COMP1819
Algorithms and Data Structures

Lecture 10: MergeSort & QuickSort

Dr. Tuan Vuong

30/03/2021

LEARNING
DATA STRUCTURE & ALGORITHM
IS IMPORTANT

Content

- Lab 09 Walk-through/CW Q&A
- Merge Sort
- Quick Sort
- Reinforcement

You can check for sample code here: https://github.com/vptuan/COMP1819ADS

1. Bad recursion vs. good recursion

Using the given code for calculating Fibonacci numbers, compare the running time for the bad and good implementation of the two recursive methods.

Hints: you just need to measure the running time (import time) with a set of inputs.

Challenge: can you state the Big-O performance of the two implementations based on the running time?

2. Variety of fun trees

With given code (lecture/github), modify the recursive tree program using one of all the following ideas:

- Modify the thickness of the branches so that as the branchLen get smaller, the line gets thinner.
- Modify the colour of the branches so that as the branchLen gets very short it is coloured like a leaf
- Modify the angle used in turning the turtle so that at each branch point the angle is selected at random in some range. For example, choose the angle between 15 and 45 degrees. Play around to see what looks good.
- Modify the branchLen recursively so that instead of always subtracting the same amount you subtract a random amount in some range.

Hints: if you implement all of the above ideas, well done! You will have a very realistic looking tree.

3. Recursion: Minimum & Maximum

Write a short recursive Python function that finds the minimum and maximum values in a sequence without using any loops. What is the running time and space usage?

Just go through the process…

CW Last reminder

(Upload your report using with your lab number and member's last names e.g. Lab11_LastName1_LastName2_LastName3.pdf)

Deliverable 1 - REPORT ONLY (one per group)- Coursework submission due 23/03/2021 23:30

Deliverable 2 - Source code and files (one per group)- Coursework submission due 23/03/2021 23:30

The problem

The description of the interview question including the problem name, description, constraints and requirements, some short examples of input and output of the problem. This should NOT be longer than 300 words. The extended large input test cases be in Appendix or might need to include in separate files.

Proposed solutions

Your proposed different solutions, one for each member. Individually you need to briefly explain your own approach and solution. This should NOT be longer than 300 words. The codes need to be added to the Appendix and uploaded as files in Deliverable 2. Your solutions should cover different topics of algorithms and data structures.

CW Last reminder

Algorithm analysis

An algorithm analysis of your different solutions, including a diagram to compare the running time complexity of the different solutions with different inputs. You might need more than 5 inputs to plot clearly. State the big-O performance of each solution. This should NOT be longer than 300 words.

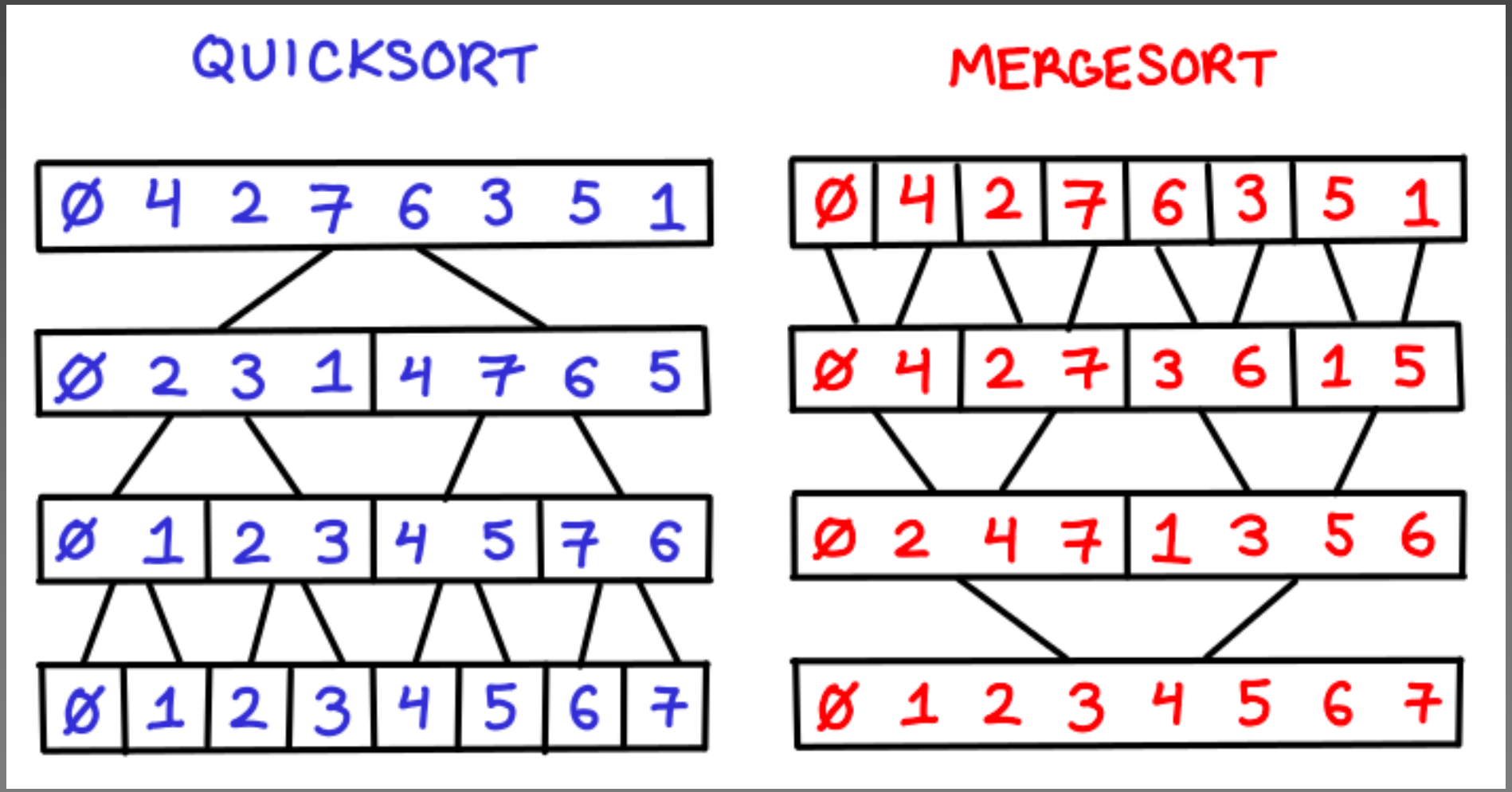
Reflection

A reflection of the team collaboration from each members and contribution mark for each member, that is agreed by your team. This should NOT be longer than 300 words.

Please note that this coursework should take an average student who is up-to-date with tutorial work **approximately 25 hours for each member** of the group.

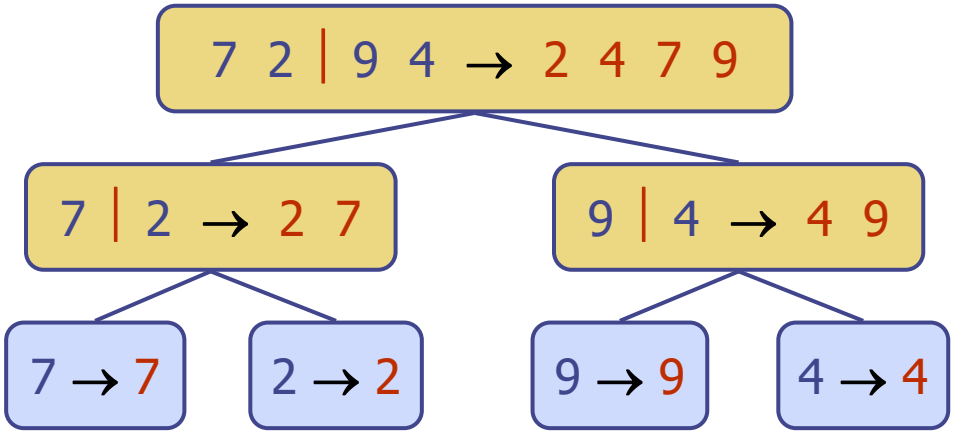
Name	Percentage of marks given agreed by team
	100% (full mark)
	50% (half mark)
	0% (zero mark)

Today



https://hackernoon.com/

Merge Sort



Watch: https://youtu.be/JSceec-wEyw

Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two disjoint subsets S_1 and S_2
  - Recur: solve the subproblems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S
- The base case for the recursion are subproblems of size 0 or 1
- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort
  - It has $O(n \log n)$ running time
- Unlike heap-sort
  - It does not use an auxiliary priority queue
  - It accesses data in a sequential manner (suitable to sort data on a disk)

Merge-Sort

- Merge-sort 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
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

Algorithm *mergeSort*(S)
  Input sequence S with n elements
  Output sequence S sorted according to C
  if $S.size() > 1$
    $(S_1, S_2) \leftarrow partition(S, n/2)$
    $mergeSort(S_1)$
    $mergeSort(S_2)$
    $S \leftarrow merge(S_1, S_2)$

Merging Two Sorted Sequences

- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

Algorithm *merge*(A, B)

Input sequences A and B with $n/2$ elements each

Output sorted sequence of $A \cup B$

$S \leftarrow$ empty sequence

while $\neg A.isEmpty() \wedge \neg B.isEmpty()$

if $A.first().element() < B.first().element()$

$S.addLast(A.remove(A.first()))$

else

$S.addLast(B.remove(B.first()))$

while $\neg A.isEmpty()$

$S.addLast(A.remove(A.first()))$

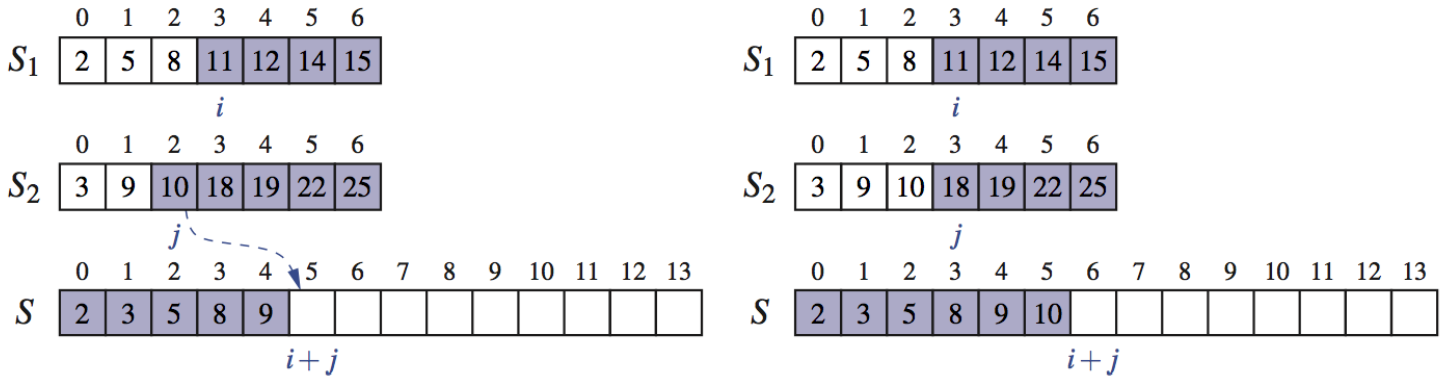
while $\neg B.isEmpty()$

$S.addLast(B.remove(B.first()))$

return S

Python Merge Implementation

```
1  def merge(S1, S2, S):
2    """Merge two sorted Python lists S1 and S2 into properly sized list S."""
3    i = j = 0
4    while i + j < len(S):
5      if j == len(S2) or (i < len(S1) and S1[i] < S2[j]):
6        S[i+j] = S1[i]                  # copy ith element of S1 as next item of S
7        i += 1
8      else:
9        S[i+j] = S2[j]                  # copy jth element of S2 as next item of S
10       j += 1
```

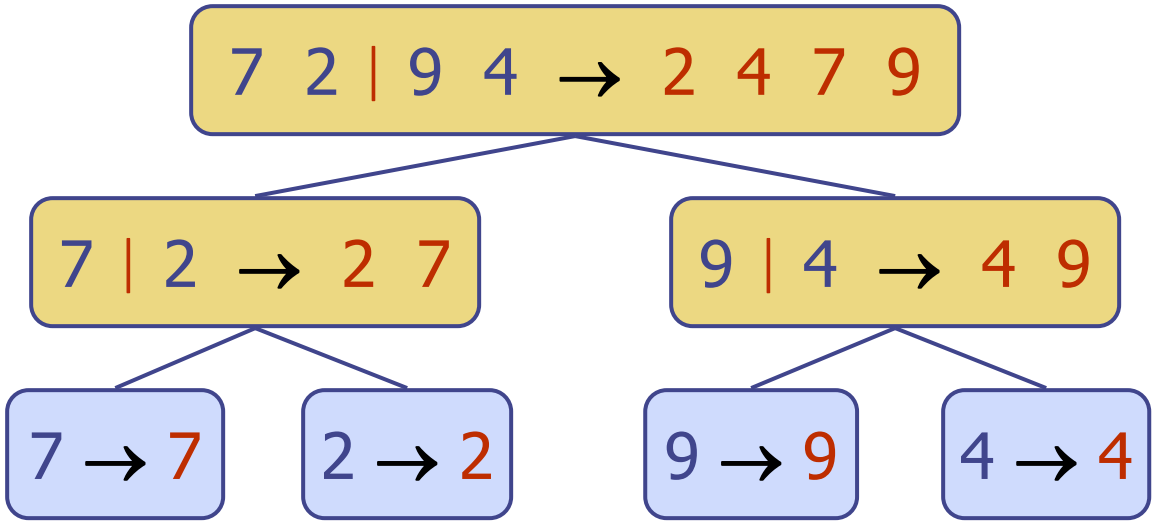


Python Merge-Sort Implementation

```
1  def merge_sort(S):
2    """Sort the elements of Python list S using the merge-sort algorithm."""
3    n = len(S)
4    if n < 2:
5      return                          # list is already sorted
6    # divide
7    mid = n // 2
8    S1 = S[0:mid]                     # copy of first half
9    S2 = S[mid:n]                     # copy of second half
10   # conquer (with recursion)
11   merge_sort(S1)                    # sort copy of first half
12   merge_sort(S2)                    # sort copy of second half
13   # merge results
14   merge(S1, S2, S)                  # merge sorted halves back into S
```

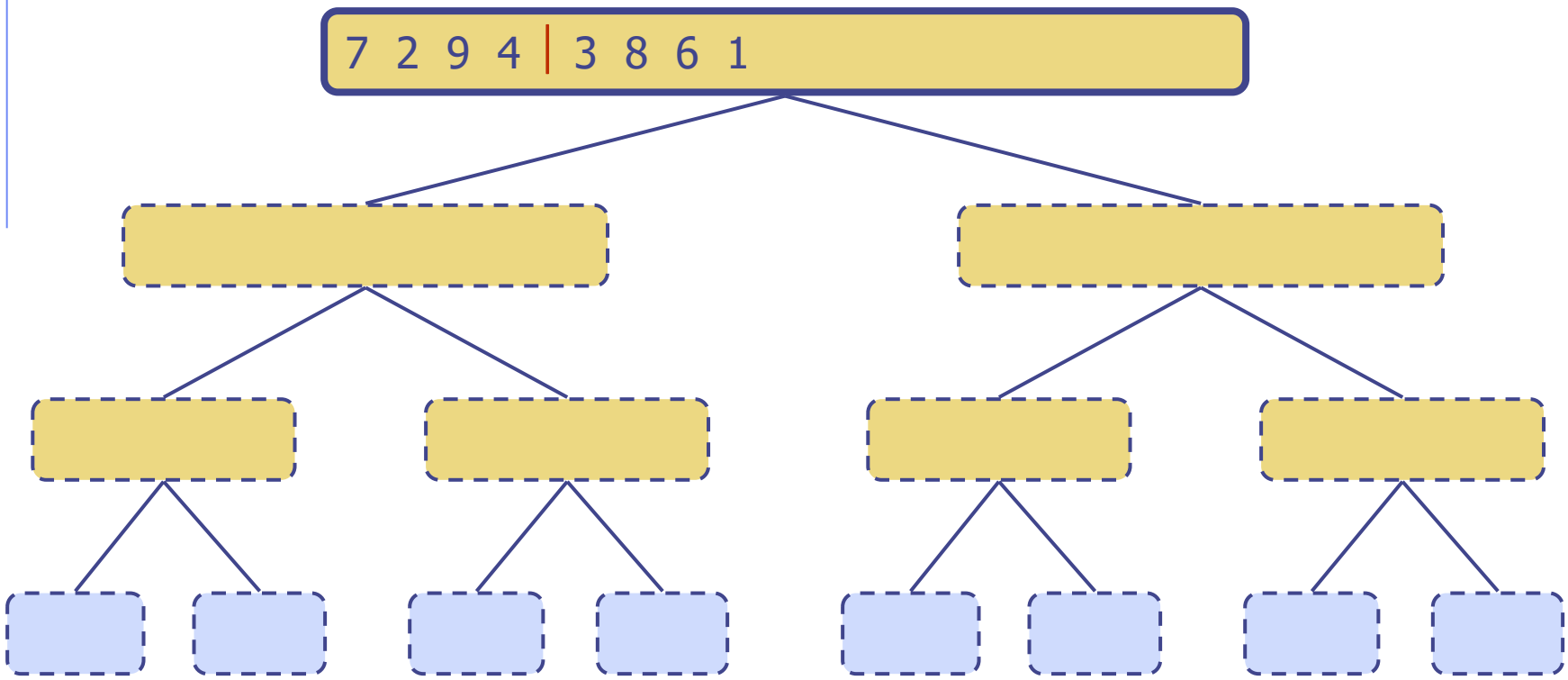
Merge-Sort Tree

- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - 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



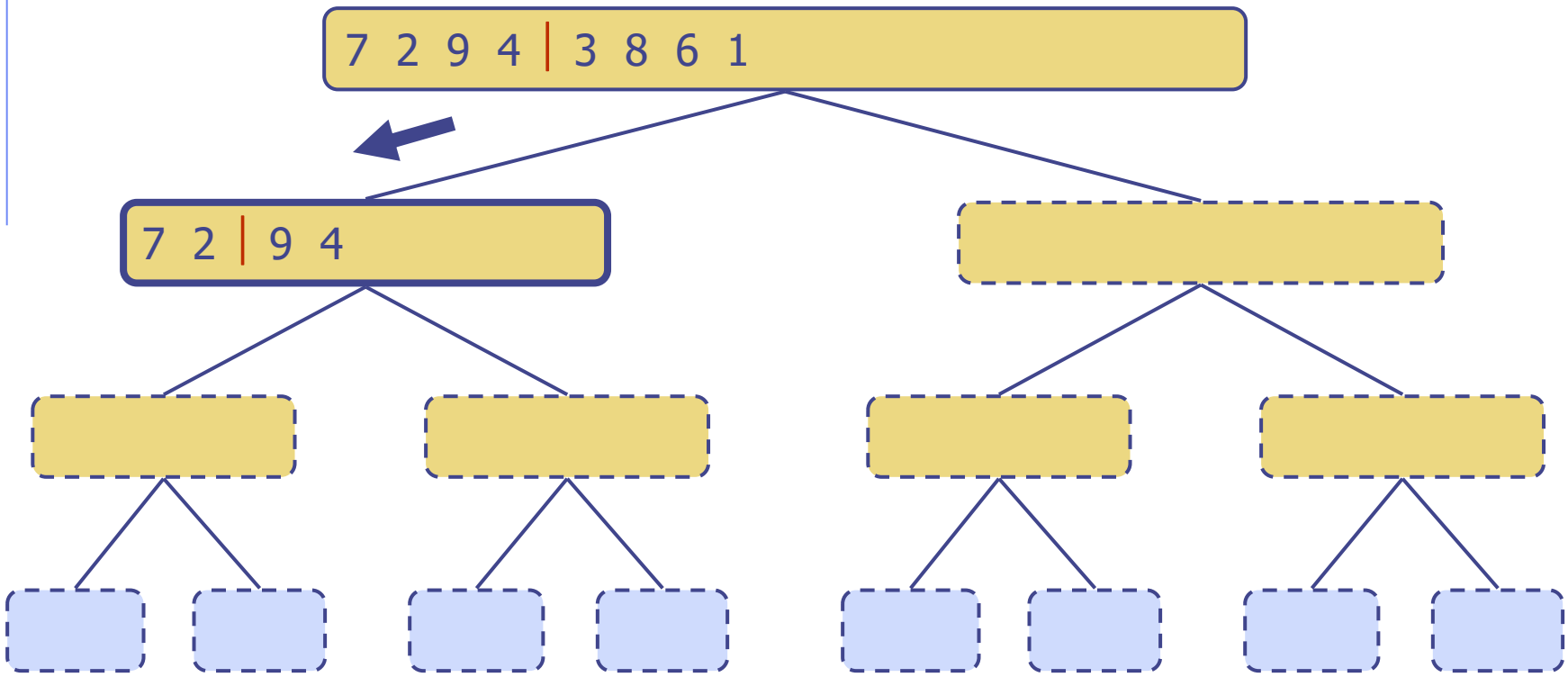
Execution Example

- Partition



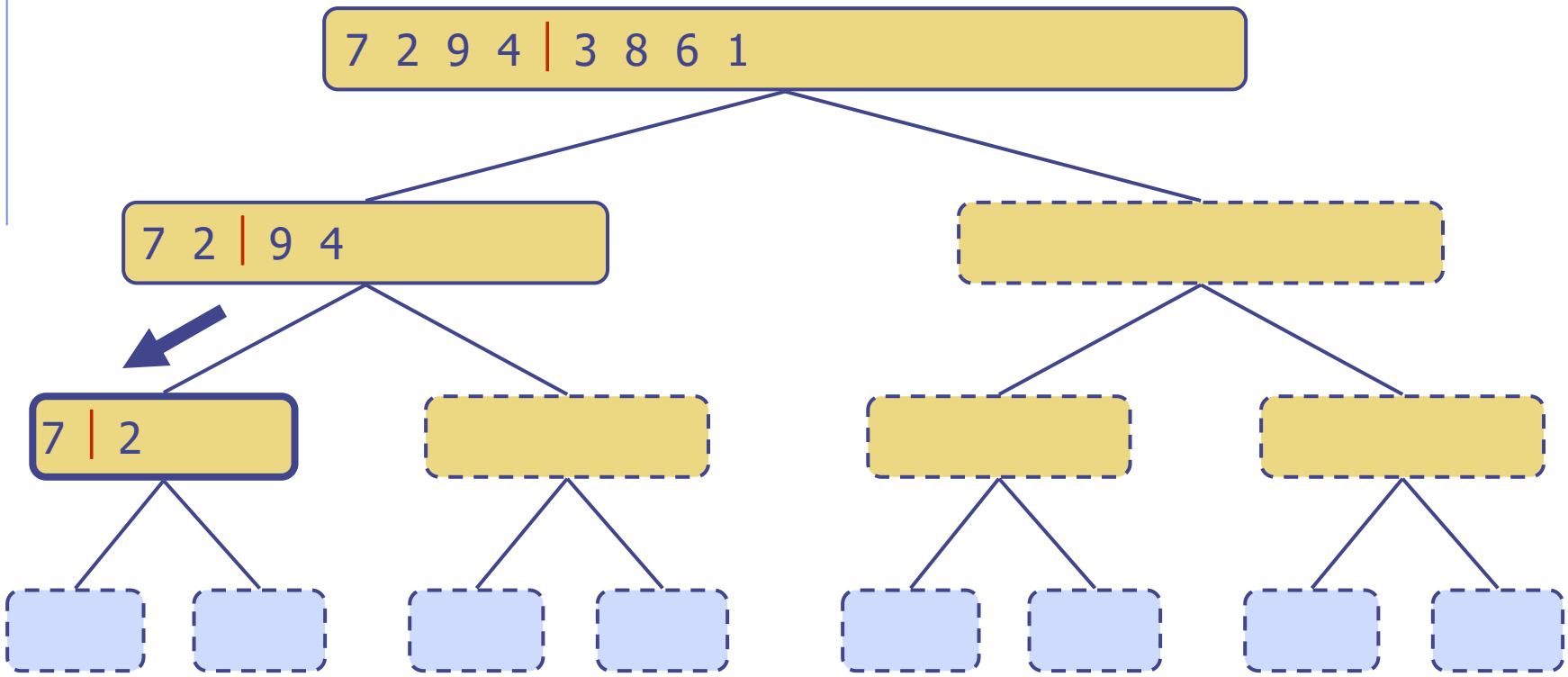
Execution Example (cont.)

- Recursive call, partition



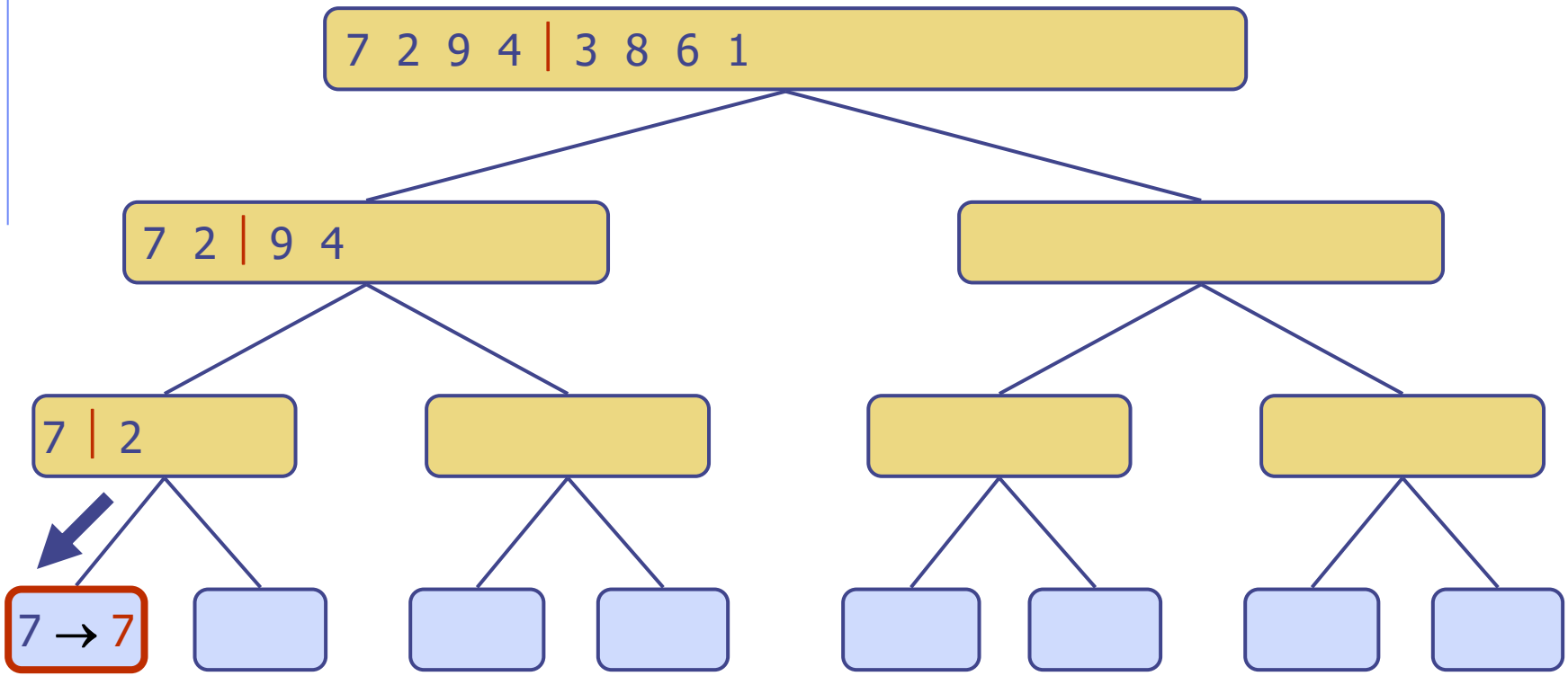
Execution Example (cont.)

- Recursive call, partition



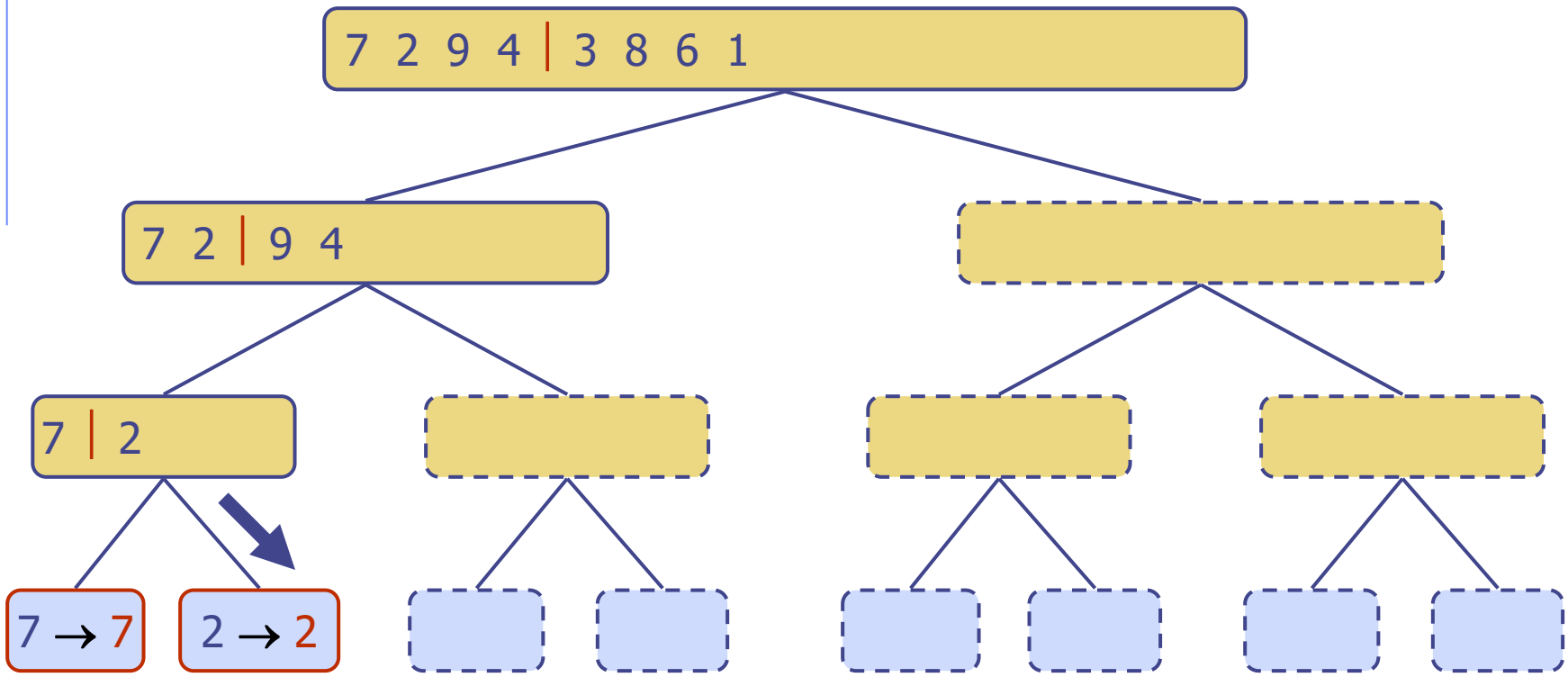
Execution Example (cont.)

- Recursive call, base case



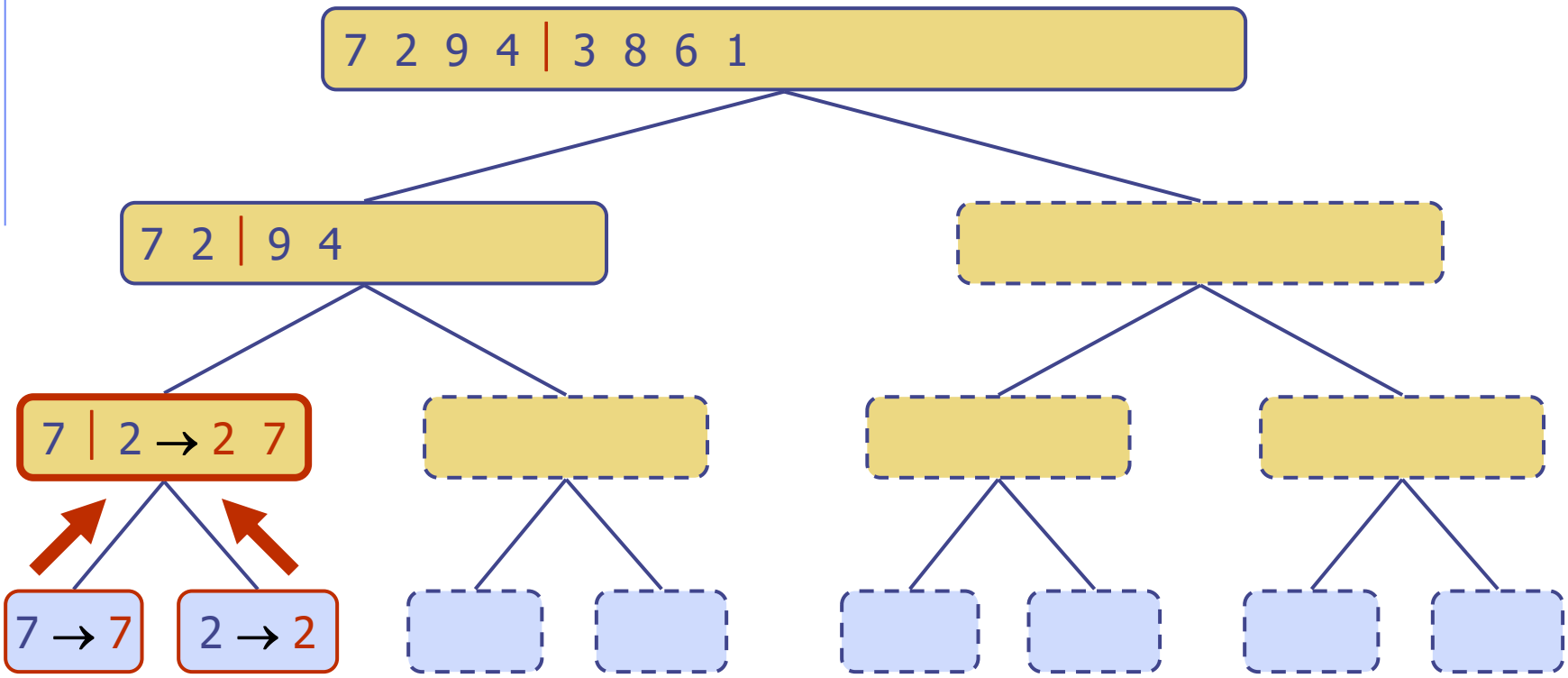
Execution Example (cont.)

- Recursive call, base case



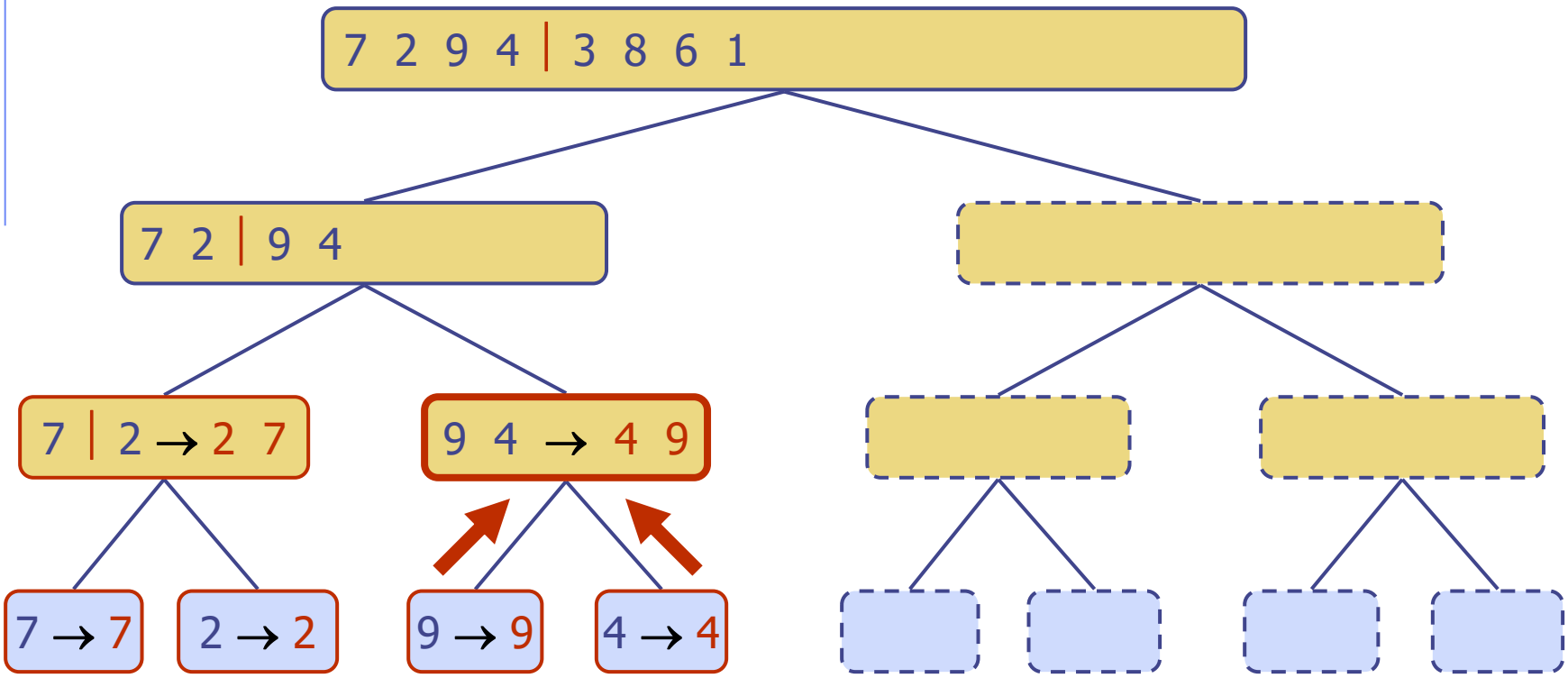
Execution Example (cont.)

- Merge



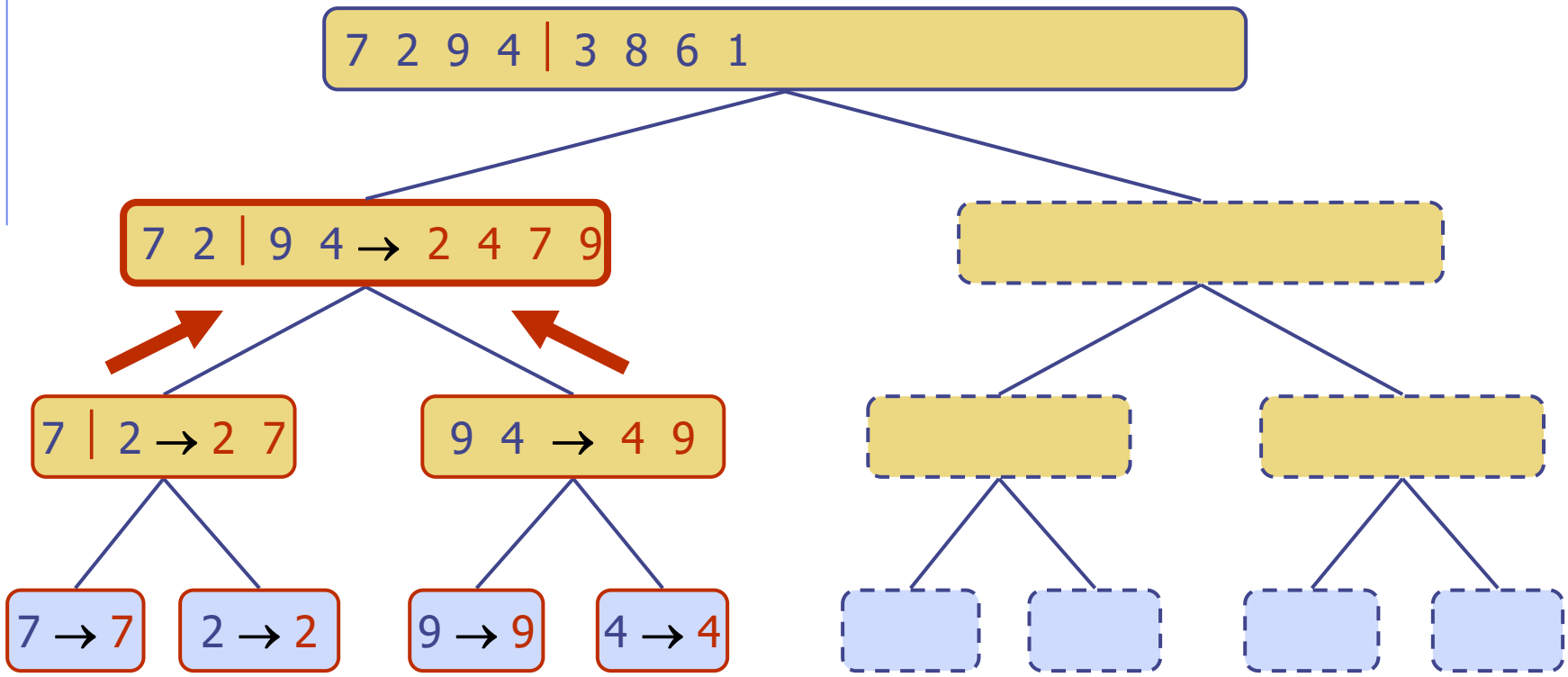
Execution Example (cont.)

- Recursive call, …, base case, merge



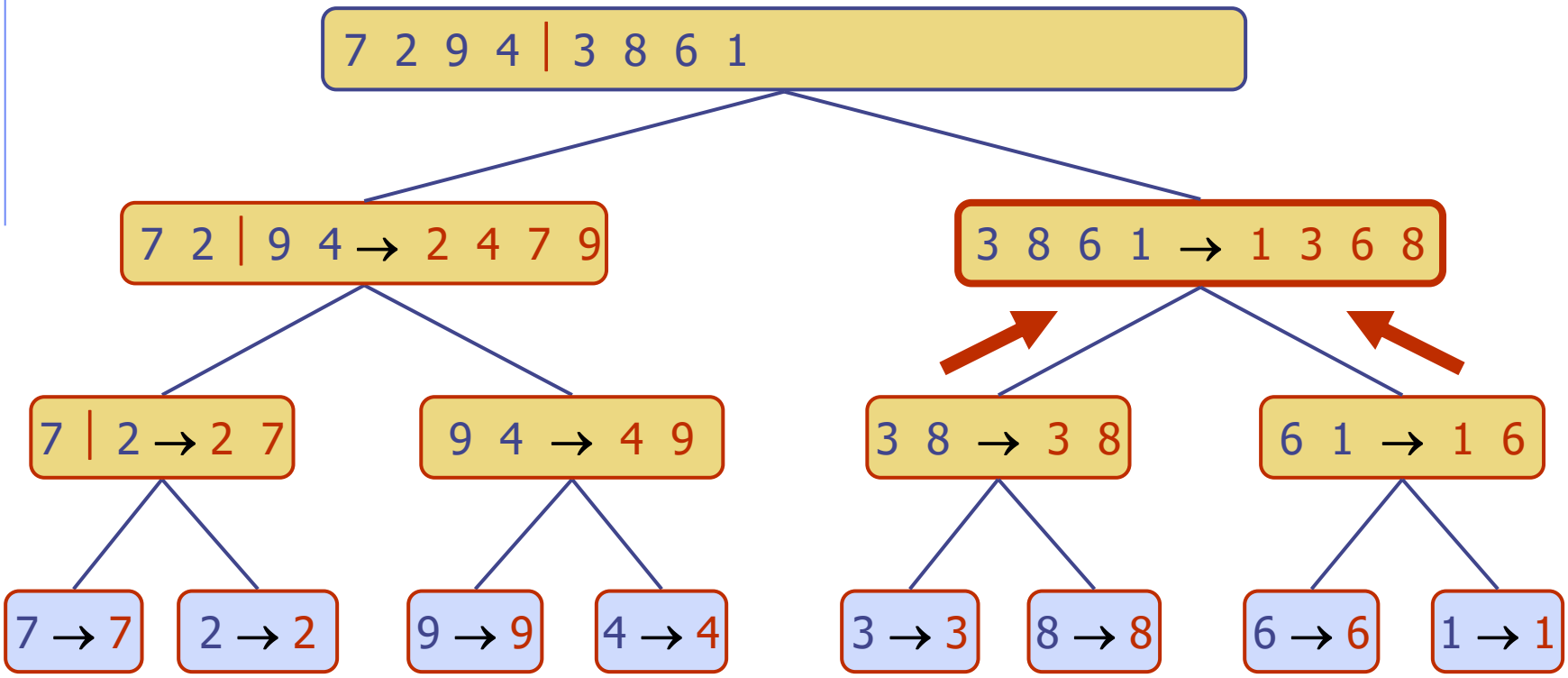
Execution Example (cont.)

- Merge



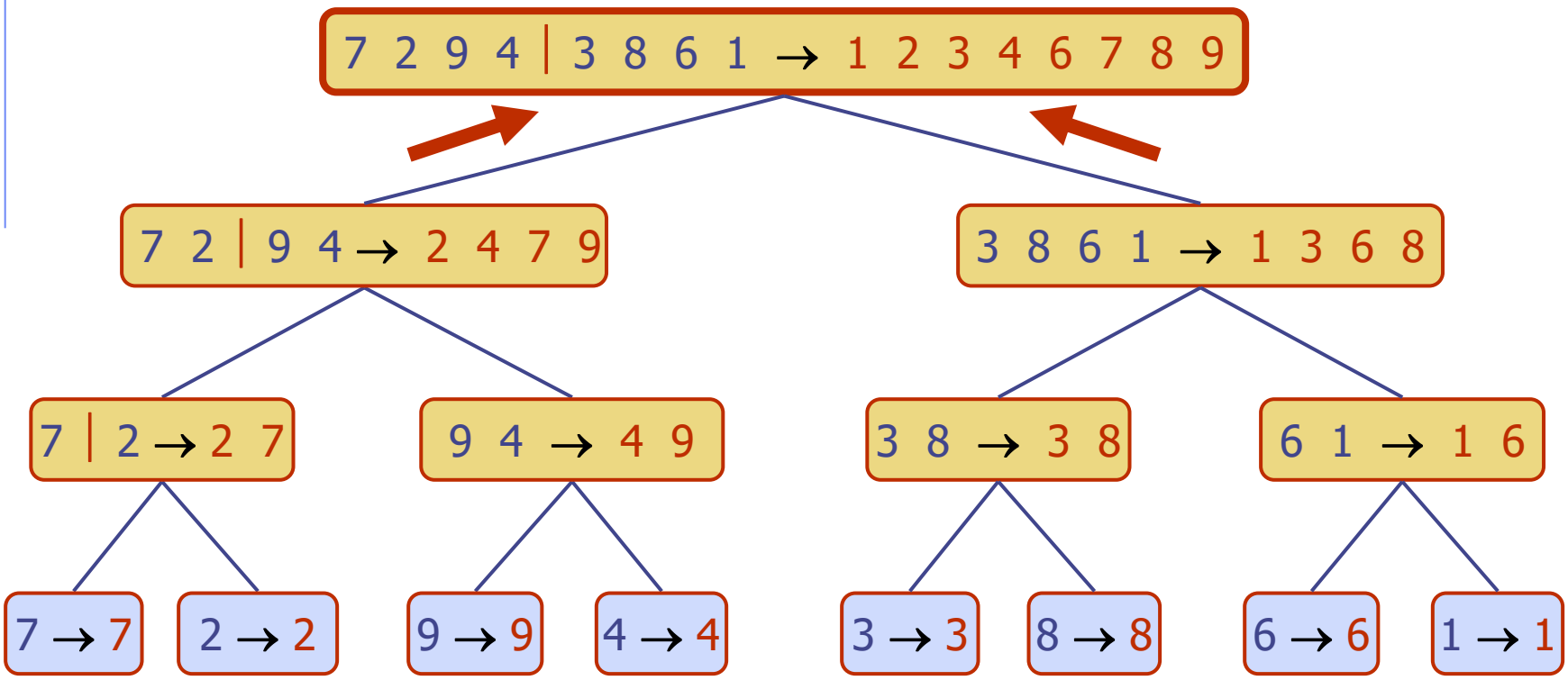
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

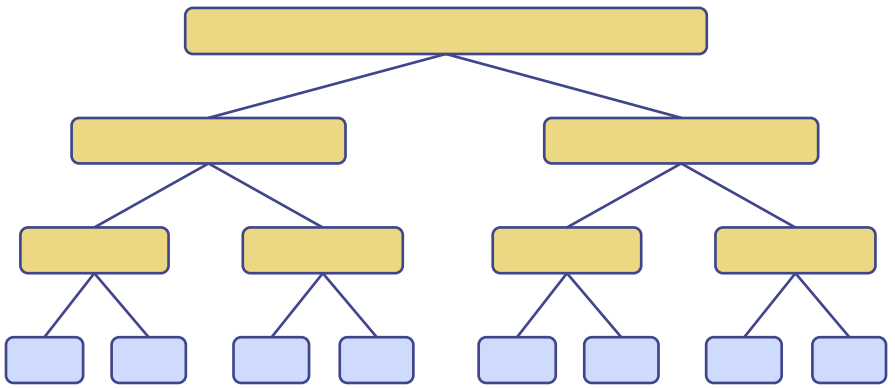
- Merge



Analysis of Merge-Sort

- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

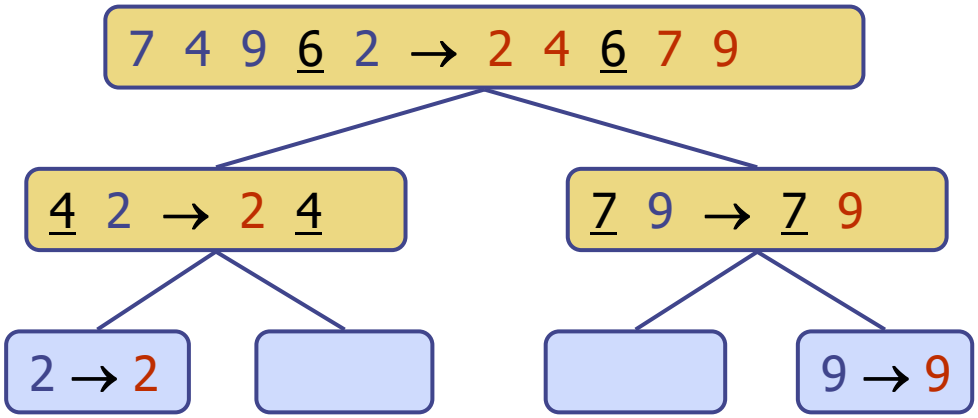
depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
insertion-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
merge-sort	$O(n \log n)$	▪ fast ▪ sequential data access ▪ for huge data sets (> 1M)

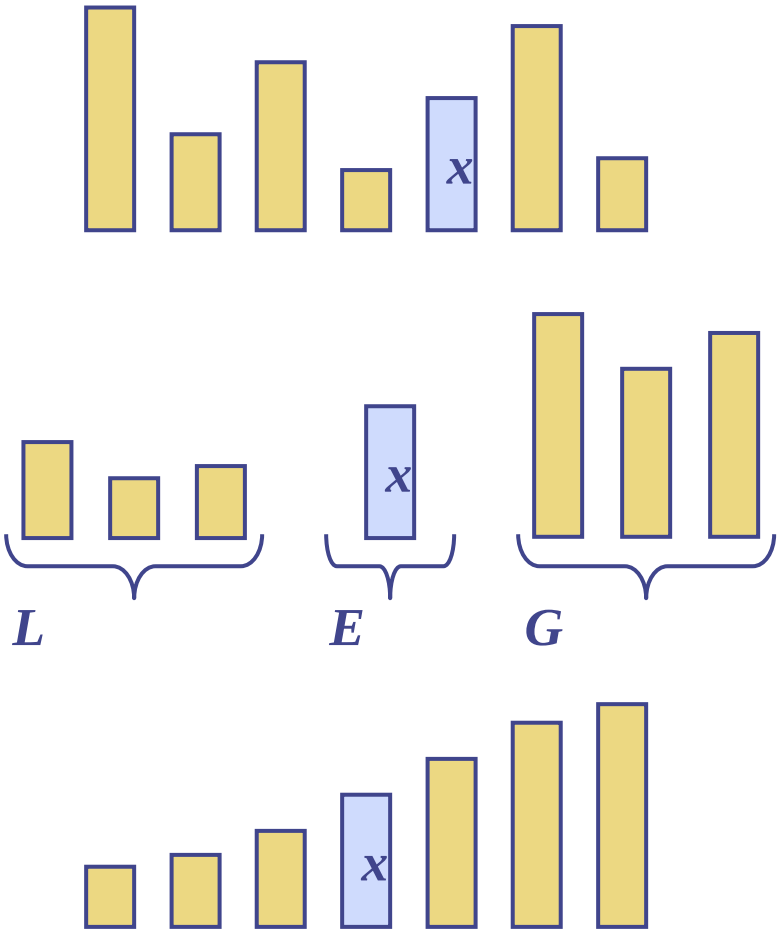
Quick-Sort



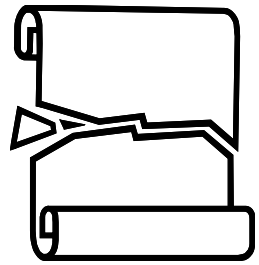
Watch: https://youtu.be/PgBzjlCcFvc

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

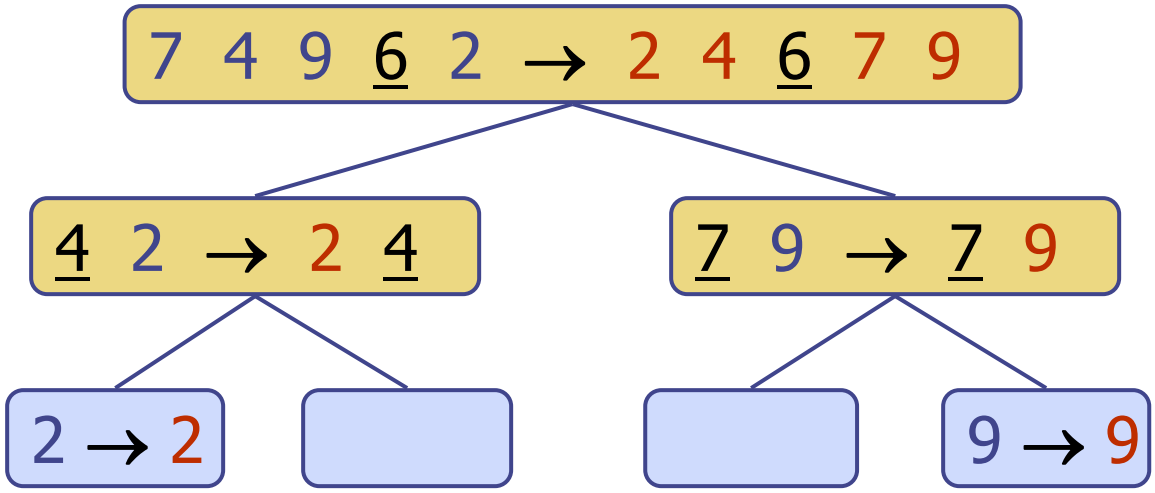
Algorithm *partition*(S, p)
  Input sequence S, position p of pivot
  Output subsequences L, E, G of the elements of S less than, equal to, or greater than the pivot, resp.
  $L, E, G \leftarrow$ empty sequences
$x \leftarrow S.remove(p)$
while $\neg S.isEmpty()$
  $y \leftarrow S.remove(S.first())$
  if $y < x$
    $L.addLast(y)$
  else if $y = x$
    $E.addLast(y)$
  else $\{ y > x \}$
    $G.addLast(y)$
return L, E, G

Python Implementation

```
1  def quick_sort(S):
2    """Sort the elements of queue S using the quick-sort algorithm."""
3    n = len(S)
4    if n < 2:
5      return                          # list is already sorted
6    # divide
7    p = S.first( )                    # using first as arbitrary pivot
8    L = LinkedQueue( )
9    E = LinkedQueue( )
10   G = LinkedQueue( )
11   while not S.is_empty( ):          # divide S into L, E, and G
12     if S.first( ) < p:
13       L.enqueue(S.dequeue( ))
14     elif p < S.first( ):
15       G.enqueue(S.dequeue( ))
16     else:                           # S.first() must equal pivot
17       E.enqueue(S.dequeue( ))
18   # conquer (with recursion)
19   quick_sort(L)                     # sort elements less than p
20   quick_sort(G)                     # sort elements greater than p
21   # concatenate results
22   while not L.is_empty( ):
23     S.enqueue(L.dequeue( ))
24   while not E.is_empty( ):
25     S.enqueue(E.dequeue( ))
26   while not G.is_empty( ):
27     S.enqueue(G.dequeue( ))
```

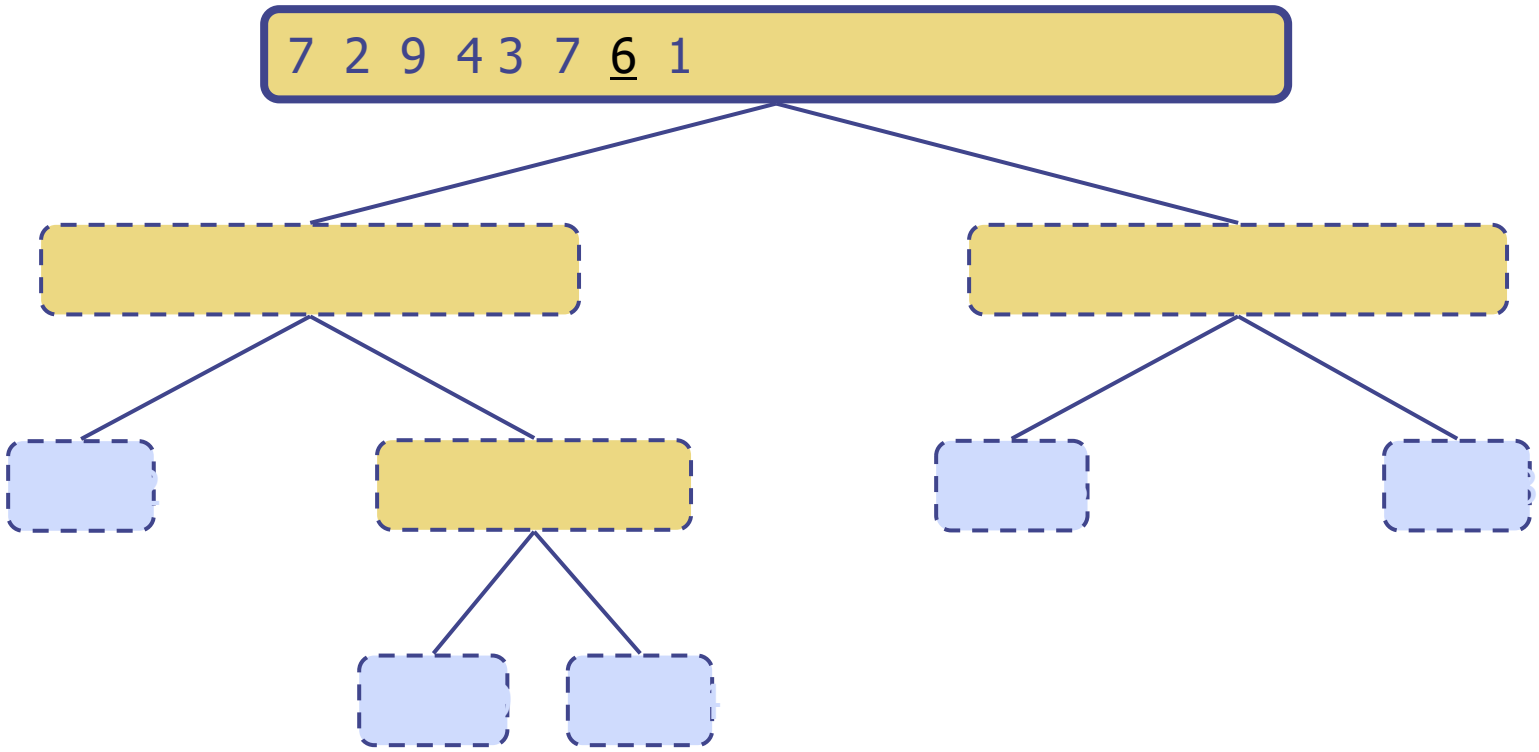
Quick-Sort Tree

- 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



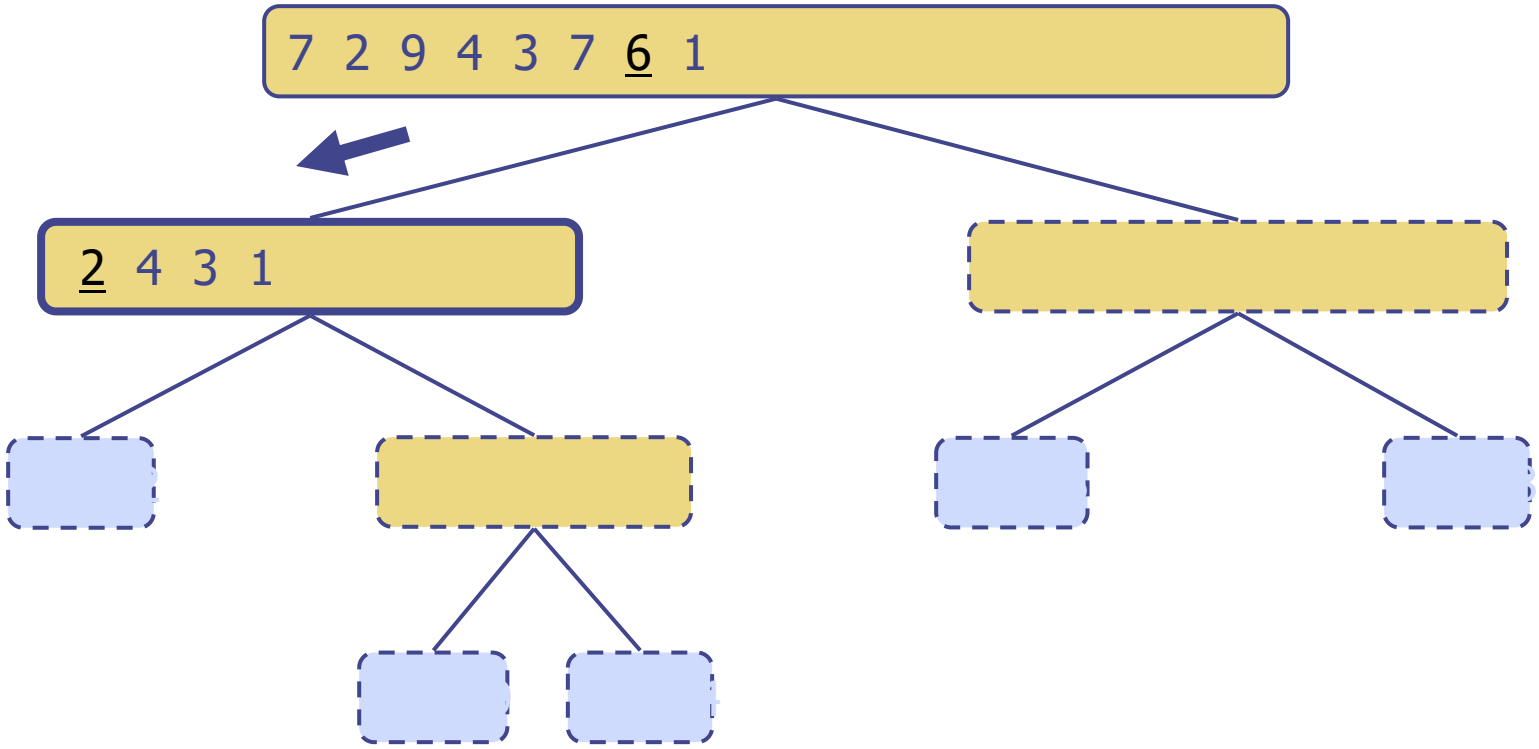
Execution Example

- Pivot selection



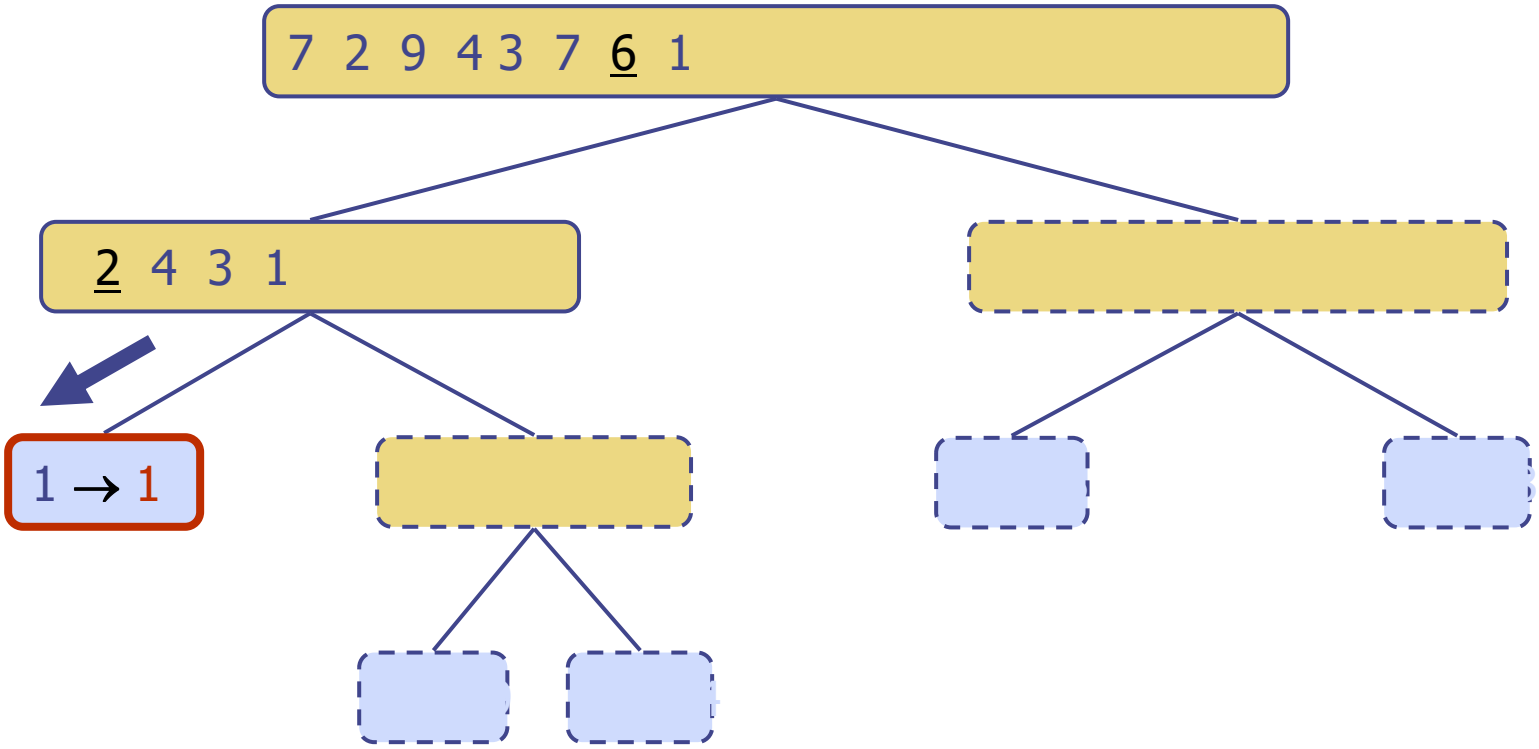
Execution Example (cont.)

- Partition, recursive call, pivot selection



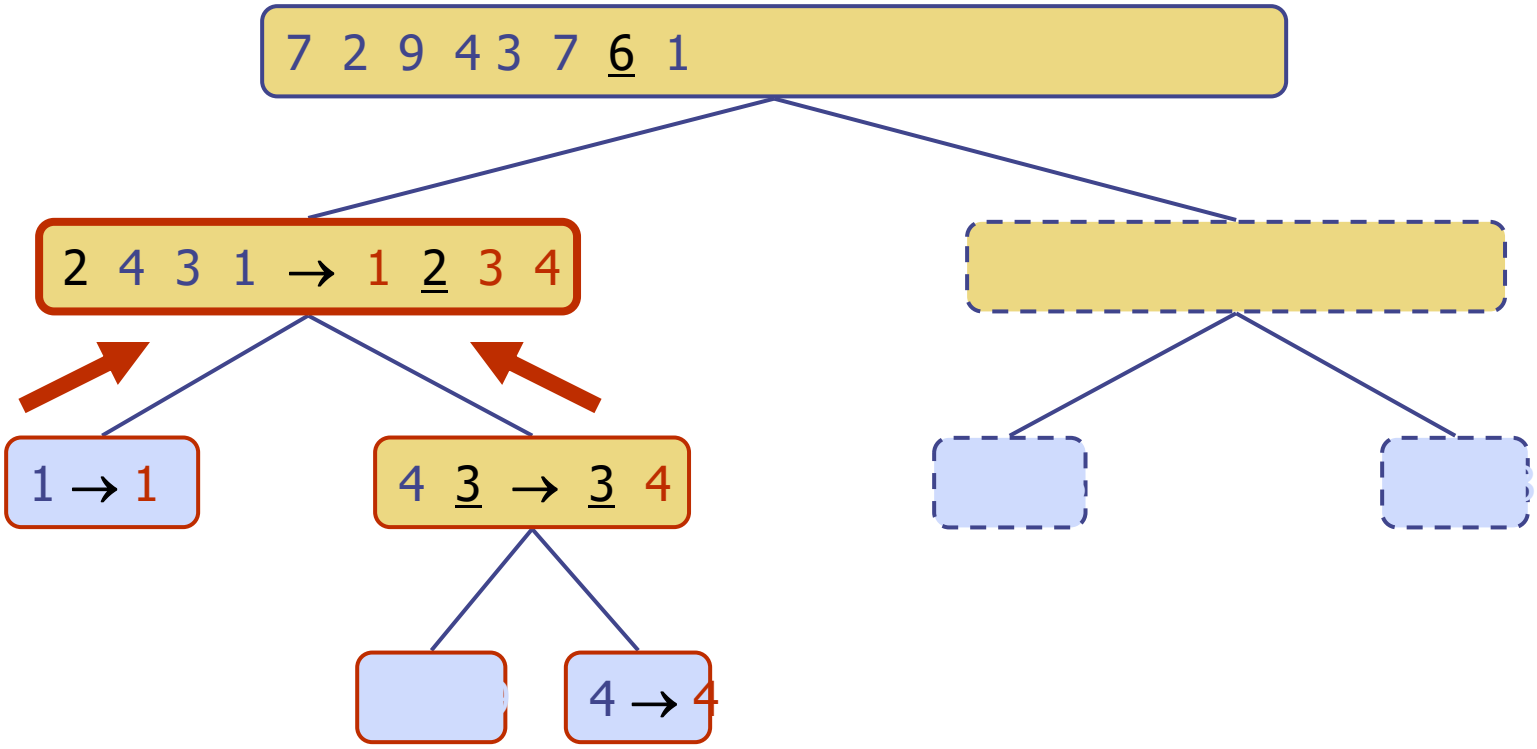
Execution Example (cont.)

- Partition, recursive call, base case



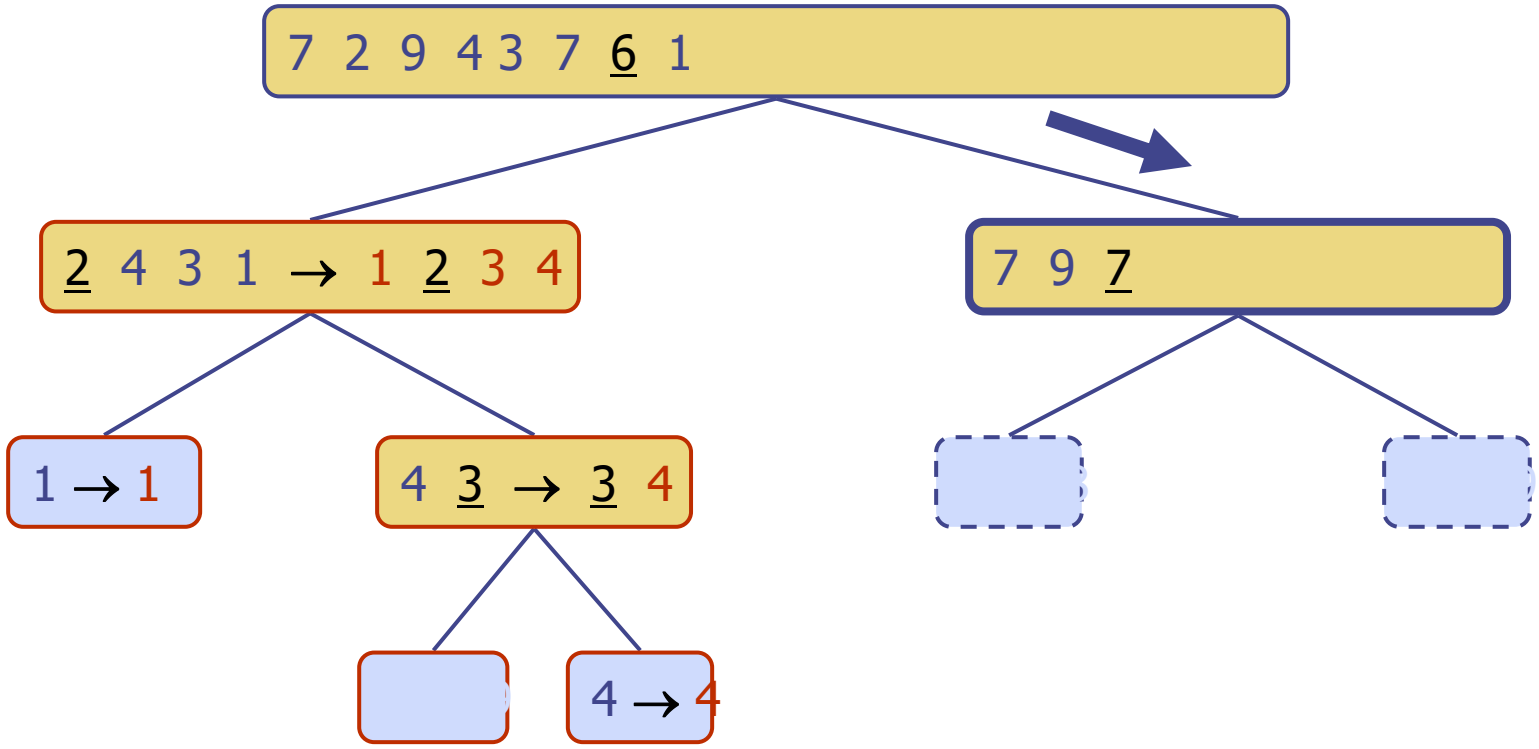
Execution Example (cont.)

- Recursive call, …, base case, join



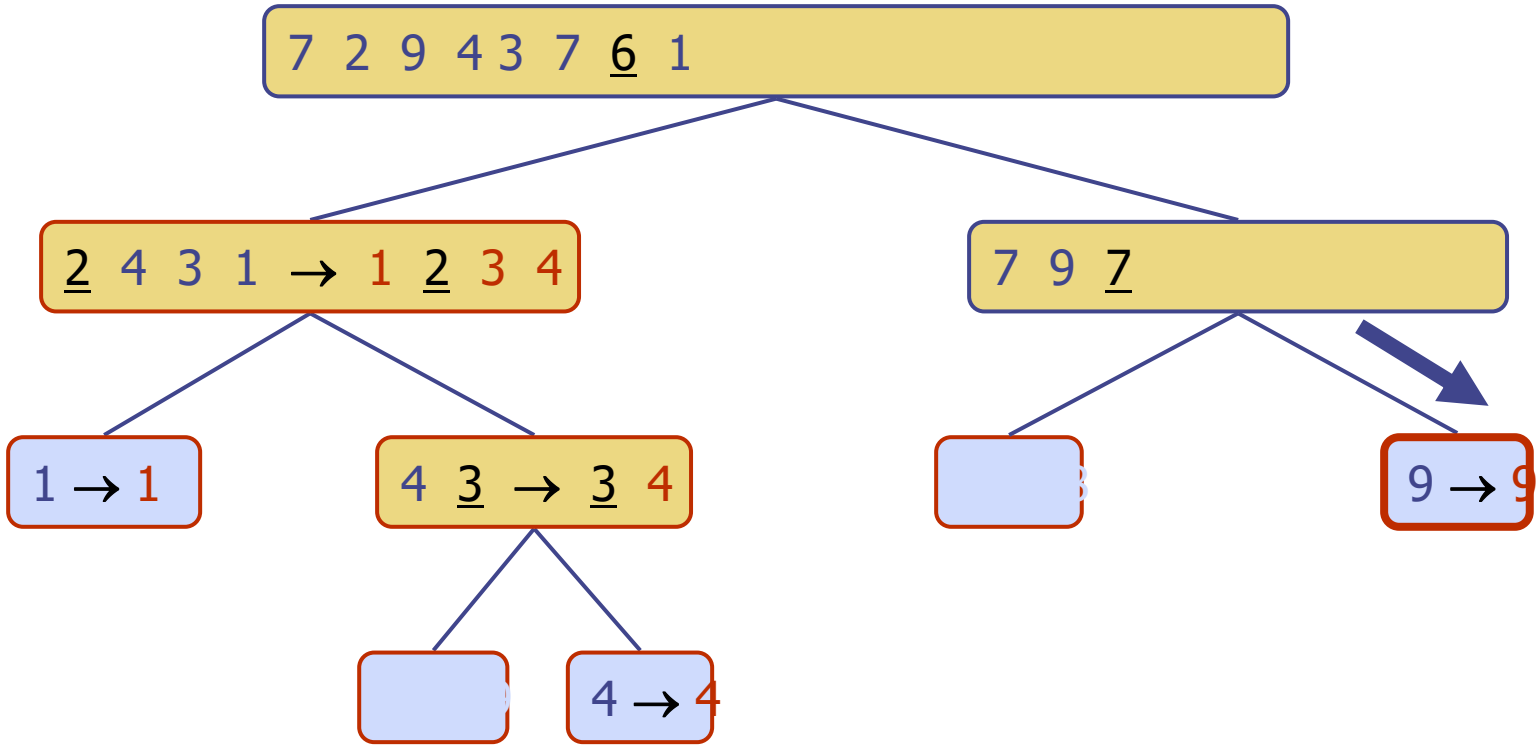
Execution Example (cont.)

- Recursive call, pivot selection



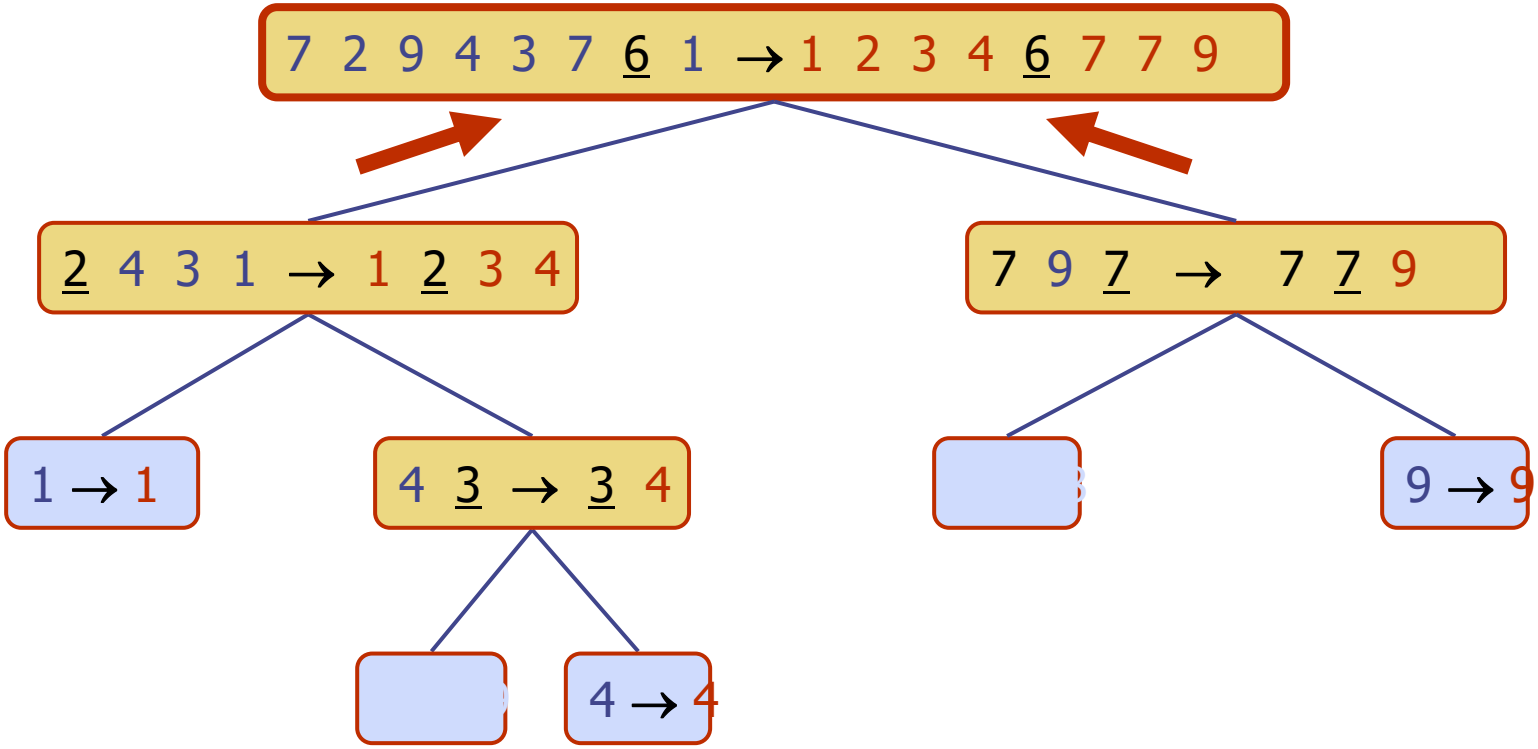
Execution Example (cont.)

- Partition, ..., recursive call, base case



Execution Example (cont.)

- Join, join

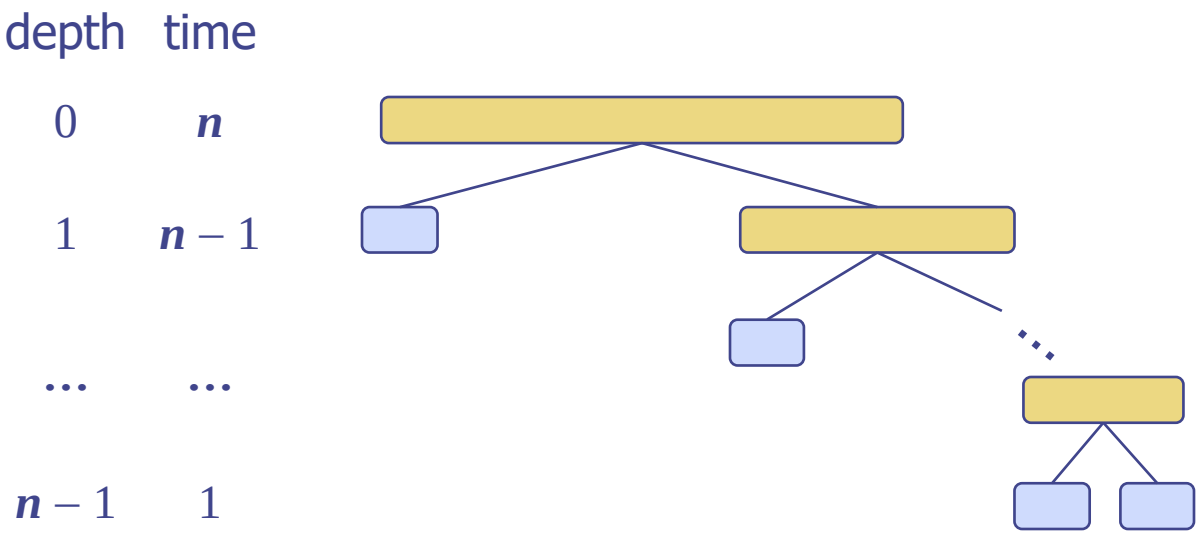


Worst-case Running Time

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

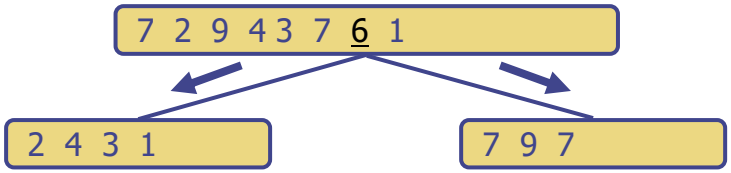
$$n + (n - 1) + \ldots + 2 + 1$$

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

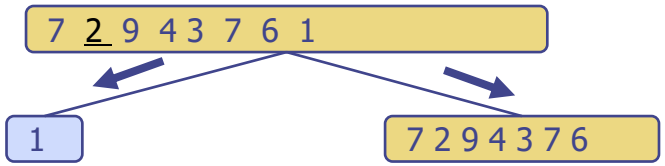


Expected Running Time

- 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

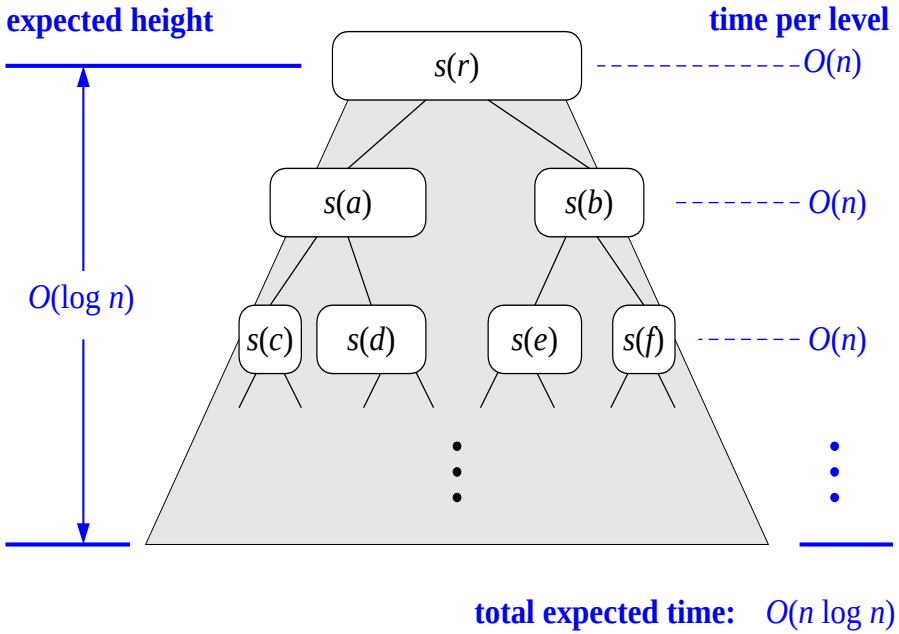


Bad call

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

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
  - $i/2$ ancestors are good calls
  - 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
  - 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 $O(n)$
- Thus, 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

$x \leftarrow S.elemAtRank(i)$

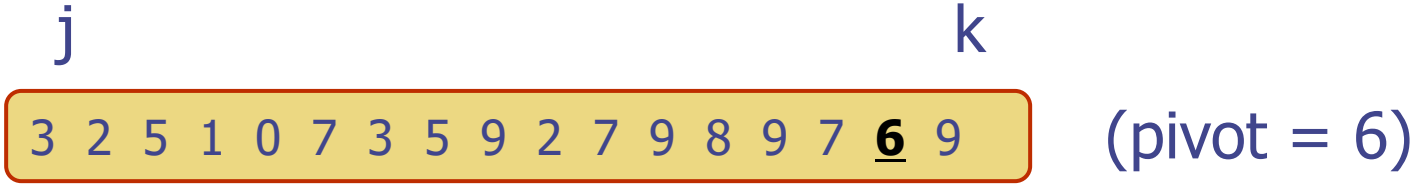
$(h, k) \leftarrow inPlacePartition(x)$

$inPlaceQuickSort(S, l, h - 1)$

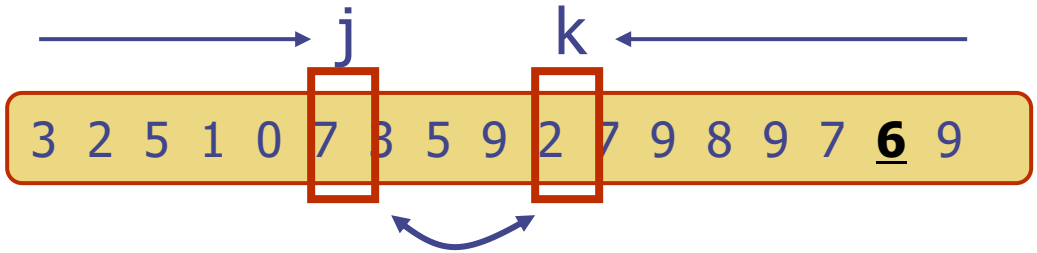
$inPlaceQuickSort(S, k + 1, r)$

In-Place Partitioning

- Perform the partition using two indices to split S into L and E U G (a similar method can split E U G into E and G).



- Repeat until j and k cross:
  - 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



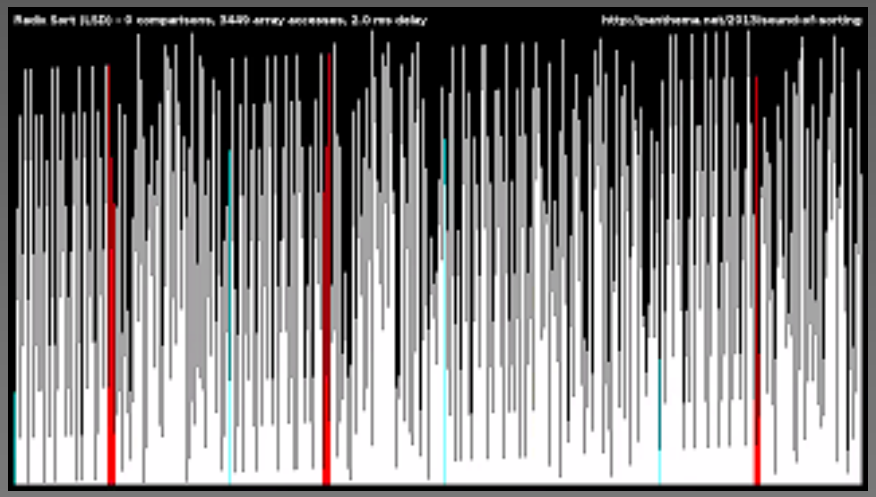
Python Implementation

```
1  def inplace_quick_sort(S, a, b):
2    """Sort the list from S[a] to S[b] inclusive using the quick-sort algorithm."""
3    if a >= b: return                          # range is trivially sorted
4    pivot = S[b]                               # last element of range is pivot
5    left = a                                   # will scan rightward
6    right = b-1                                # will scan leftward
7    while left <= right:
8      # scan until reaching value equal or larger than pivot (or right marker)
9      while left <= right and S[left] < pivot:
10       left += 1
11     # scan until reaching value equal or smaller than pivot (or left marker)
12     while left <= right and pivot < S[right]:
13       right -= 1
14     if left <= right:                          # scans did not strictly cross
15       S[left], S[right] = S[right], S[left]                  # swap values
16       left, right = left + 1, right - 1                      # shrink range
17
18   # put pivot into its final place (currently marked by left index)
19   S[left], S[b] = S[b], S[left]
20   # make recursive calls
21   inplace_quick_sort(S, a, left - 1)
22   inplace_quick_sort(S, left + 1, b)
```

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)
merge-sort	$O(n \log n)$	▪ sequential data access ▪ fast (good for huge inputs)

Reinforcement



https://youtu.be/kPRA0W1kECg

Question 1

Consider the following list of integers: [1,2,3,4,5,6,7,8,9,10].
Show how this list is sorted by the following algorithms:

- bubble sort
- selection sort
- insertion sort
- merge sort
- quick sort

Question 2

Consider the list of characters: ['P','Y','T','H','O','N']. Show how this list is sorted using the following algorithms:

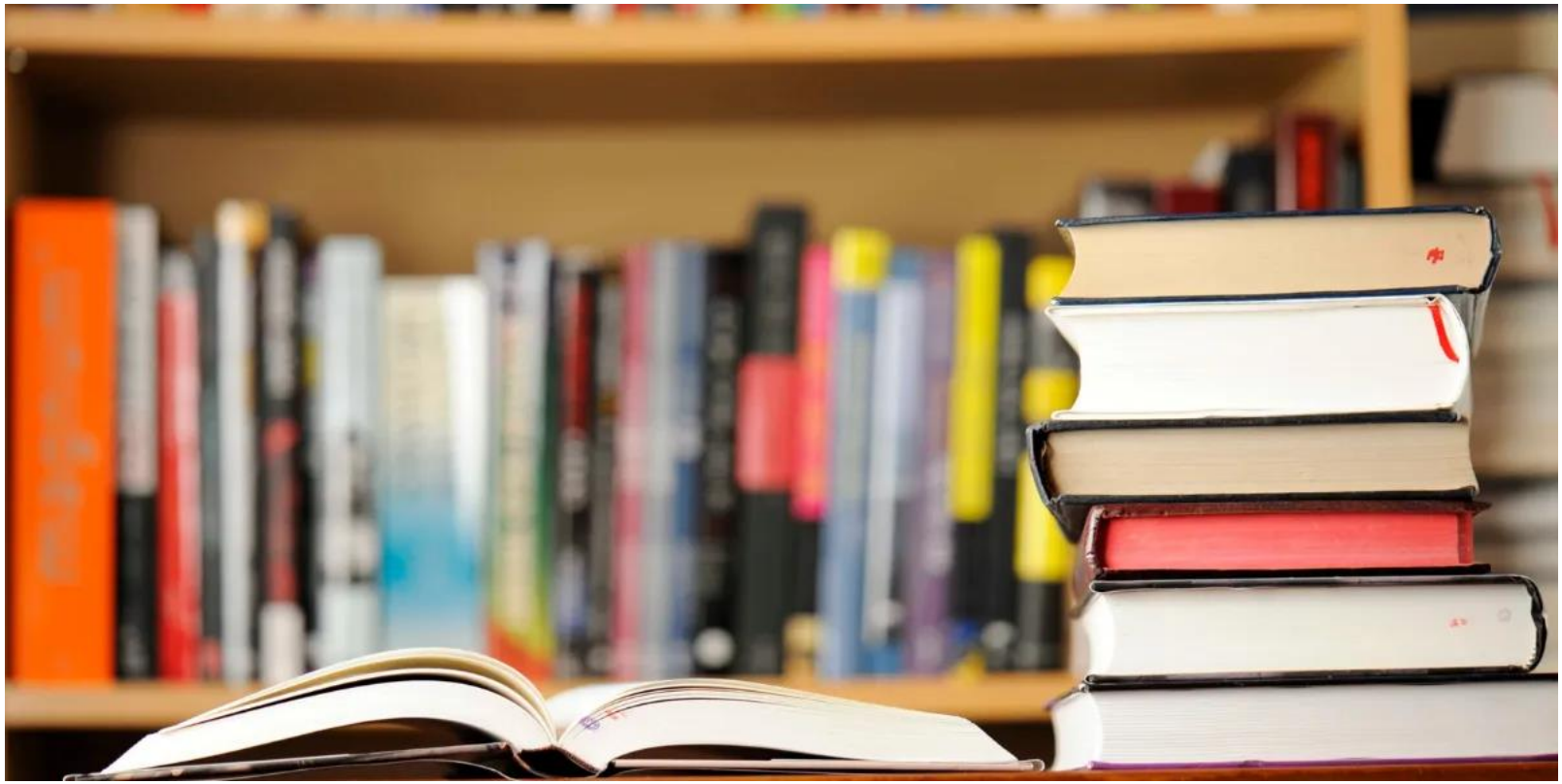
- bubble sort
- selection sort
- insertion sort
- merge sort
- quick sort

Quick overview

- A merge sort is O (n log n), but requires additional space for the merging process
- A quick sort is O (n log n), but may degrade to O (n^2) if the split points are not near the middle of the list. It does not require additional space.

Extra reading

- Shell Sort



The Greedy Method and Text Compression

Next week!