

Sorting

Data structures and Algorithms

Acknowledgement:

These slides are adapted from slides provided with *Data Structures and Algorithms in C++*
Goodrich, Tamassia and Mount (Wiley, 2004)

Outline

- ◆ Bubble Sort
- ◆ Insertion Sort
- ◆ Merge Sort
- ◆ Quick Sort

Bubble Sort

Algorithm

1. Compare each pair of adjacent elements from the beginning of an array and, if they are in reversed order, swap them.
2. If at least one swap has been done, repeat step 1.

Reference: http://www.algolist.net/Algorithms/Sorting/Bubble_sort

1st pass

5 1 12 -5 16

unsorted

5 1 12 -5 16

5 > 1, swap

1 5 12 -5 16

5 < 12, ok

1 5 12 -5 16

12 > -5, swap

1 5 -5 12 16

12 < 16, ok

2nd pass

1 5 -5 12 16

1 < 5, ok

1 5 -5 12 16

5 > -5, swap

1 -5 5 12 16

5 < 12, ok

3rd pass

1 -5 5 12 16

1 > -5, swap

-5 1 5 12 16

1 < 5, ok

4th pass

-5 1 5 12 16

-5 < 1, ok

-5 1 5 12 16

sorted

Bubble Sort pseudocode

Algorithm *bubbleSort*(S, C)

Input sequence S with n elements, comparator C

Output sequence S sorted according to C

do

$swapped \leftarrow false$

for each i **in** 1 **to** $length(S) - 1$ **inclusive do:**

if $S[i - 1] > S[i]$ *according to* C **then**

$swap(S[i - 1], S[i])$

$swapped \leftarrow true$

while $swapped$

```

For pass in 1 ... n-1:
    for j in 1..n-pass
        if S[j-1]>S[j]:
            swap(s[j-1], s[j])

```

```

For i in 1 ... n-1:
    for j in 1..n-i
        if S[j-1]>S[j]:
            swap(s[j-1], s[j])

```

5	1	12	-5	16
---	---	----	----	----

unsorted

5	1	12	-5	16
---	---	----	----	----

5 > 1, swap

1	5	12	-5	16
---	---	----	----	----

5 < 12, ok

1	5	12	-5	16
---	---	----	----	----

12 > -5, swap

1	5	-5	12	16
---	---	----	----	----

12 < 16, ok

1	5	-5	12	16
---	---	----	----	----

1 < 5, ok

1	5	-5	12	16
---	---	----	----	----

5 > -5, swap

1	-5	5	12	16
---	----	---	----	----

5 < 12, ok

1	-5	5	12	16
---	----	---	----	----

1 > -5, swap

-5	1	5	12	16
----	---	---	----	----

1 < 5, ok

-5	1	5	12	16
----	---	---	----	----

-5 < 1, ok

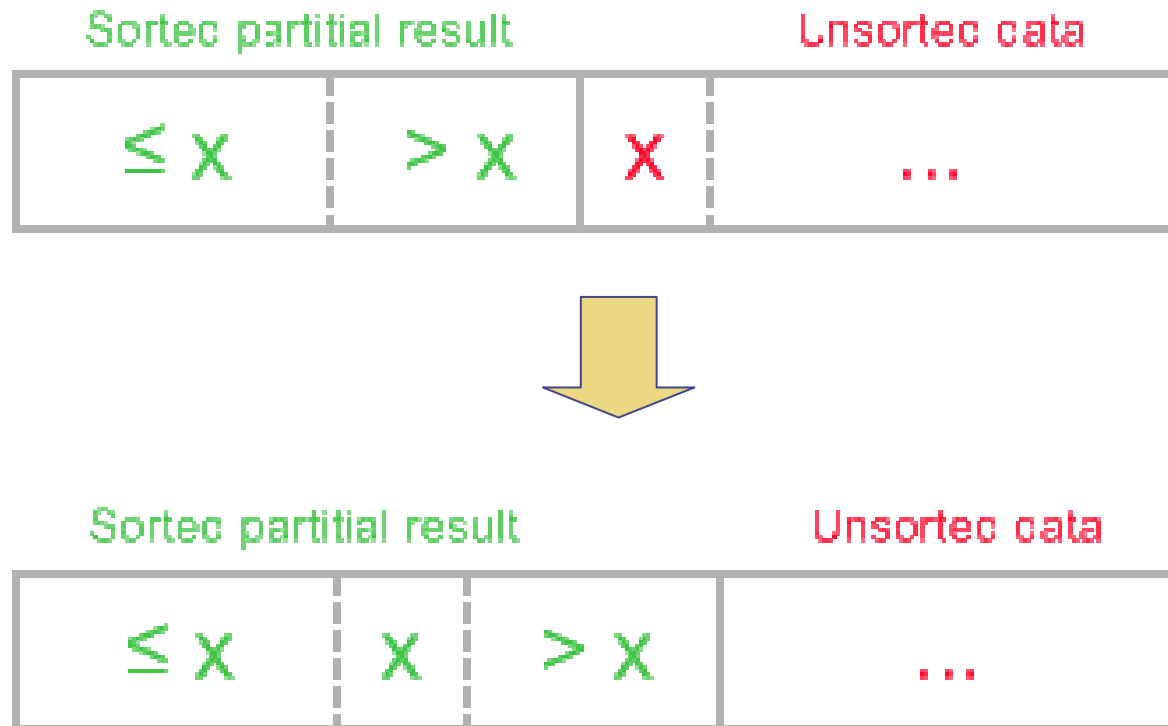
-5	1	5	12	16
----	---	---	----	----

sorted

Bubble Sort performance

- ◆ Worst-case & average-case: $O(n^2)$
- ◆ Best-case: (over an already-sorted list) : $O(n)$

Insertion Sort



Reference: http://www.algolist.net/Algorithms/Sorting/Insertion_sort

1st pass

7	-5	2	16	4
---	----	---	----	---

unsorted

7	-5	2	16	4
---	----	---	----	---

-5 to be inserted

?	7	2	16	4
---	---	---	----	---

7 > -5, shift

-5	7	2	16	4
----	---	---	----	---

reached left boundary, insert -5

2nd pass

-5	7	2	16	4
----	---	---	----	---

2 to be inserted

-5	?	7	16	4
----	---	---	----	---

7 > 2, shift

-5	2	7	16	4
----	---	---	----	---

-5 < 2, insert 2

3rd pass

-5	2	7	16	4
----	---	---	----	---

16 to be inserted

-5	2	7	16	4
----	---	---	----	---

7 < 16, insert 16

4th pass

-5	2	7	16	4
----	---	---	----	---

4 to be inserted

-5	2	7	?	16
----	---	---	---	----

16 > 4, shift

-5	2	?	7	16
----	---	---	---	----

7 > 4, shift

-5	2	4	7	16
----	---	---	---	----

2 < 4, insert 4

-5	2	4	7	16
----	---	---	---	----

sorted

Insertion Sort pseudocode

Algorithm *insertionSort*(S, C)

Input sequence S with n elements, comparator C

Output sequence S sorted according to C

for i **from** 1 **to** $length(S)$ **do**

$j \leftarrow i$

while $j > 0$ **&&** $S[j - 1] > S[j]$ **then**

$swap(S[j - 1], S[j])$

$j--$

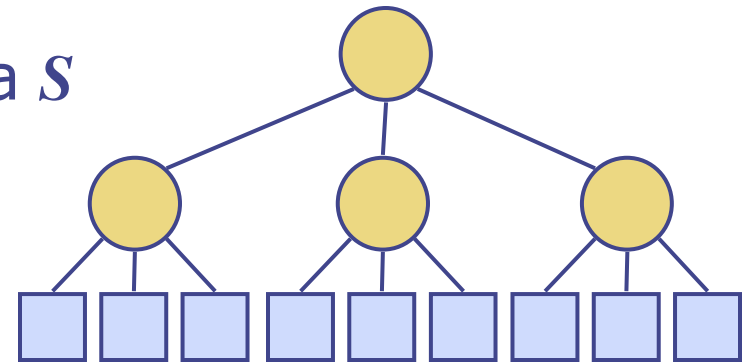
Insertion Sort performance

- ◆ Worst-case & average-case: $O(n^2)$
- ◆ Best-case: (over an already-sorted list) : $O(n)$
- ◆ adaptive (performance adapts to the initial order of elements);
- ◆ stable (insertion sort retains relative order of the same elements);
- ◆ in-place (requires constant amount of additional space);
- ◆ online (new elements can be added during the sort).

Divide-and-Conquer

◆ **Divide-and conquer** is a general algorithm design paradigm:

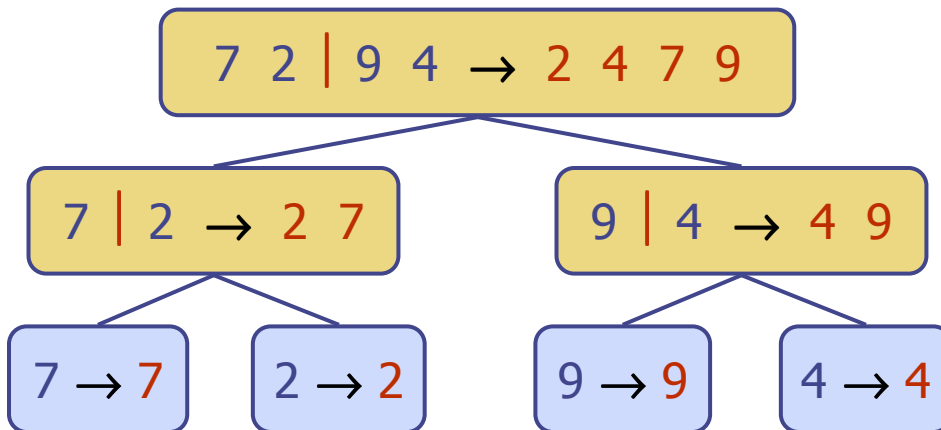
- **Divide**: divide the input data S in two or more disjoint subsets S_1, S_2, \dots
- **Recur**: solve the subproblems recursively
- **Conquer**: combine the solutions for S_1, S_2, \dots , into a solution for S



◆ The base case for the recursion are subproblems of constant size

◆ Analysis can be done using **recurrence equations**

Merge Sort



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, C)

Input sequence S with n elements, comparator C

Output sequence S sorted according to C

if $S.size() > 1$

$(S_1, S_2) \leftarrow partition(S, n/2)$

mergeSort(S_1, C)

mergeSort(S_2, C)

$S \leftarrow merge(S_1, S_2)$

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

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

else

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

while $\neg A.isEmpty()$

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

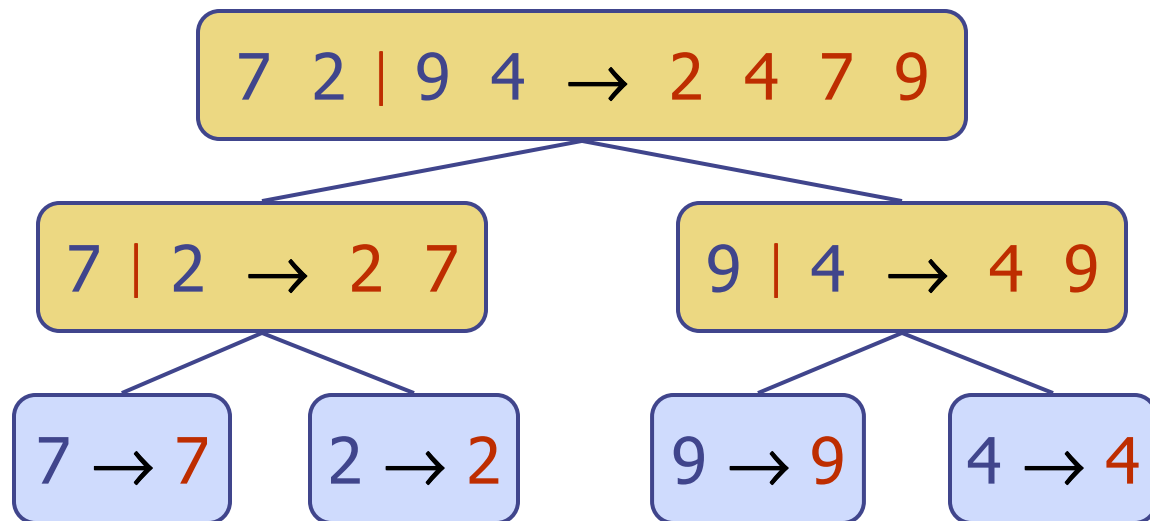
while $\neg B.isEmpty()$

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

return S

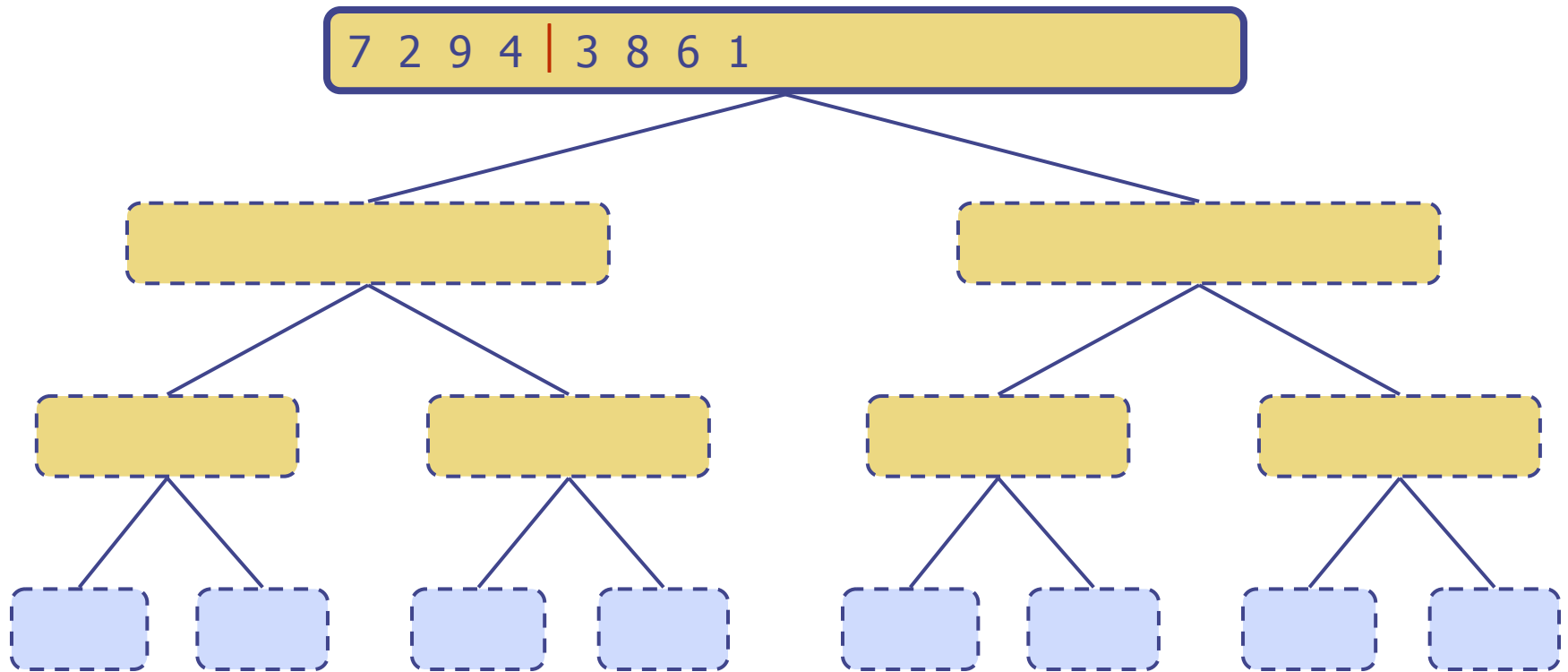
Merge-Sort Tree

- ◆ 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



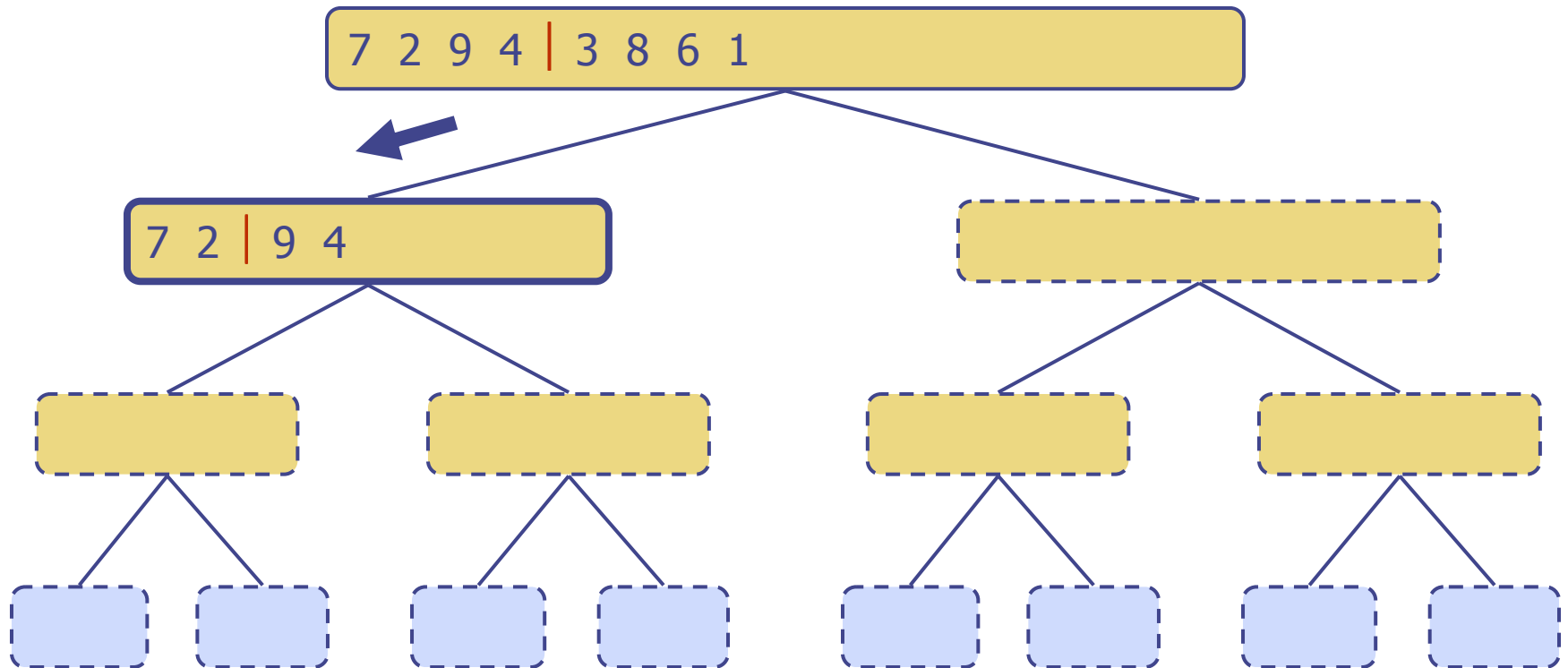
Execution Example

◆ Partition



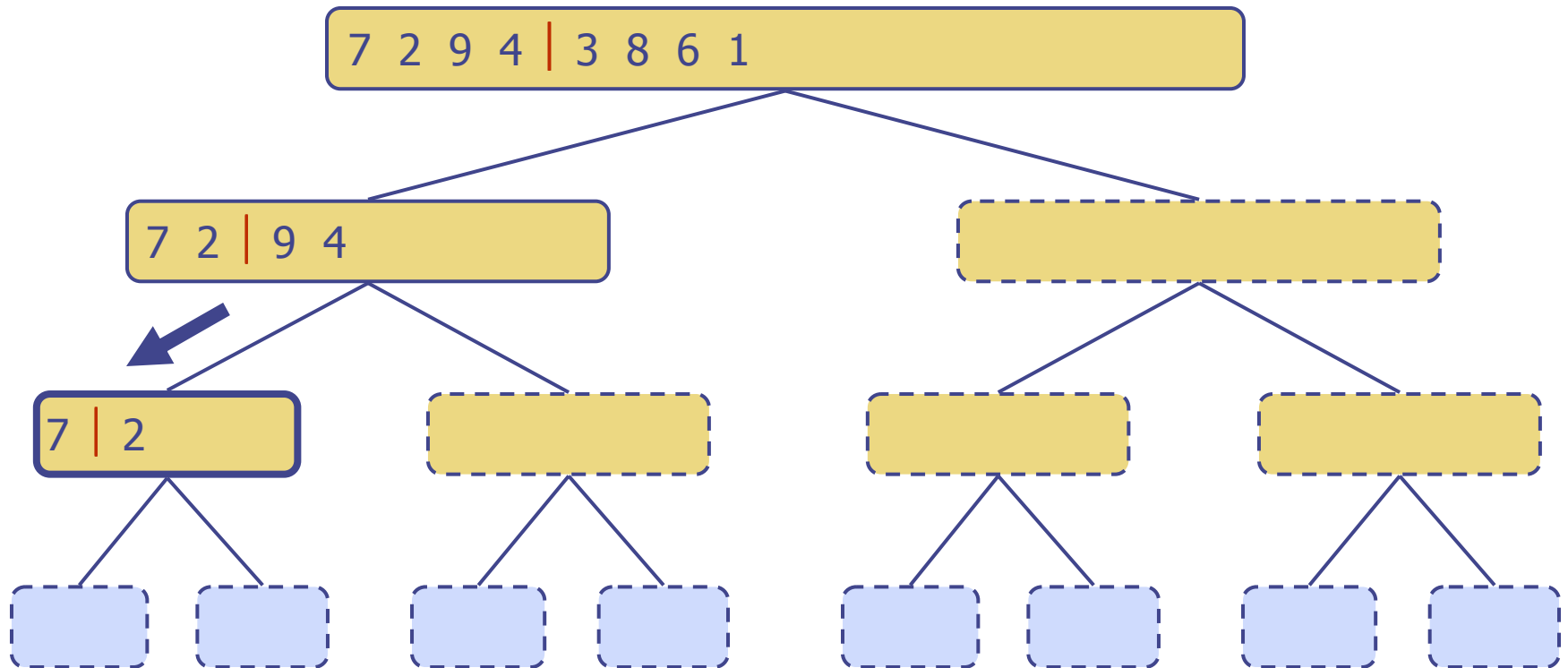
Execution Example (cont.)

◆ Recursive call, partition



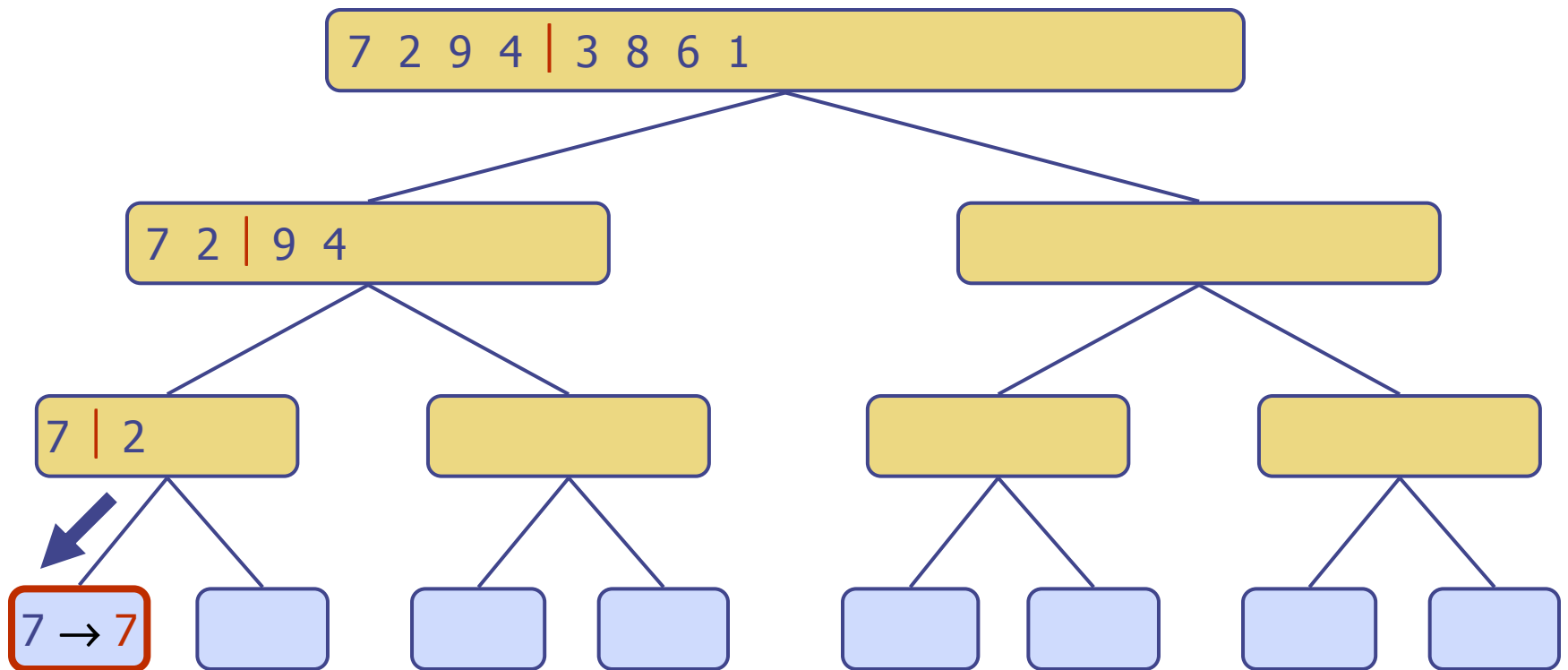
Execution Example (cont.)

◆ Recursive call, partition



Execution Example (cont.)

◆ Recursive call, base case

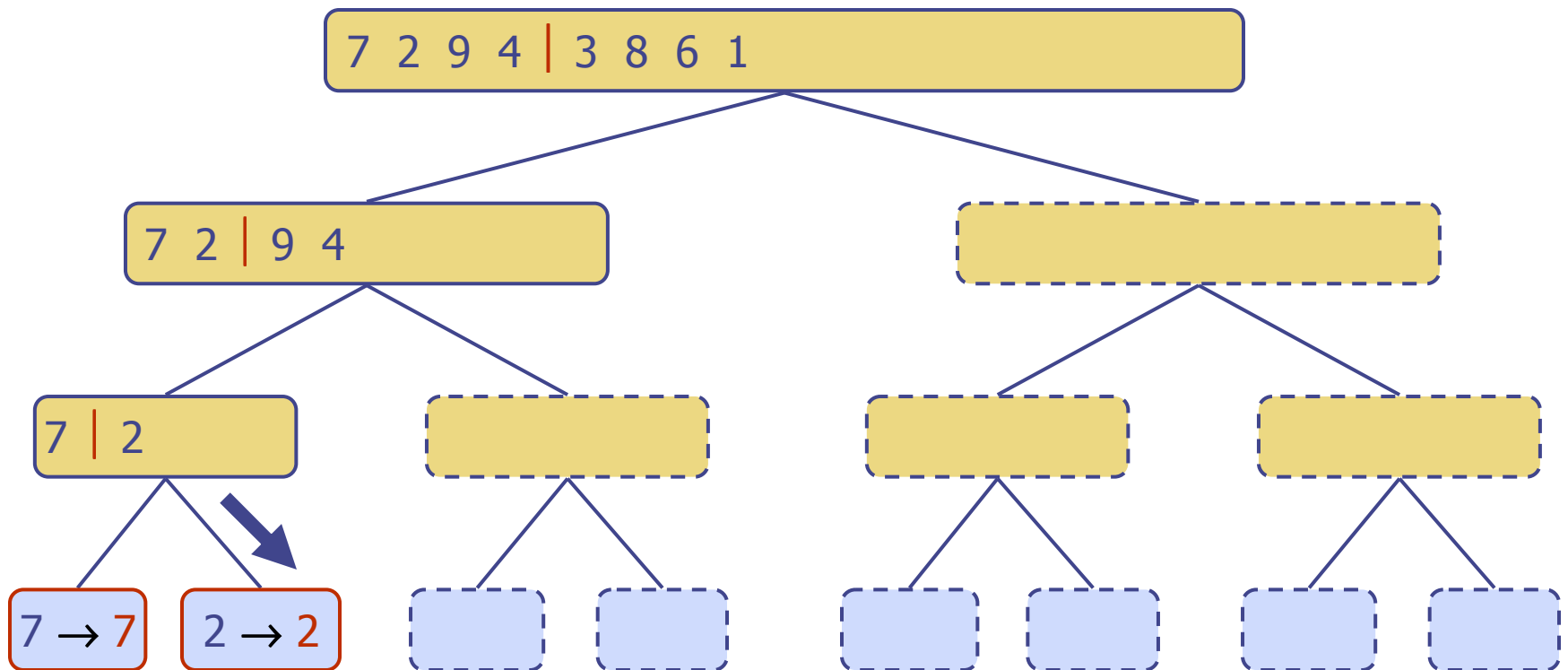


Sorting

20

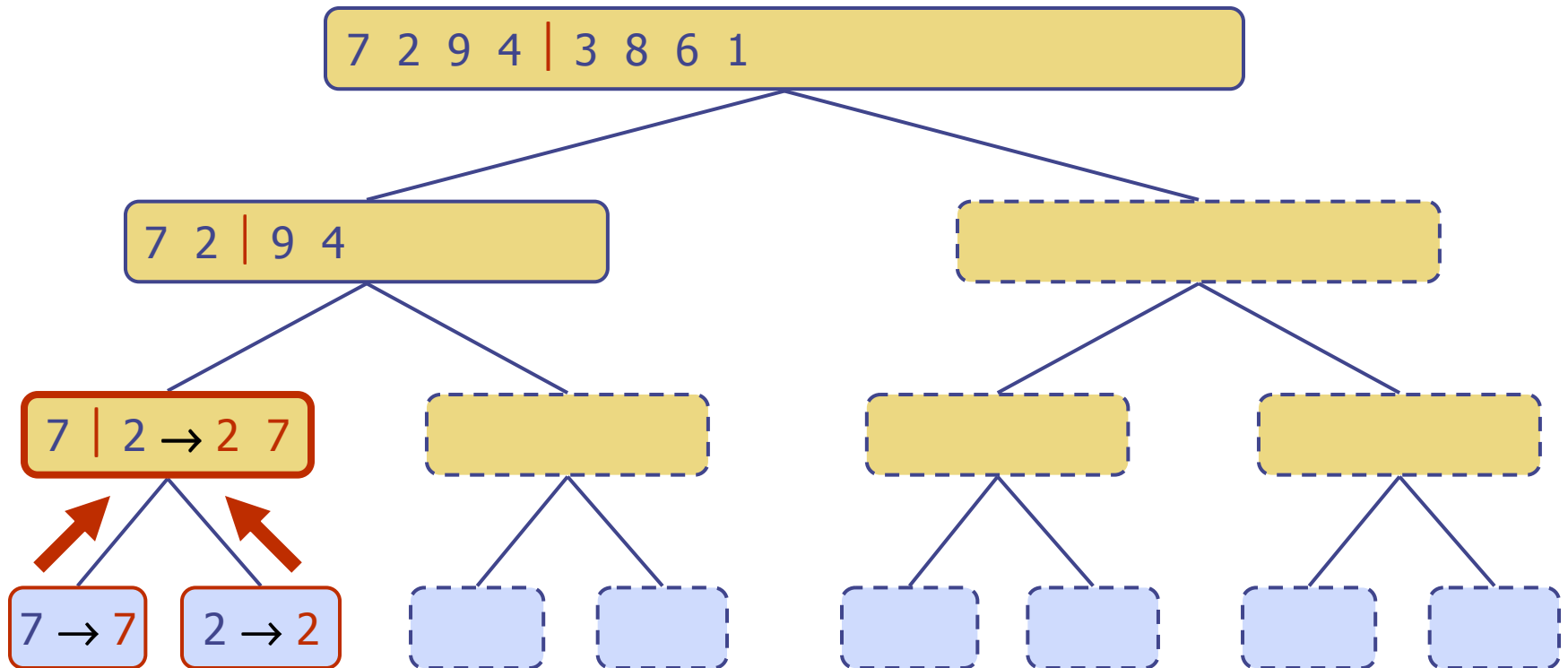
Execution Example (cont.)

◆ Recursive call, base case



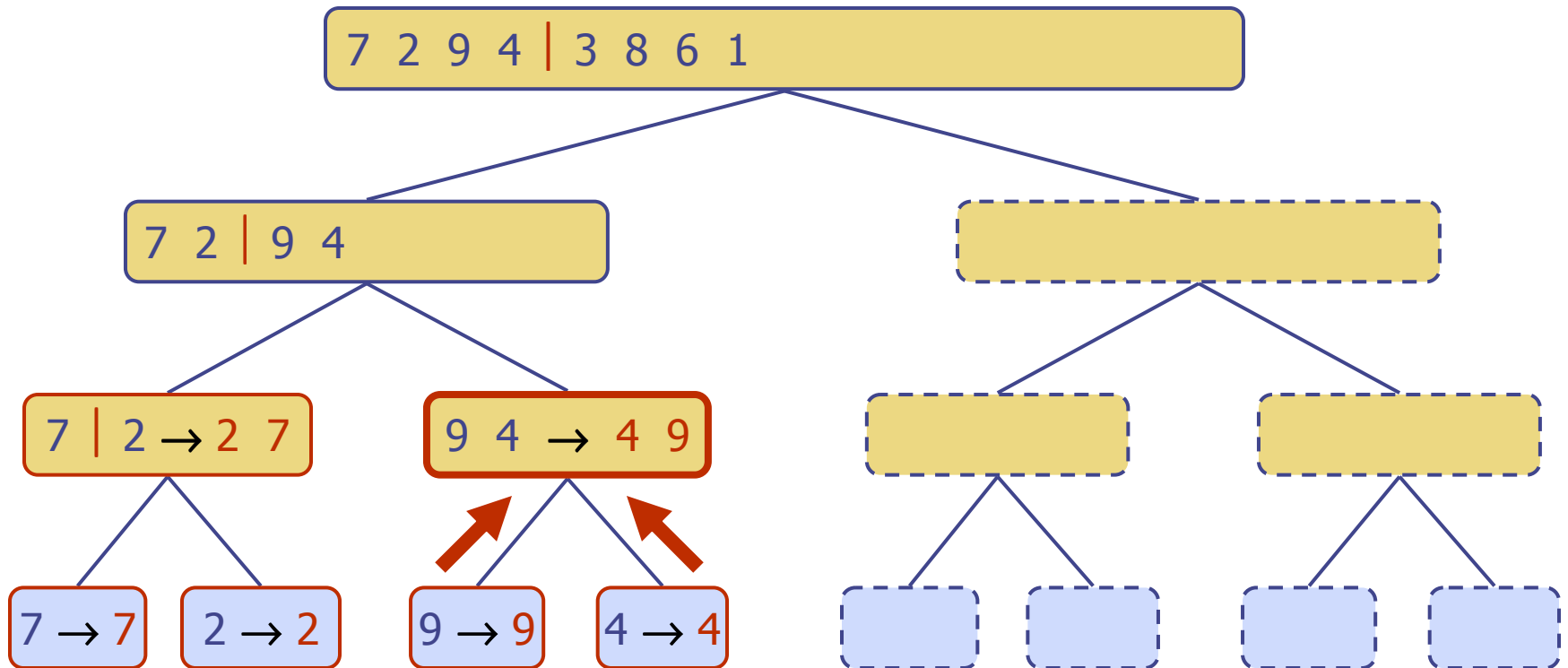
Execution Example (cont.)

◆ Merge



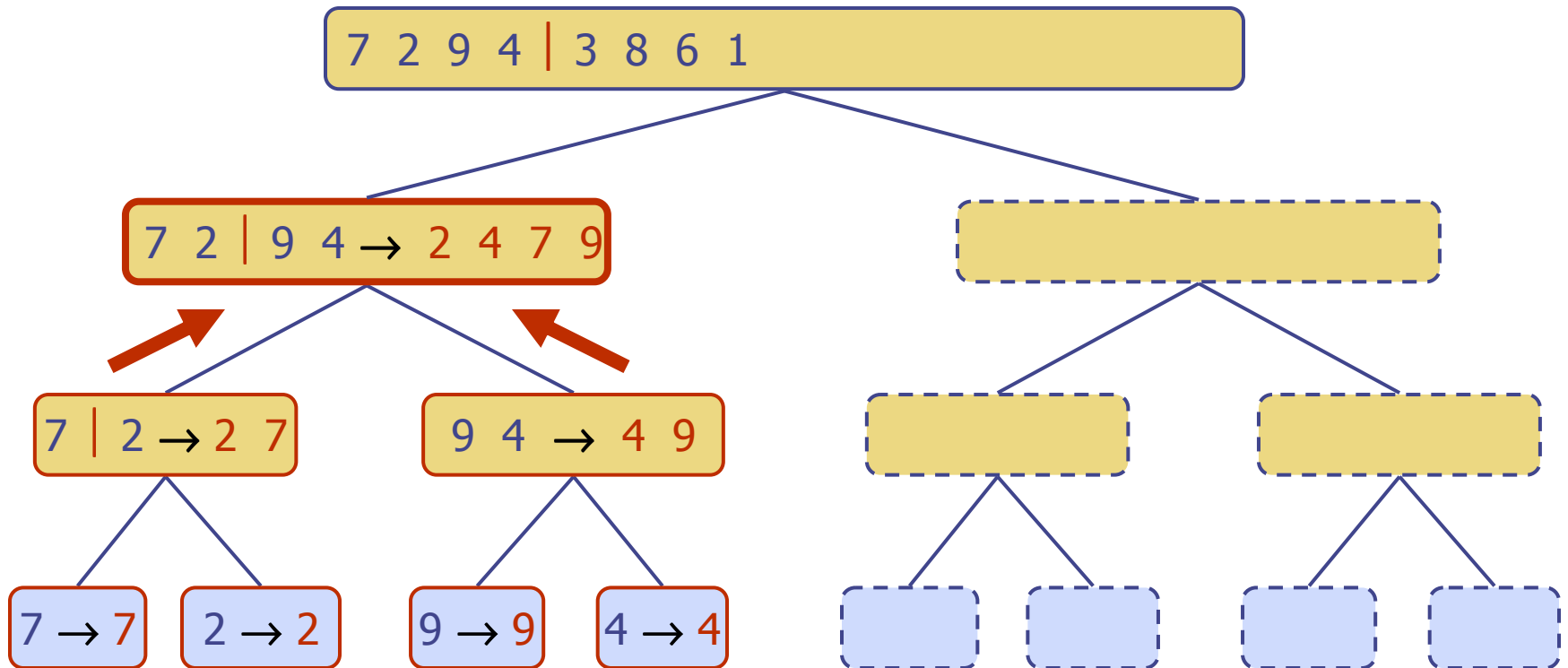
Execution Example (cont.)

◆ Recursive call, ..., base case, merge



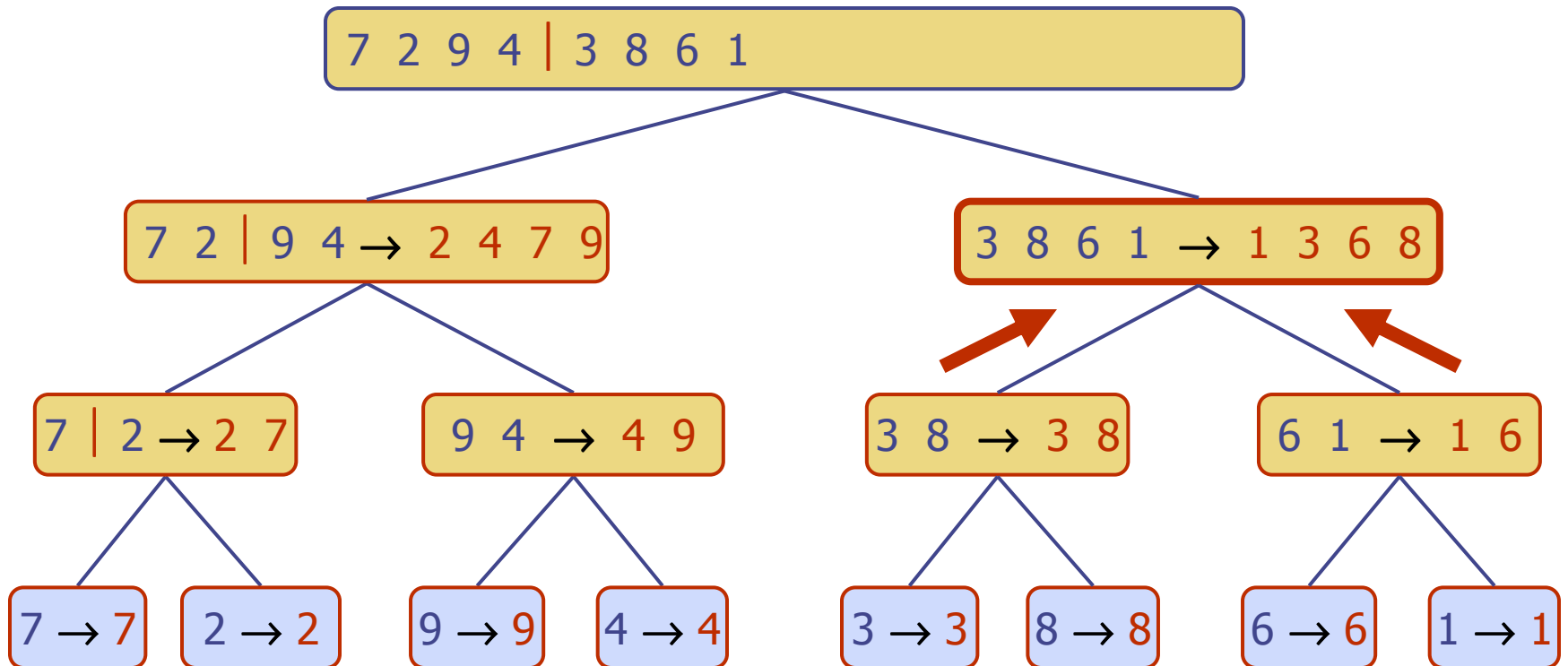
Execution Example (cont.)

◆ Merge



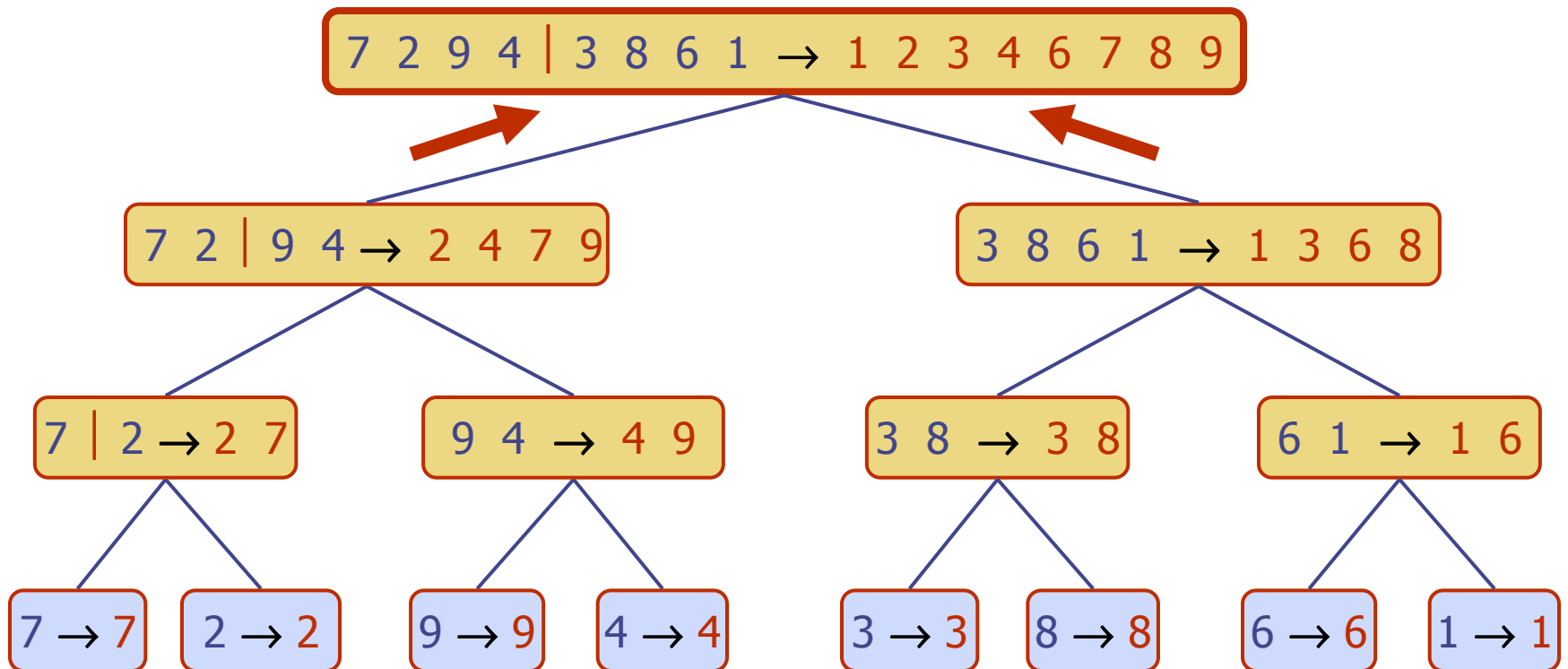
Execution Example (cont.)

◆ 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 of 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 $O(n \log n)$

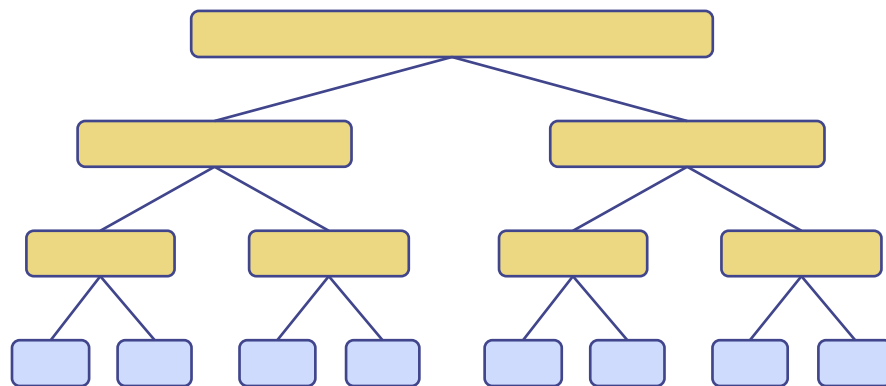
depth	#seqs	size
-------	-------	------

0	1	n
---	---	-----

1	2	$n/2$
---	---	-------

i	2^i	$n/2^i$
-----	-------	---------

...
-----	-----	-----



Sorting

Analysis of Merge-Sort using Recurrence Relations

- Merge(S1,S2) takes time $O(n)$, where n is the size of S1 and S2
- $T(n) = 2T(n/2) + O(n)$
- Solving, get $T(n)=O(n\log n)$

Algorithm *mergeSort*(S, C)

Input sequence S with n elements, comparator C

Output sequence S sorted according to C

if $S.size() > 1$

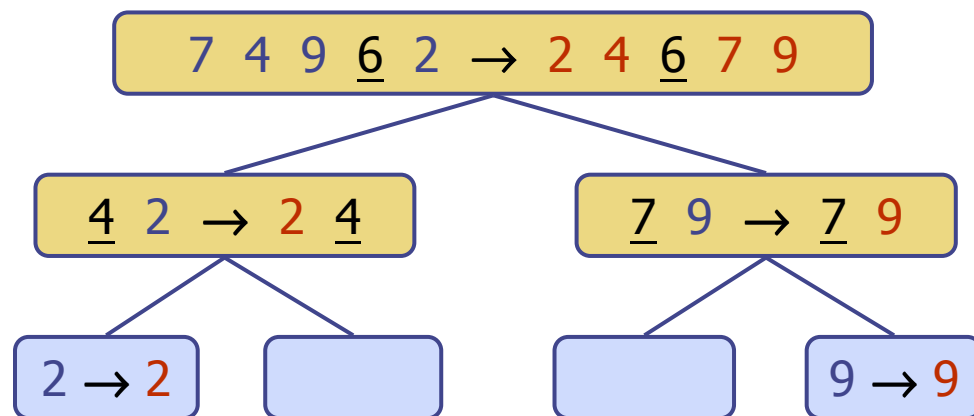
$(S_1, S_2) \leftarrow partition(S, n/2)$

mergeSort(S_1, C)

mergeSort(S_2, C)

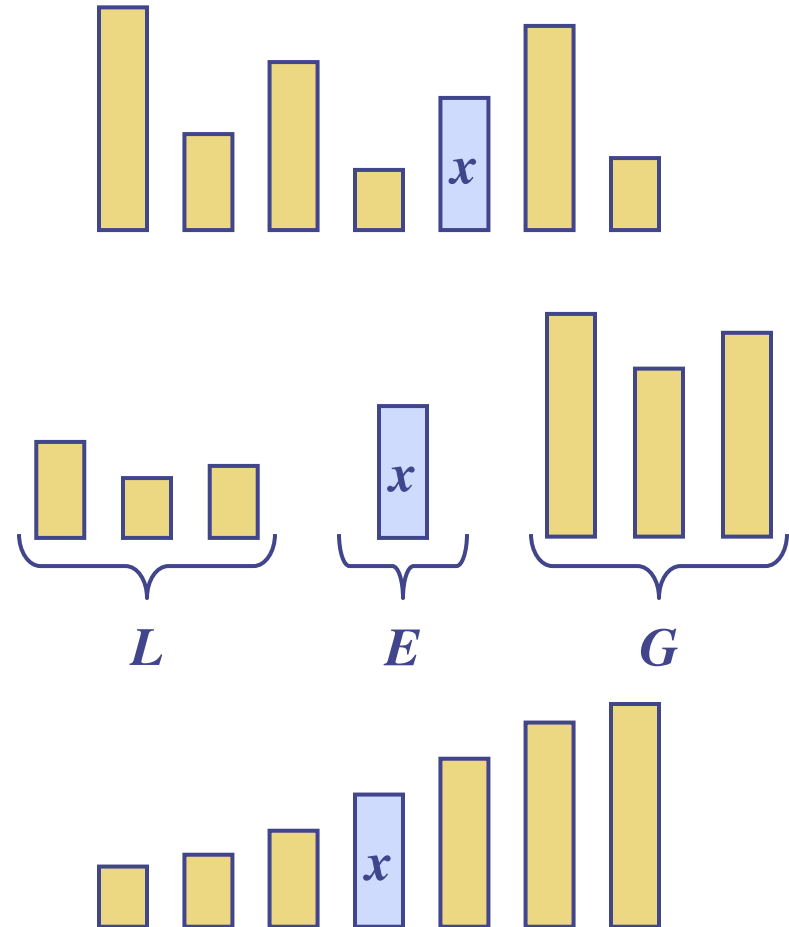
$S \leftarrow merge(S_1, S_2)$

Quick-Sort

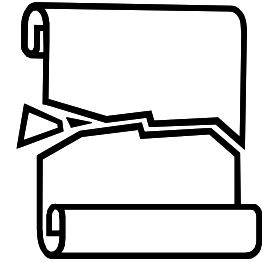


Quick-Sort

- ◆ **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 $y < x$

$L.insertLast(y)$

else if $y = x$

$E.insertLast(y)$

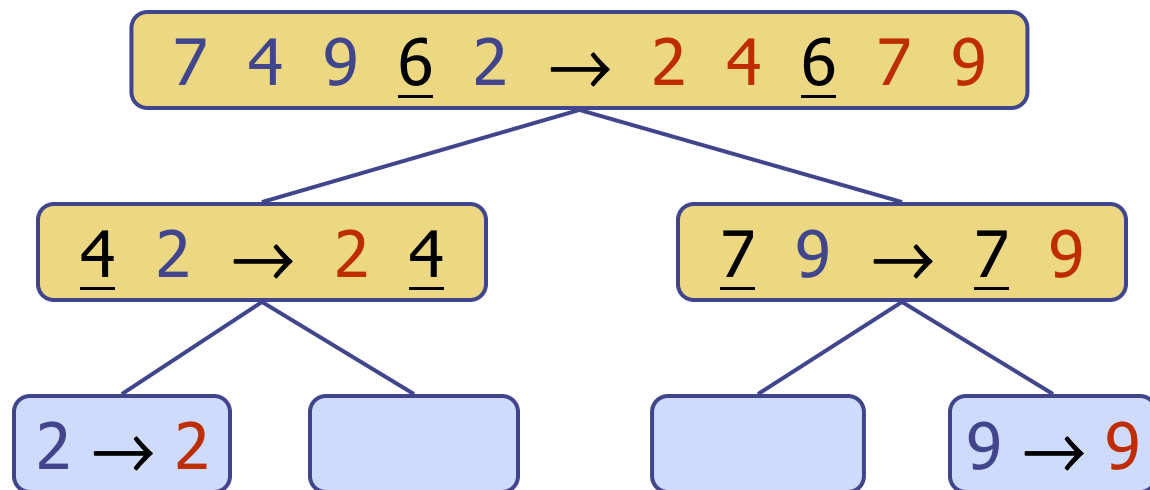
else { $y > x$ }

$G.insertLast(y)$

return L, E, G

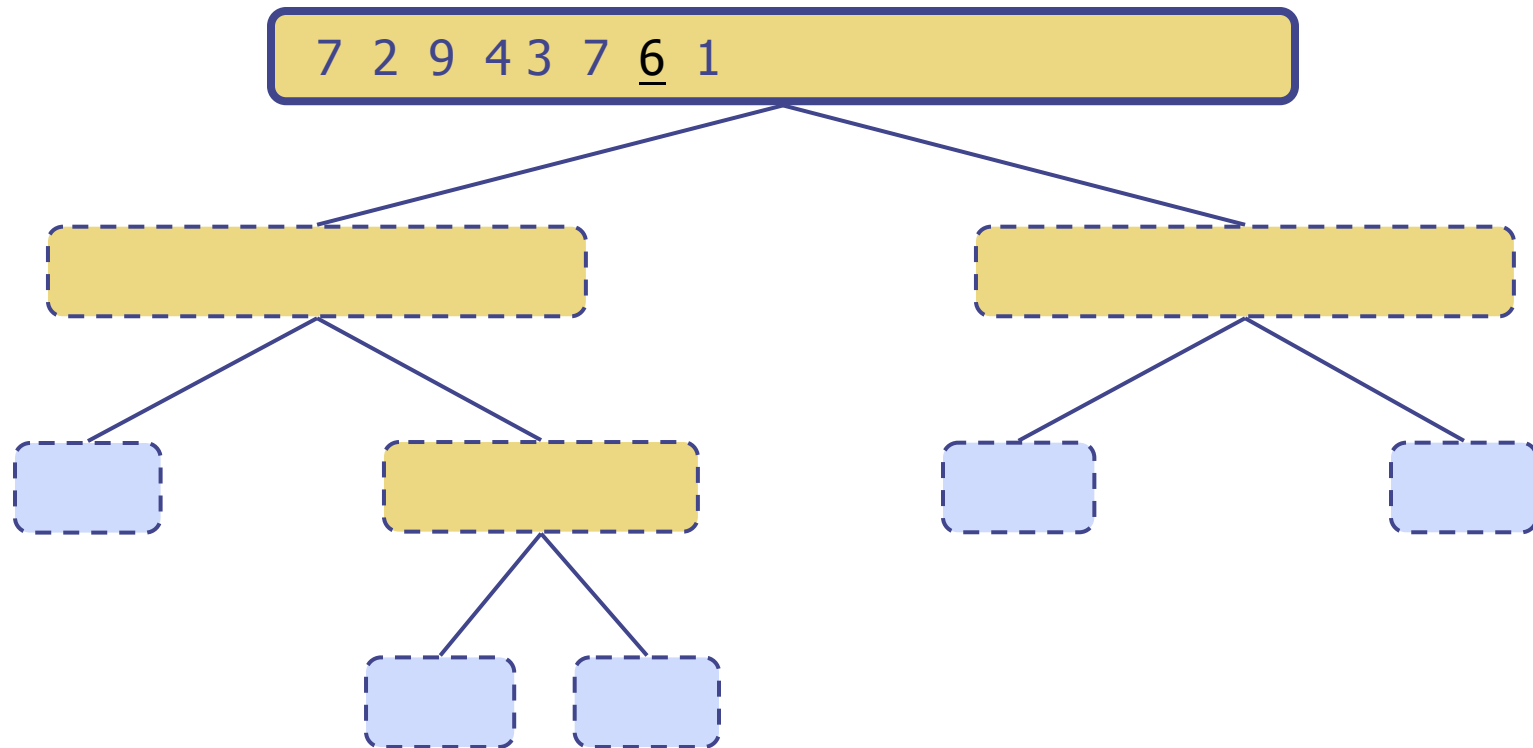
Quick-Sort Tree

- ◆ 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



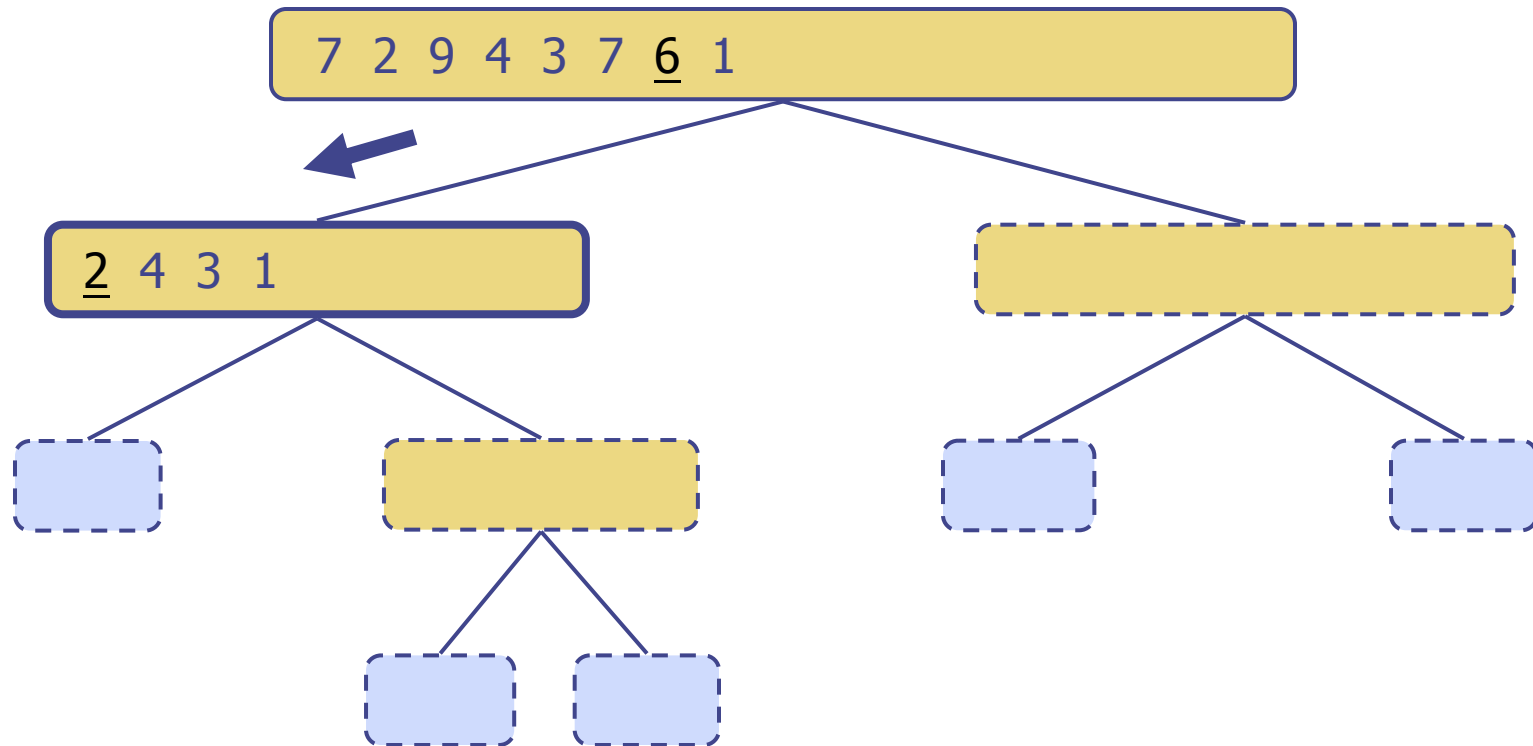
Execution Example

◆ Pivot selection



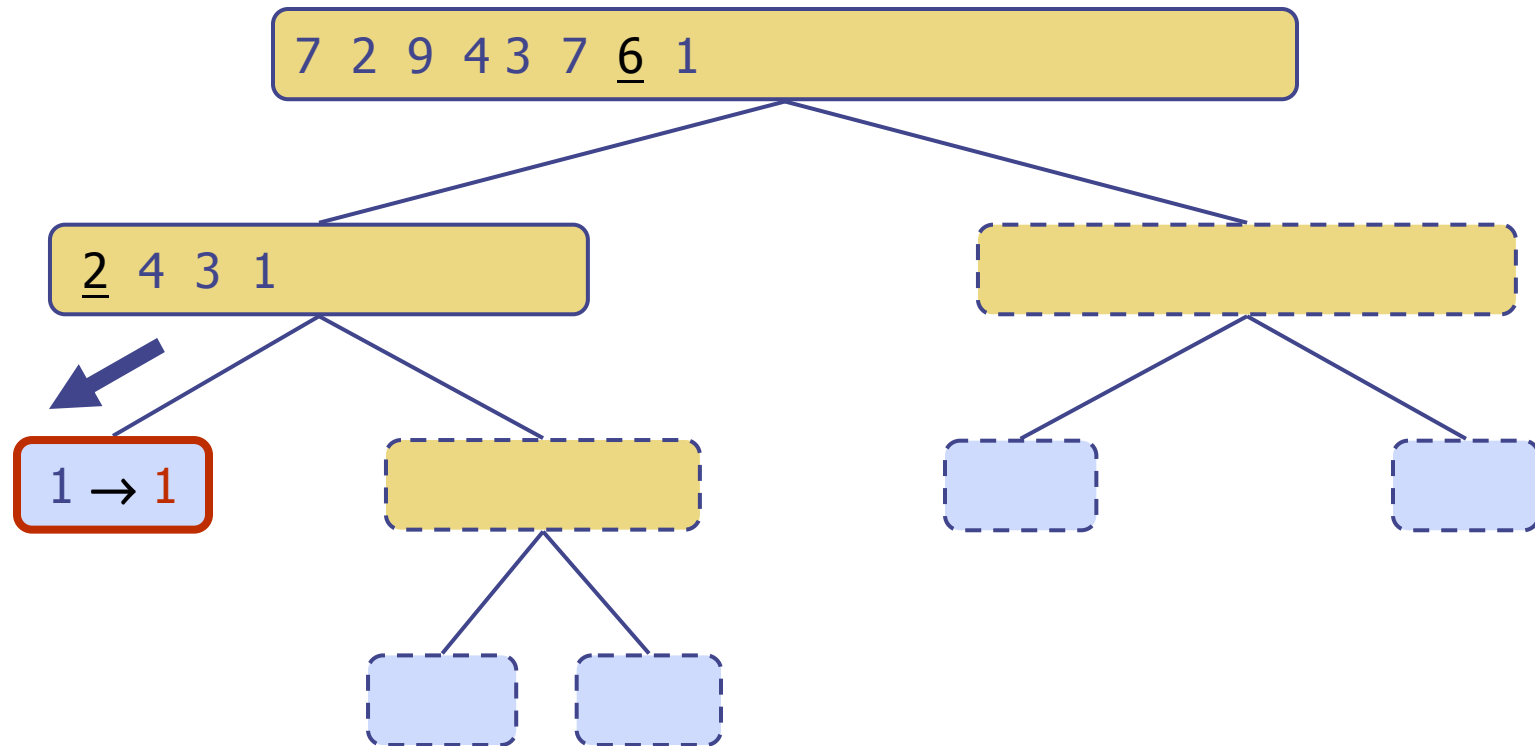
Execution Example (cont.)

◆ Partition, recursive call, pivot selection



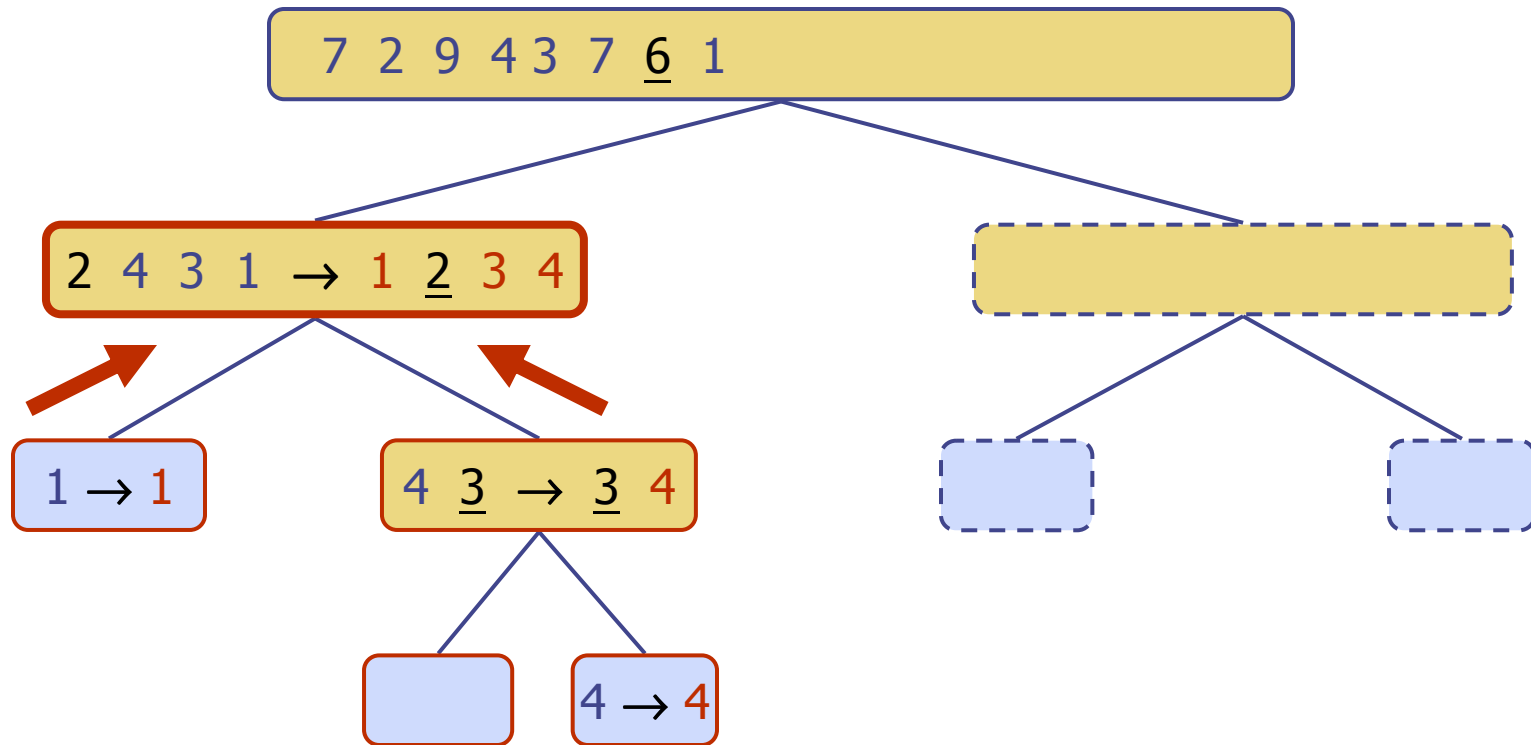
Execution Example (cont.)

◆ Partition, recursive call, base case



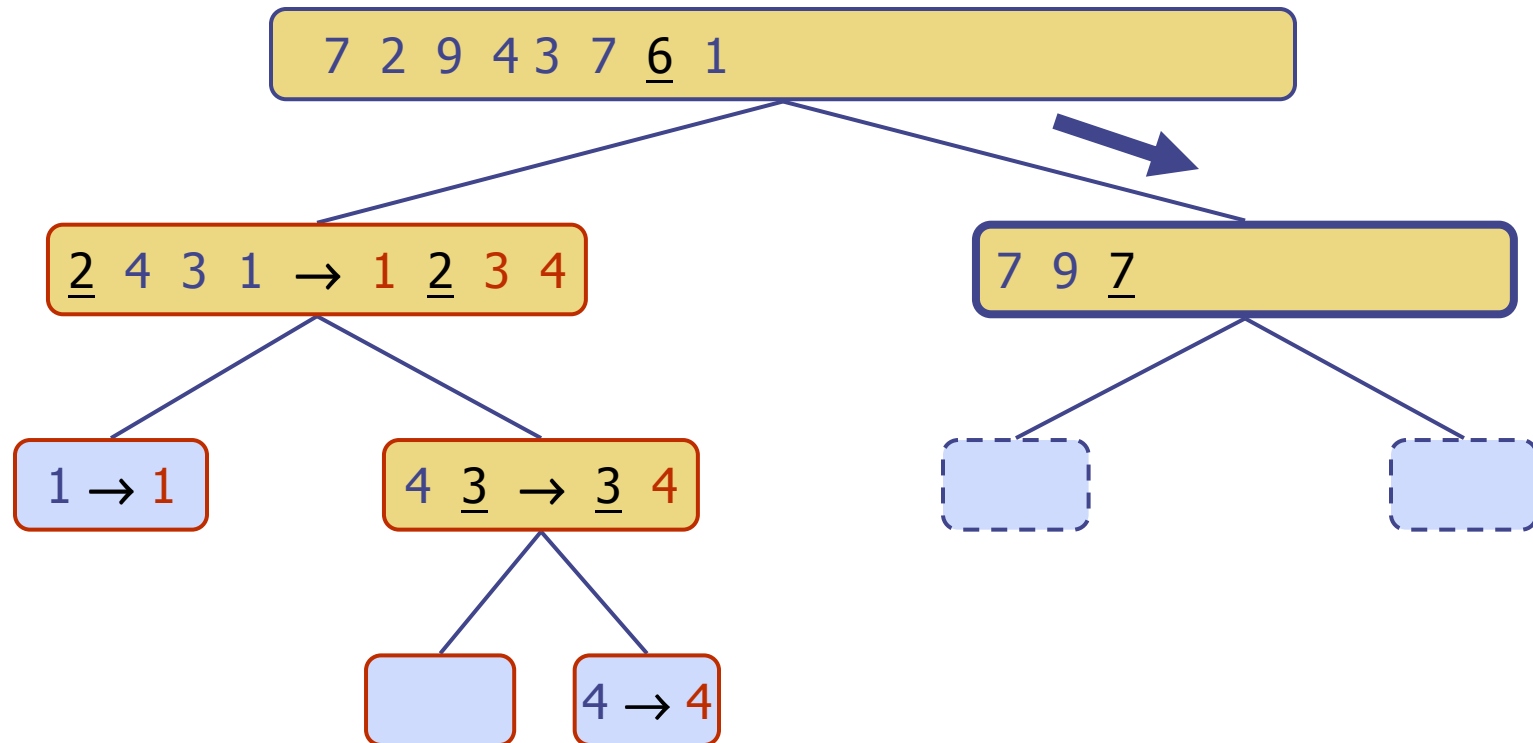
Execution Example (cont.)

◆ Recursive call, ..., base case, join



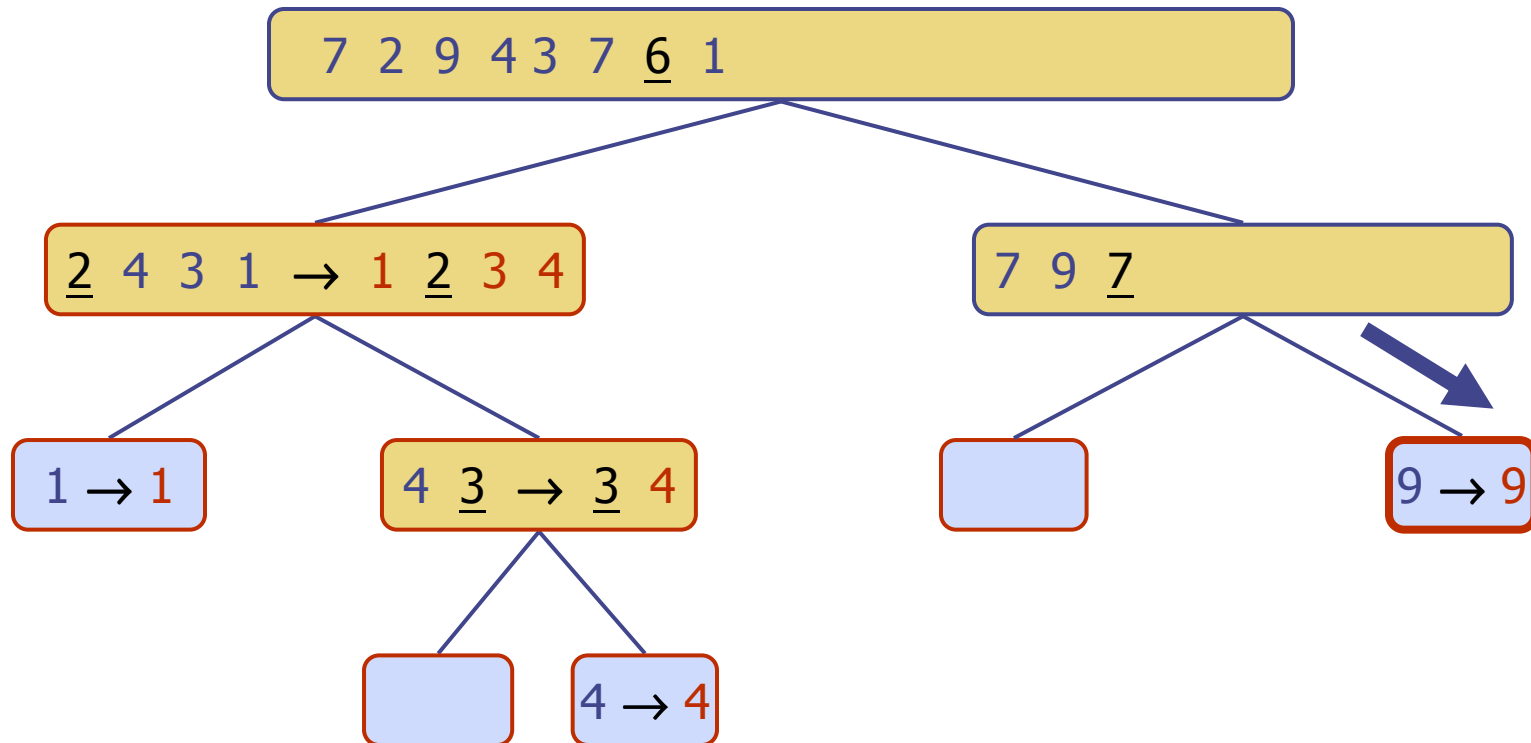
Execution Example (cont.)

◆ Recursive call, pivot selection



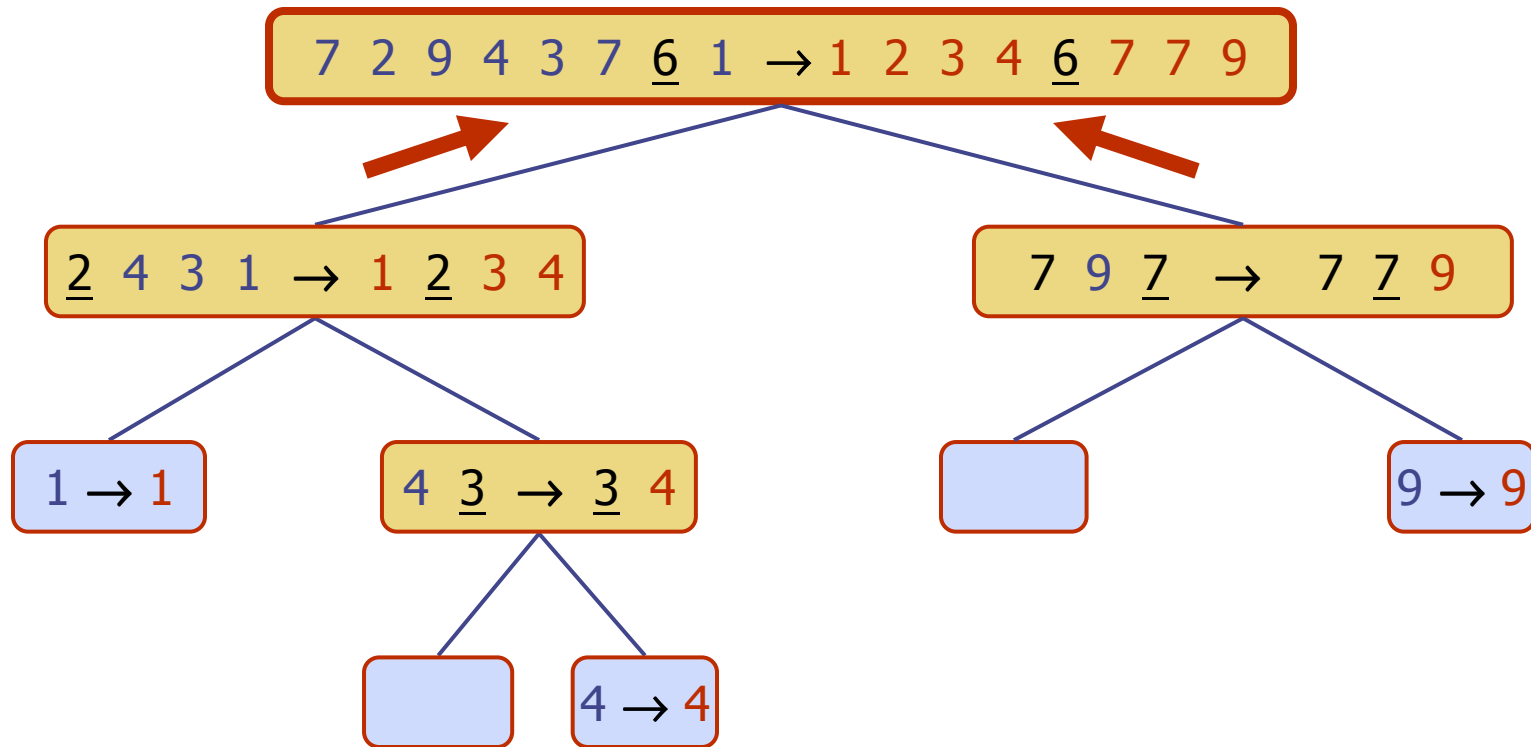
Execution Example (cont.)

◆ Partition, ..., recursive call, base case



Execution Example (cont.)

◆ Join, join



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

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

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

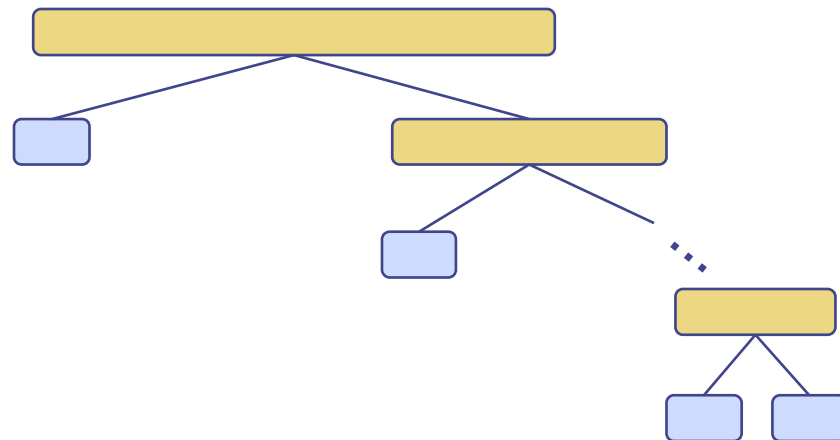
depth time

0 n

1 $n - 1$

...

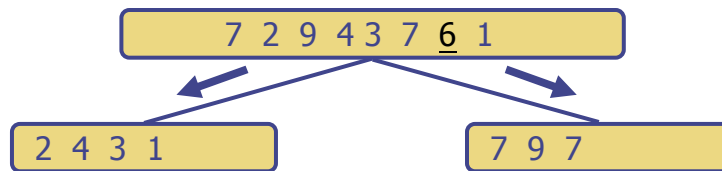
$n - 1$ 1



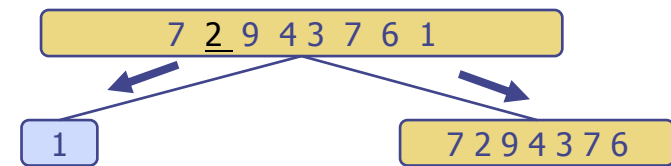
Sorting

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



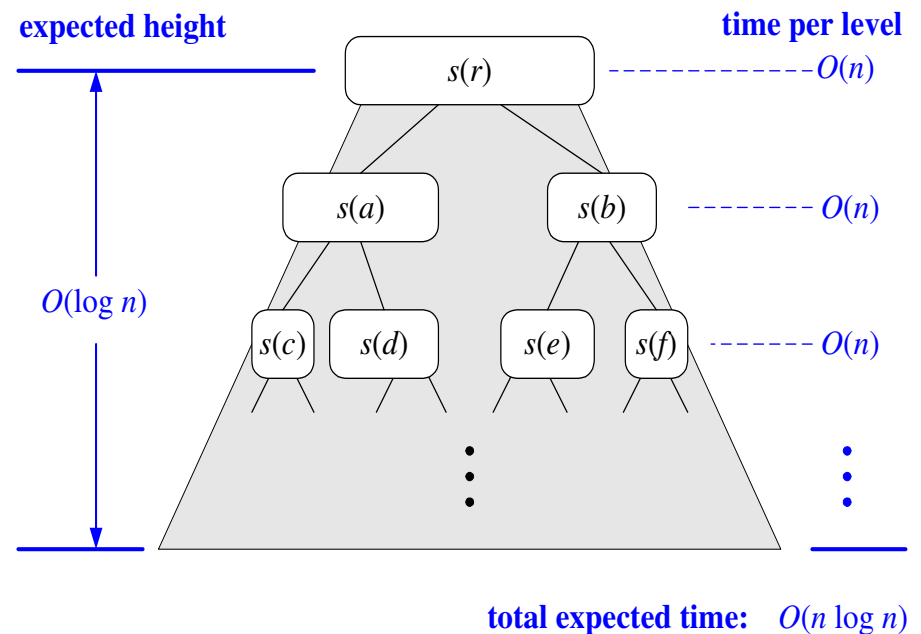
Bad call

- ◆ 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 of 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

$x \leftarrow S.\text{elemAtRank}(i)$

$(h, k) \leftarrow \text{inPlacePartition}(x)$

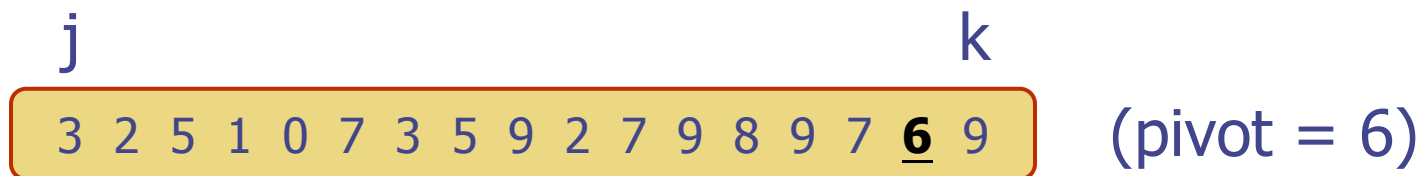
inPlaceQuickSort($S, l, h - 1$)

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

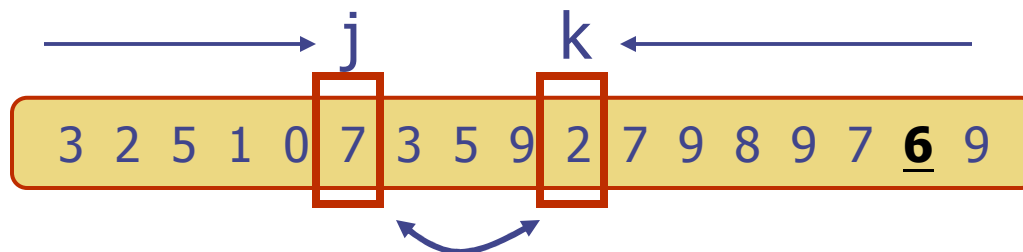
In-Place Partitioning



- ◆ Perform the partition using two indices to split S into L and EYG (a similar method can split EYG 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



Sorting

Summary of Sorting Algorithms

Algorithm	Time	Notes
bubble-sort	$O(n^2)$	<ul style="list-style-type: none">◆ in-place◆ slow (good for small inputs)
insertion-sort	$O(n^2)$	<ul style="list-style-type: none">◆ in-place◆ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	<ul style="list-style-type: none">◆ in-place, randomized◆ fastest (good for large inputs)
heap-sort	$O(n \log n)$	<ul style="list-style-type: none">◆ in-place◆ fast (good for large inputs)
merge-sort	$O(n \log n)$	<ul style="list-style-type: none">◆ sequential data access◆ fast (good for huge inputs)