IT1312

Data Structures & Algorithms

Topic 06: Advanced Sort

version 1.0

(Content adapted from:

- *Data Structures & Algorithms using Python. Rance D. Necaise, Wiley, 1st Edition, 2011.*
- *Data Structures & Algorithms in Python. Michael T. Goodrich, Roberto Tamassia & Michael H. Goldwasser, Wiley, 1st Edition, 2013.)*

Sorting Algorithms

- Basic Sort (Topic 04)
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
- **Advanced Sort (Topic 06)**
  - **Merge Sort**
  - **Quick Sort**

Merge Sort

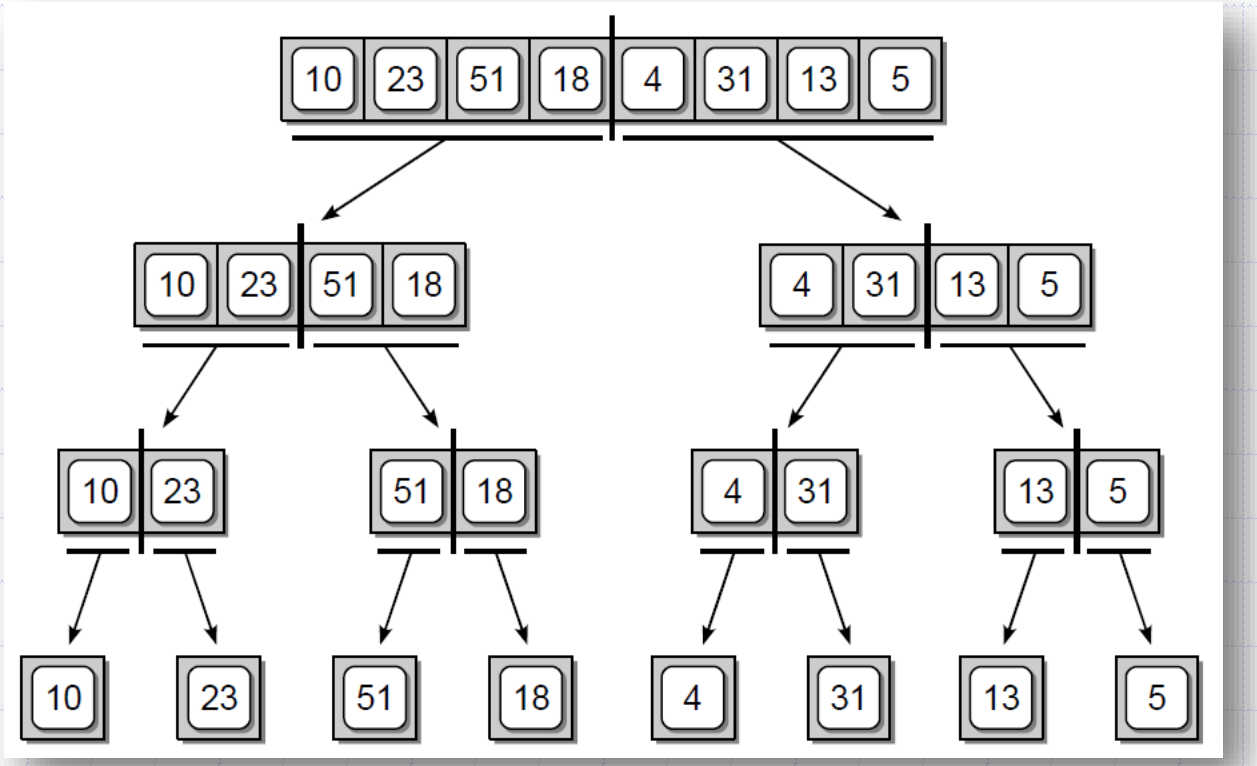
- Merge Sort is a sorting algorithm based on the ***divide-and-conquer*** strategy:
  - Recursively breaking down a problem into two or more sub-problems, until these become simple enough to be solved directly.
  - The solutions to the sub-problems are then combined to give a solution to the original problem.

Merge Sort

- Merge Sort on an input list S with n elements consists of **three steps**:
  - **Divide**: partition S into two sublists S_1 and S_2 of about $n/2$ elements each
  - **Recur**: recursively sort S_1 and S_2
  - **Conquer**: merge S_1 and S_2 into a unique sorted list

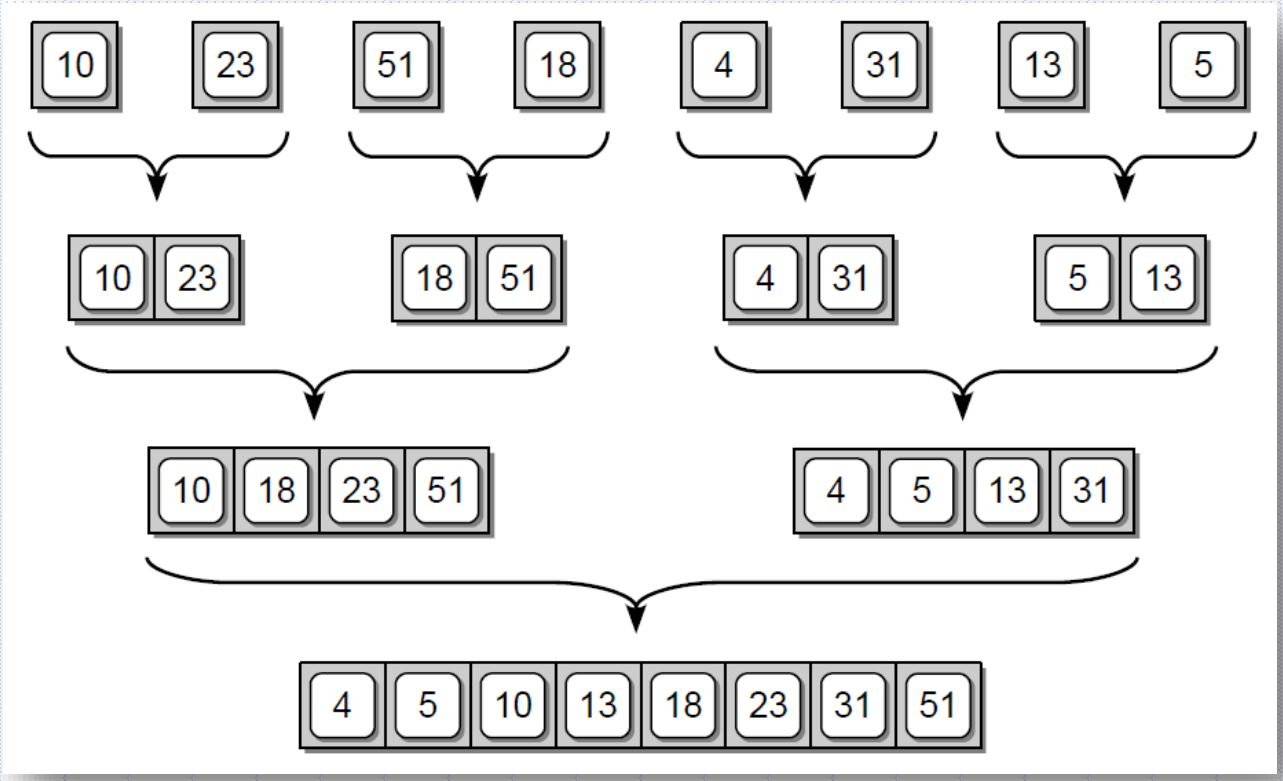
Merge Sort

- **DIVIDE**: Recursively splitting a list until each element is contained within its own list:



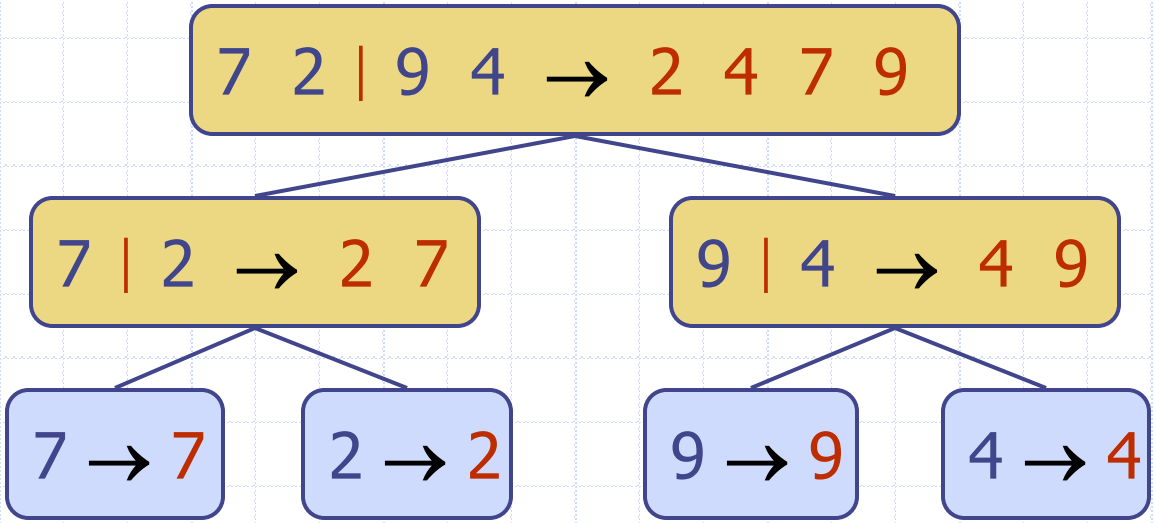
Merge Sort

- **CONQUER**: The sublists are merged back together to create a sorted list:



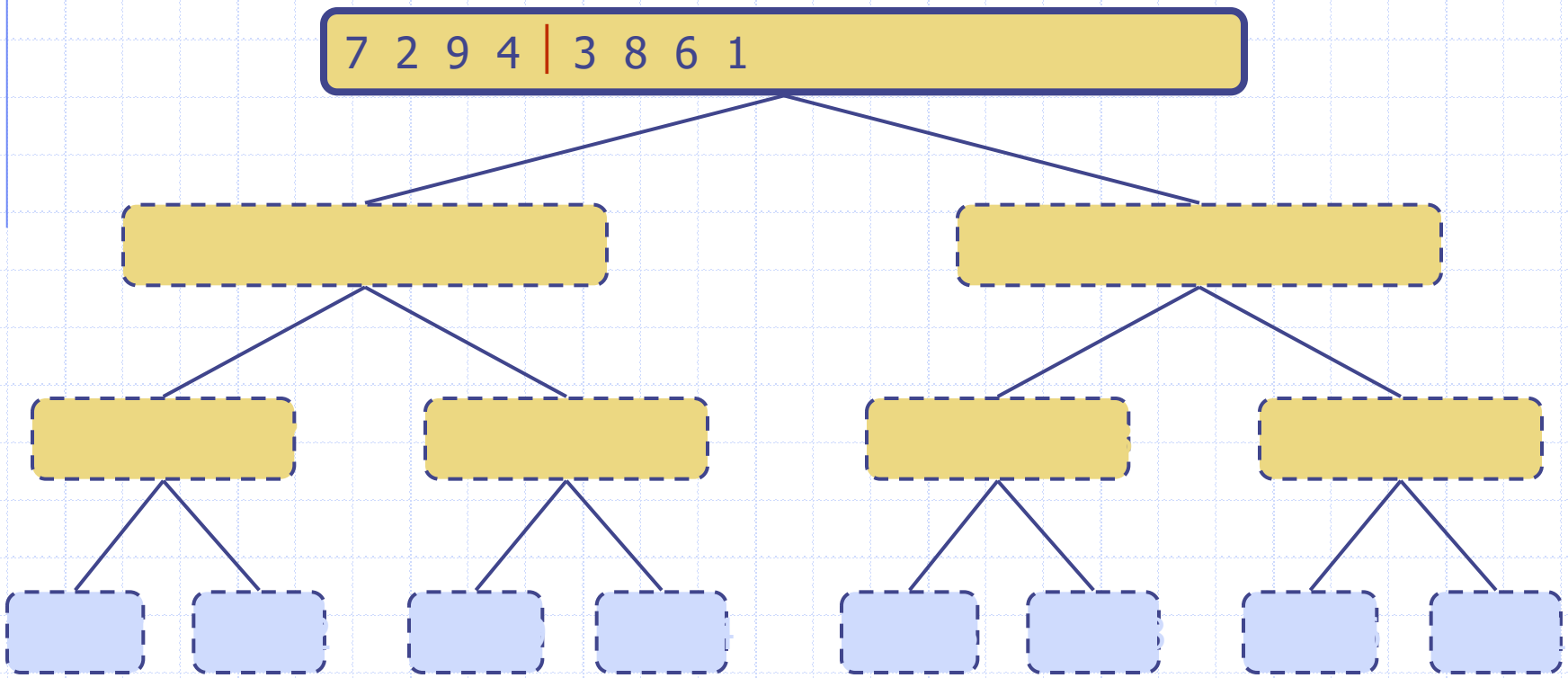
Merge Sort Tree

- An execution of Merge Sort is depicted by a binary tree
  - each node represents a recursive call of Merge Sort and stores
    - unsorted list before the execution and its partition
    - sorted list at the end of the execution
  - the root is the initial call
  - the leaves are calls on sublists of size 0 or 1



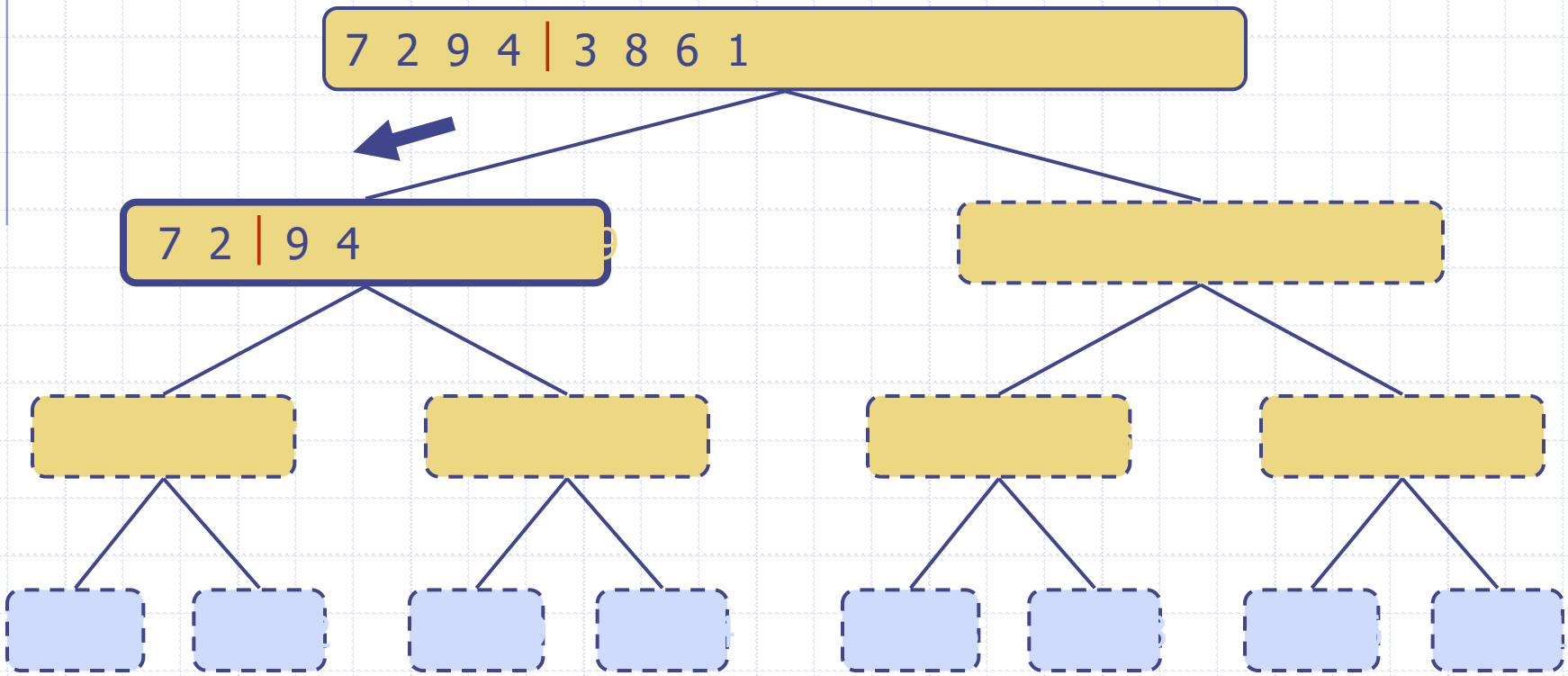
Execution Example

- Partition



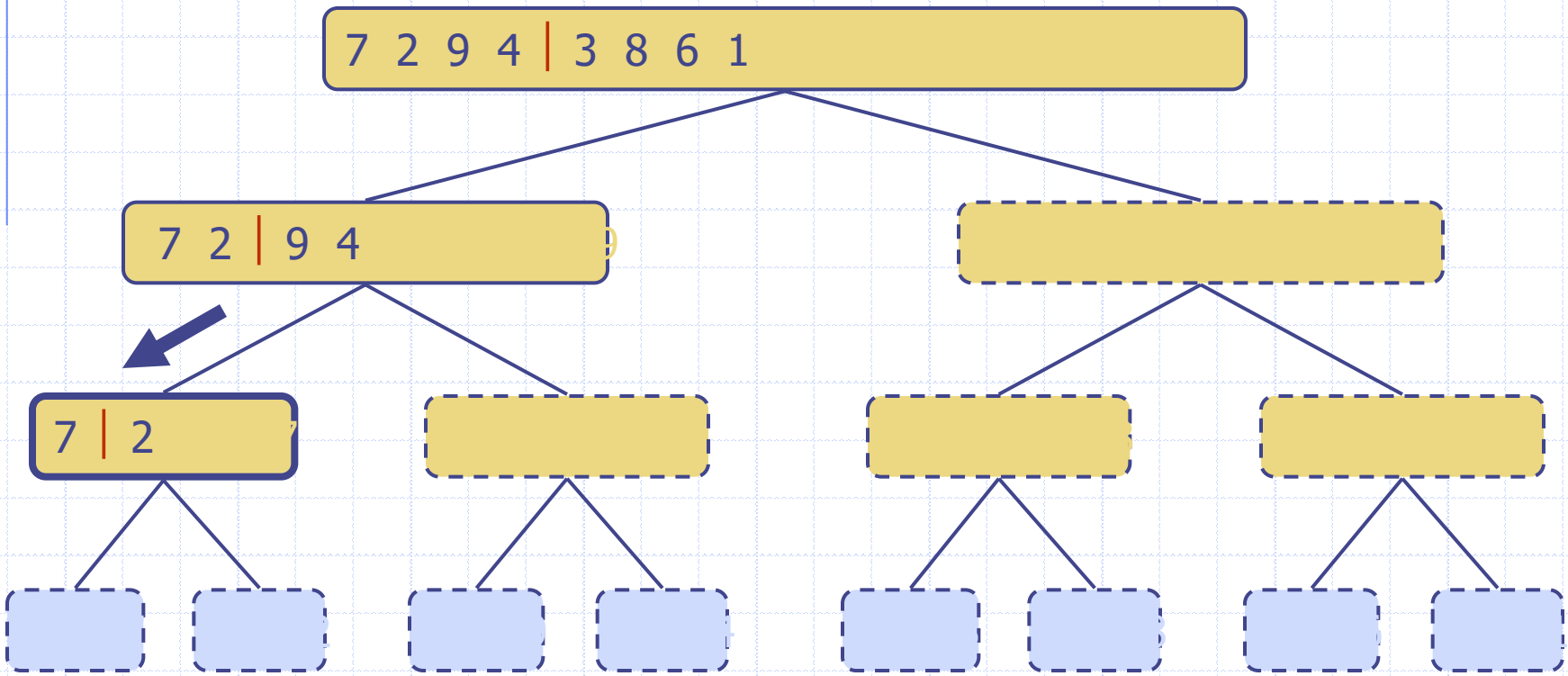
Execution Example (cont.)

- Recursive call, partition



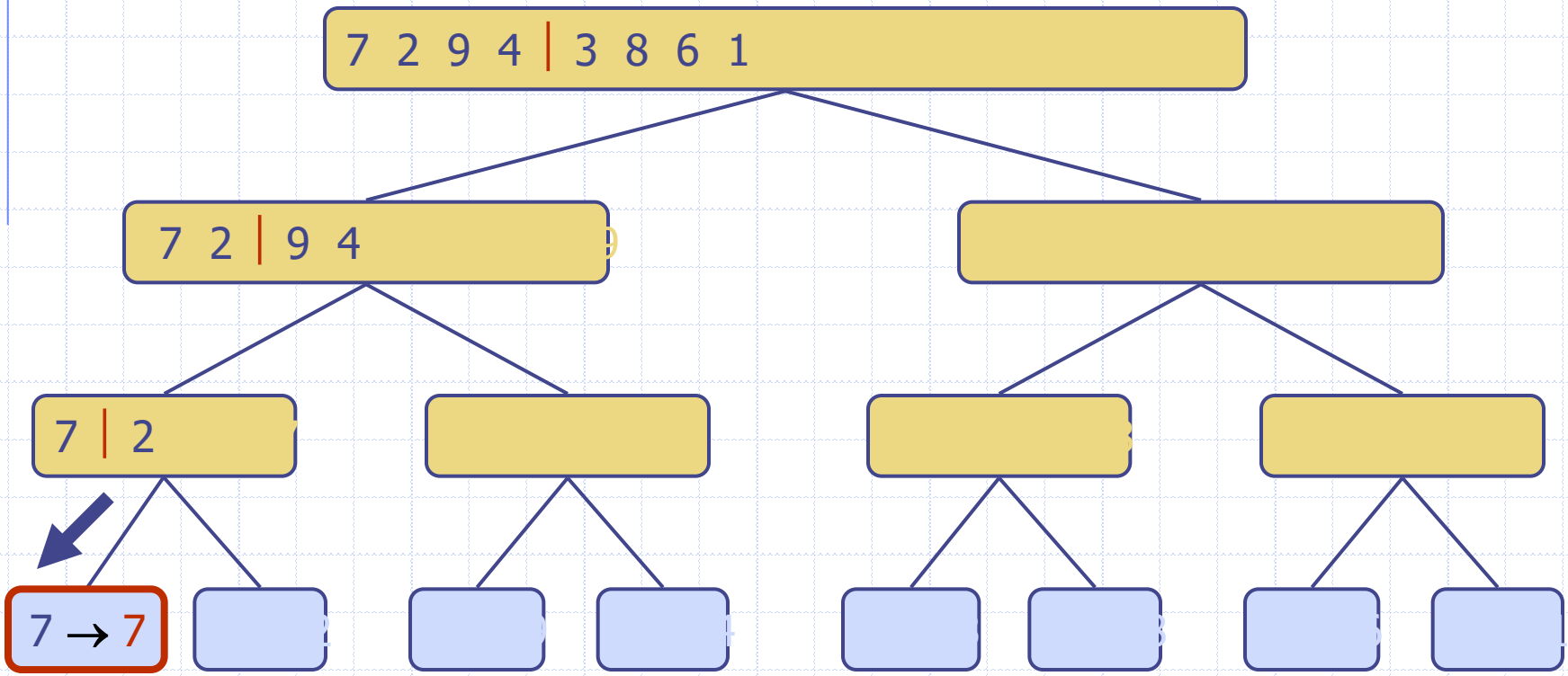
Execution Example (cont.)

- Recursive call, partition



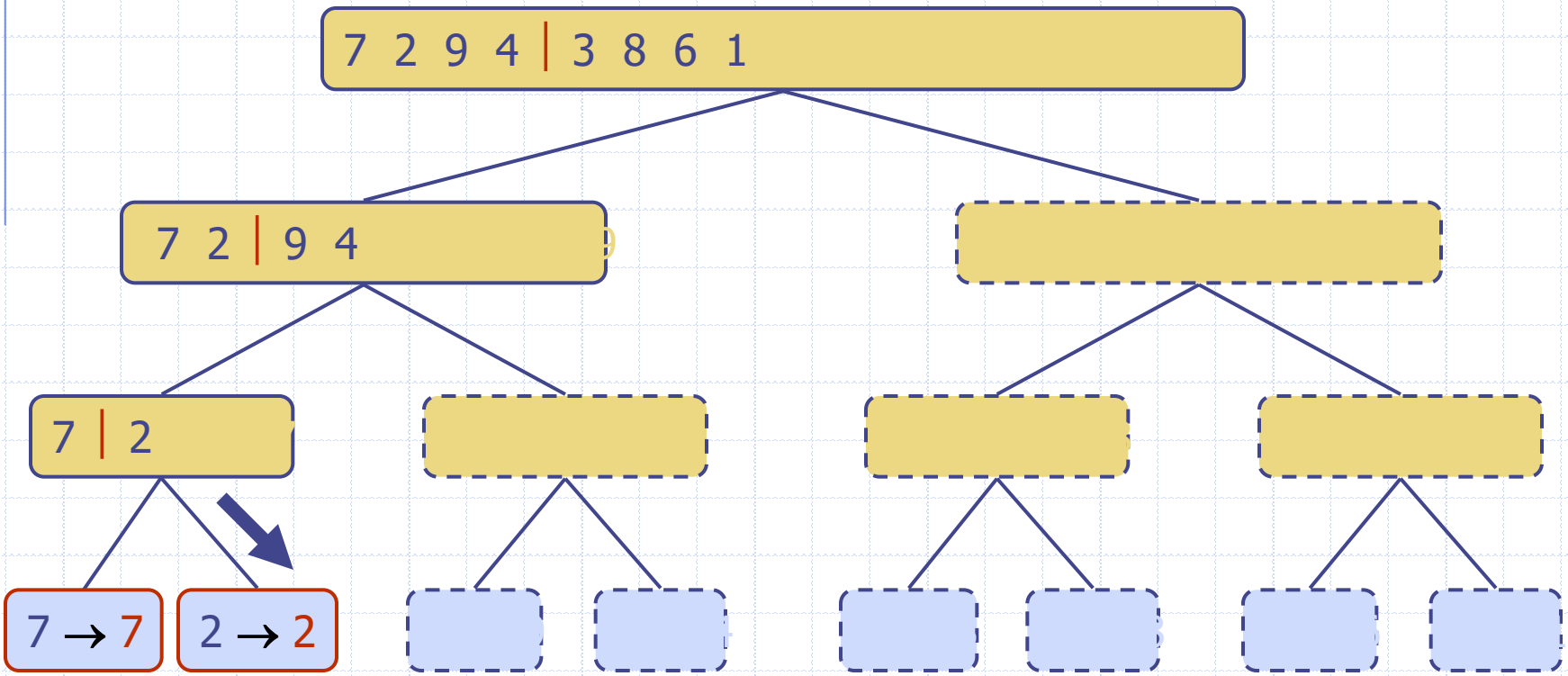
Execution Example (cont.)

- Recursive call, base case



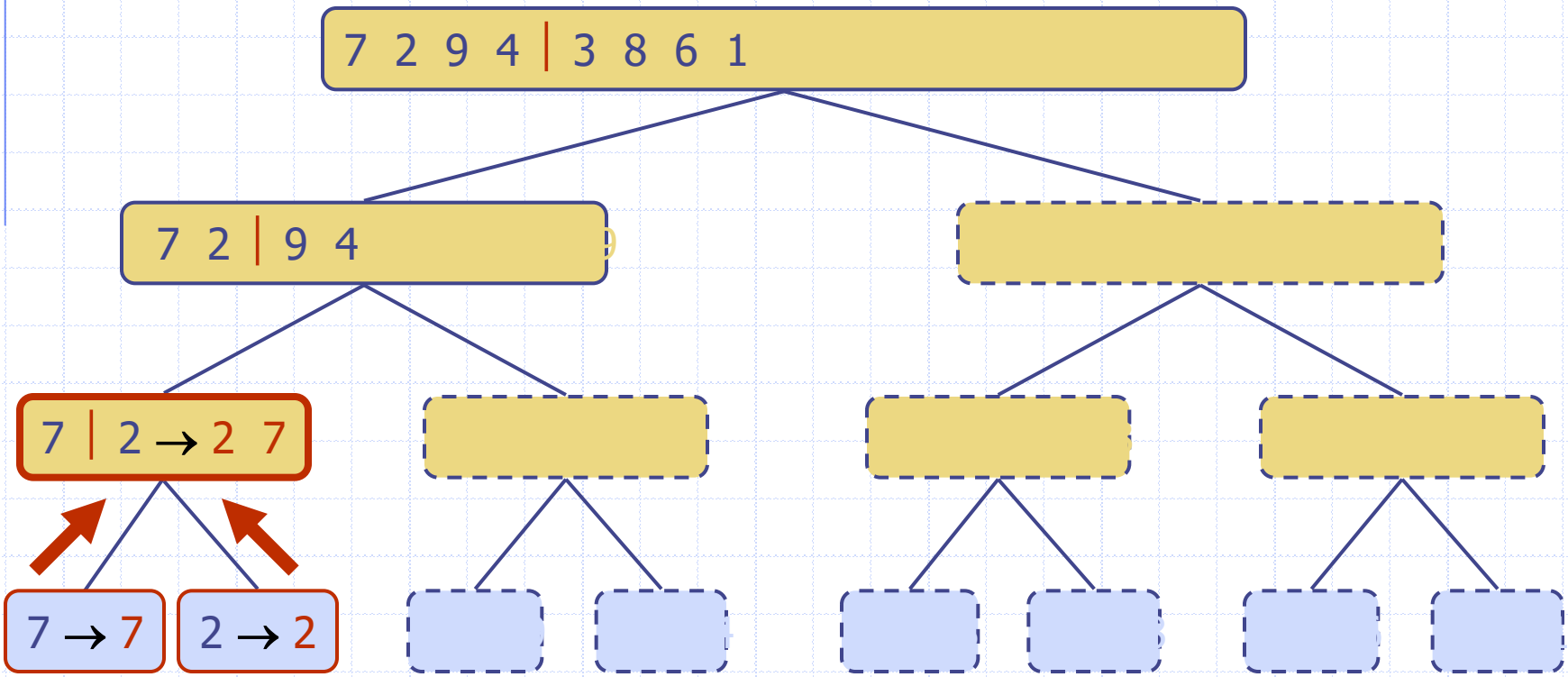
Execution Example (cont.)

- Recursive call, base case



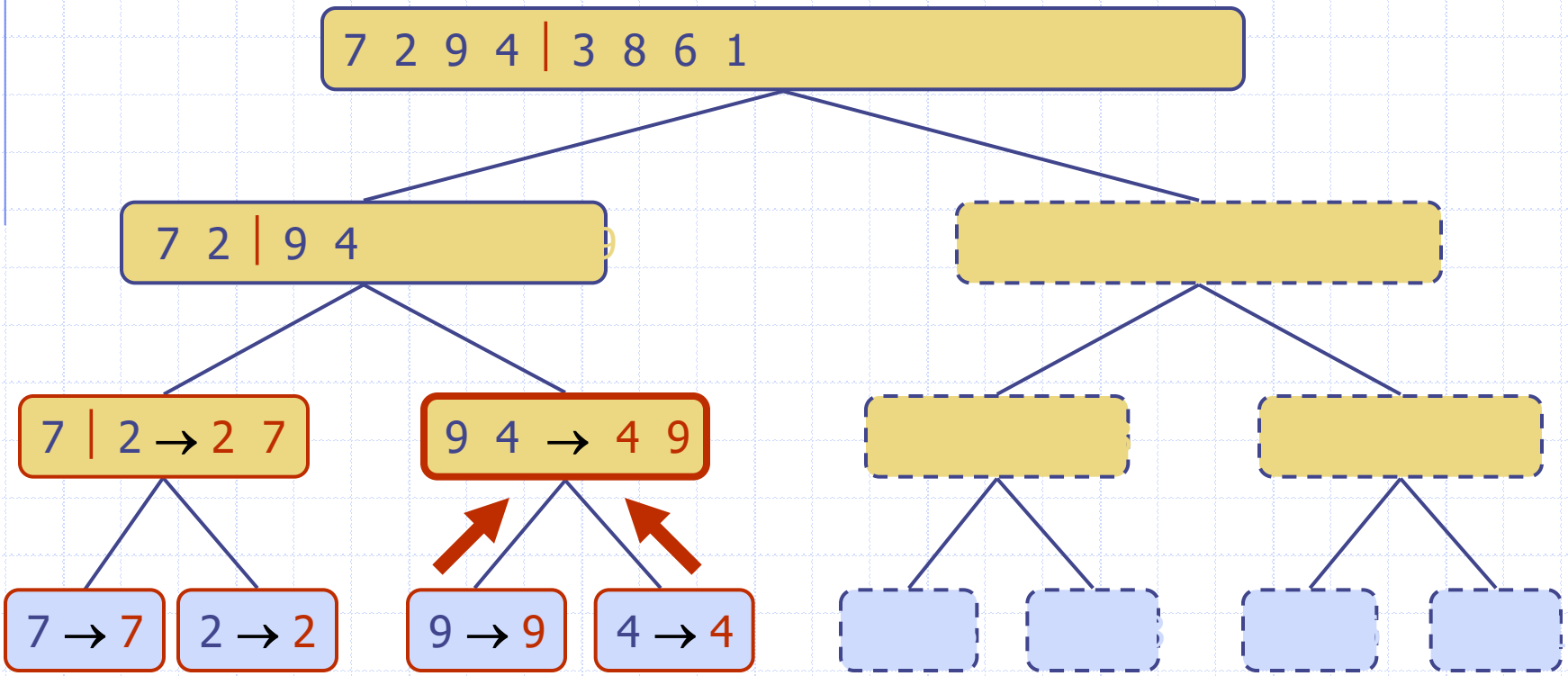
Execution Example (cont.)

- Merge



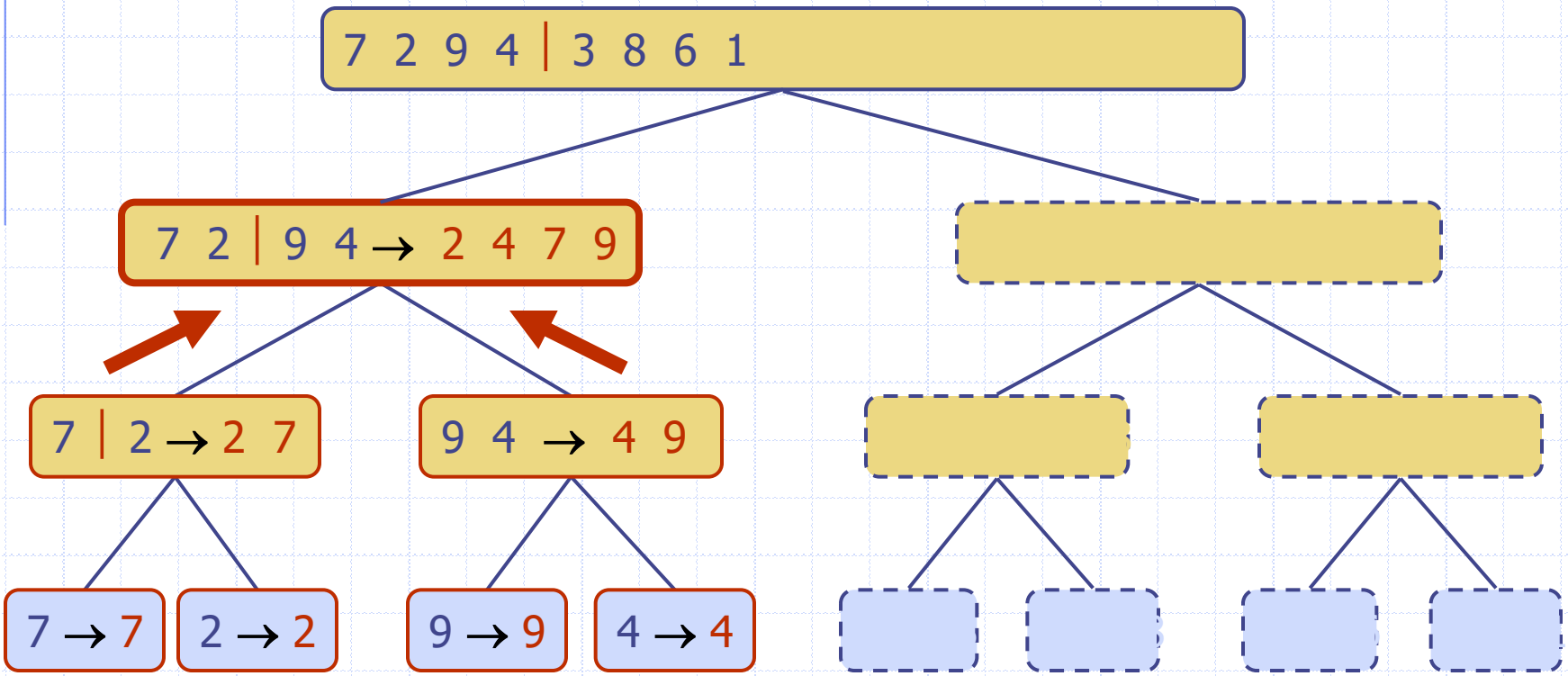
Execution Example (cont.)

- Recursive call, …, base case, merge



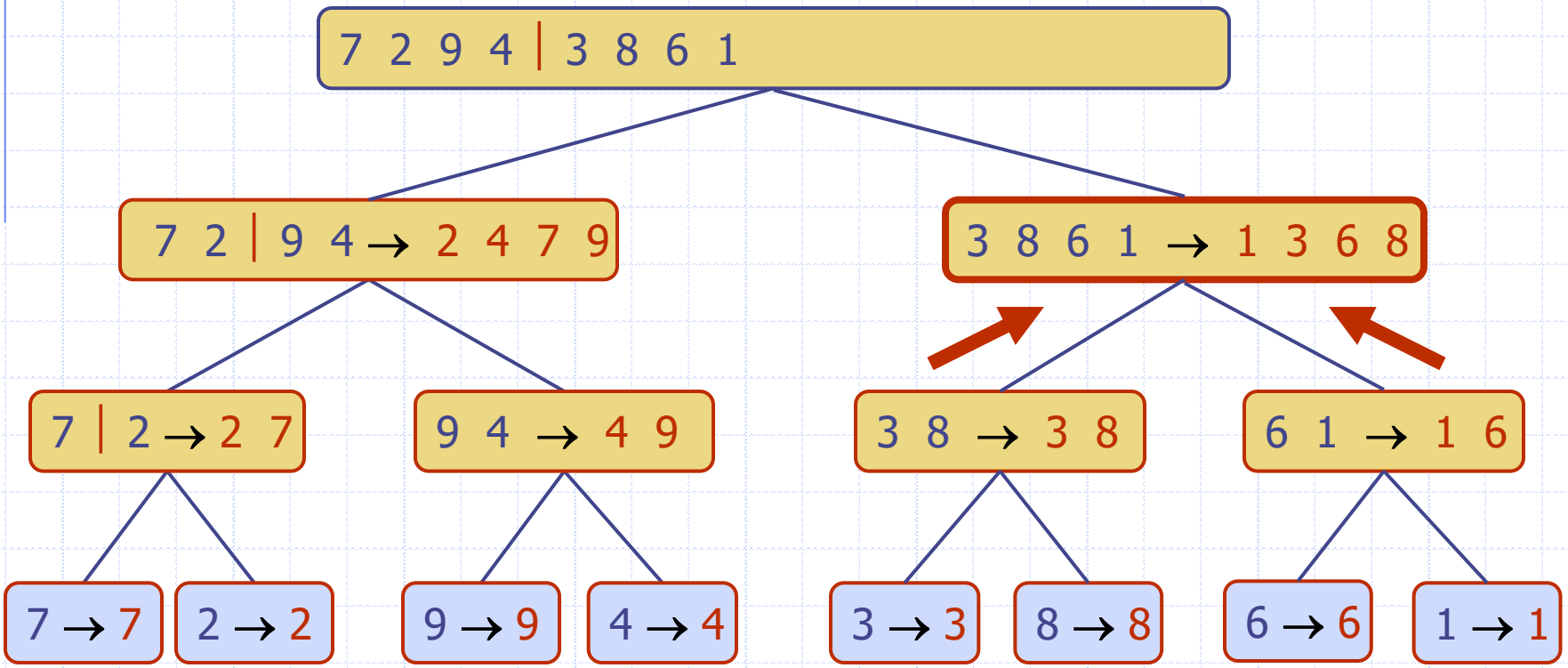
Execution Example (cont.)

- Merge



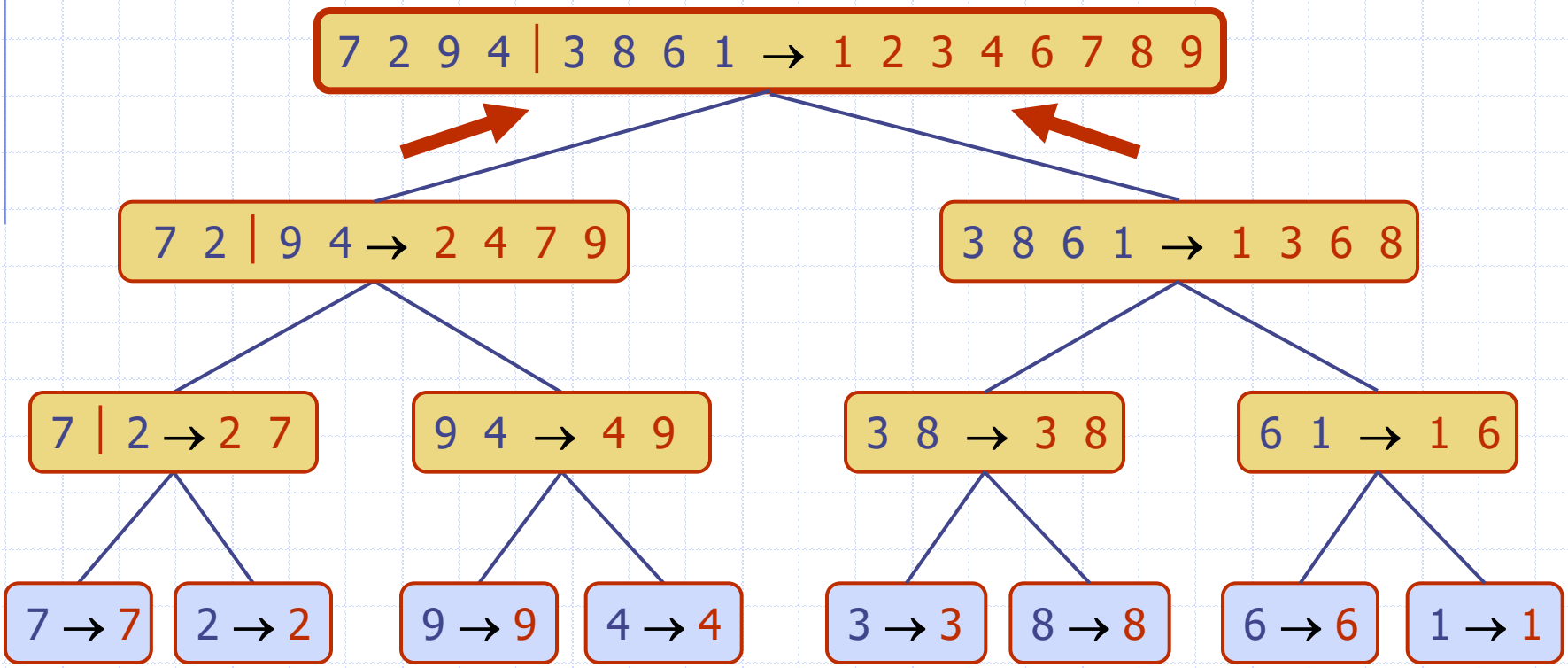
Execution Example (cont.)

- Recursive call, ..., merge, merge



Execution Example (cont.)

- Merge

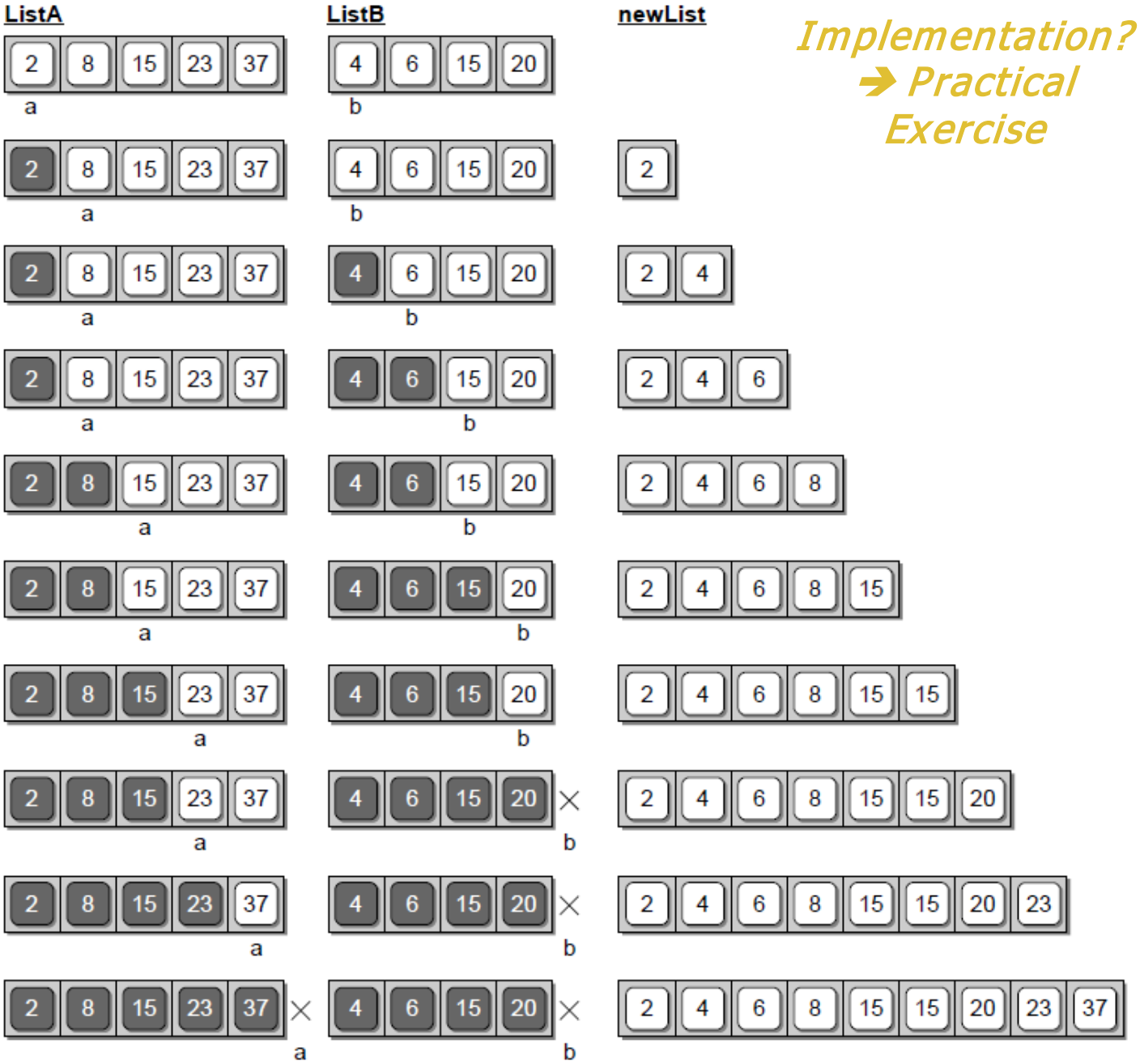


Merge Sort

```
# Sorts a Python list in ascending order using
# the merge sort algorithm
def mergeSort( theList ):
    # Check the base case - the list contains a single item
    if len(theList) <= 1:
        return theList
    else:
        # Compute the midpoint
        mid = len(theList) // 2

        # Split the list and perform the recursive step
        leftHalf= mergeSort( theList[ :mid ] )
        rightHalf = mergeSort( theList[ mid: ] )

        # Merge the two sorted sublists
        newList = mergeSortedLists( leftHalf, rightHalf)
        return newList
```

Merging Two Sorted Lists

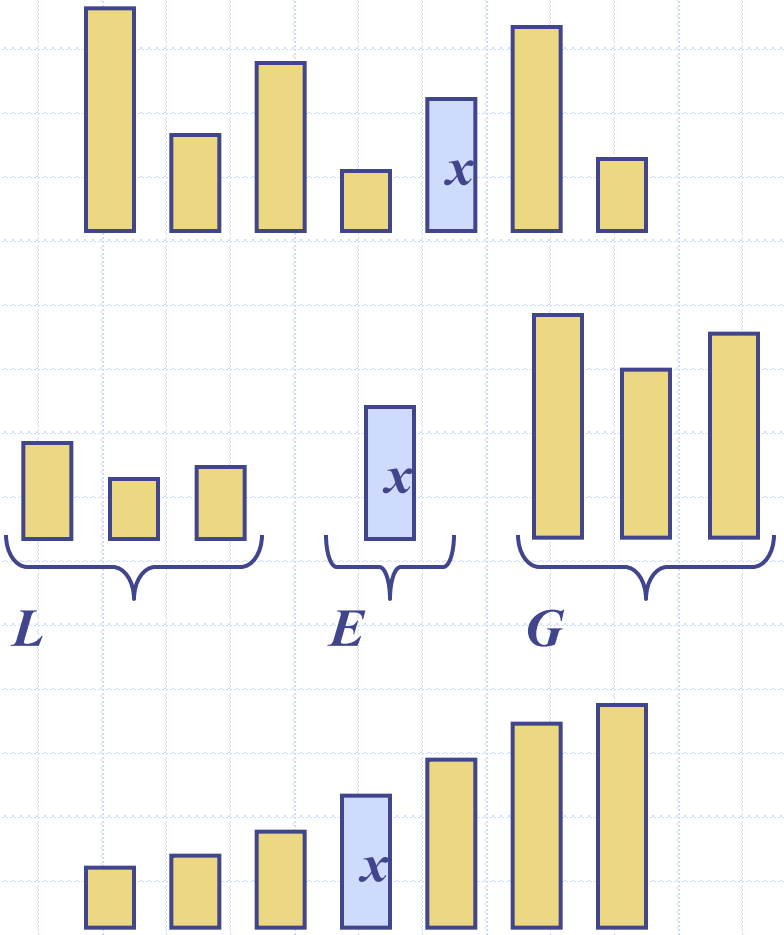
*NOTE: **a** and **b** are index variables indicating the next value to be merged from the respective list.*



Implementation? ➔ Practical Exercise

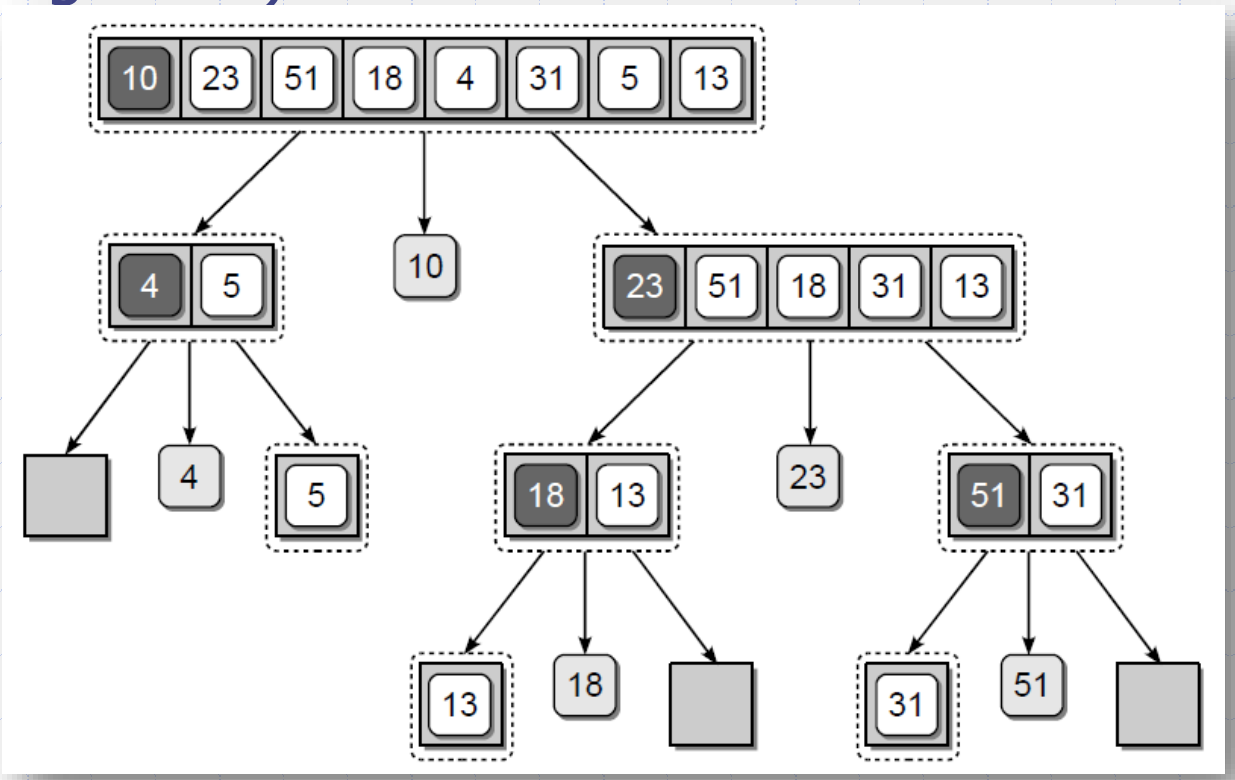
Quick Sort

- Quick-sort is also a sorting algorithm based on the *divide and conquer* strategy:
  - 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



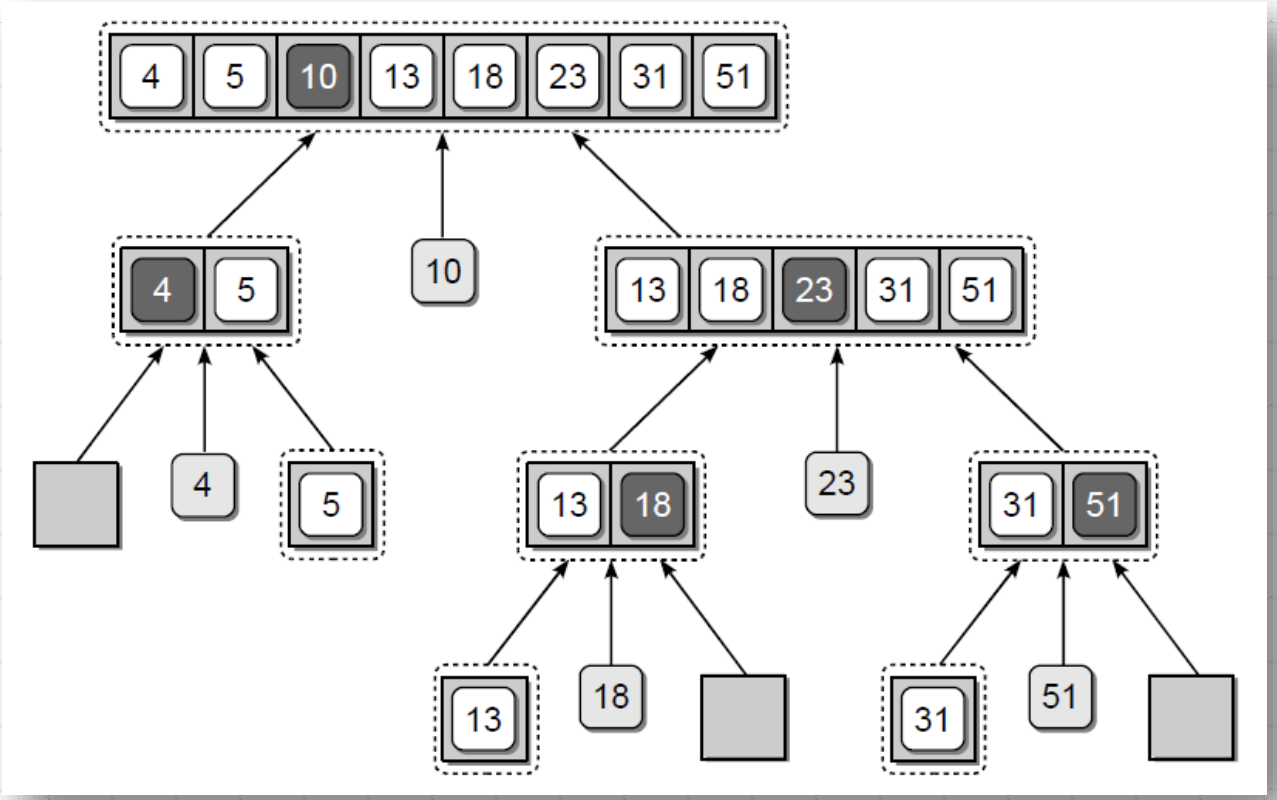
Quick Sort

- **DIVIDE**: Partitions the sequence into segments based on the pivot value (shown in gray background):



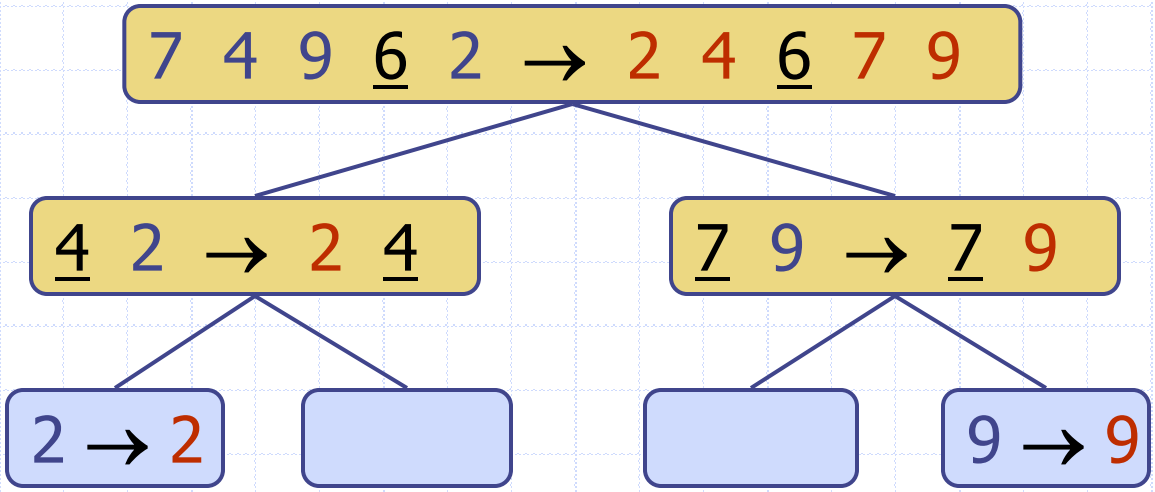
Quick Sort

- **CONQUER**: Merges the sorted segments and pivot value back into the original sequence:



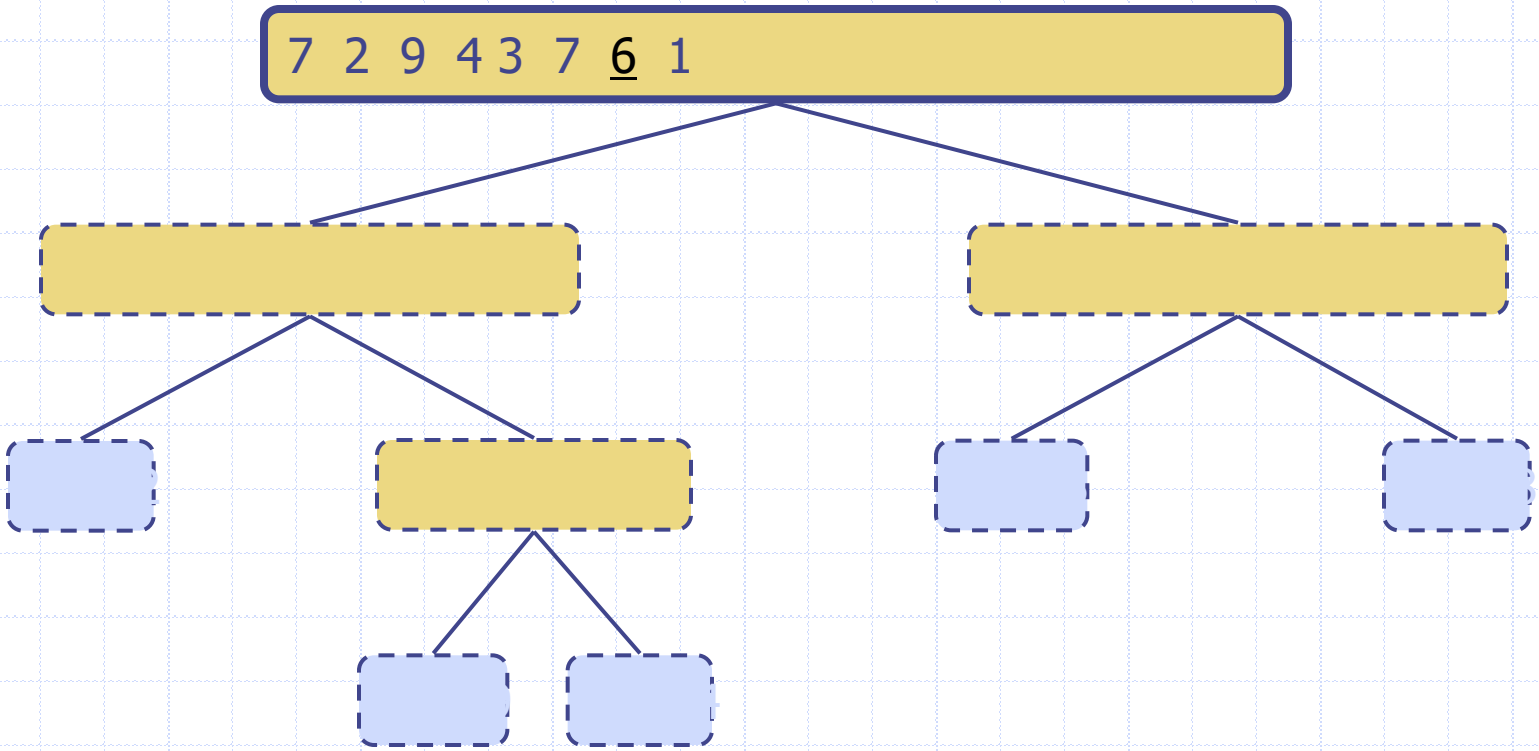
Quick-Sort Tree

- 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



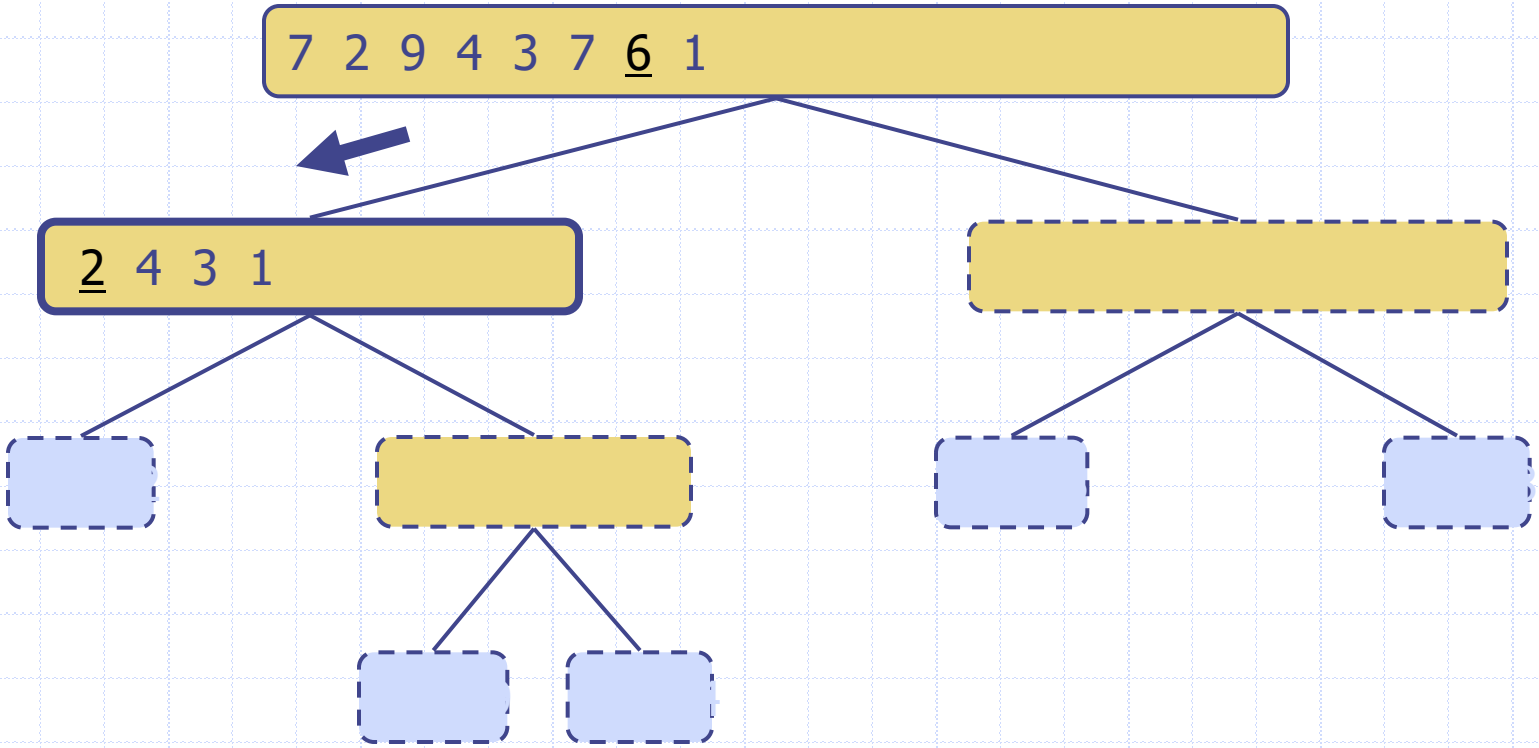
Execution Example

- Pivot selection



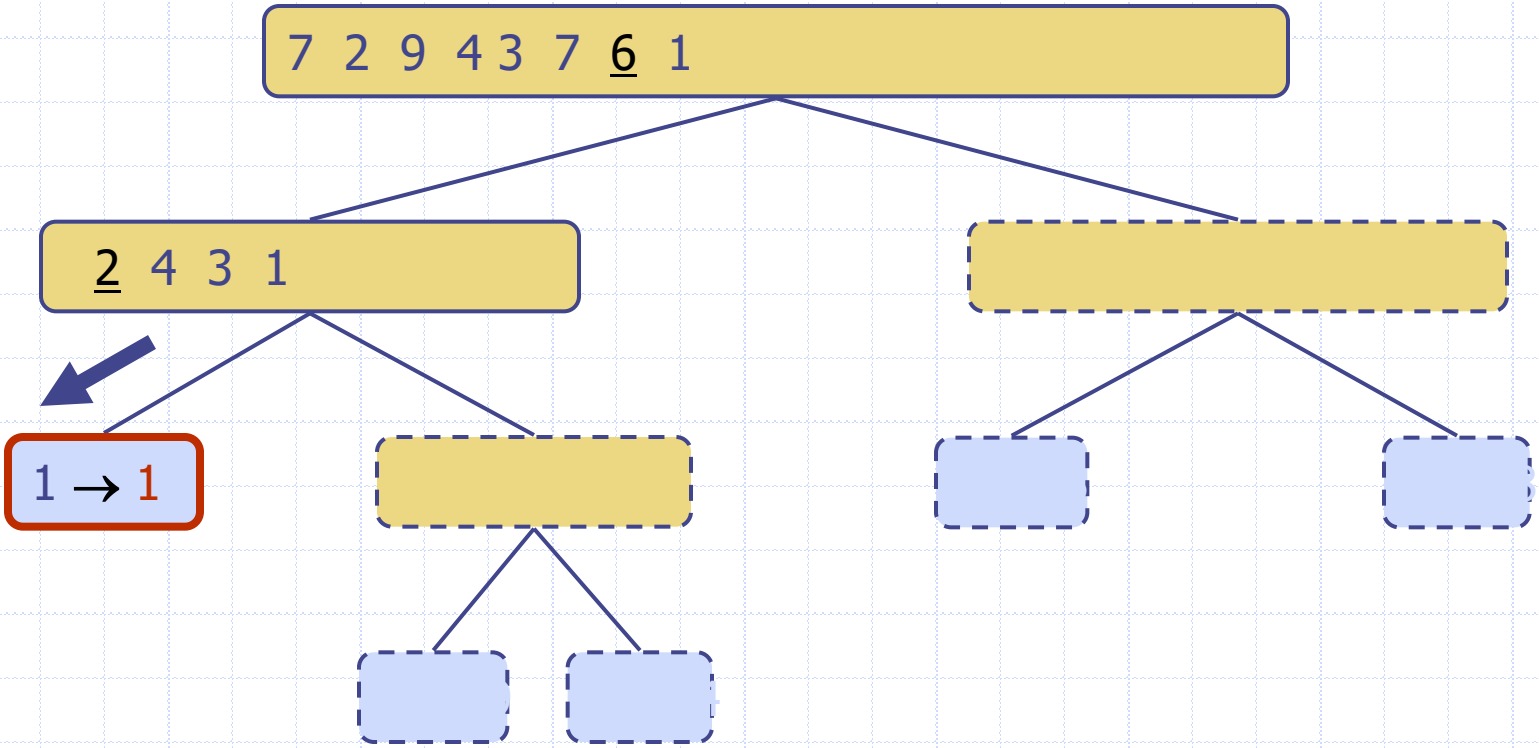
Execution Example (cont.)

- Partition, recursive call, pivot selection



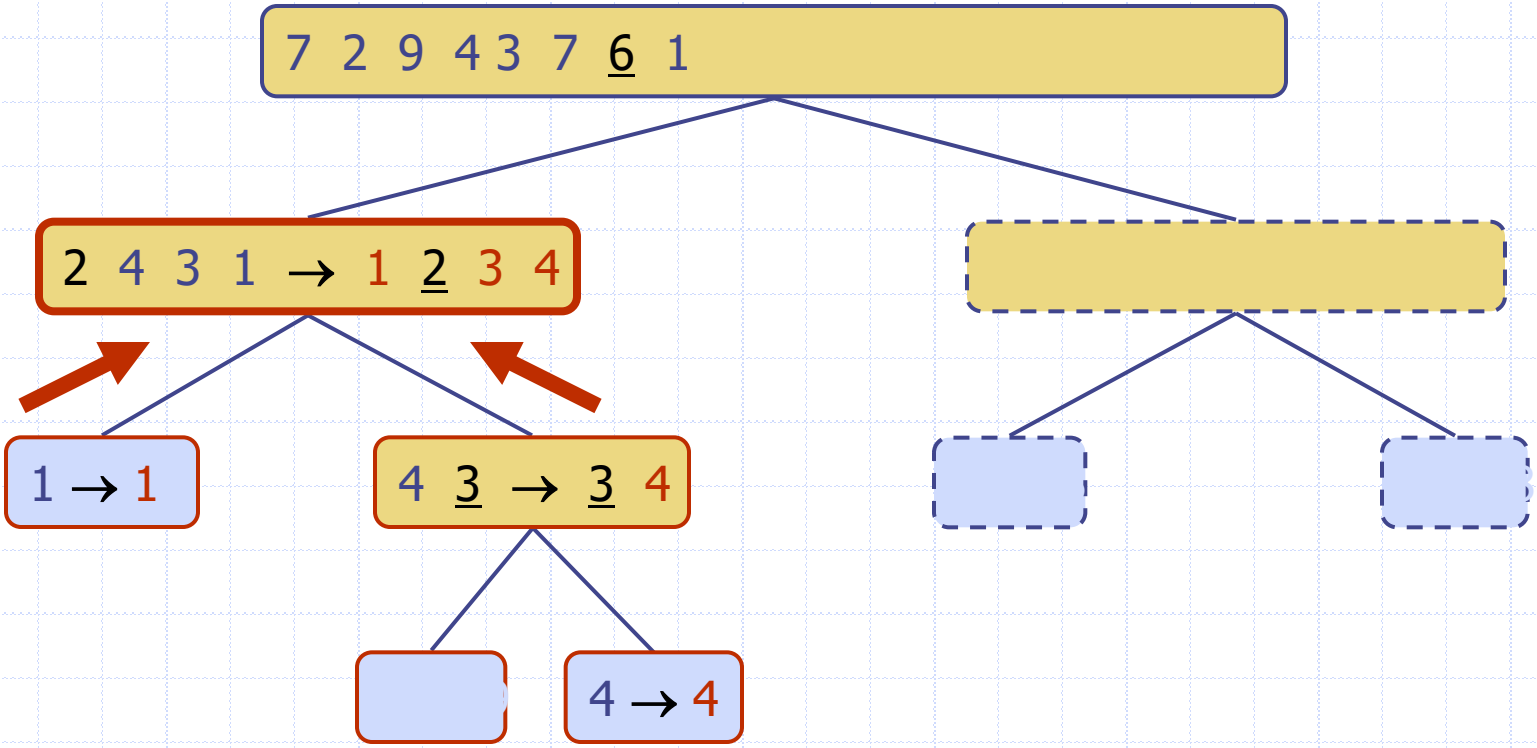
Execution Example (cont.)

- Partition, recursive call, base case



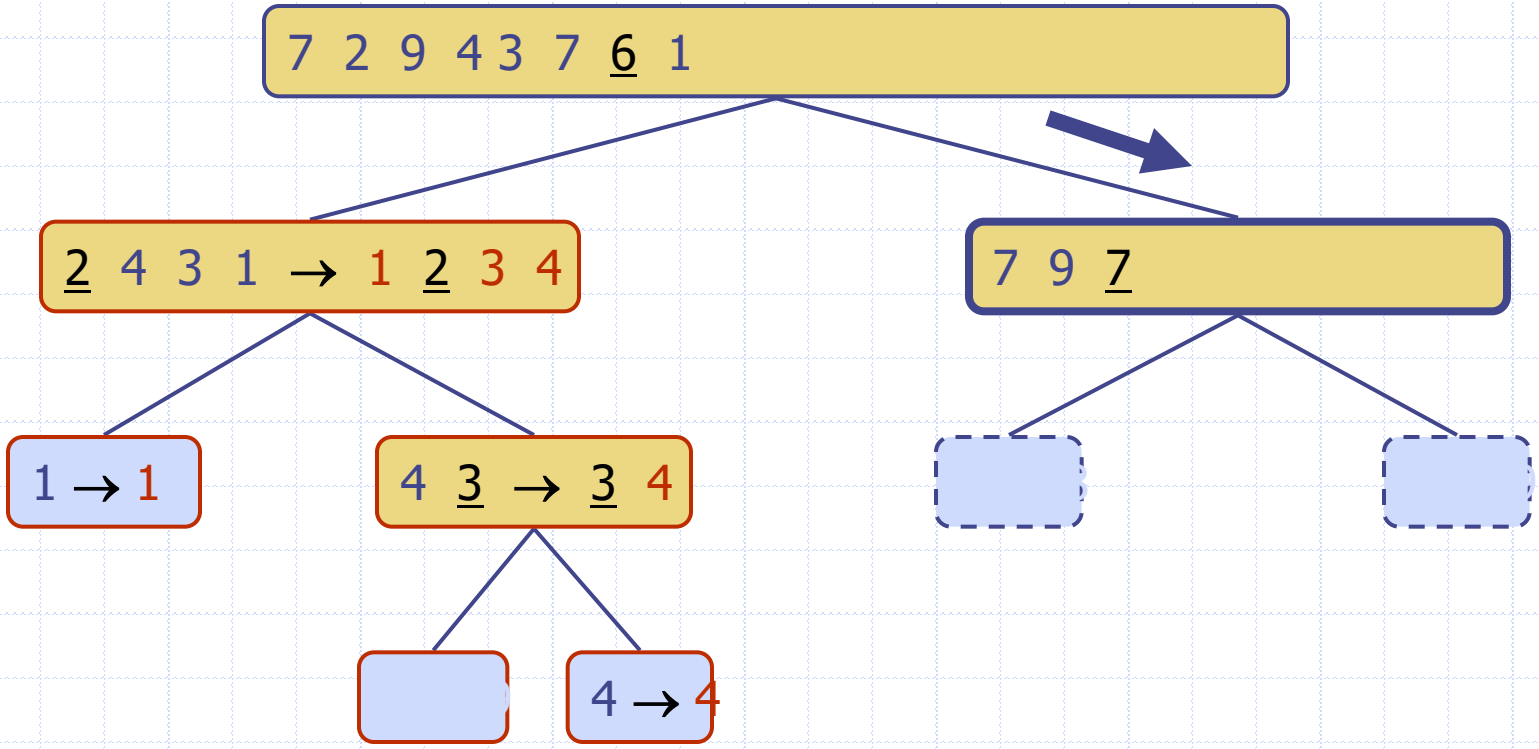
Execution Example (cont.)

- Recursive call, …, base case, join



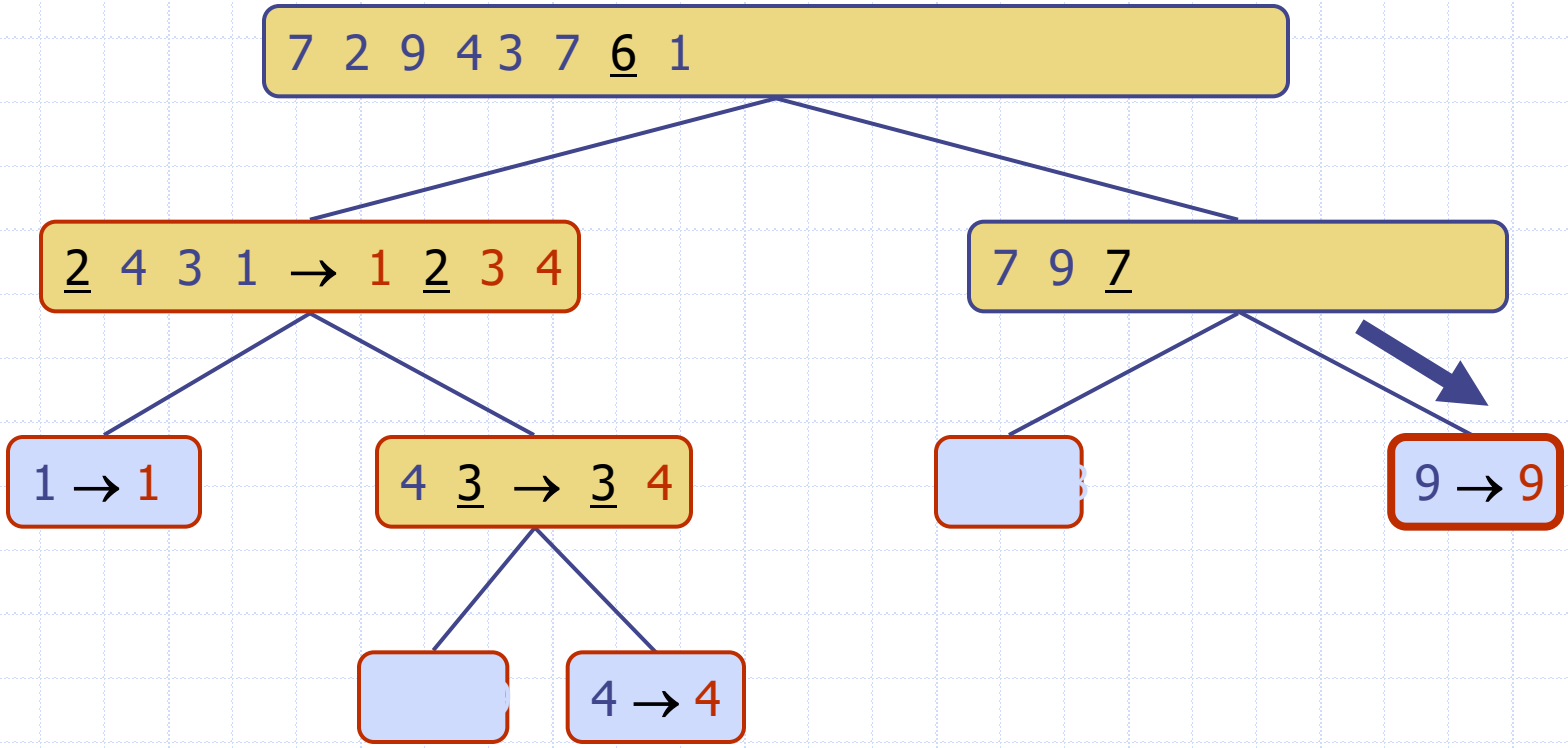
Execution Example (cont.)

- Recursive call, pivot selection



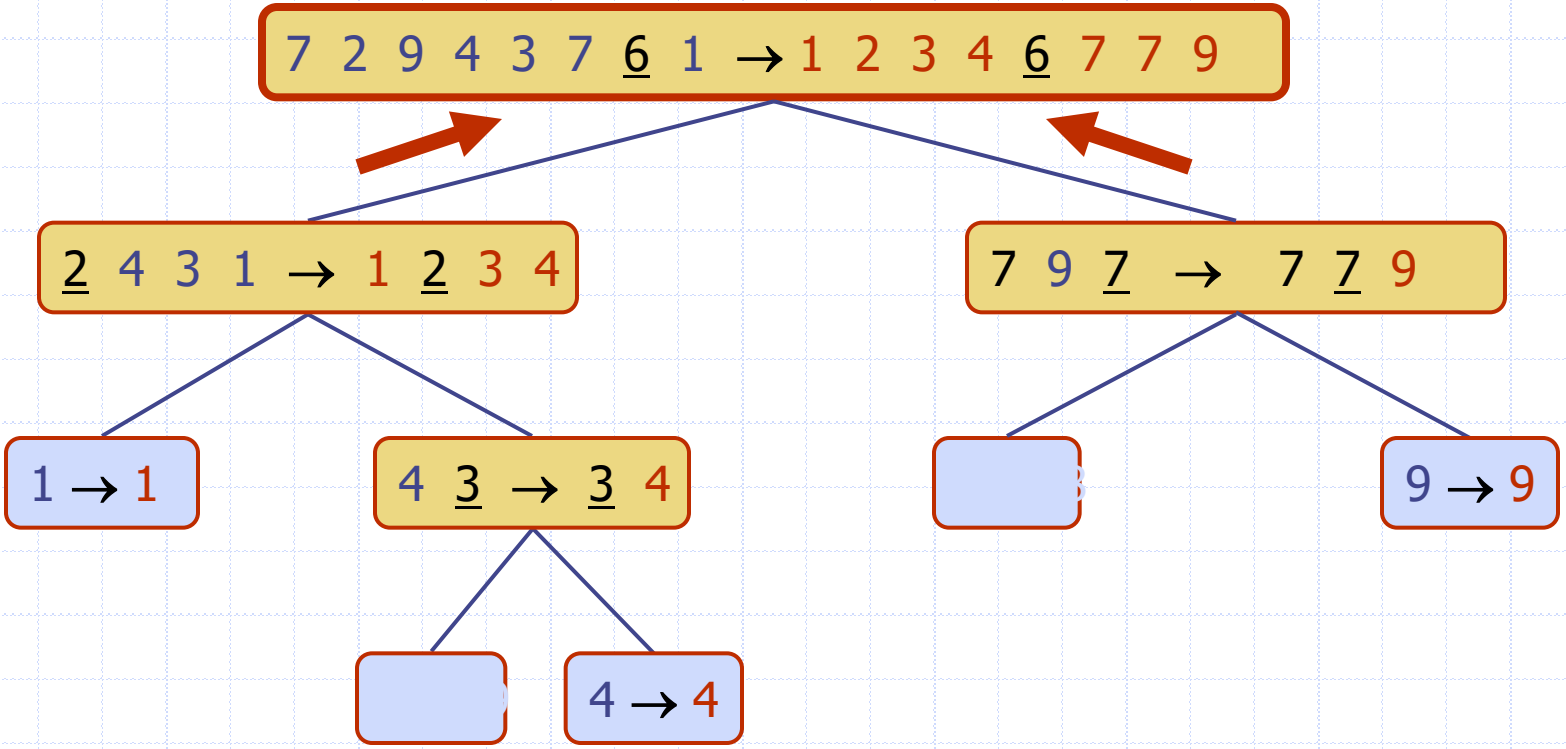
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



Quick Sort (Part 1 of 2)

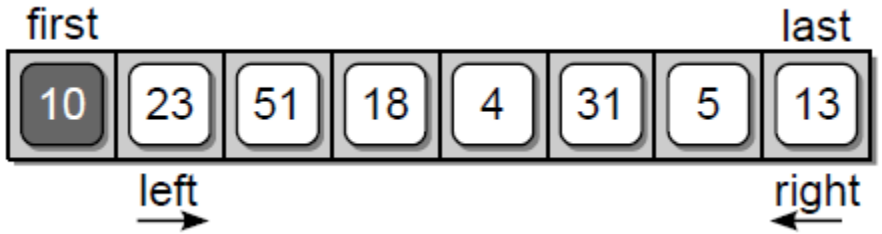
```
# Sorts an array or list using the recursive quick sort algorithm
def quickSort( theSeq ):
    n = len( theSeq )
    recQuickSort( theSeq, 0, n-1 )


# The recursive "in-place" implementation
def recQuickSort( theSeq, first, last ):
    # Check the base case (range is trivially sorted)
    if first >= last:
        return
    else:
        # Partition the sequence and obtain the pivot position
        pos = partitionSeq( theSeq, first, last )

        # Repeat the process on the two subsequences
        recQuickSort( theSeq, first, pos - 1 )
        recQuickSort( theSeq, pos + 1, last )
```

Quick Sort (Part 2 of 2)

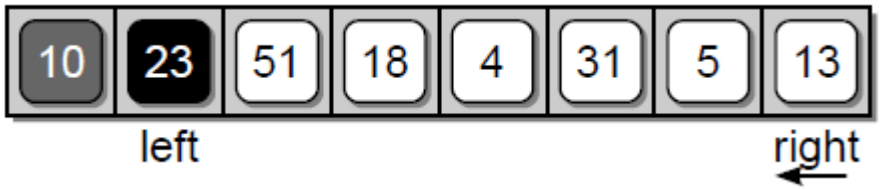
```
# Partitions the subsequence using the first key as the pivot
def partitionSeq(theSeq, first, last):
    # Save a copy of the pivot value.
    pivot = theSeq[first]    # first element of range is pivot

    # Find the pivot position and move the elements around the pivot
    left = first + 1         # will scan rightward
    right = last             # will scan leftward
    while left <= right:
        # Scan until reaches value equal or larger than pivot (or right marker)
        while left <= right and theSeq[left] < pivot:
            left += 1

        # Scan until reaches value equal or smaller than pivot (or left marker)
        while left <= right and theSeq[right] > pivot:
            right -= 1

        # Scans did not strictly cross
        if left <= right:
            # swap values
            theSeq[left], theSeq[right] = theSeq[right], theSeq[left]

            # Shrink range (Recursion: Progress towards base case)
            left += 1
            right -= 1

    # Put the pivot in the proper position (marked by the right index)
    theSeq[first], theSeq[right] = theSeq[right], pivot

    # Return the index position of the pivot value.
    return right
```

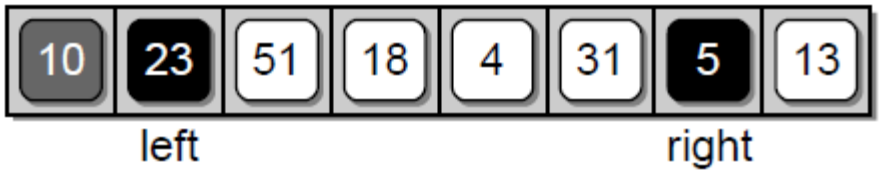
partitionSeq()

- First element is the **pivot** value.
- `left` marker initialised to the first position following the pivot.
- `right` marker initialised to the last position within the segment.



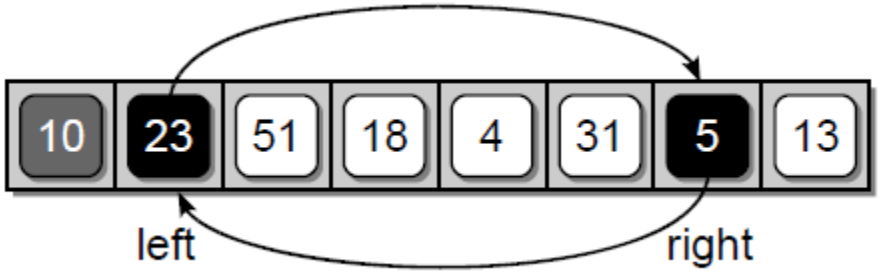
- The main loop is executed until one of the two markers crosses the other.
- `left` marker is shifted to the right until a key value equal or larger than the pivot is reached, or the `left` marker crosses the `right` marker.
- `right` marker is shifted to the left until a key value equal or lesser than the pivot is reached, or the `right` marker crosses the `left` marker.



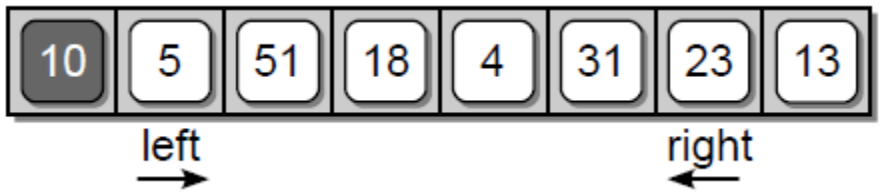


partitionSeq()

- The two keys located at the positions marked by `left` and `right` are then swapped, which will place them within the proper segment once the location of the pivot is found.

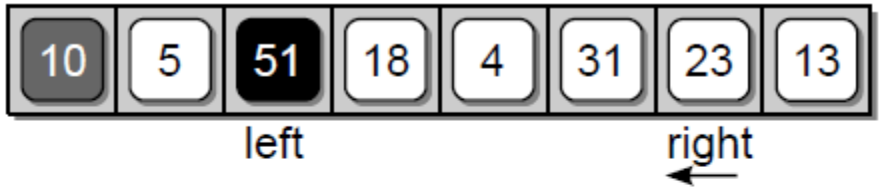


- After the two keys are swapped, the two markers are again shifted starting from where they left off.

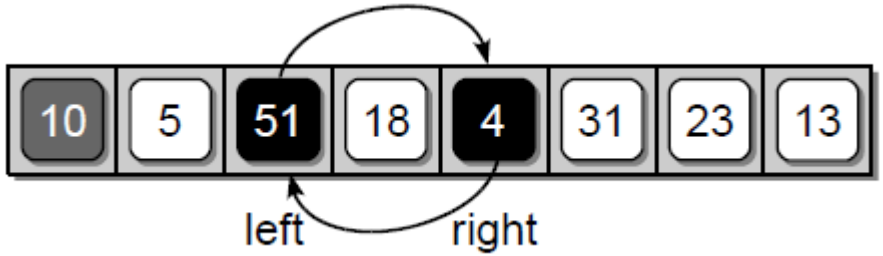


partitionSeq()

- The `left` marker will be shifted to key value 51, and the `right` marker to value 4.

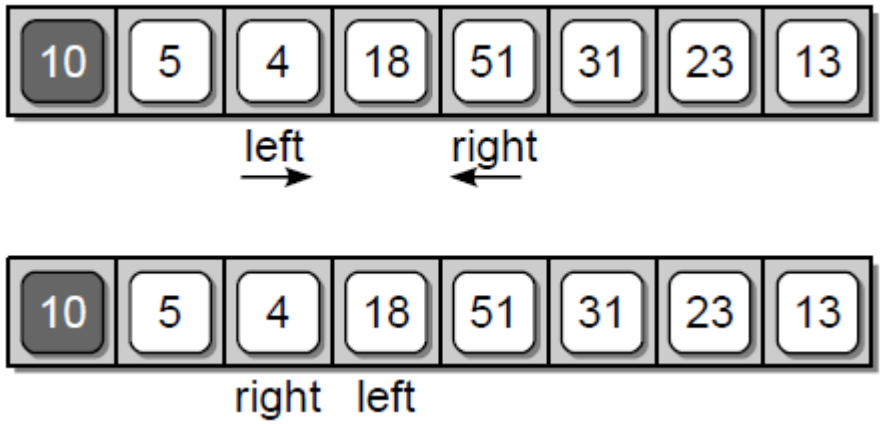


- Once the two markers are shifted, the corresponding keys are swapped and the process is repeated



partitionSeq()

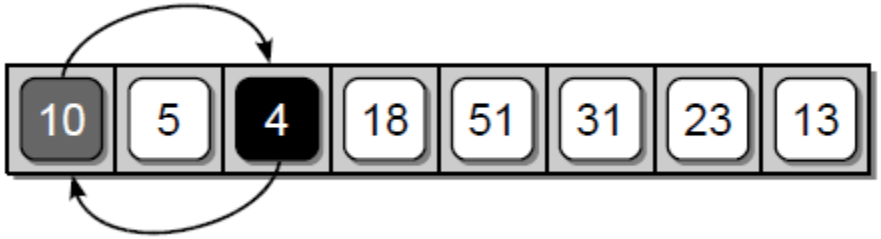
- Next, the `left` marker will stop at the value 18, while the `right` marker will stop at the value 4.



- Note that the `right` marker has crossed the `left` such that `right < left` ➔ termination of the outer `while` loop.

partitionSeq()

- When the two markers cross, the `right` marker indicates the final position of the pivot value in the resulting sorted list.
- Thus, the pivot value (currently located in the first element) and the element marked by `right` have to be swapped.



- Finally, the function returns the pivot position for use in splitting the sequence into two segments.