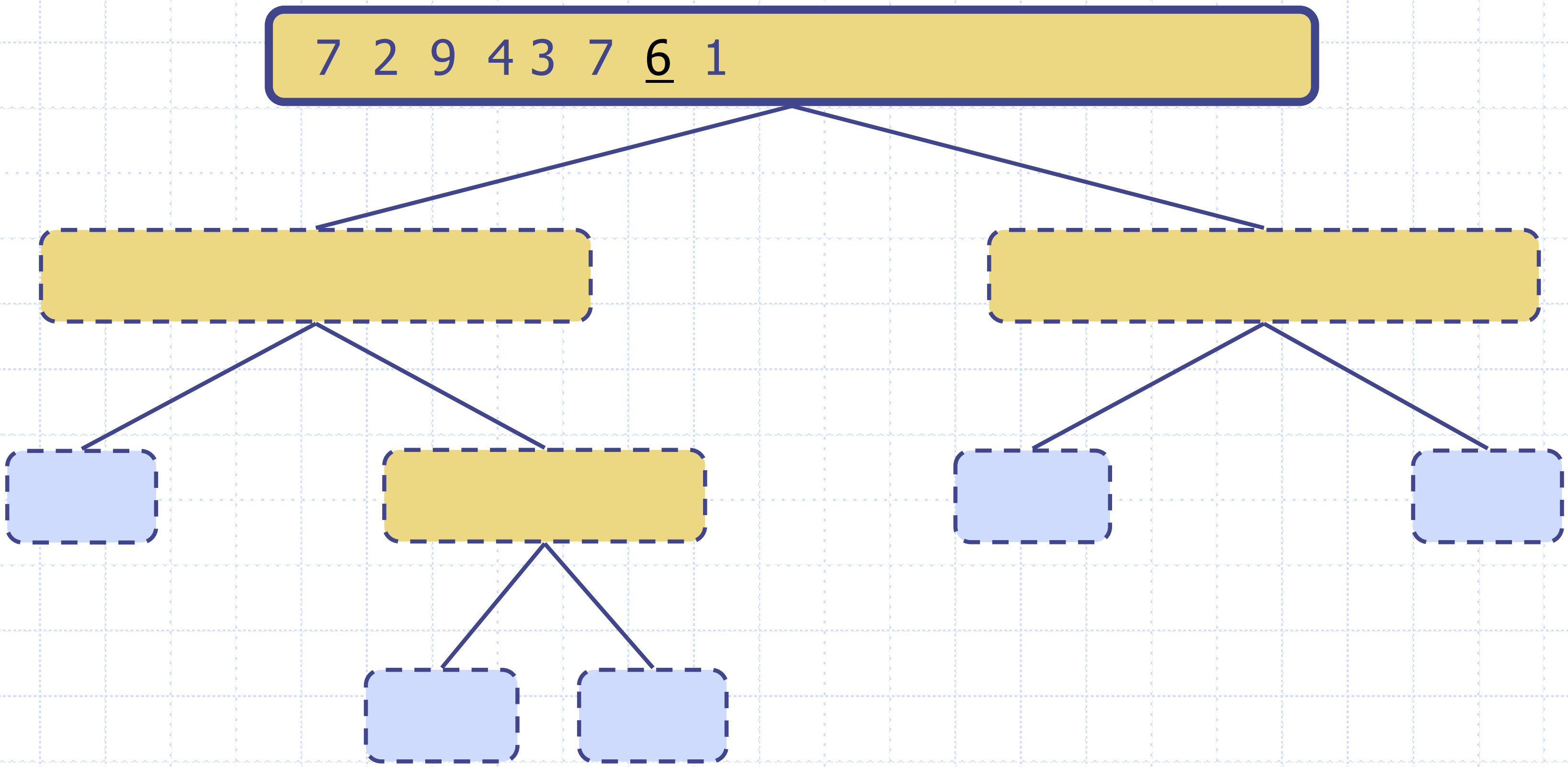
QuickSort Tree

- 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



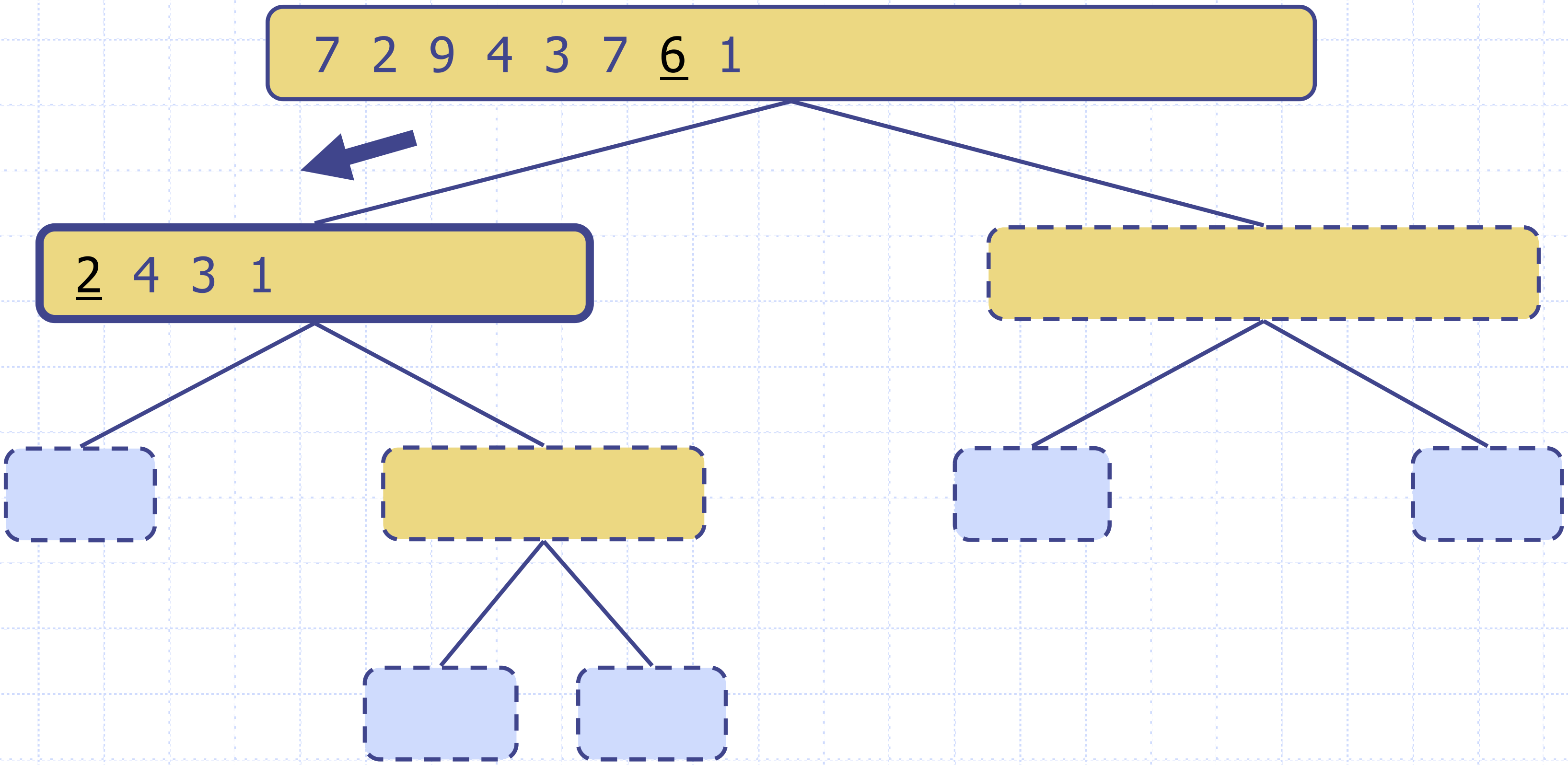
Execution Example

- Pivot selection



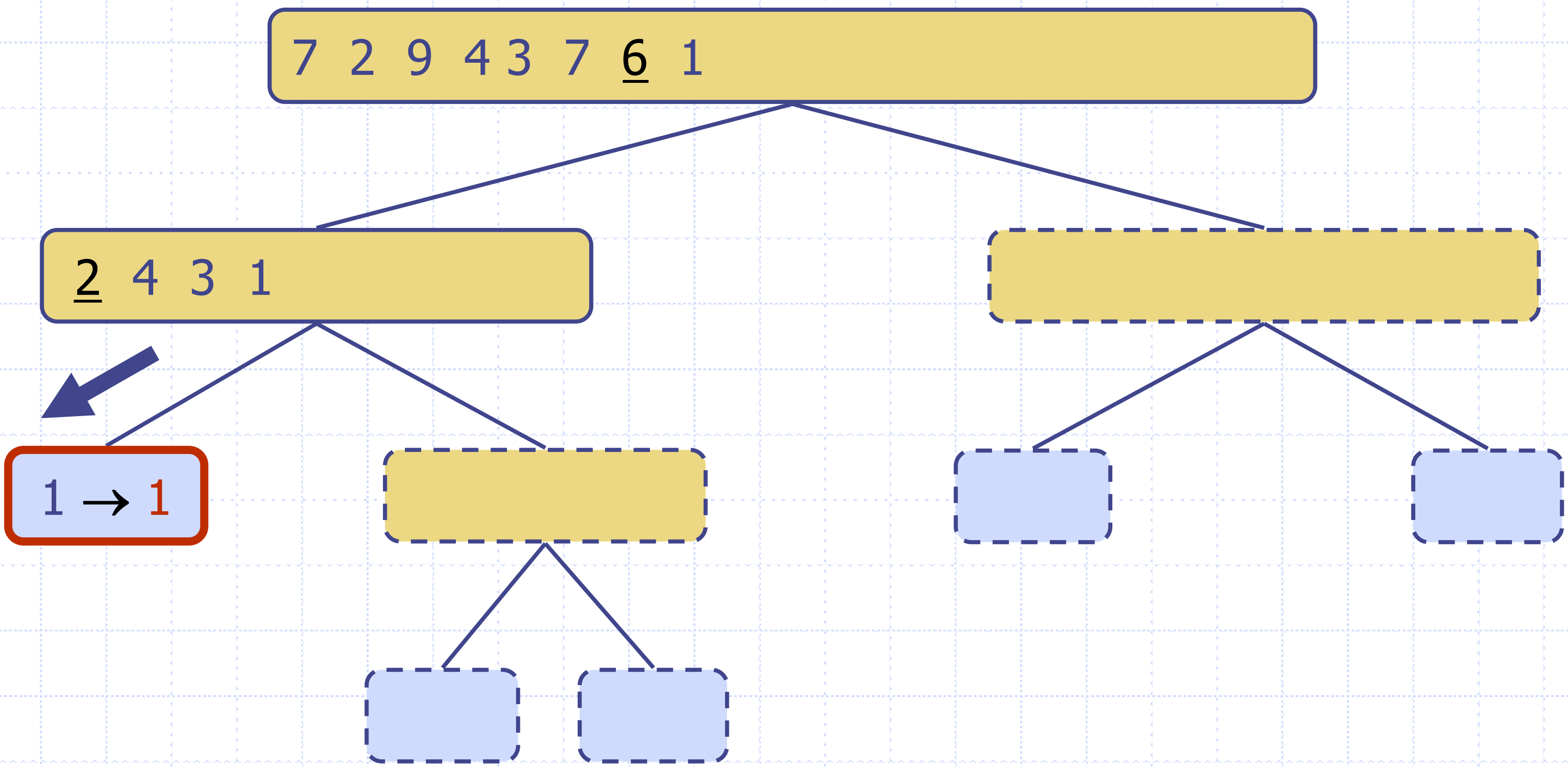
Execution Example (cont.)

- Partition, recursive call, pivot selection



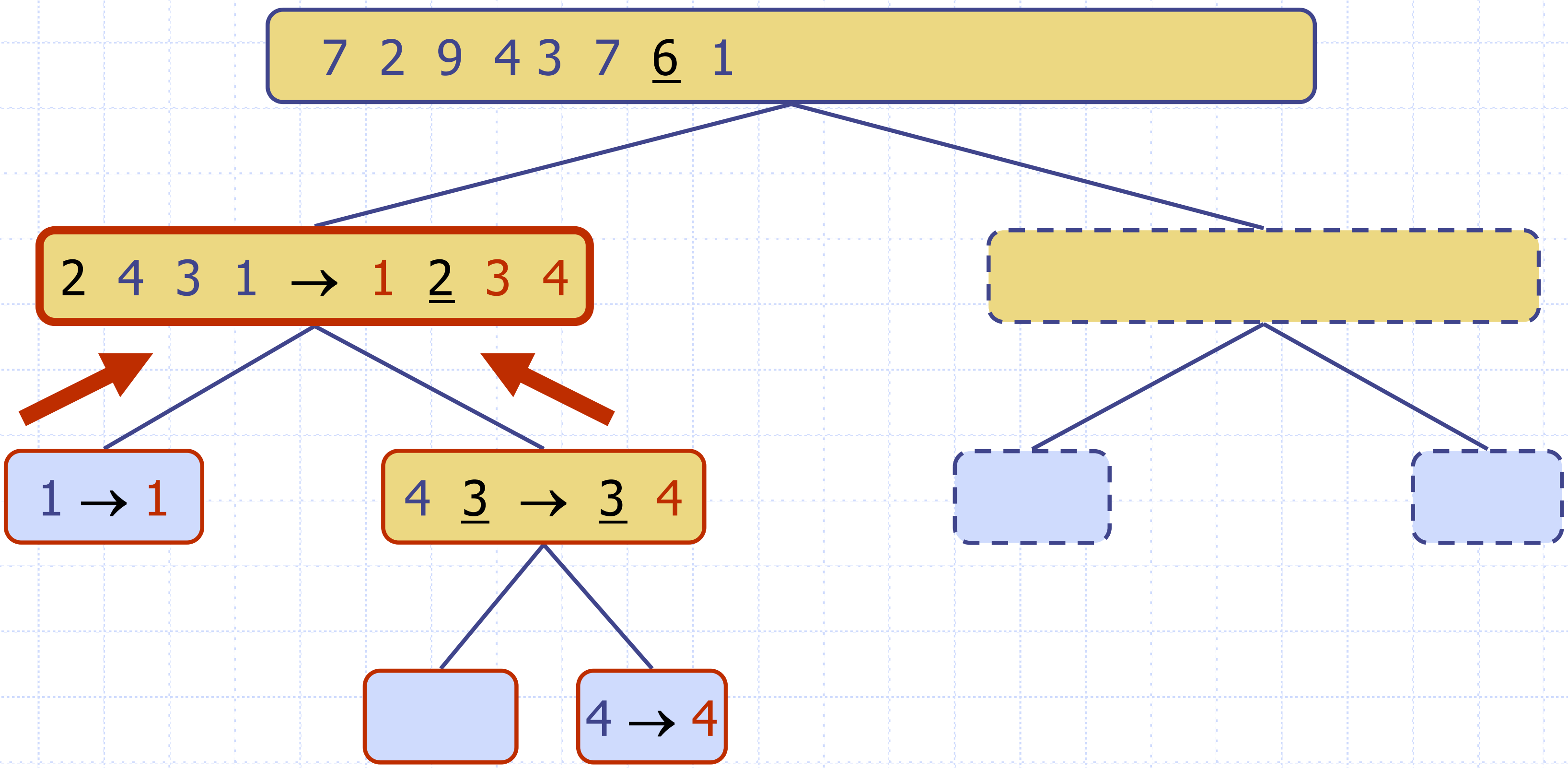
Execution Example (cont.)

- Partition, recursive call, base case



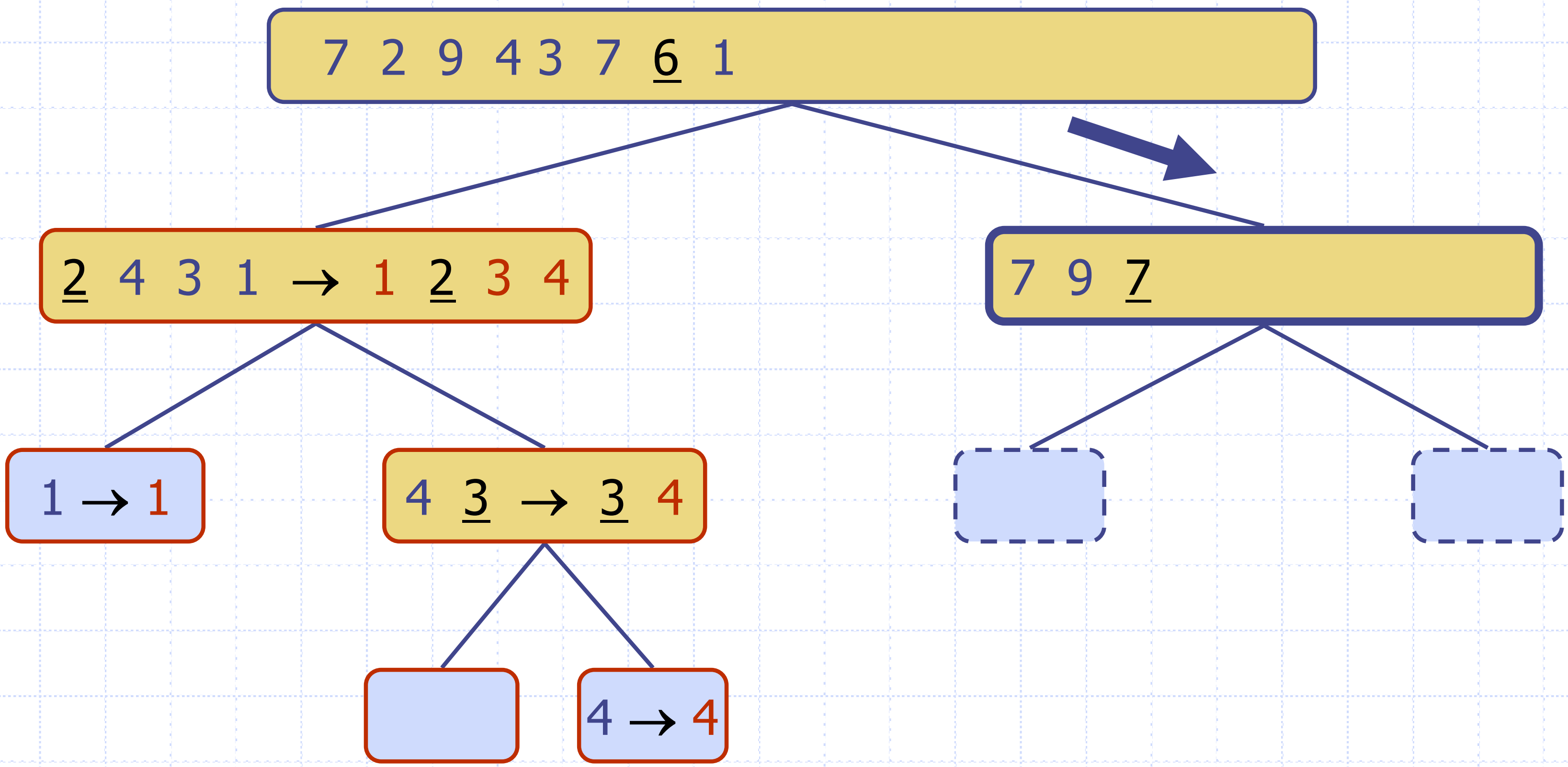
Execution Example (cont.)

- Recursive call, …, base case, join



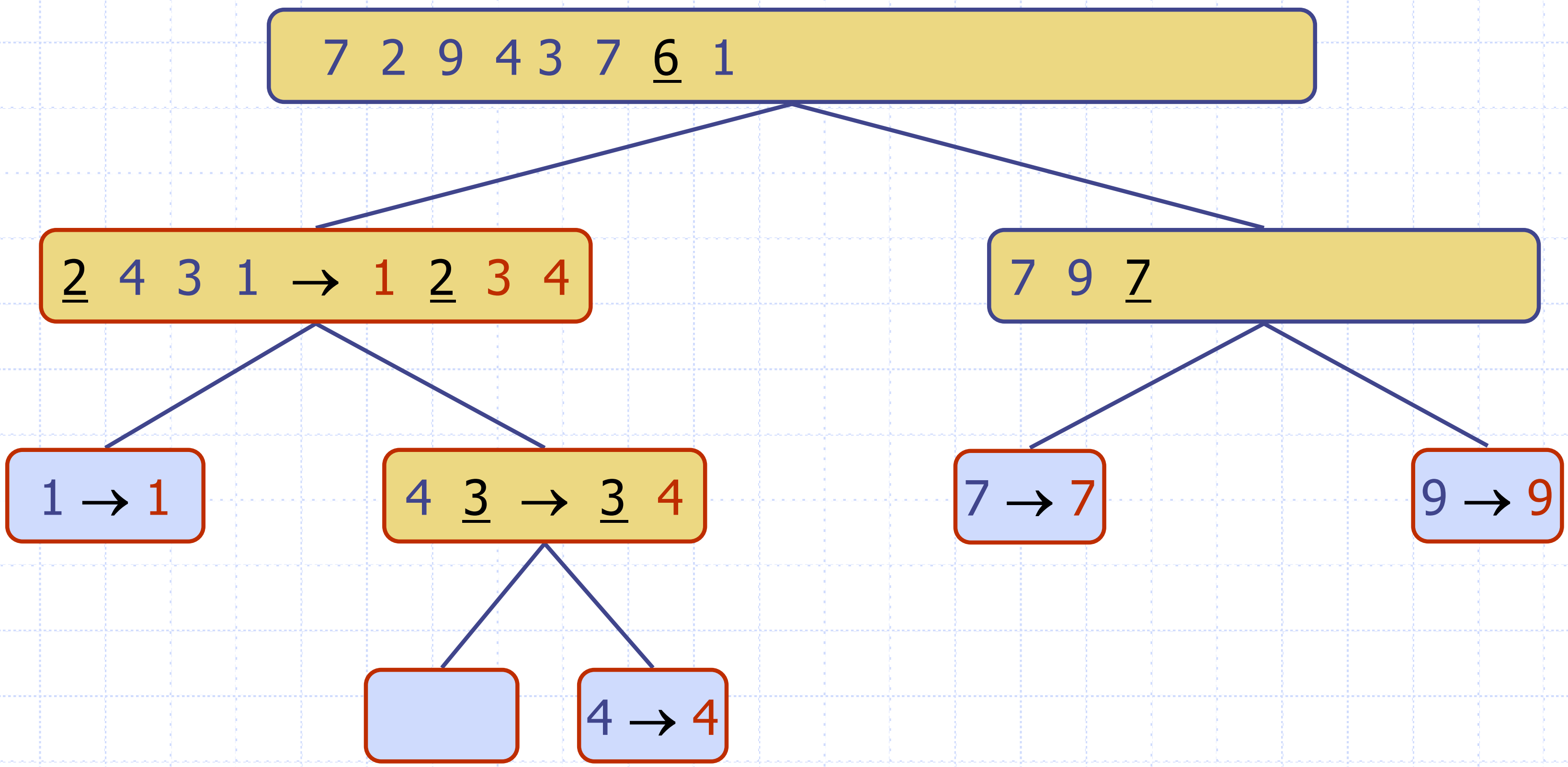
Execution Example (cont.)

- Recursive call, pivot selection



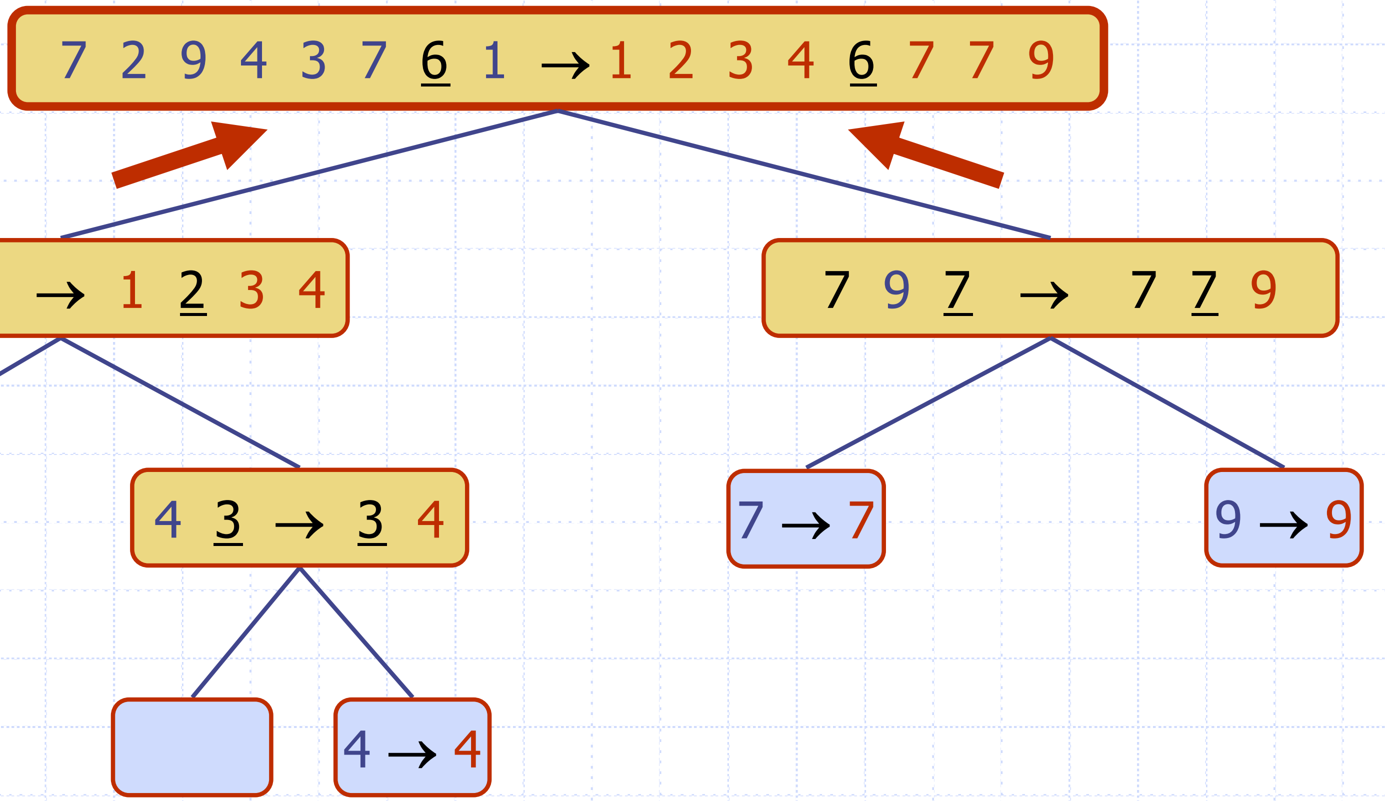
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



QuickSort Visualization

Sorting Algorithms