COMP 352
Data Structures and Algorithms

MERGE & QUICK SORT

Chapter 12

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
- Conquer: solve the sub-problems associated with S_1 and S_2
- Combine: take the solutions for S_1 and S_2 and combine into a solution for S

The base case for the recursion are sub-problems of size 0 or 1

Merge-Sort

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

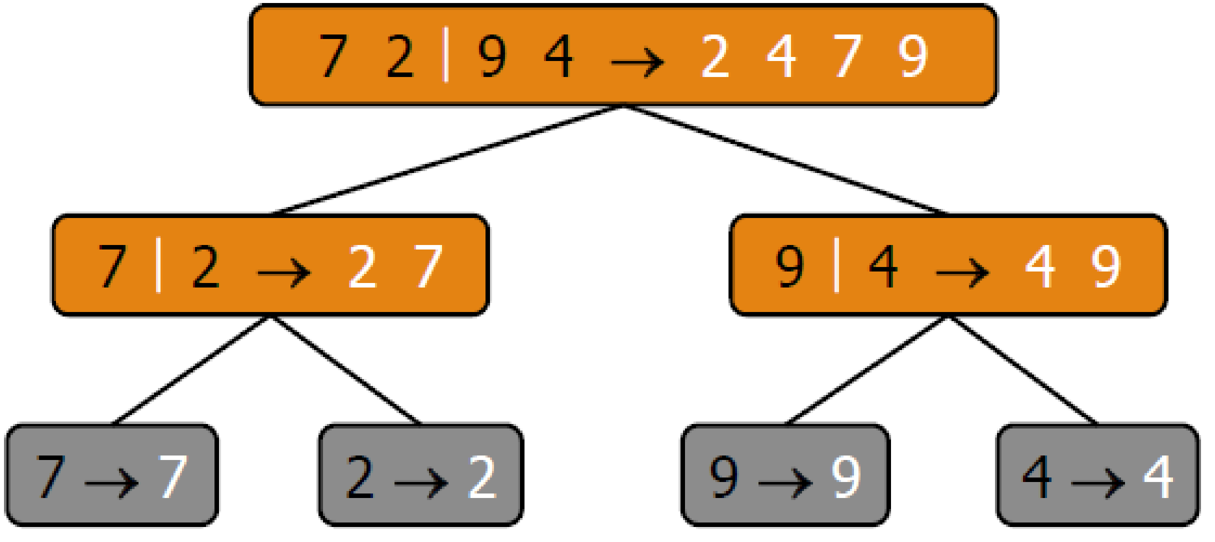
Algorithm *merge*(A, B)
  Input sequences A and B with $n/2$ elements each
  Output sorted sequence of $A \cup B$

  $S \leftarrow$ 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

Merge-Sort Tree

An execution of merge-sort is depicted by a binary tree

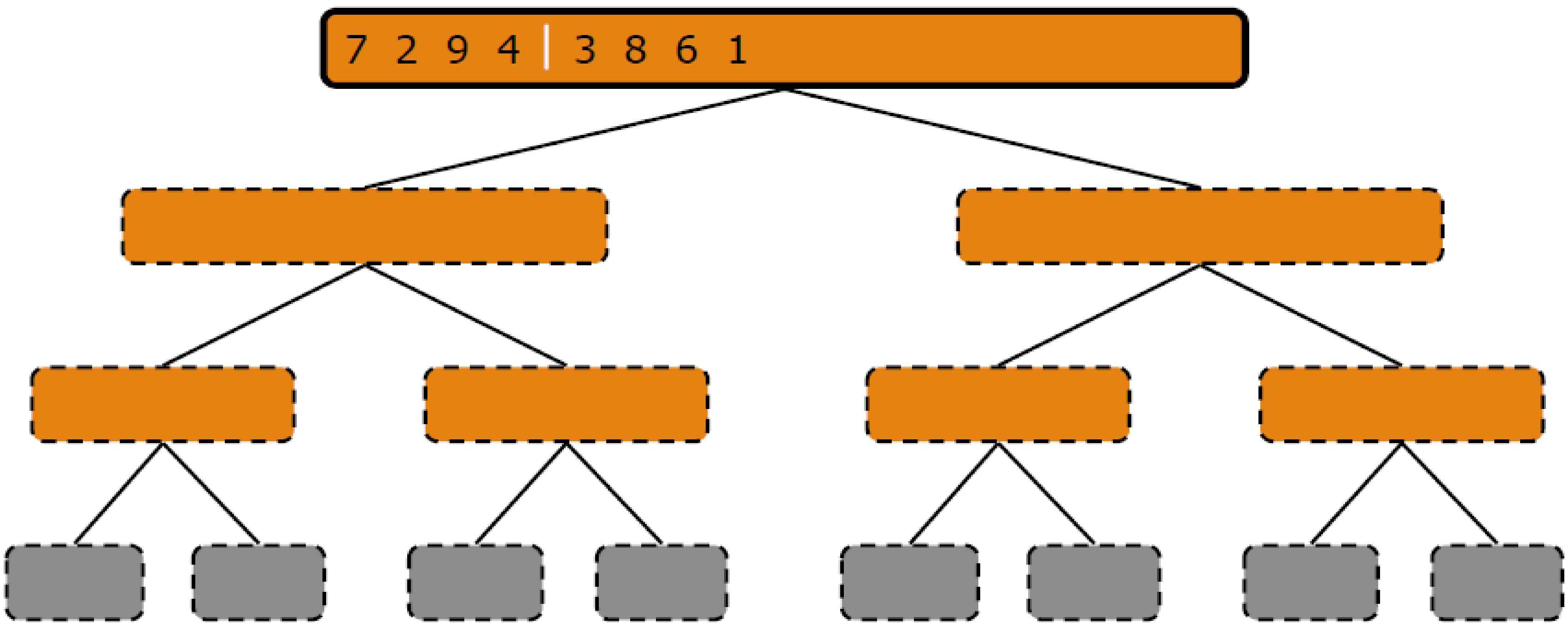
- each node represents a recursive call of merge-sort and stores
  - unsorted sequence before the execution and its partition
  - sorted sequence at the end of the execution
- the root is the initial call
- the leaves are calls on subsequences of size 0 or 1



Execution Example

if $S.size() > 1$
  $(S_1, S_2) \leftarrow partition(S, n/2)$
  $mergeSort(S_1)$
  $mergeSort(S_2)$
  $S \leftarrow merge(S_1, S_2)$

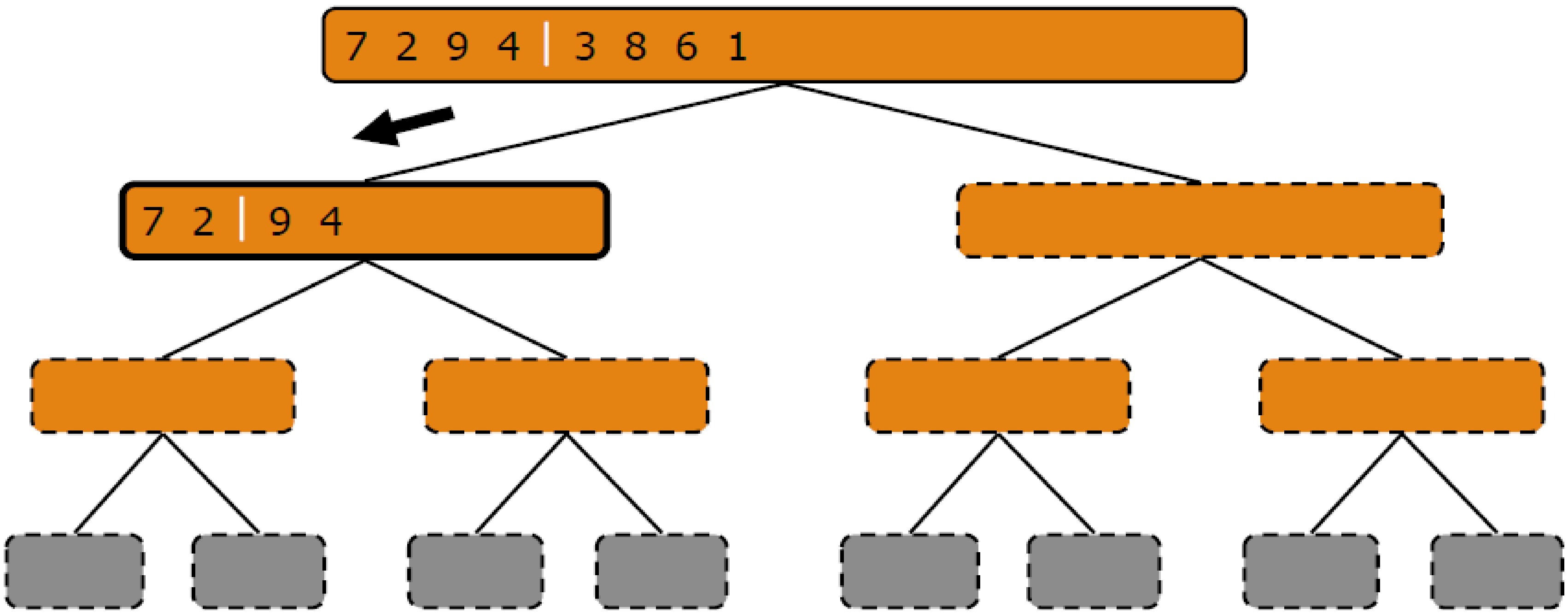
Partition



Execution Example

if $S.size() > 1$
  $(S_1, S_2) \leftarrow partition(S, n/2)$
  $mergeSort(S_1)$
  $mergeSort(S_2)$
  $S \leftarrow merge(S_1, S_2)$

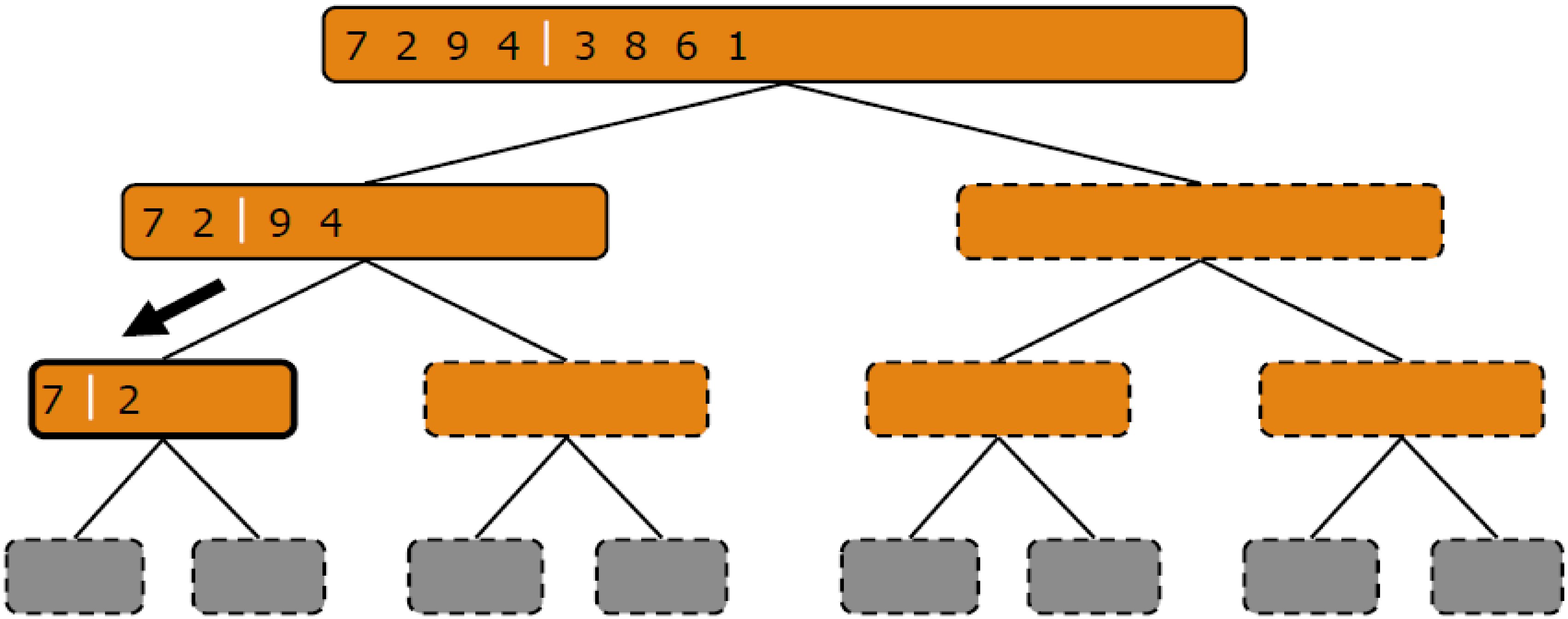
Recursive call, partition



Execution Example

if $S.size() > 1$
  $(S_1, S_2) \leftarrow partition(S, n/2)$
  $mergeSort(S_1)$
  $mergeSort(S_2)$
  $S \leftarrow merge(S_1, S_2)$

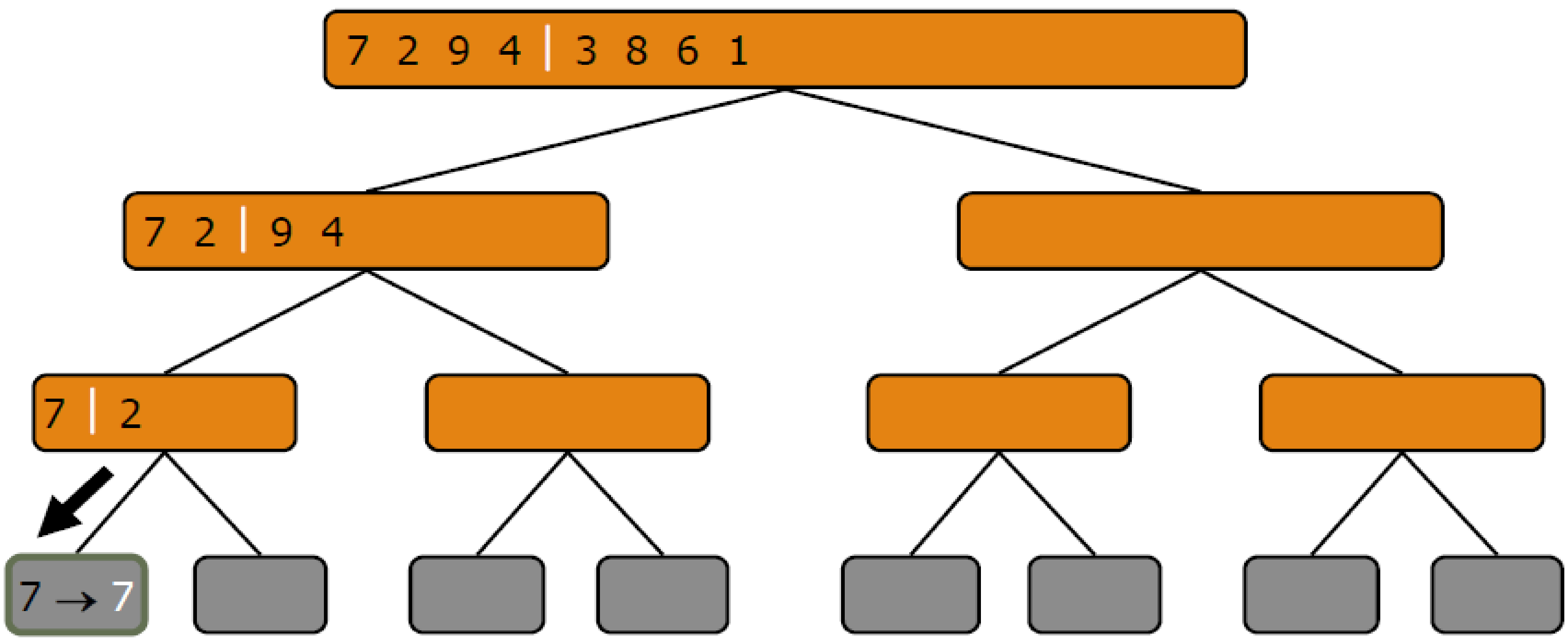
Recursive call, partition



Execution Example

if $S.size() > 1$
$(S_1, S_2) \leftarrow partition(S, n/2)$
$mergeSort(S_1)$
$mergeSort(S_2)$
$S \leftarrow merge(S_1, S_2)$

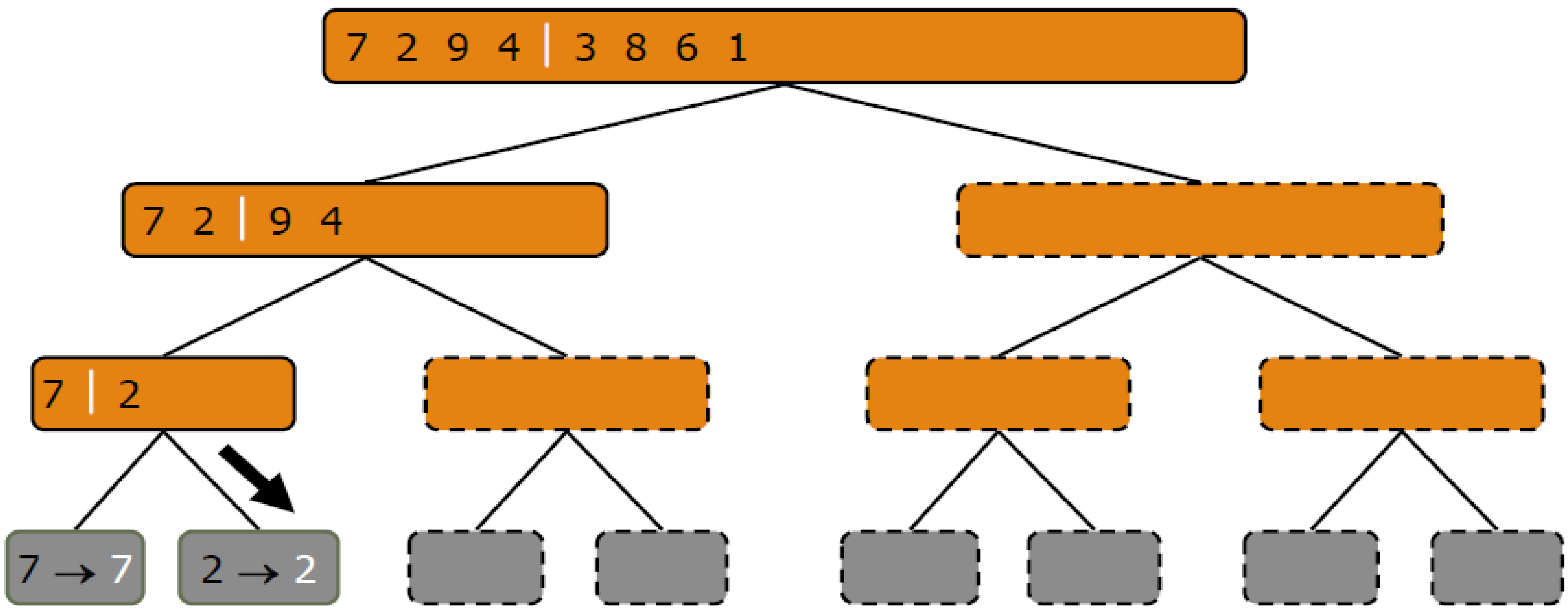
Recursive call, base case



Execution Example

if $S.size() > 1$
  $(S_1, S_2) \leftarrow partition(S, n/2)$
  $mergeSort(S_1)$
  $mergeSort(S_2)$
  $S \leftarrow merge(S_1, S_2)$

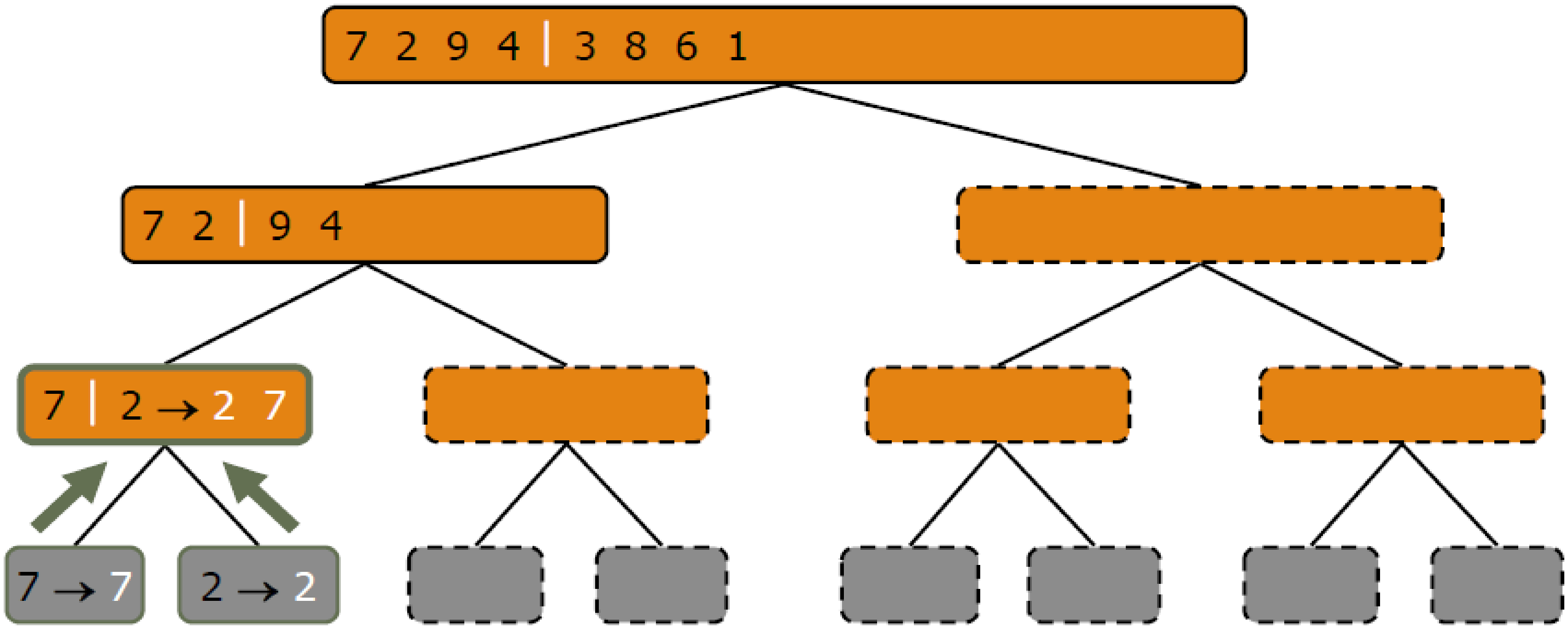
Recursive call, base case



Execution Example

```
if S.size() > 1
  (S1, S2) ← partition(S, n/2)
  mergeSort(S1)
  mergeSort(S2)
  S ← merge(S1, S2)
```

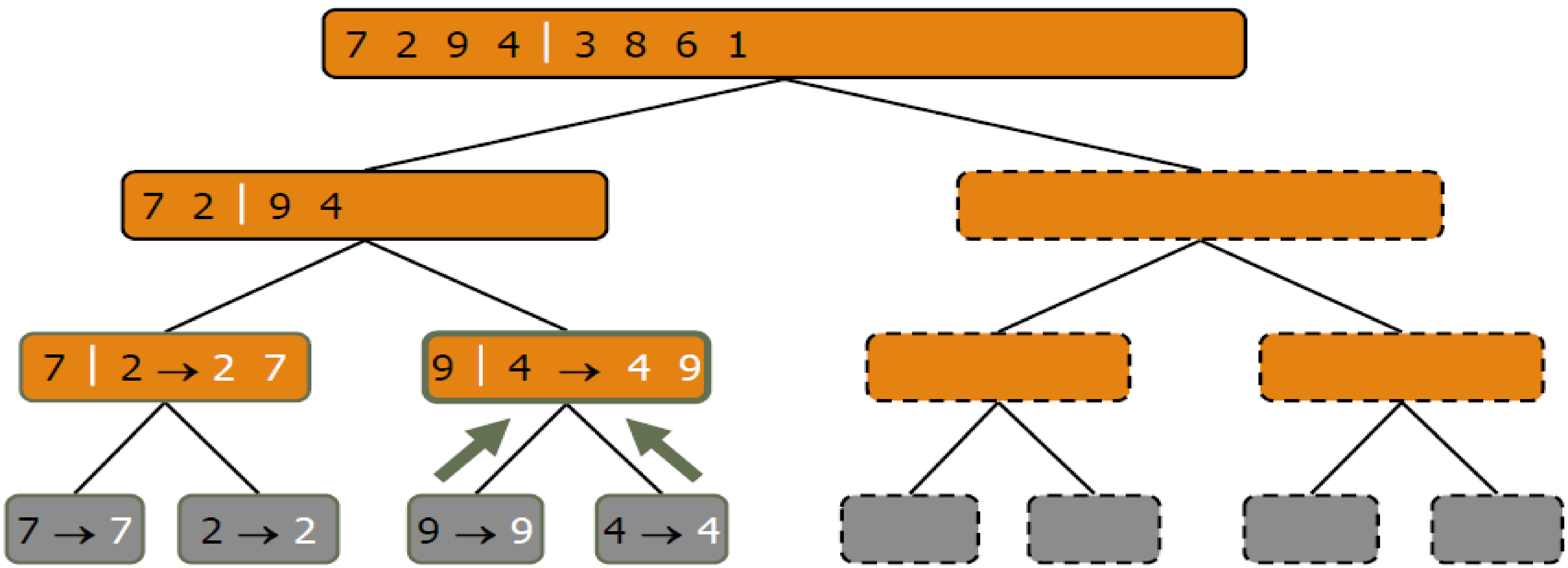
Merge



Execution Example

if $S.size() > 1$
  $(S_1, S_2) \leftarrow partition(S, n/2)$
  $mergeSort(S_1)$
  $mergeSort(S_2)$
  $S \leftarrow merge(S_1, S_2)$

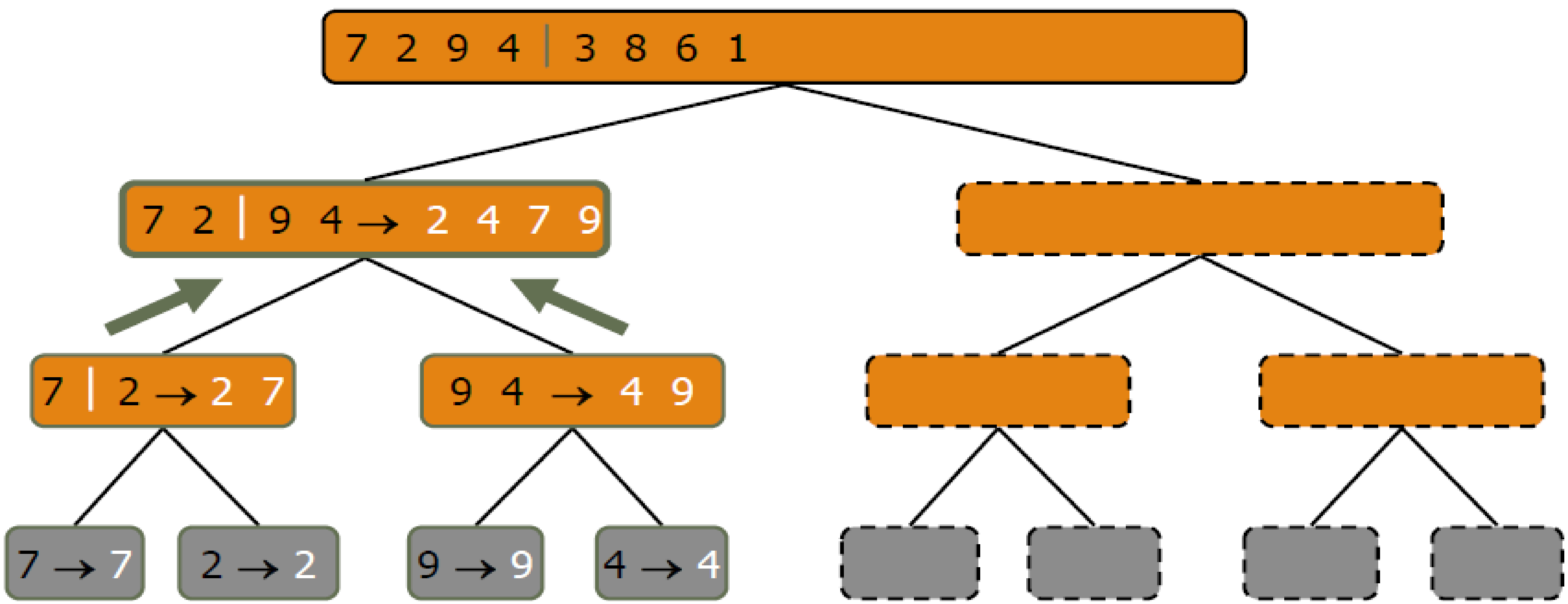
Recursive call, …, base case, merge



Execution Example

if $S.size() > 1$
$(S_1, S_2) \leftarrow partition(S, n/2)$
$mergeSort(S_1)$
$mergeSort(S_2)$
$S \leftarrow merge(S_1, S_2)$

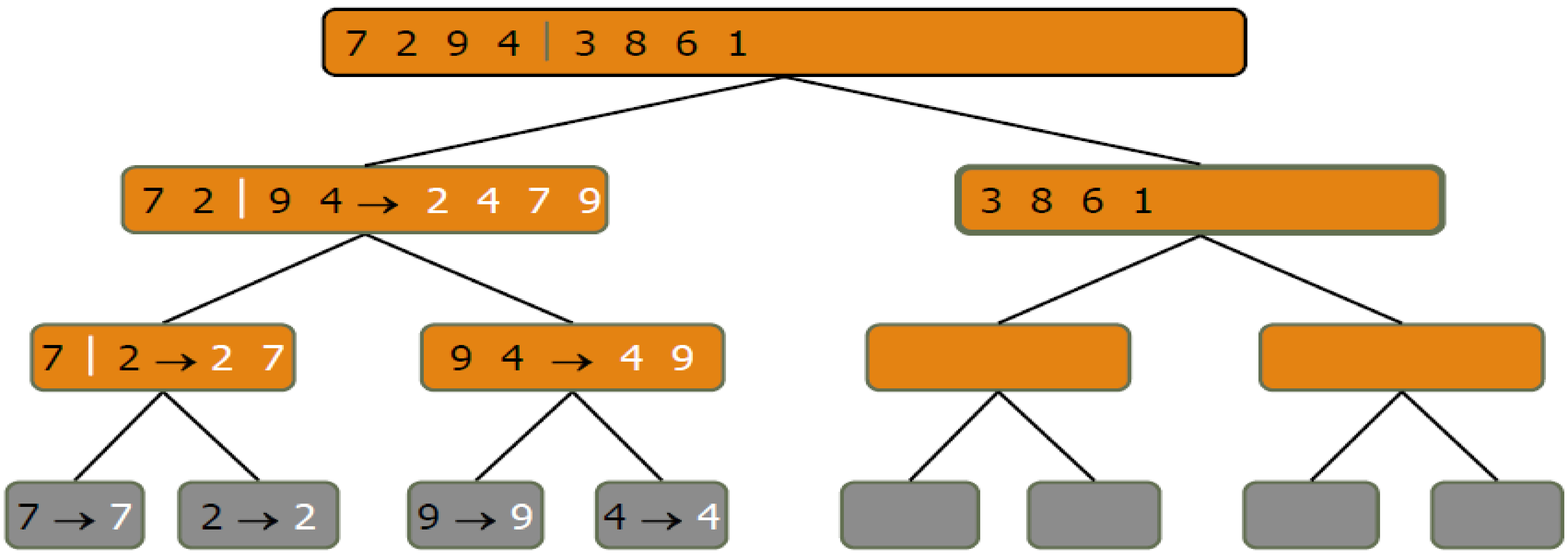
Merge



Execution Example

if $S.size() > 1$
  $(S_1, S_2) \leftarrow partition(S, n/2)$
  $mergeSort(S_1)$
  $mergeSort(S_2)$
  $S \leftarrow merge(S_1, S_2)$

Recursive call, ..., merge, merge



You try

Trace the steps that a merge sort takes when sorting the following array into ascending order: 9 6 2 4 8 7 5 3

Analysis of Merge-Sort

The height ***h*** of the merge-sort tree is ***O***(log ***n***)

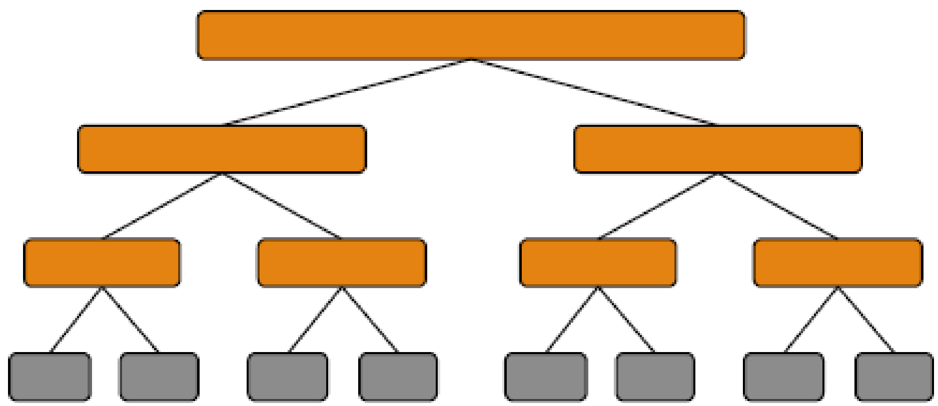
- at each recursive call we divide in half the sequence,

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	$n/2^i$
…	…	…



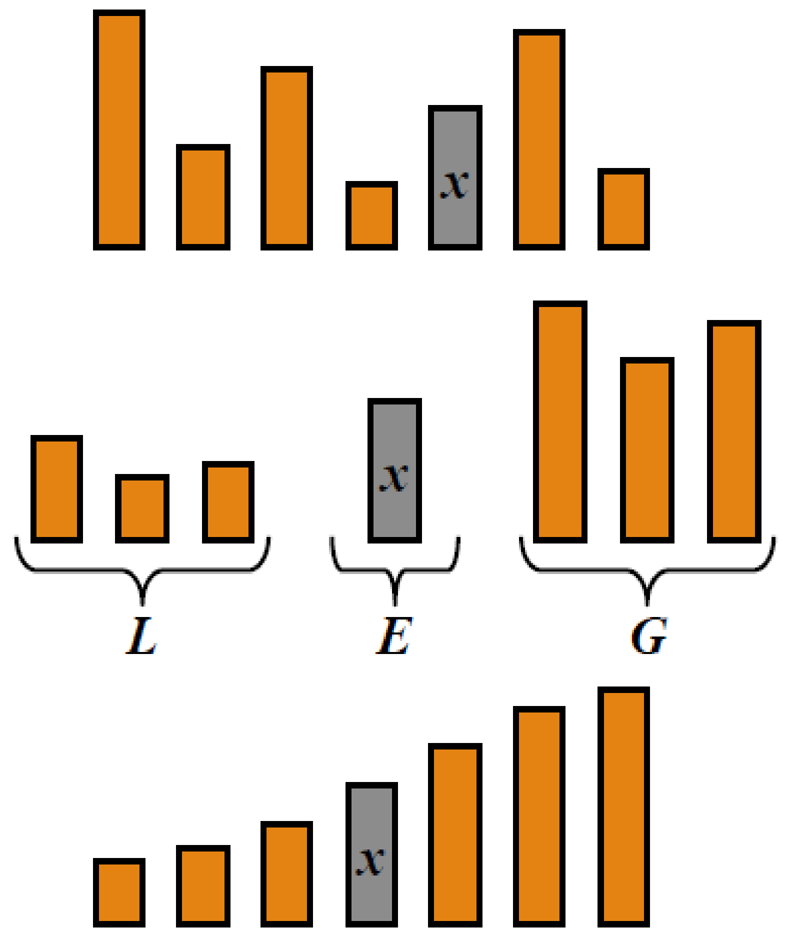
Summary of Sorting Algorithms

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

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***
- Conquer : recursively sort ***L*** and ***G***
- Combine: 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

Partition

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 ← empty sequences
x ← S.remove(p)
while ¬S.isEmpty()
    y ← 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
```

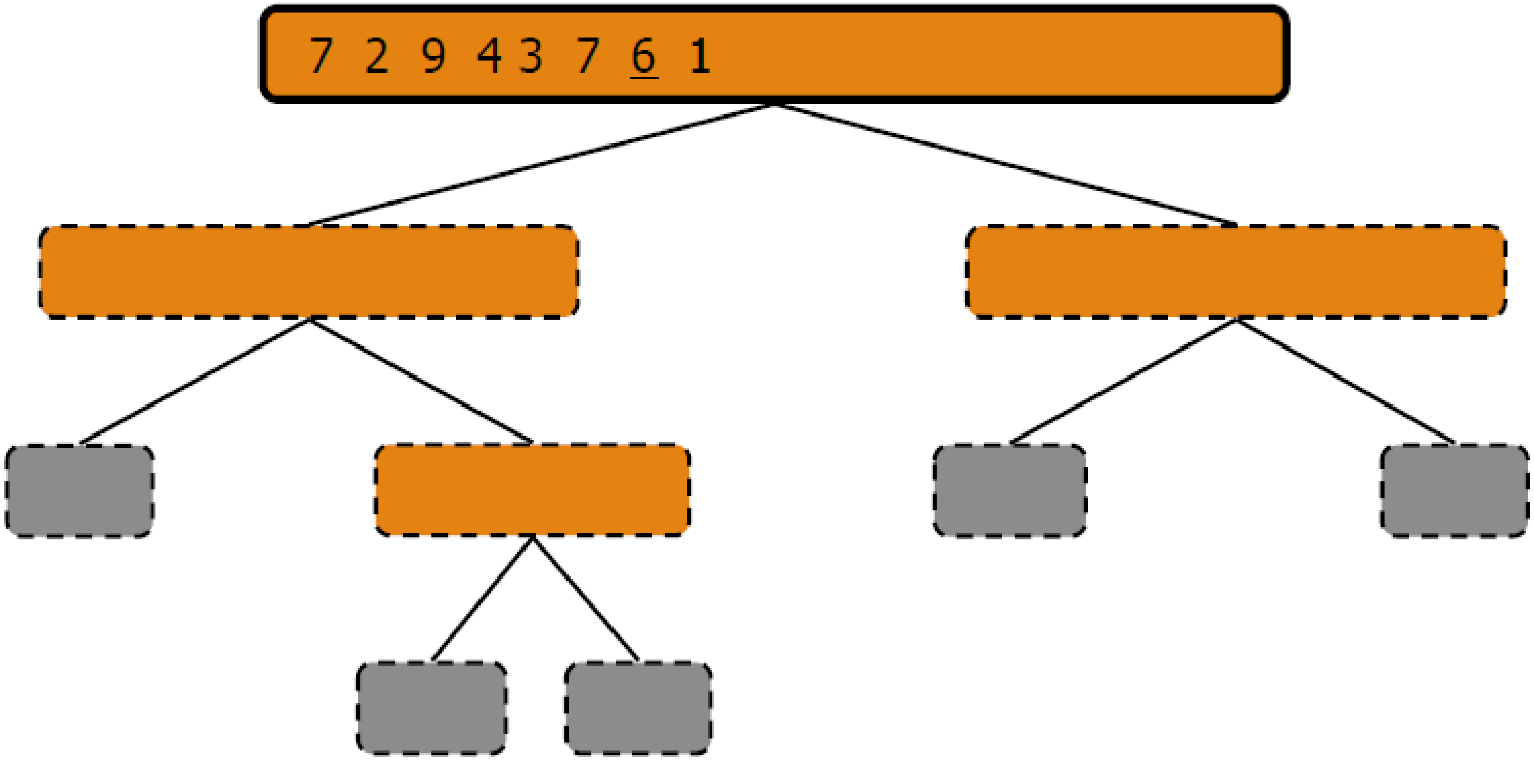
Quick-Sort Tree

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

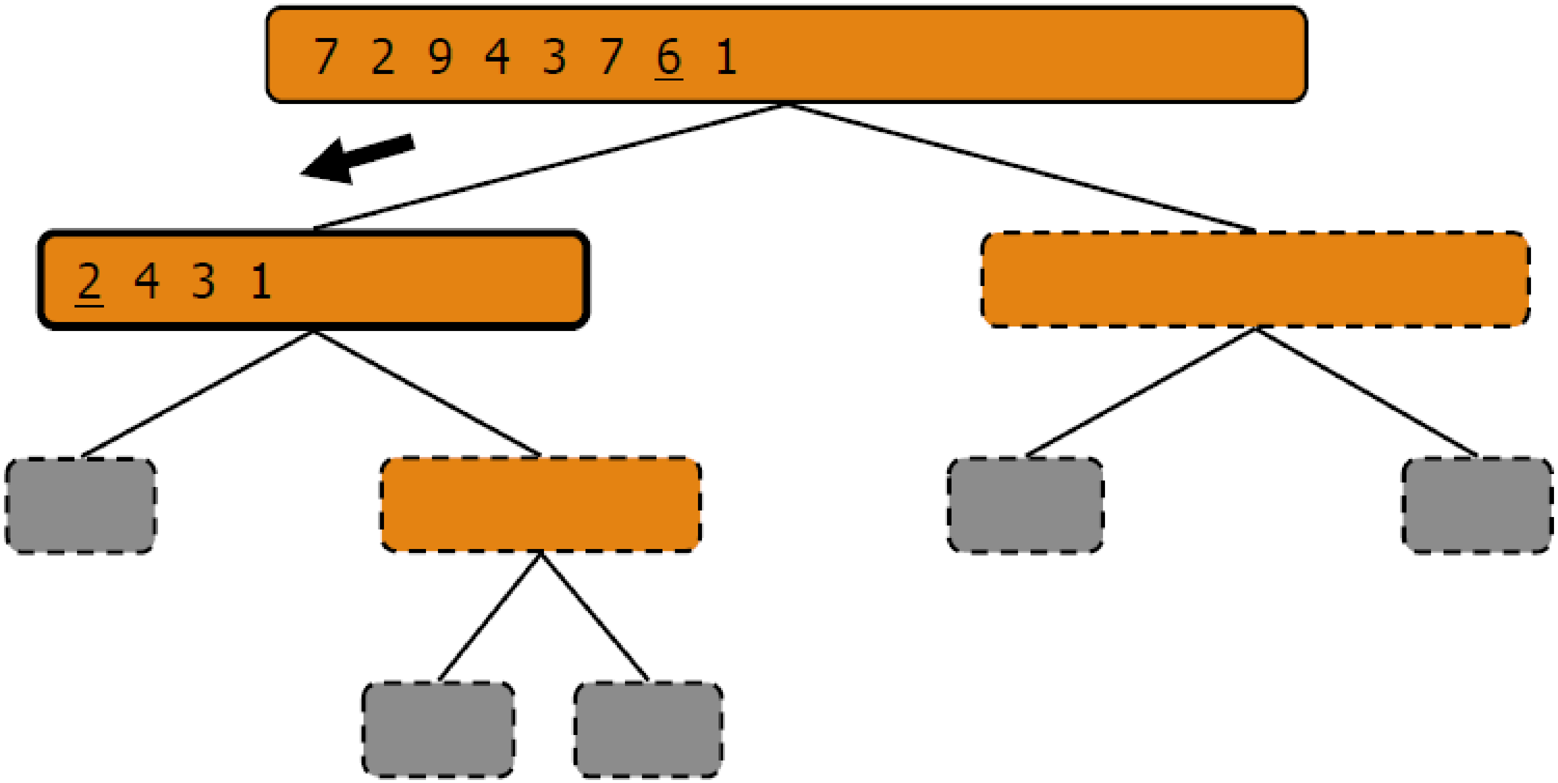
Execution Example

Pivot selection



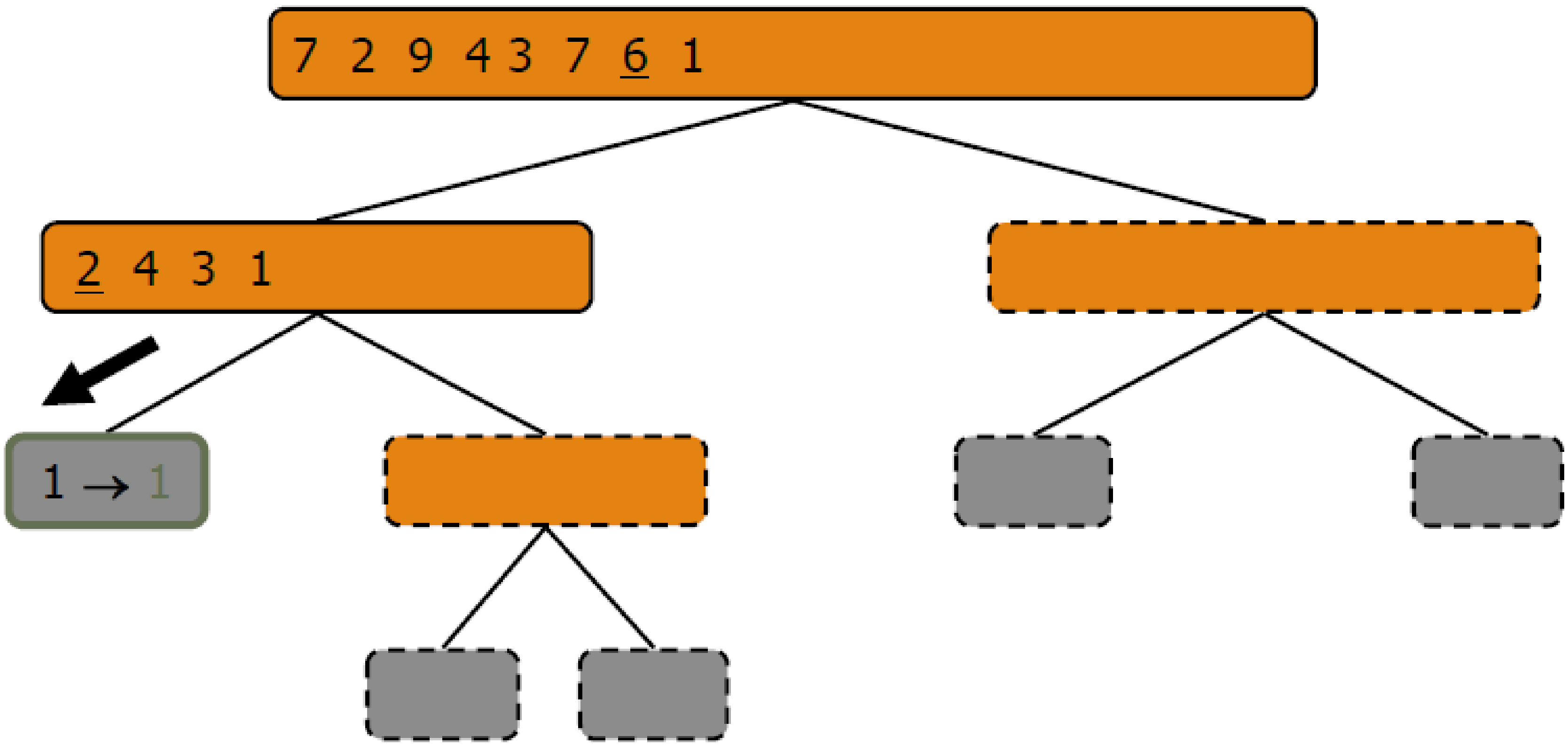
Execution Example (cont.)

Partition, recursive call, pivot selection



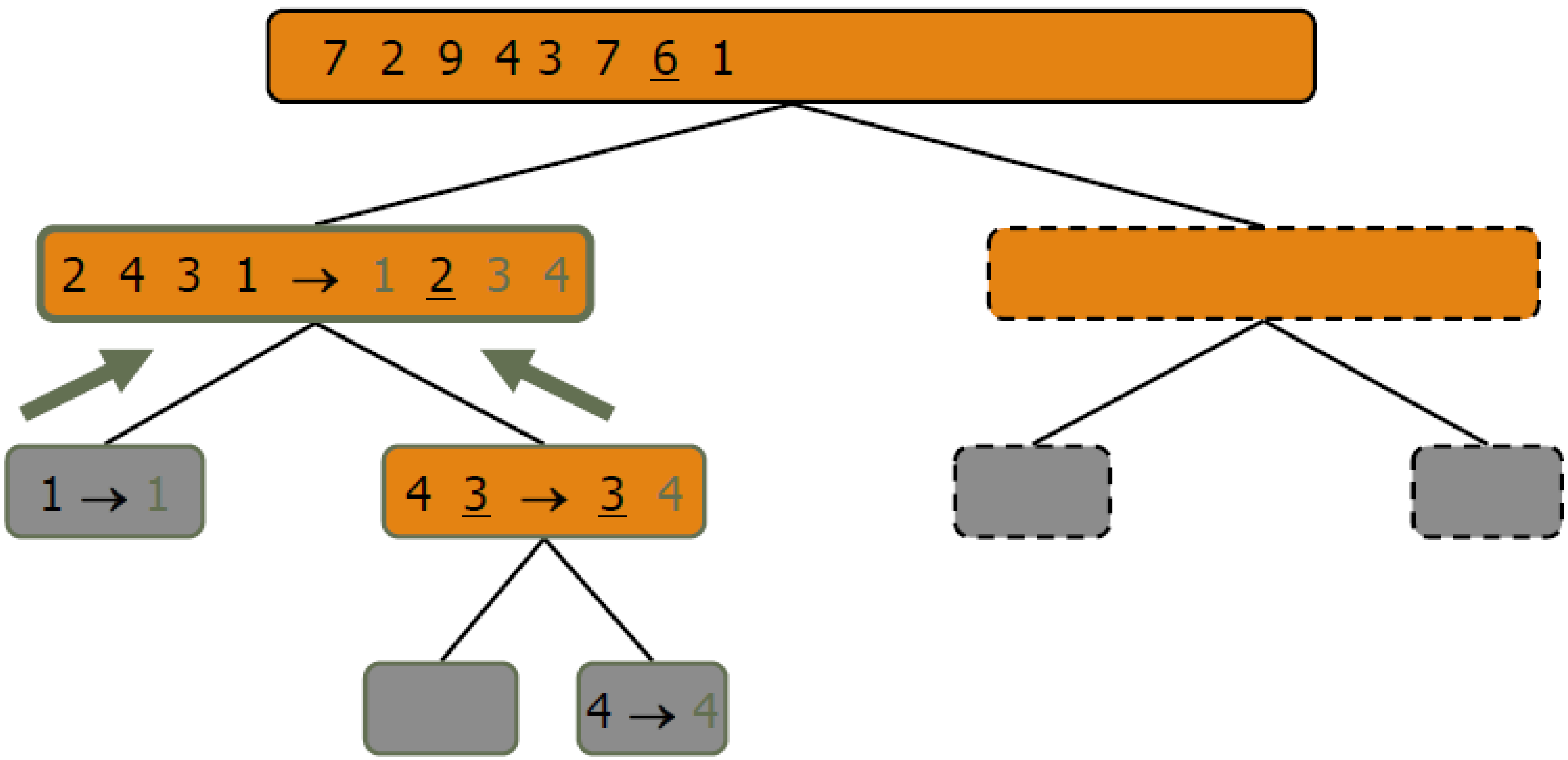
Execution Example (cont.)

Partition, recursive call, base case



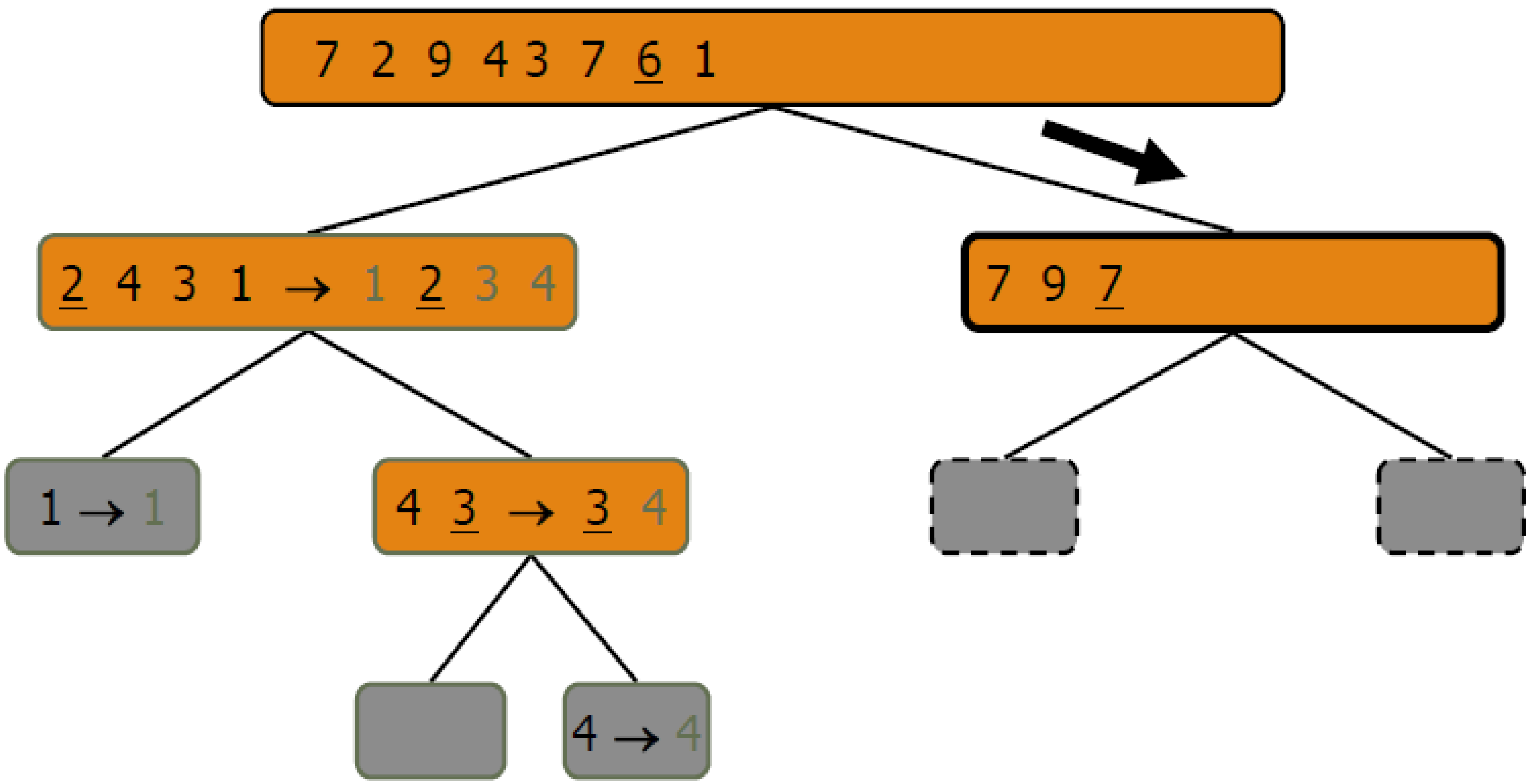
Execution Example (cont.)

Recursive call, ..., base case, join



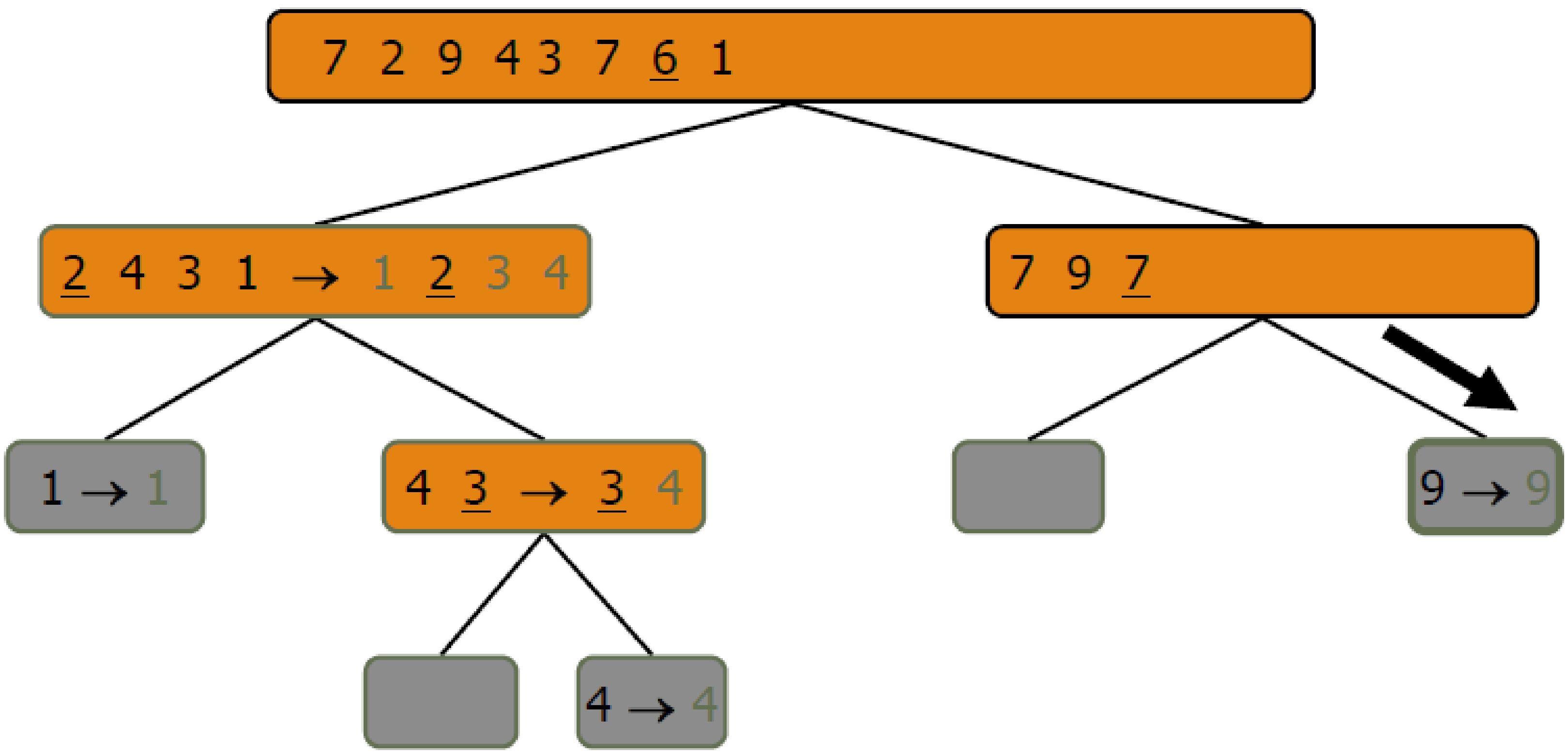
Execution Example (cont.)

Recursive call, pivot selection



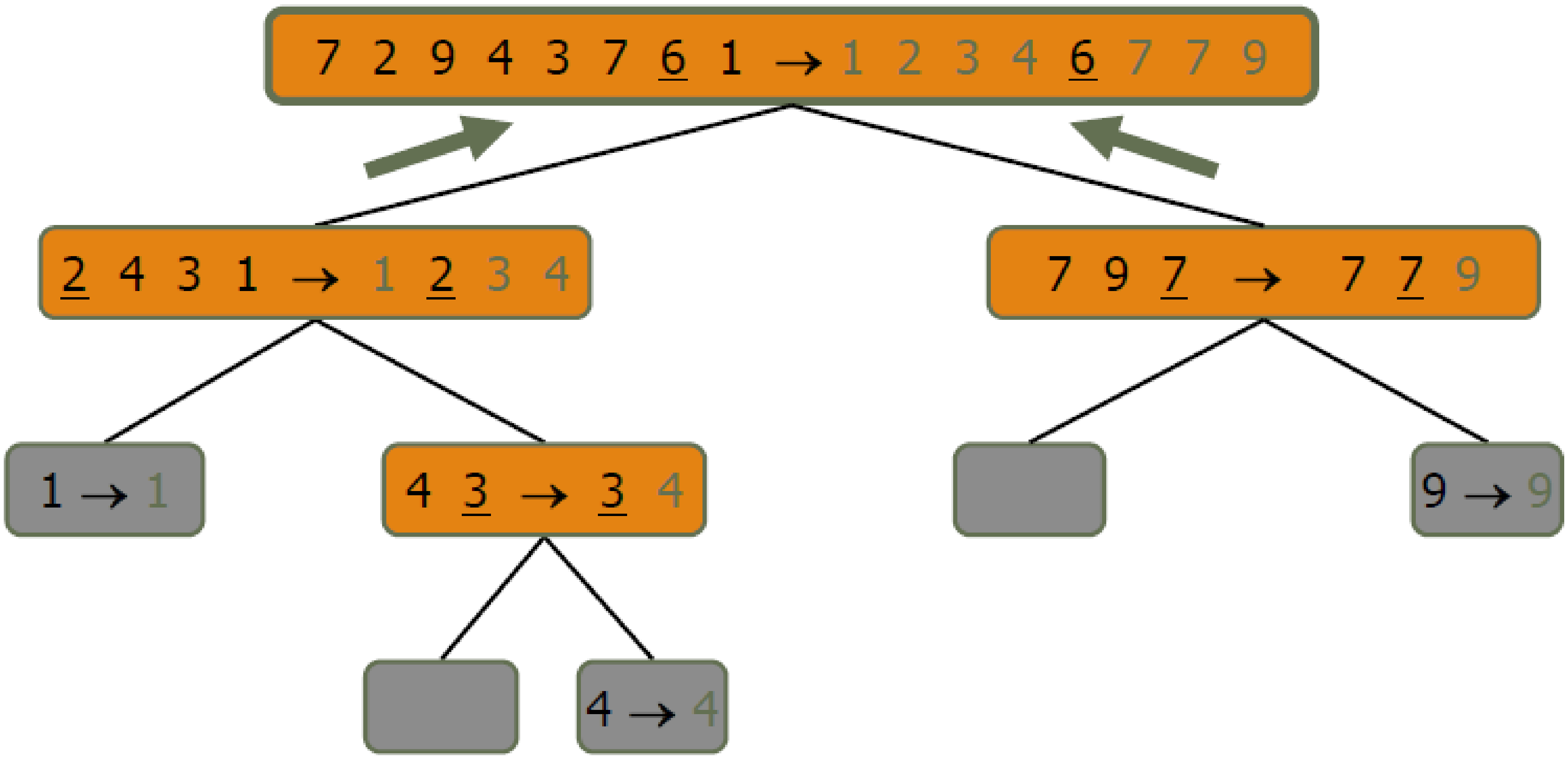
Execution Example (cont.)

Partition, ..., recursive call, base case



Execution Example (cont.)

Join, join



You try

Trace the steps that a quick sort takes when sorting the following array into ascending order: 9 6 2 4 8 7 5 3

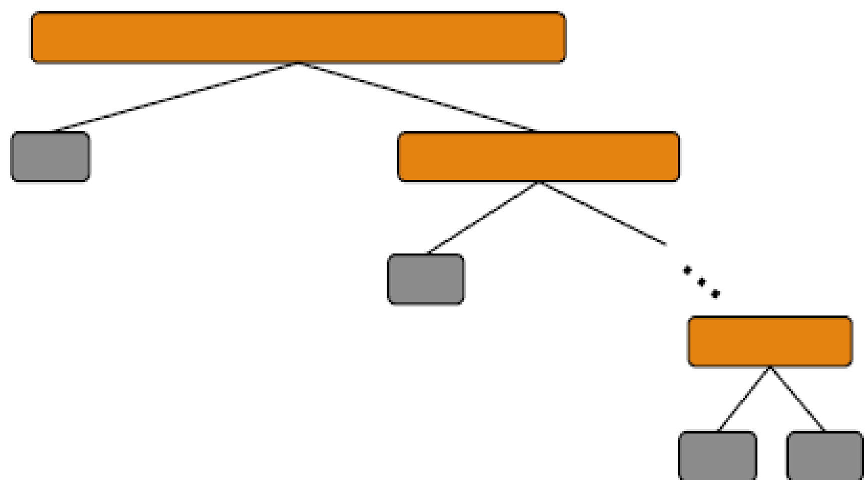
Worst-case Running Time

- 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 $O(n^2)$

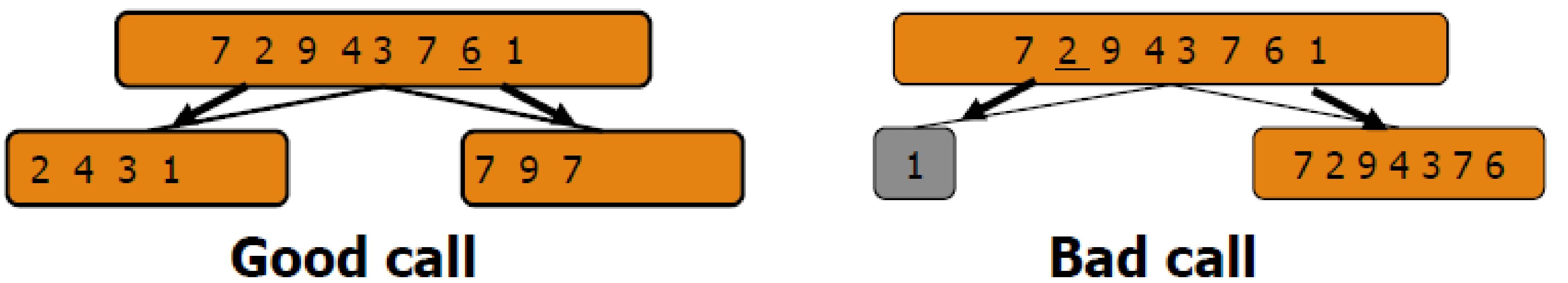
depth	time
0	n
1	$n - 1$
…	…
$n - 1$	1



Expected Running Time

Consider a recursive call of quick-sort on a sequence of size s

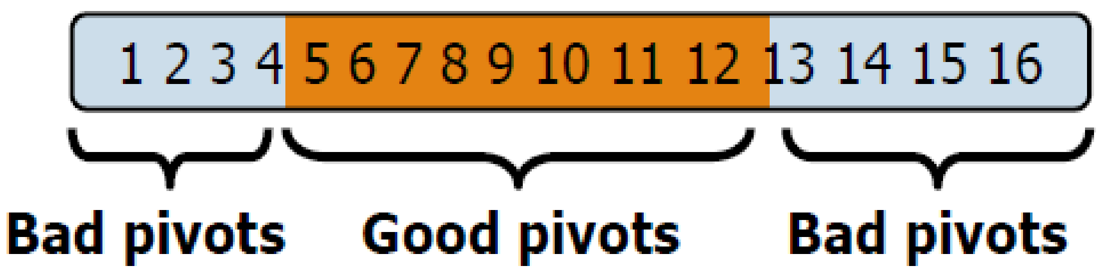
- **Good call:** the sizes of L and G are each less than $3s/4$
- **Bad call:** one of L and G has size greater than $3s/4$



Expected Running Time …

A call is good with probability 1/2

- 1/2 of the possible pivots cause good calls:



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 and G. (< and =>)

h k

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 **<u>6</u>** 9 (pivot = 6)

Repeat until h and k cross:

- Scan h to the right until finding an element $\geq$ x.
- Scan k to the left until finding an element < x.
- Swap elements at indices h and k

h k

3 2 5 1 0 $\boxed{7}$ 3 5 9 $\boxed{2}$ 7 9 8 9 7 **<u>6</u>** 9

You try

Trace the steps that a quick sort with in place partitioning takes when sorting the following array into ascending order:
9 6 2 4 8 7 5 3

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

References

These slides has been extracted, modified and updated from original slides of :

1. Data Structures and Algorithms in Java, 6th edition. John Wiley& Sons,
2. Introduction to Algorithms, 3rd Edition. Thomas H. Cormen and Charles E. Leiserson