FALL 2018 CISC 311

OBJECT & STRUCTURE & ALGORITHM I

– Chapter 6: Sorting Algorithms

Outline

- Bubble sort
- Selection sort
- Insertion sort
- Divide-and-conquer
- Quick sort
- Merge sort

- Heap sort
- Bucket sort
- Radix sort
- Counting sort

Heap

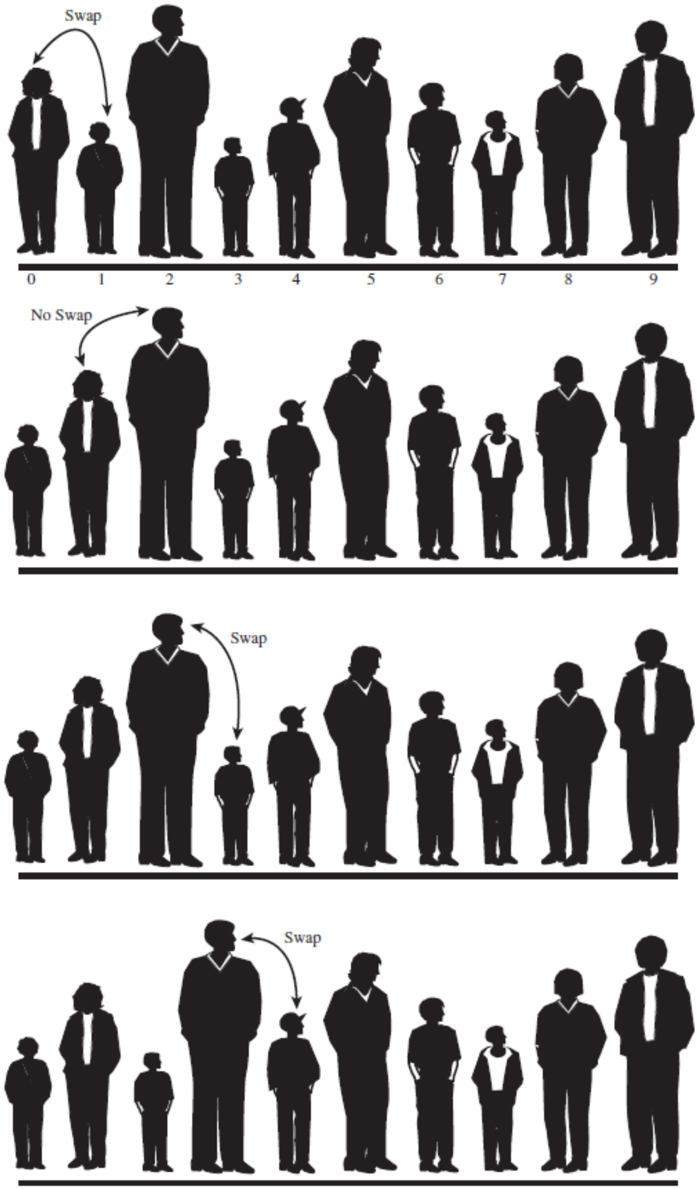
Hash table

Basic Sorting Algorithms

- Basic sorting algorithms:
  - Bubble sort
  - Selection sort
  - Insertion sort
- The three algorithms all involve two steps, executed over and over until the data is sorted:
  - Compare two items
  - Swap two items, or copy one item

Bubble Sort

- Idea: starting from the beginning of the sequence, compare every adjacent pair, swap their position if they are not in the right order. After each iteration, one less element (the last one) is needed to be compared until there are no more elements left to be compared


Swap
0
1
2
3
4
5
6
7
8
9
No Swap
Swap
Swap

Original array:		5	3	1	9	8
Pass *i* = 0	*j*	**0**	**1**	**2**	**3**	**4**
	0	3	5	1	9	8
	1	3	1	5	9	8
	2	3	1	5	9	8
	3	3	1	5	8	9
Pass *i* = 1	*j*	**0**	**1**	**2**	**3**	**4**
	0	1	3	5	8	9
	1	1	3	5	8	9
	2	1	3	5	8	9
Pass *i* = 2	*j*	**0**	**1**	**2**	**3**	**4**
	0	1	3	5	8	9
	1	1	3	5	8	9
Pass *i* = 3	*j*	**0**	**1**	**2**	**3**	**4**
	0	1	3	5	8	9

Bubble Sort (cont’d.)

```
Algorithm bubbleSort($A$)
    Input $A$ array $A$
    Output $A$ sorted array
    for $i \leftarrow 0$ to $A.length - 2$ do
        for $j \leftarrow 0$ to $A.length - i - 2$ do
            if $A[j] > A[j+1]$ then
                $A[j] \leftrightarrow A[j+1]$
    return $A$
```

BubbleSort.java

Bubble Sort Efficiency (cont'd.)

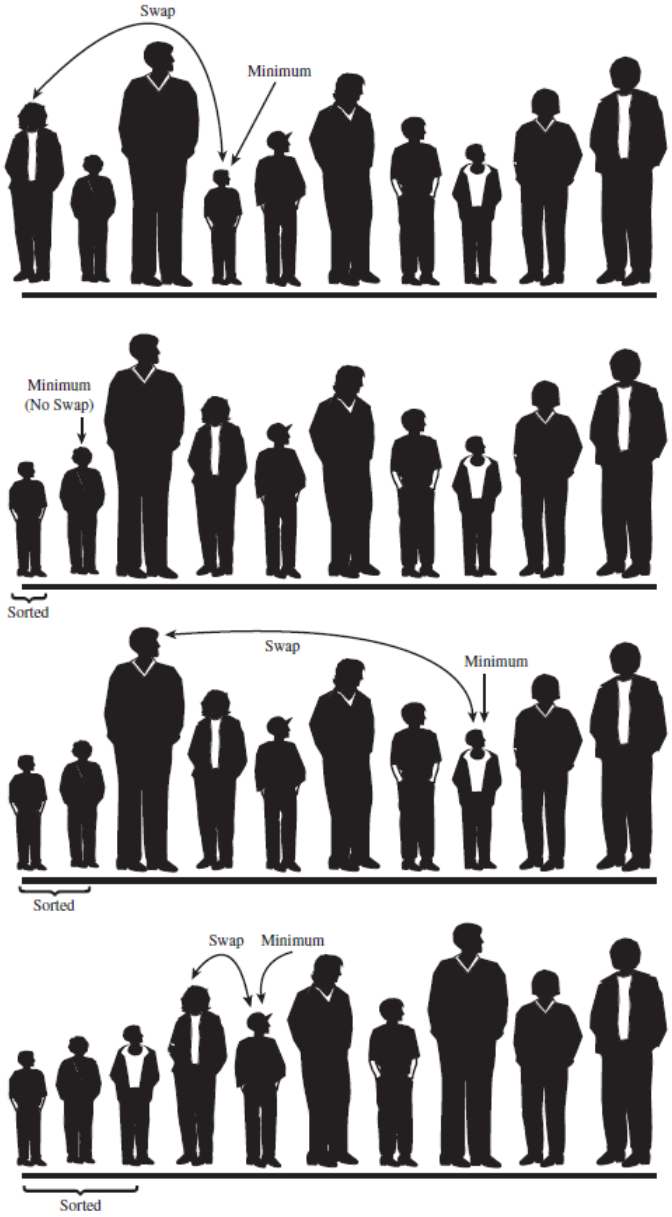
- The bubble sort is notoriously slow, but it's conceptually the simplest of the sorting algorithms and for that reason is a good beginning for our exploration of sorting techniques

	Worst-case	Best-case	Average	Space
Bubble sort	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n)$

Selection Sort

```
Algorithm selectionSort(A)
    Input A array A
    Output A sorted array
    for $i \leftarrow 0$ to $A.length - 2$ do
        $min \leftarrow i$
        for $j \leftarrow i + 1$ to $A.length - 1$ do
            if $A[j] < A[min]$ then
                $min \leftarrow j$
        $A[i] \leftrightarrow A[min]$
    return A
```

SelectionSort.java


Swap
Minimum
Minimum
(No Swap)
Sorted
Swap
Minimum
Sorted
Swap
Minimum
Sorted

minimum

Swap

Original array:		5	3	1	9	8
Pass *i* = 0	*j*	**0**	**1**	**2**	**3**	**4**
		1	3	5	9	8
Pass *i* = 1	*j*	**0**	**1**	**2**	**3**	**4**
		1	**3**	5	9	8
Pass *i* = 2	*j*	**0**	**1**	**2**	**3**	**4**
		1	**3**	**5**	9	8
Pass *i* = 3	*j*	**0**	**1**	**2**	**3**	**4**
		1	3	5	**8**	9

Compare $O(n^2)$ times   Swap $O(n)$ times

Selection Sort (cont'd.)

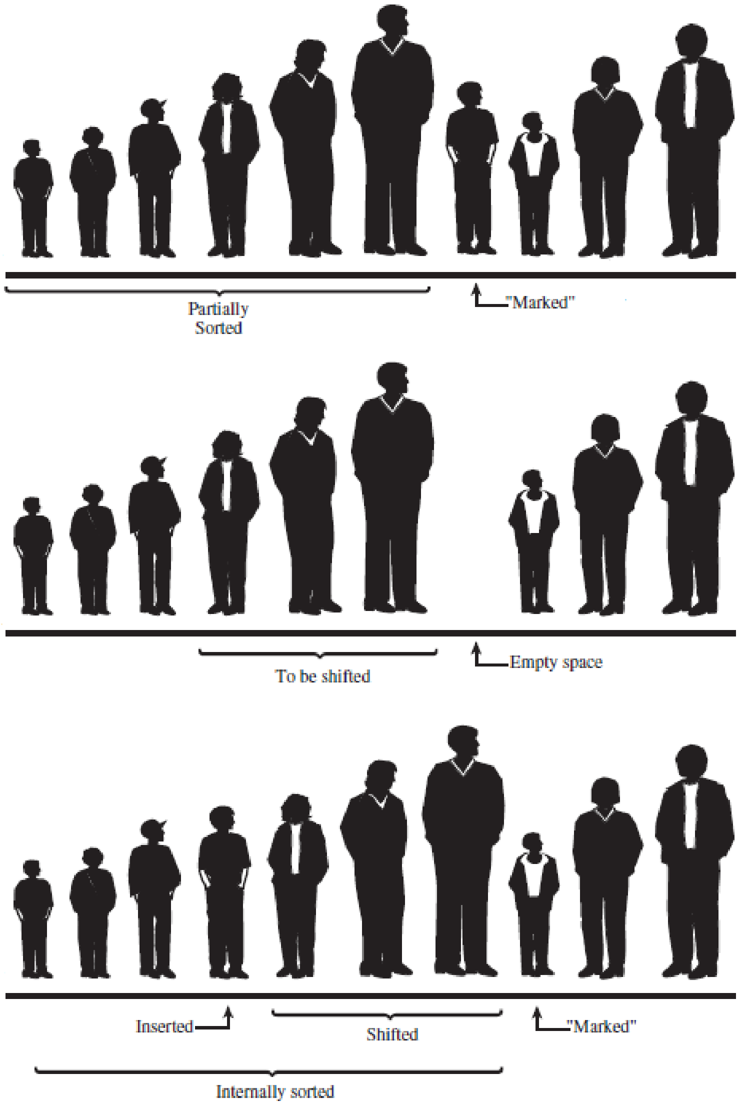
- The selection sort improves on the bubble sort by reducing the number of swaps necessary from $O(n^2)$ to $\mathrm{O}(n)$
- Unfortunately, the number of comparisons remains $O(n^2)$
- However, the selection sort can still offer a significant improvement for large records that must be physically moved around in memory, causing the swap time to be much more important than the comparison time

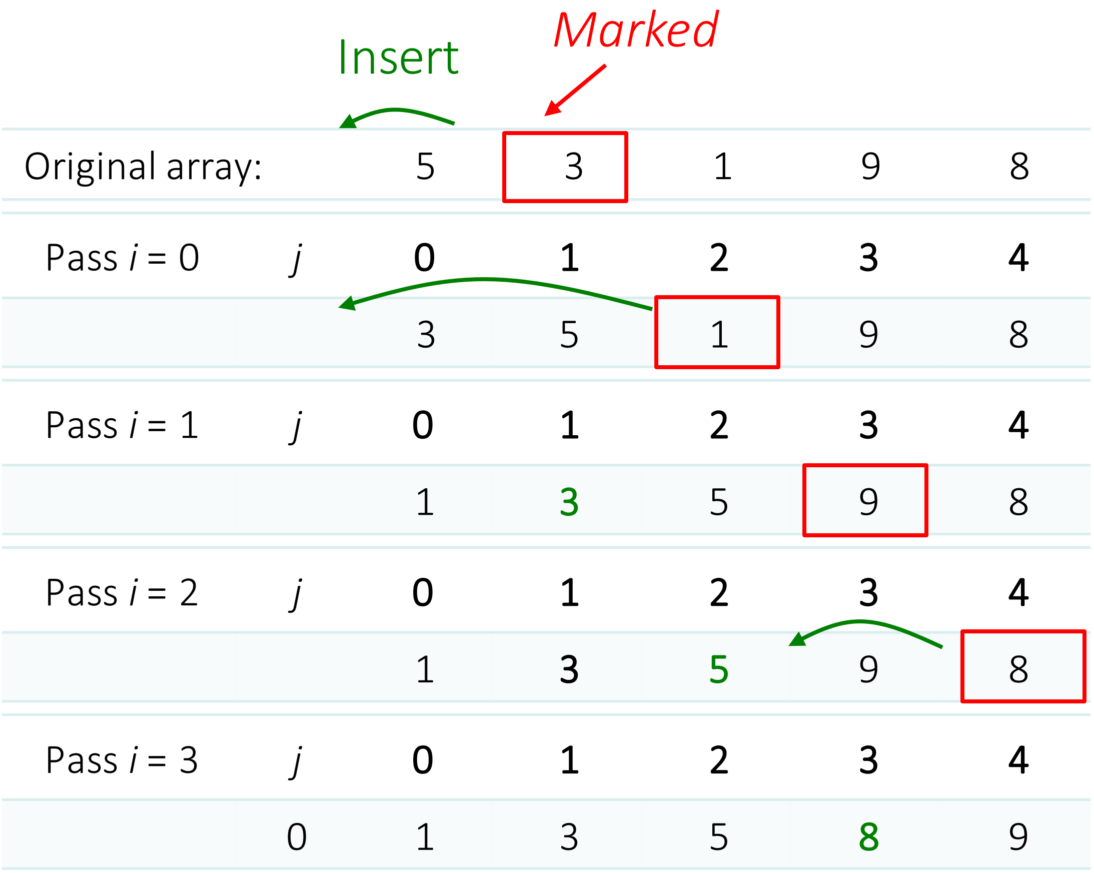
	Worst-case	Best-case	Average	Space
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n)$

Insertion Sort

```
Algorithm insertionSort($A$)
    Input $A$ array $A$
    Output $A$ sorted array
    for $i \leftarrow 1$ to $A.length - 1$ do
        $marked \leftarrow A[i]$
        $j \leftarrow i$
        while $j >= 1$ and $A[j\text{-}1] > marked$ do
            $A[j] \leftarrow A[j - 1]$
            $j \leftarrow j - 1$
        $A[j] \leftarrow marked$
    return $A$
```

InsertionSort.java


Partially Sorted
"Marked"
To be shifted
Empty space
Inserted
Shifted
"Marked"
Internally sorted



Original array:		5	3	1	9	8
Pass *i* = 0	*j*	**0**	**1**	**2**	**3**	**4**
		3	5	1	9	8
Pass *i* = 1	*j*	**0**	**1**	**2**	**3**	**4**
		1	**3**	5	9	8
Pass *i* = 2	*j*	**0**	**1**	**2**	**3**	**4**
		1	**3**	**5**	9	8
Pass *i* = 3	*j*	**0**	**1**	**2**	**3**	**4**
	0	1	3	5	**8**	9

Insertion Sort (cont'd.)

- For data that is already sorted or almost sorted, the insertion sort does much better
- When data is in order, the condition in the while loop is never true, so it becomes a simple statement in the outer loop, which executes n-1 times. In this case the algorithm runs in $O(n)$ time. If the data is almost sorted, insertion sort runs in almost $O(n)$ time, which makes it a simple and efficient way to order a file that is only slightly out of order

	Worst-case	Best-case	Average	Space
Insertion sort	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n)$

Comparison

- There's probably no point in using the bubble sort. The bubble sort is so simple that you can write it from memory. Even so, it's practical only if the amount of data is small
- The selection sort minimizes the number of swaps, but the number of comparisons is still high. This sort might be useful when the amount of data is small and swapping data items is very time-consuming compared with comparing them
- The insertion sort is the most versatile of the three and is the best bet in most situations, assuming the amount of data is small or the data is almost sorted. For larger amounts of data, quicksort is generally considered the fastest approach

Invariants

- In many algorithms there are conditions that remain unchanged as the algorithm proceeds. These conditions are called *invariants*
- Recognizing invariants can be useful in understanding the algorithm. In certain situations they may also be helpful in debugging; you can repeatedly check that the invariant is true, and signal an error if it isn't
- Example: In bubble sort, the invariant is that the data items to the right of `i` are sorted. This remains true throughout the running of the algorithm

In-place Sorting

- An in-place sort is a sorting algorithm which transforms input using no auxiliary data structure
  - However a small amount of extra storage space is allowed for auxiliary variables
  - In-place algorithm updates input sequence only through replacement or swapping of elements

Stable Sorting

- A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted
- Formally stability may be defined as,
  - Let A be an array, and let < be a strict weak ordering on the elements of A
  - A sorting algorithm is stable if $i < j$ and $A[i] \equiv A[j]$ implies $\pi(i) < \pi(j)$, where π is the sorting permutation (sorting moves $A[i]$ to position $\pi(i)$)
- Informally, stability means that equivalent elements retain their relative positions, after sorting

Summary: Basic Sorting Algorithms

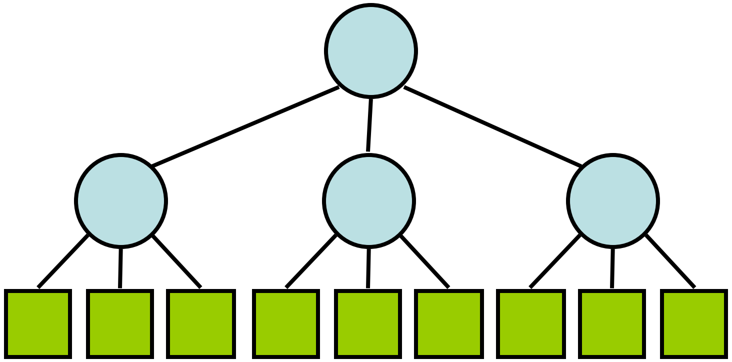
- All the three algorithms execute in $O(n^2)$ time. Nevertheless, some can be substantially faster than others
- An invariant is a condition that remains unchanged while an algorithm runs
- The bubble sort is the least efficient, but the simplest, sort
- The insertion sort is the most commonly used of the $O(n^2)$ sorts
- None of the three sorts require more than a single temporary variable, in addition to the original array

Algorithm Design Paradigm

- In chapter 2, we briefly introduced algorithmic design patterns
- Algorithmic patterns:
  - Recursion
  - Amortization
  - Divide-and-conquer
  - Prune-and-search
  - Brute force
  - Dynamic programming
  - The greedy method

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, …
  - Conquer: solve the sub-problems recursively
  - Combine: combine the solutions for S_1, S_2, …, into a solution for S
- The base case for the recursion are sub-problems of constant size
- Analysis can be done using recurrence equations

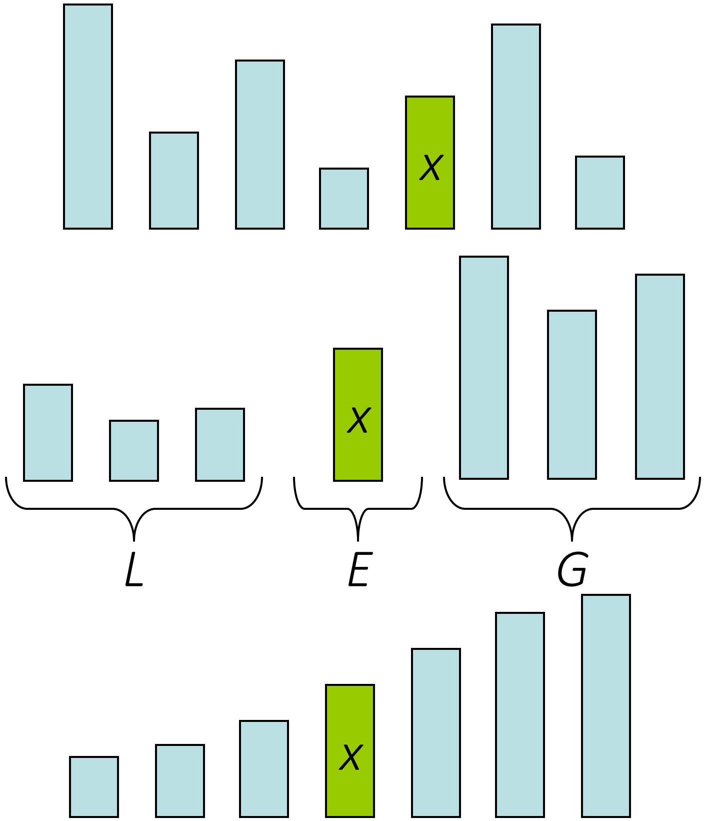


Divide-and-conquer-based Sorting Algorithms

- Quick sort
- Merge 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

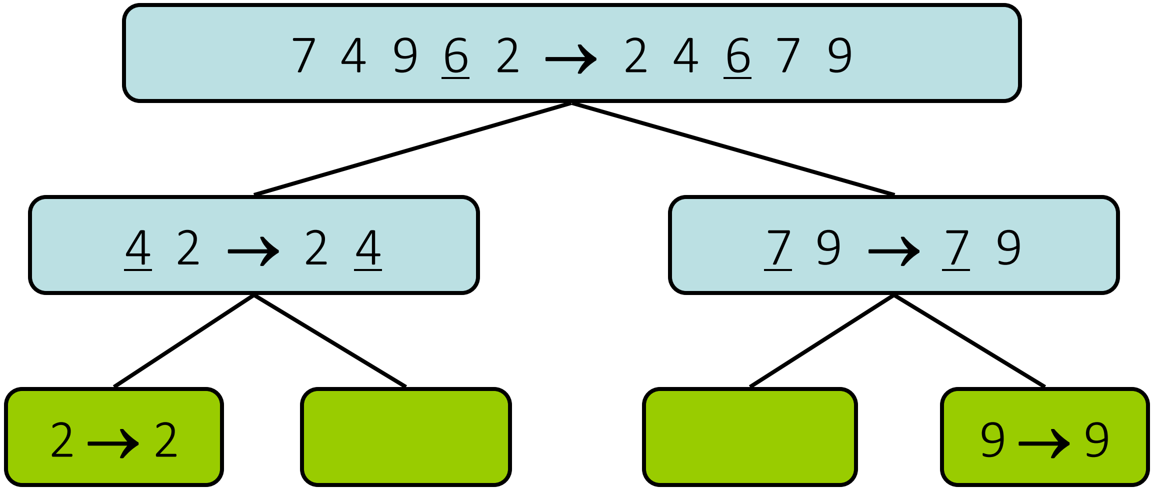
```
Algorithm partition(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the elements of S
            less than, equal to, or greater than the pivot, resp.
    L, E, G ← empty sequences
    x ← S.remove(p)
    while ¬S.isEmpty() do
        y ← S.remove(S.first())
        if y < x then
            L.addLast(y)
        else if y = x
            E.addLast(y)
        else if y > x
            G.addLast(y)
    return L, E, G
```

Additional space
Not in-place

OutPlaceQuickSort.java

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



7 4 9 $\underline{6}$ 2 → 2 4 $\underline{6}$ 7 9

$\underline{4}$ 2 → 2 $\underline{4}$

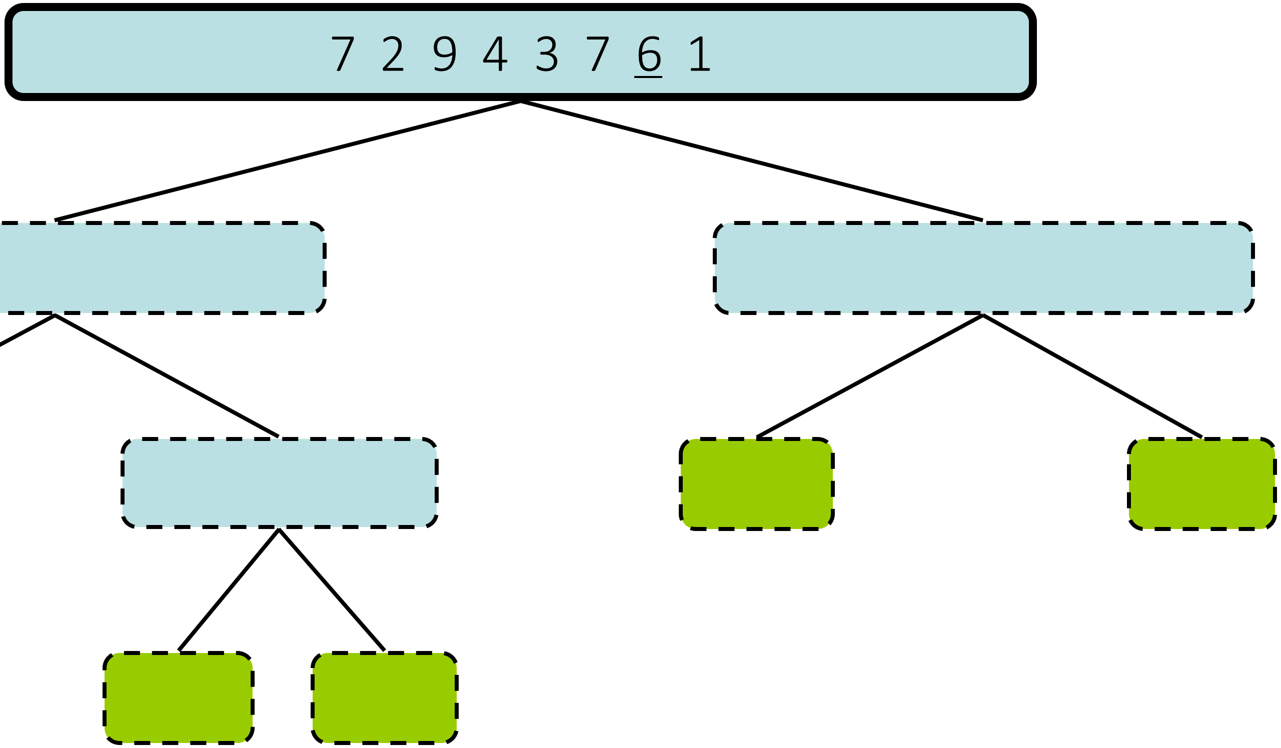
$\underline{7}$ 9 → $\underline{7}$ 9

2 → 2

9 → 9

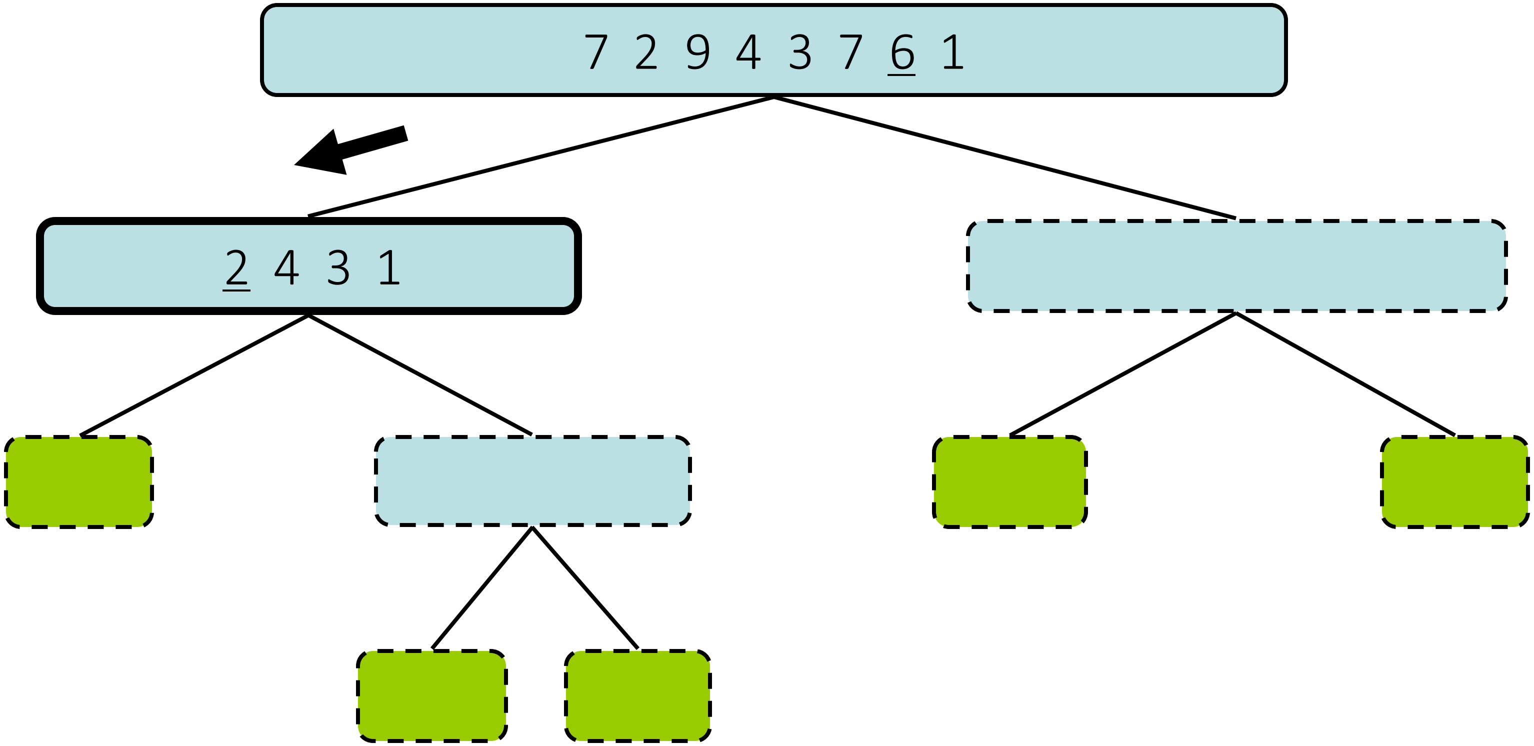
Execution Example

- Pivot selection



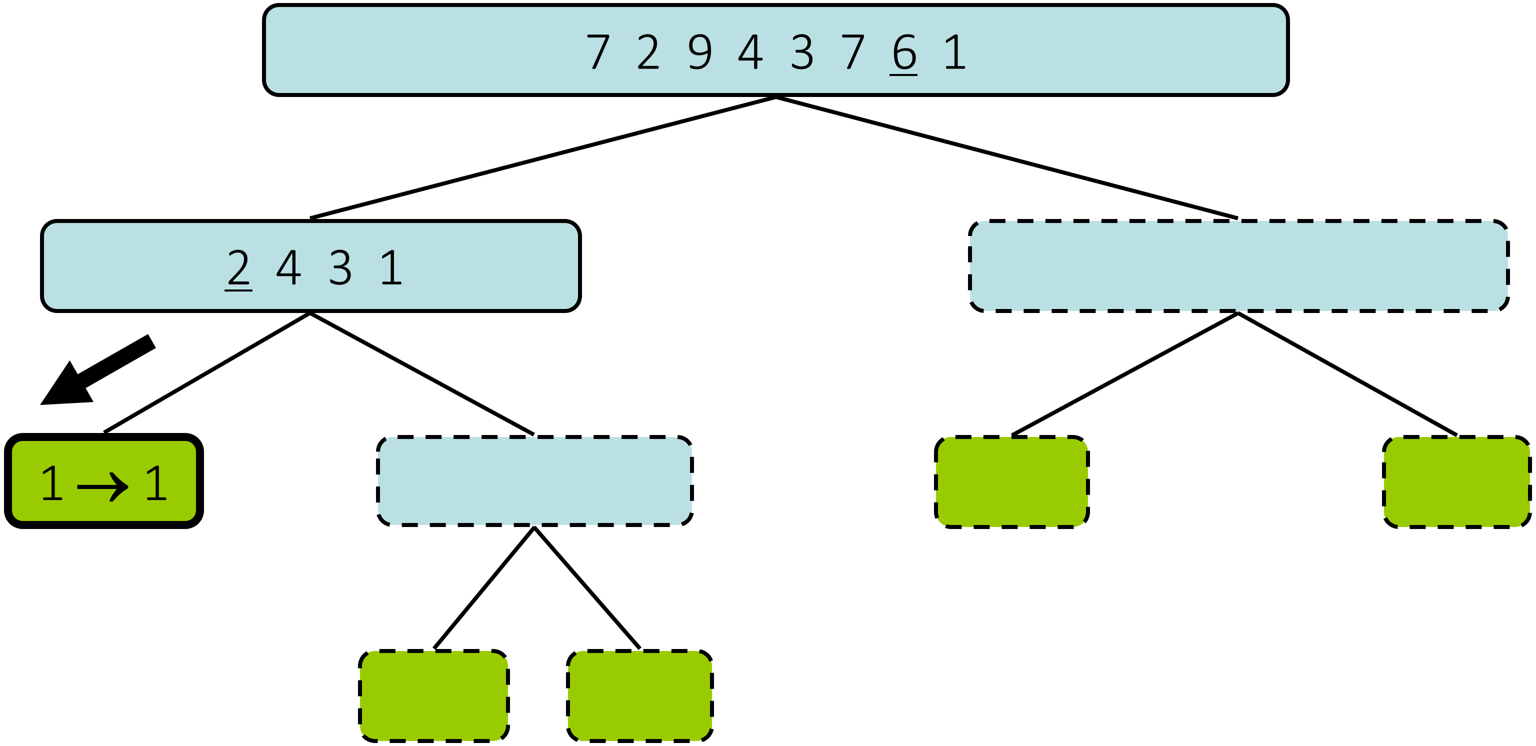
Execution Example (cont'd.)

- Partition, recursive call, pivot selection



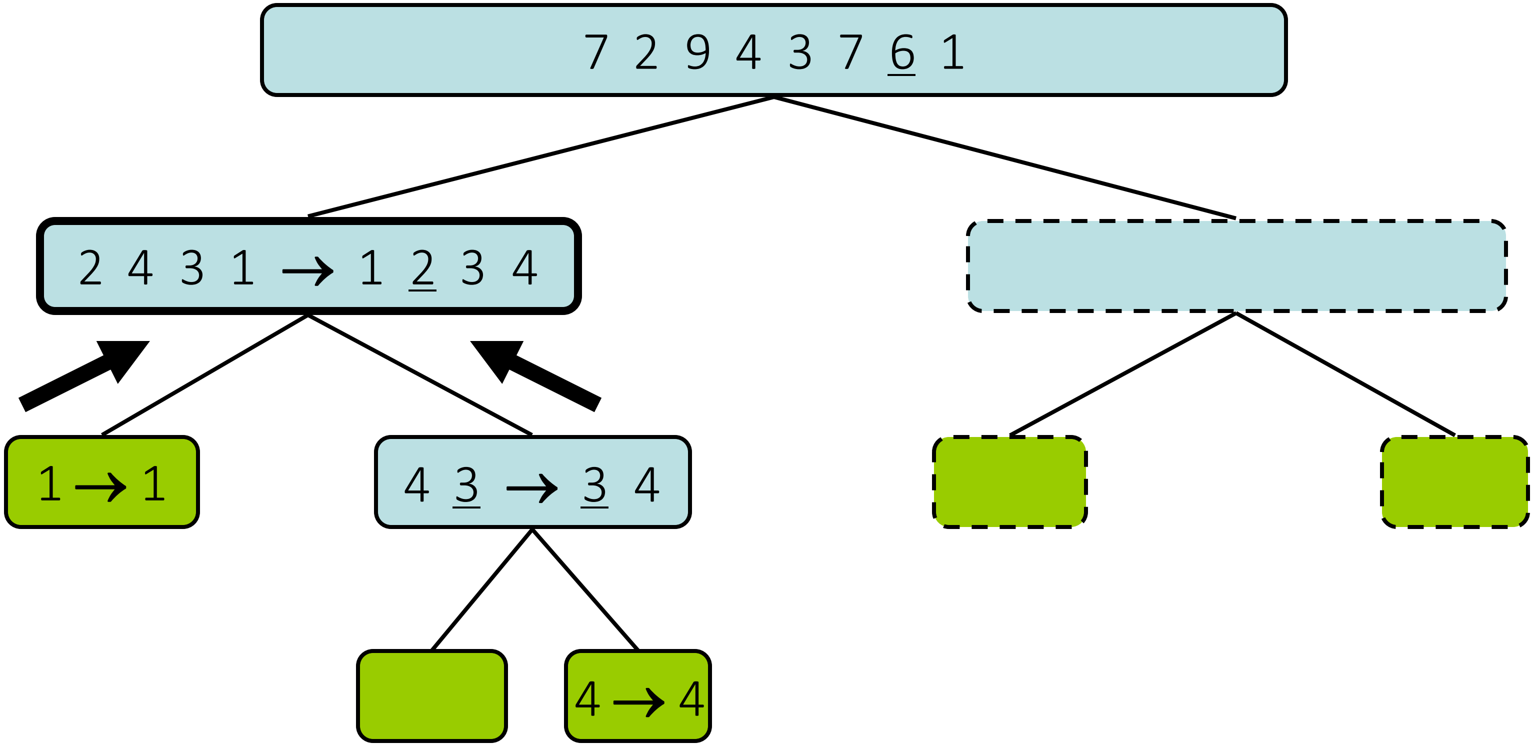
Execution Example (cont'd.)

- Partition, recursive call, base case



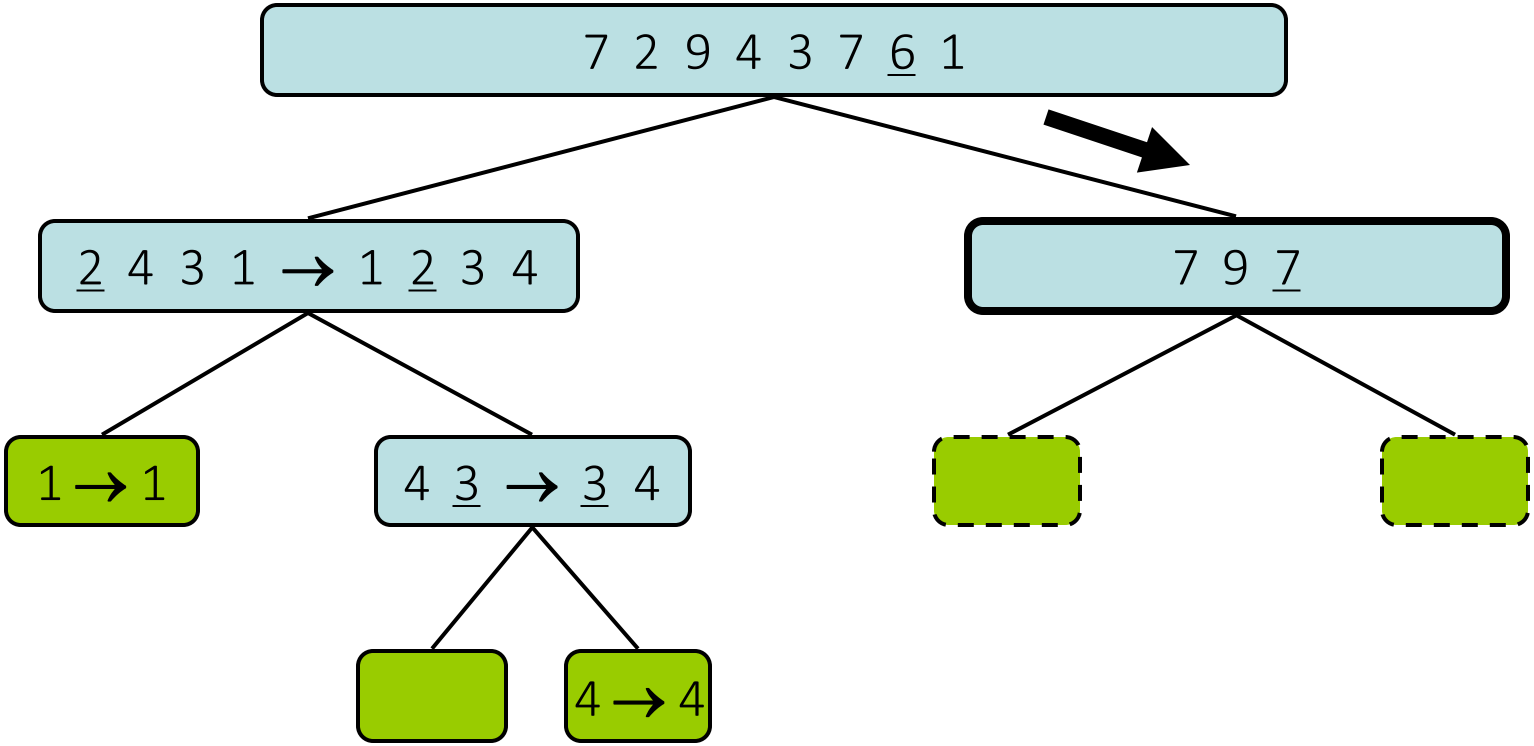
Execution Example (cont'd.)

- Recursive call, …, base case, join



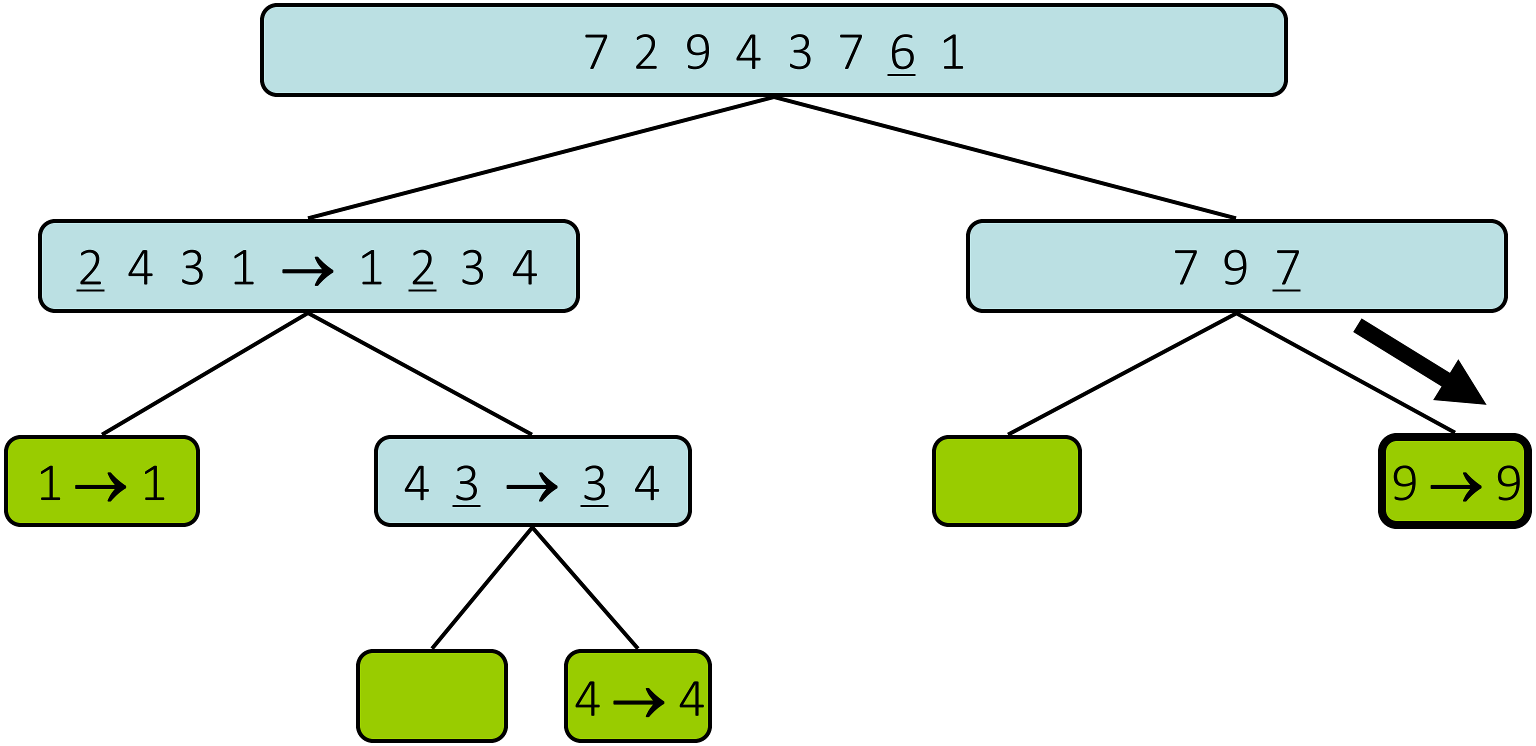
Execution Example (cont’d.)

- Recursive call, pivot selection



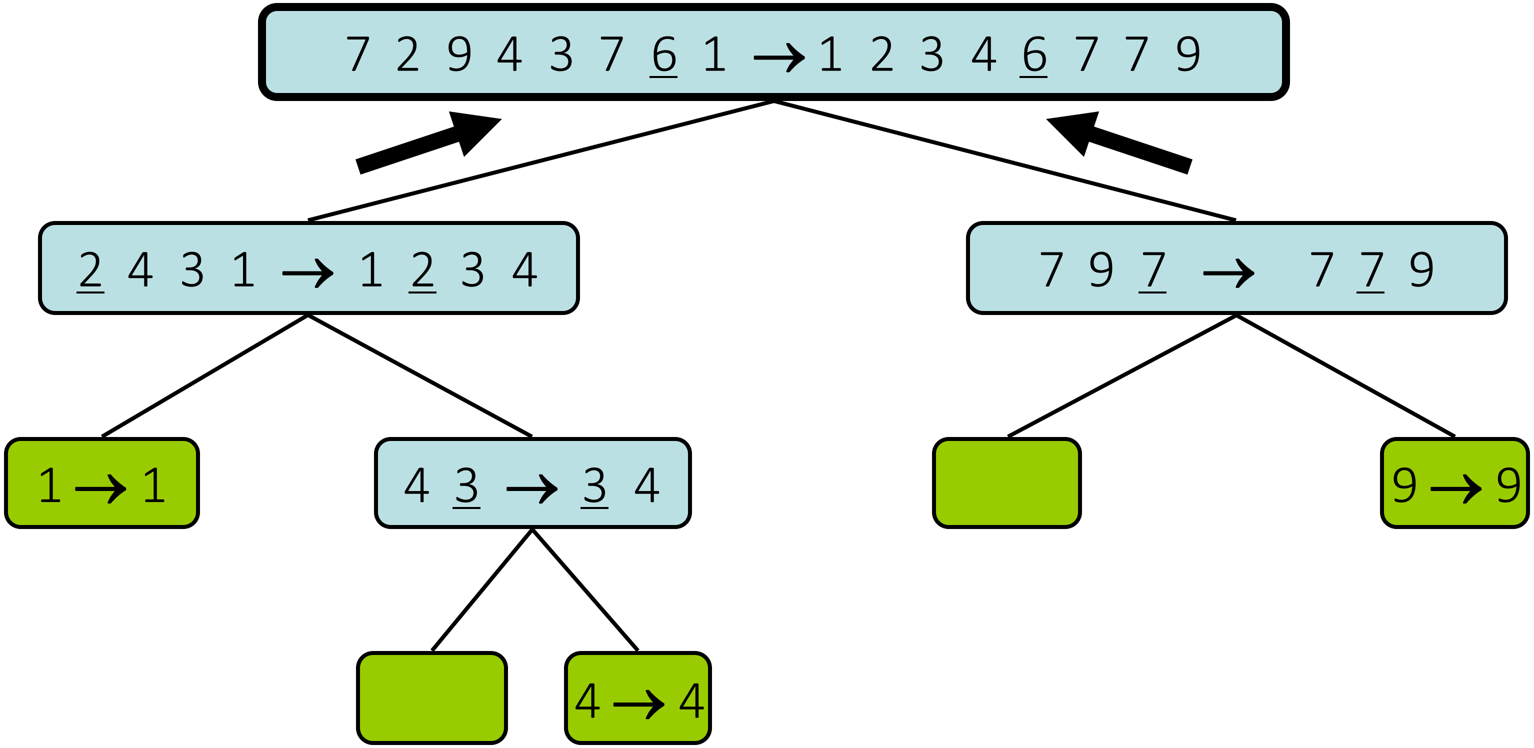
Execution Example (cont'd.)

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



Execution Example (cont’d.)

- Join, join



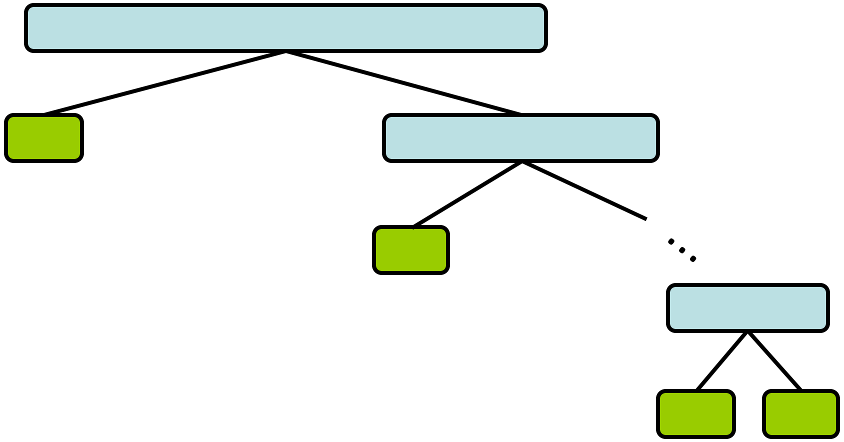
Worst-case Running Time

- 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

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

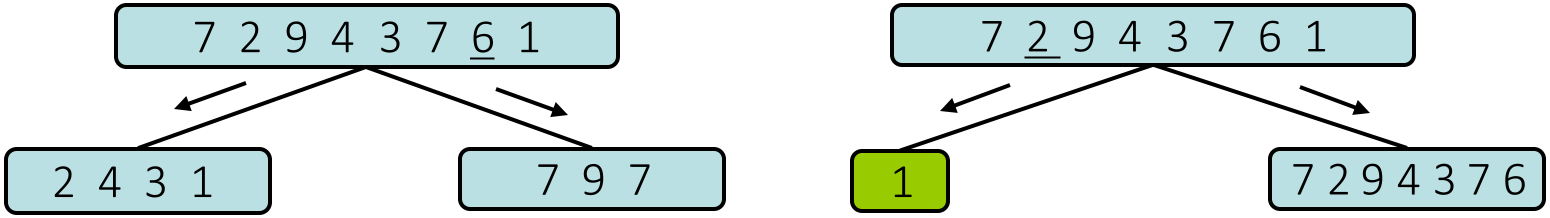
- Thus, the worst-case running time of quick sort is $O(n^2)$

depth	time
0	*n*
1	*n* - 1
…	…
n - 1	1



Expected Running Time

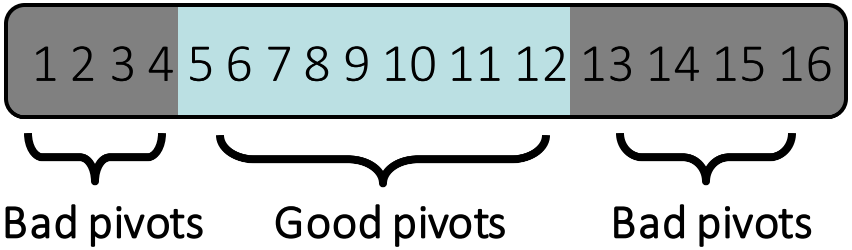
- Consider a recursive call of quick sort on a sequence of size s
  - Good call: the sizes of L and G are each less than $3s/4$
  - Bad call: one of L and G has size greater than $3s/4$



Good call

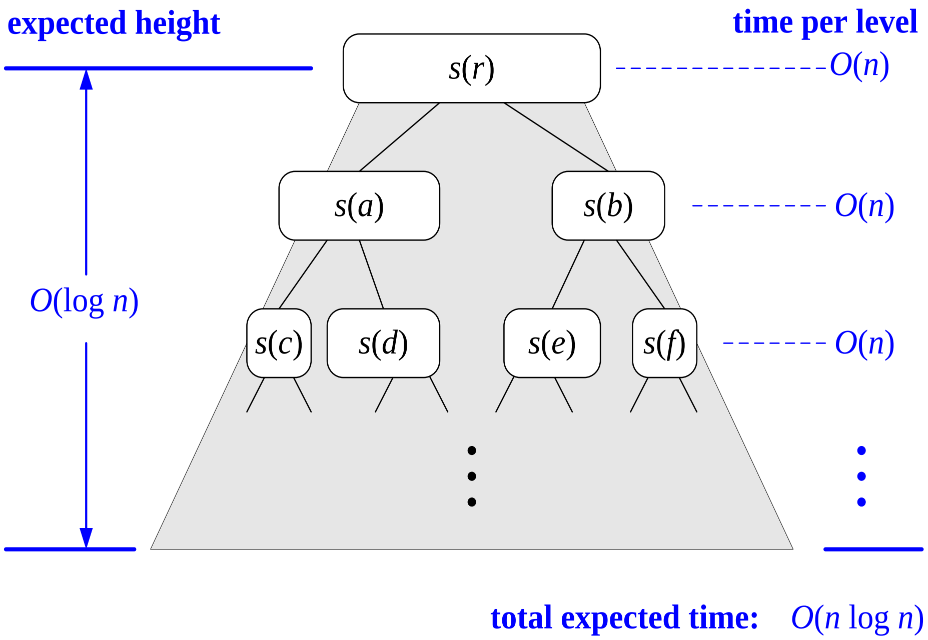
Bad call

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



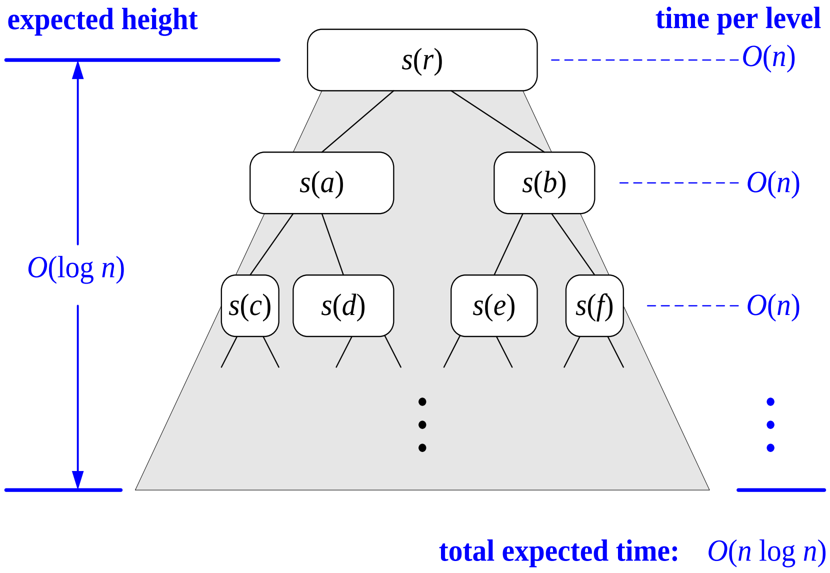
Expected Running Time (cont'd.)

- 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 $(3/4)^{i/2}n$



Expected Running Time (cont'd.)

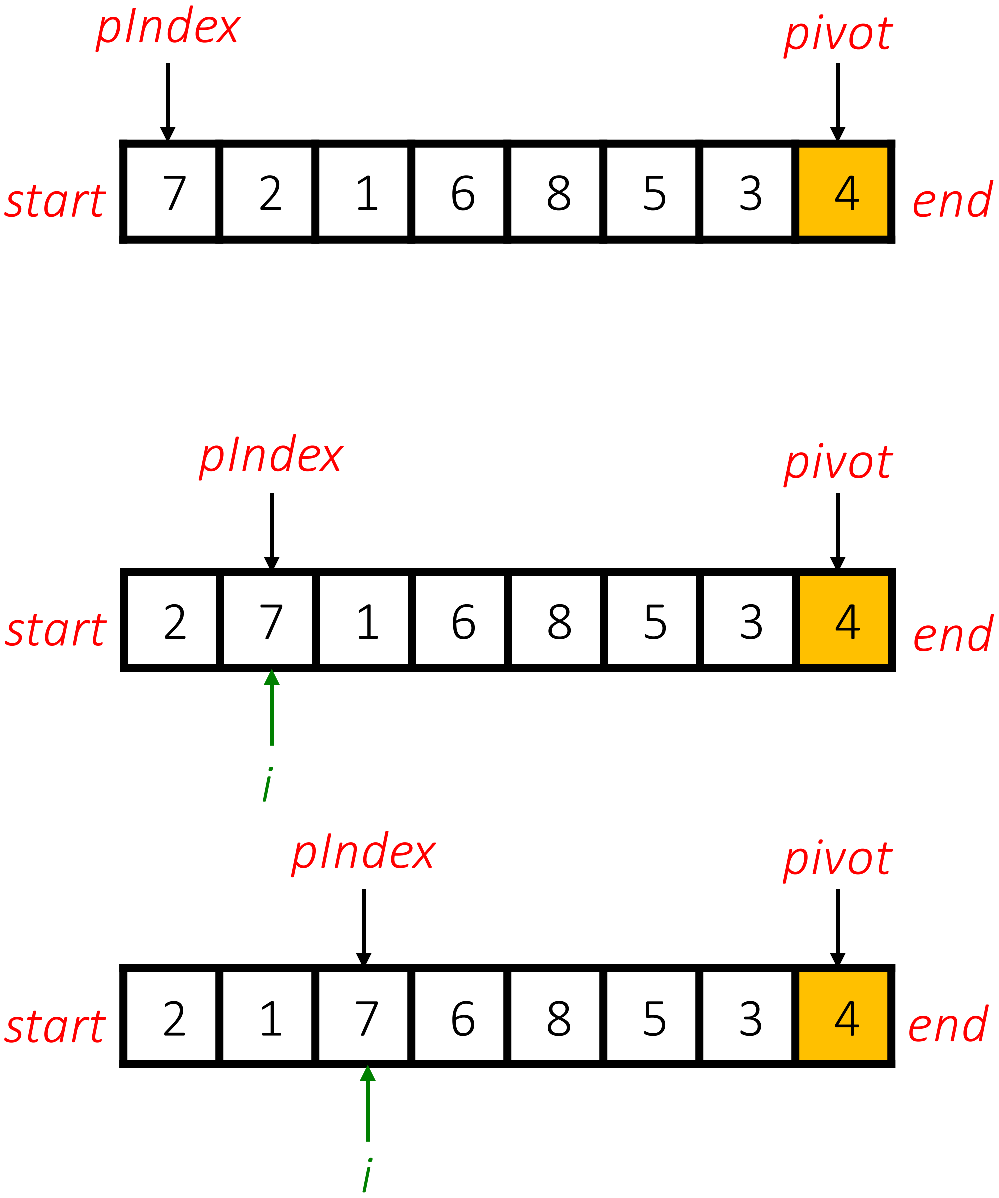
- Therefore, we have
  - For a node of depth $2\log_{4/3}n$, the expected input size is one
  - The expected height of the quick sort tree is $O(\log n)$
  - The amount or work done at the nodes of the same depth is $O(n)$
  - Thus, the expected running time of quick sort is $O(n \log n)$



In-Place Quick Sort

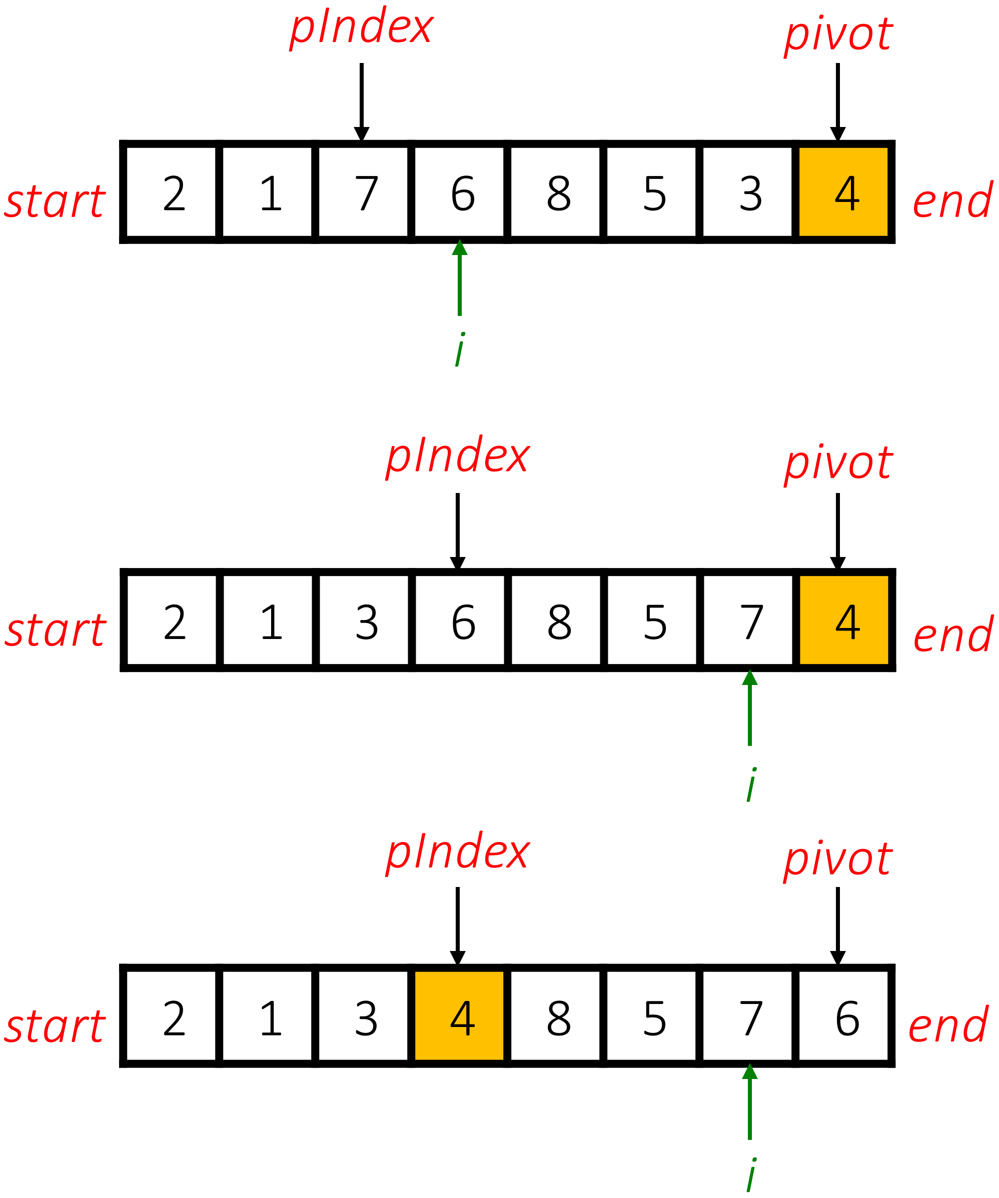
- Quick sort can be implemented to run in-place
- In the partition step, we use replace operations to rearrange the elements of the input sequence such that
  - The elements less than the pivot have rank less than *left*
  - The elements greater than the pivot have rank greater than *right*
- The recursive calls consider
  - Elements with rank less than *left*
  - Elements with rank greater than *right*

InPlaceQuickSort.java



```
Algorithm partition(S, start, end)
    Input sequence S,
          indices of segment start, end
    Output pIndex
    if start >= end then
        return
    pivot ← S[end]
    pIndex ← start
    for i ← 1 to end – 1 do
        if S[i] <= pivot then
            S[i] ↔ S[pIndex]
            pIndex ← pIndex + 1
    S[pIndex] ↔ S[end]
    return pIndex
```

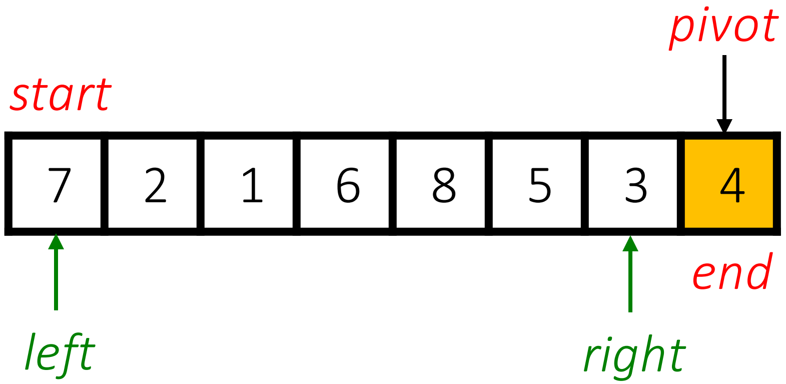
One Pointer

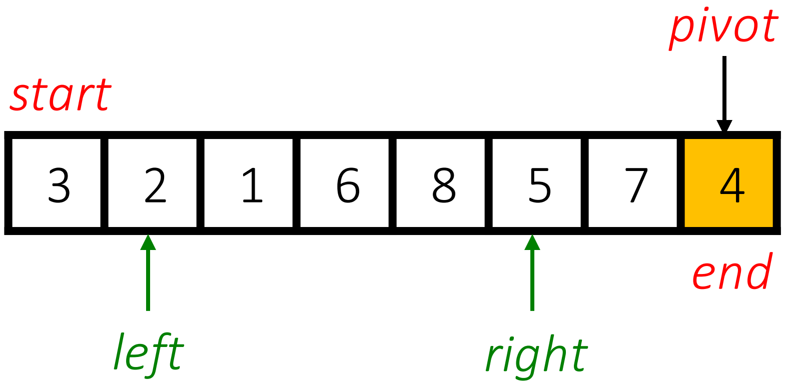


```
Algorithm partition($S$, $start$, $end$)
    Input sequence $S$,
            indices of segment $start$, $end$
    Output $pIndex$
     if $start >= end$ then
          return
    $pivot \leftarrow S[end]$
    $pIndex \leftarrow start$
    for $i \leftarrow 1$ to $end - 1$ do
        if $S[i] <= pivot$ then
            $S[i] \leftrightarrow S[pIndex]$
            $pIndex \leftarrow pIndex + 1$
    $S[pIndex] \leftrightarrow S[end]$
    return $pIndex$
```

One Pointer

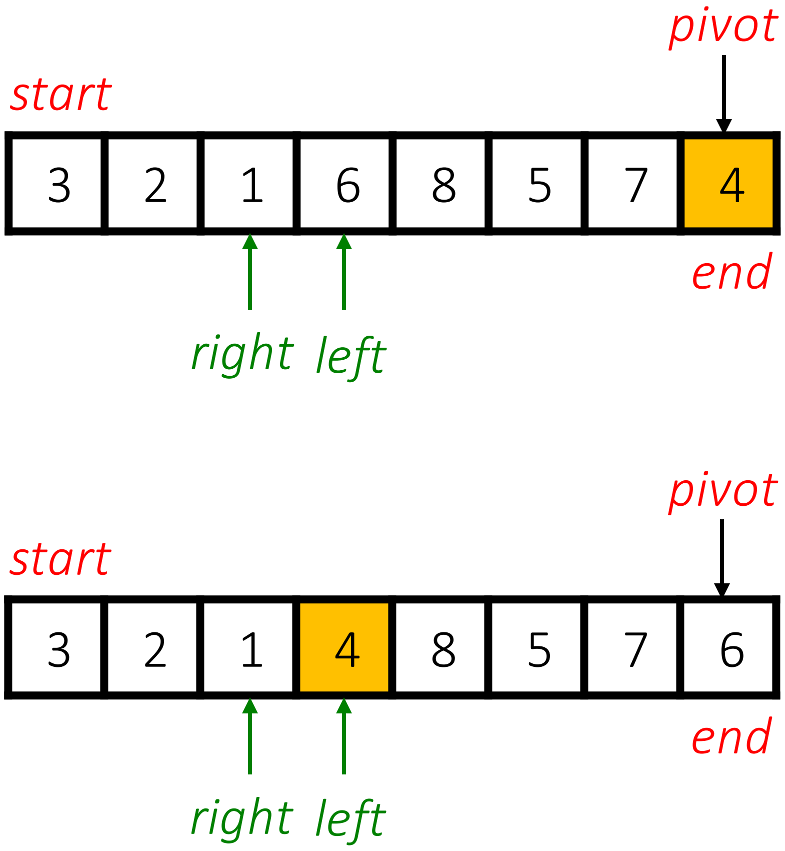
Two Pointers





```
Algorithm partition($S$, $start$, $end$)
    Input sequence $S$,
        indices of segment $start$, $end$
    Output $left$, $right$
    if $start >= end$ then return
    $pivot \leftarrow S[end]$
    $left \leftarrow start$
    $right \leftarrow end - 1$
    while $left <= right$ do
        while $S[left] <= pivot$ and $left <= right$ do
            $left \leftarrow left + 1$
        while $S[right] >= pivot$ and $left <= right$ do
            $right \leftarrow right - 1$
        if $left <= right$ then
            $S[left] \leftrightarrow S[right]$
            $left \leftarrow left + 1$
            $right \leftarrow right - 1$
    $S[left] \leftrightarrow S[end]$
    return $left$, $right$
```

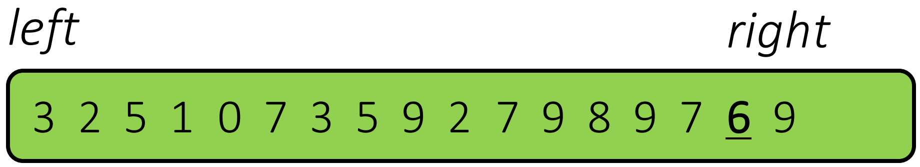
Two Pointers



```
Algorithm partition($S$, $start$, $end$)
    Input sequence $S$,
        indices of segment $start$, $end$
    Output $left$, $right$
    if $start >= end$ then return
    $pivot \leftarrow S[end]$
    $left \leftarrow start$
    $right \leftarrow end - 1$
    while $left <= right$ do
        while $S[left] <= pivot$ and $left <= right$ do
            $left \leftarrow left + 1$
        while $S[right] >= pivot$ and $left <= right$ do
            $right \leftarrow right - 1$
        if $left <= right$ then
            $S[left] \leftrightarrow S[right]$
            $left \leftarrow left + 1$
            $right \leftarrow right - 1$
    $S[left] \leftrightarrow S[end]$
    return $left$, $right$
```

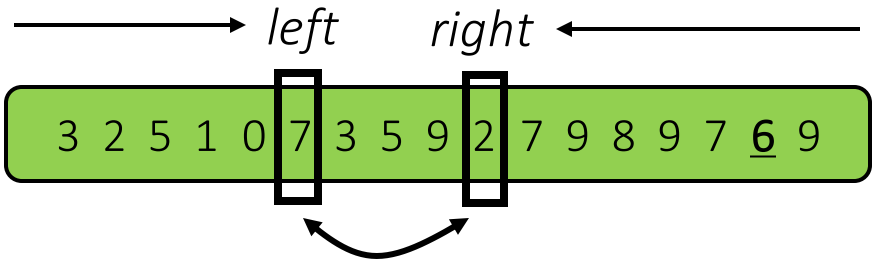
In-Place Partitioning: Two Pointer Method

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



(*pivot* = 6)

- Repeat until *left* and *right* cross:
  - Scan *left* to the right until finding an element $\geq$ *pivot*.
  - Scan *right* to the left until finding an element < *pivot*.
  - Swap elements at indices *left* and *right*



Randomized Quick Sort

```
Algorithm inPlaceQuickSort($S$, $start$, $end$)
    Input sequence $S$, ranks $start$ and $end$
    Output sequence $S$ with the elements of rank
        between $start$ and $end$ rearranged in increasing order
    if $start \geq end$ then
        return
    $i \leftarrow$ a random integer between $start$ and $end$
    $pivot \leftarrow S$.elemAtRank($i$)
    $(left, right) \leftarrow$ inPlacePartition($pivot$)
    inPlaceQuickSort($S$, $start$, $left - 1$)
    inPlaceQuickSort($S$, $left + 1$, $end$)
```

RandomizedQuickSort.java

Why Quick Sort is Unstable

- Think about this sequence:

$6\ 2_1\ 7\ \underline{5}\ 3\ 2_2\ 10$ where *pivot* = 5

- After partitioning :

$2_2\ 2_1\ 3\ \underline{5}\ 7\ 6\ 10$

Summary: Quick Sort

- Quick sort:
  - In place
  - Not stable
  - Very fast

	Worst-case	Best-case	Average	Space
Quick sort	$O(n^2)$	$O(n\log n)$	$O(n\log n)$	$O(n)$

Algorithm	Time	Notes
Selection sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs) ▪ stable
Insertion sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs) ▪ stable
Quick sort	$O(n\log n)$ expected	▪ in-place, randomized ▪ fastest (good for large inputs) ▪ not stable

Improved Quick Sort

- Solution: 3 way partition method / Dutch national flag problem

- The flag of the Netherlands consists of three colors: red, white and blue. Given balls of these three colors arranged randomly in a line (the actual number of balls does not matter), the task is to arrange them such that all balls of the same color are together and their collective color groups are in the correct order

Dutch National Flag Problem

- The solution to this problem is of interest for designing sorting algorithms; in particular, variants of the quicksort algorithm that must be robust to repeated elements need a three-way partitioning function that groups items less than a given key (red), equal to the key (white) and greater than the key (blue)
- Several solutions exist that have varying performance characteristics, tailored to sorting arrays with either small or large numbers of repeated elements
- In 3 way quick sort, an array *arr*[*start*…*end*] is divided in 3 parts:
  - `arr[start…i]` elements less than pivot
  - `arr[i+1...j-1]` elements equal to pivot
  - `arr[j...end]` elements greater than pivot

DnfQuickSort.java

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

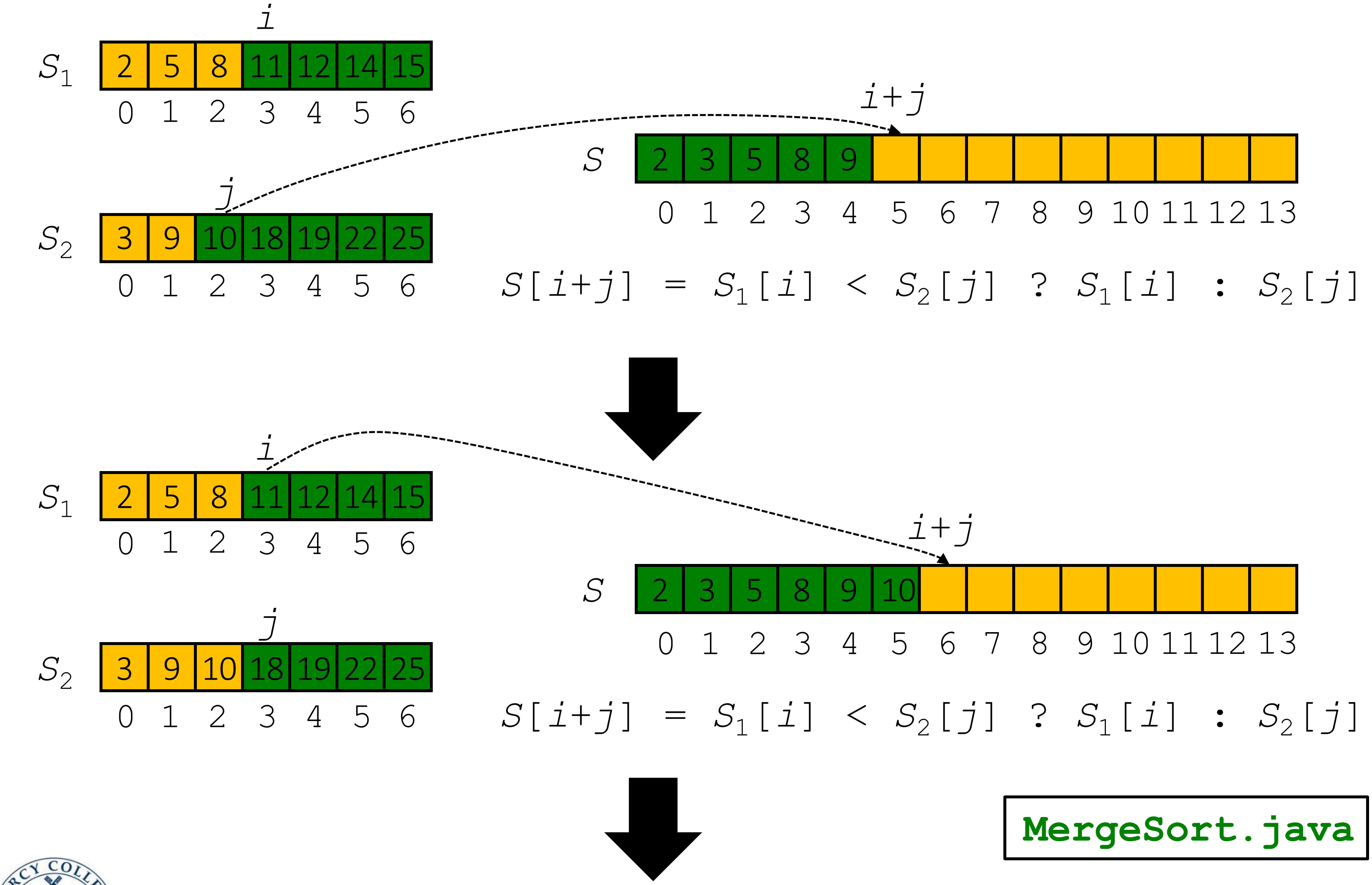
```
Algorithm mergeSort($S$)
    Input sequence $S$ with $n$ elements
    Output sequence $S$ sorted
    if $S$.size() $> 1$ then
        $(S_1, S_2) \leftarrow$ partition($S$, $n/2$)
        mergeSort($S_1$)
        mergeSort($S_2$)
        $S \leftarrow$ merge($S_1$, $S_2$)
```

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() do
        if $A$.first().element() $<$ $B$.first().element() then
            $S$.addLast($A$.remove($A$.first()))
        else
            $S$.addLast($B$.remove($B$.first()))
    while $\neg A$.isEmpty()    do
            $S$.addLast($A$.remove($A$.first()))
    while $\neg B$.isEmpty()    do
            $S$.addLast($B$.remove($B$.first()))
    return $S$
```

QueueMergeSort.java



				i			
S_1	2	5	8	11	12	14	15
	0	1	2	3	4	5	6

			j				
S_2	3	9	10	18	19	22	25
	0	1	2	3	4	5	6

						$i+j$								
S	2	3	5	8	9									
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

$S[i+j] = S_1[i] < S_2[j]\ ?\ S_1[i] : S_2[j]$

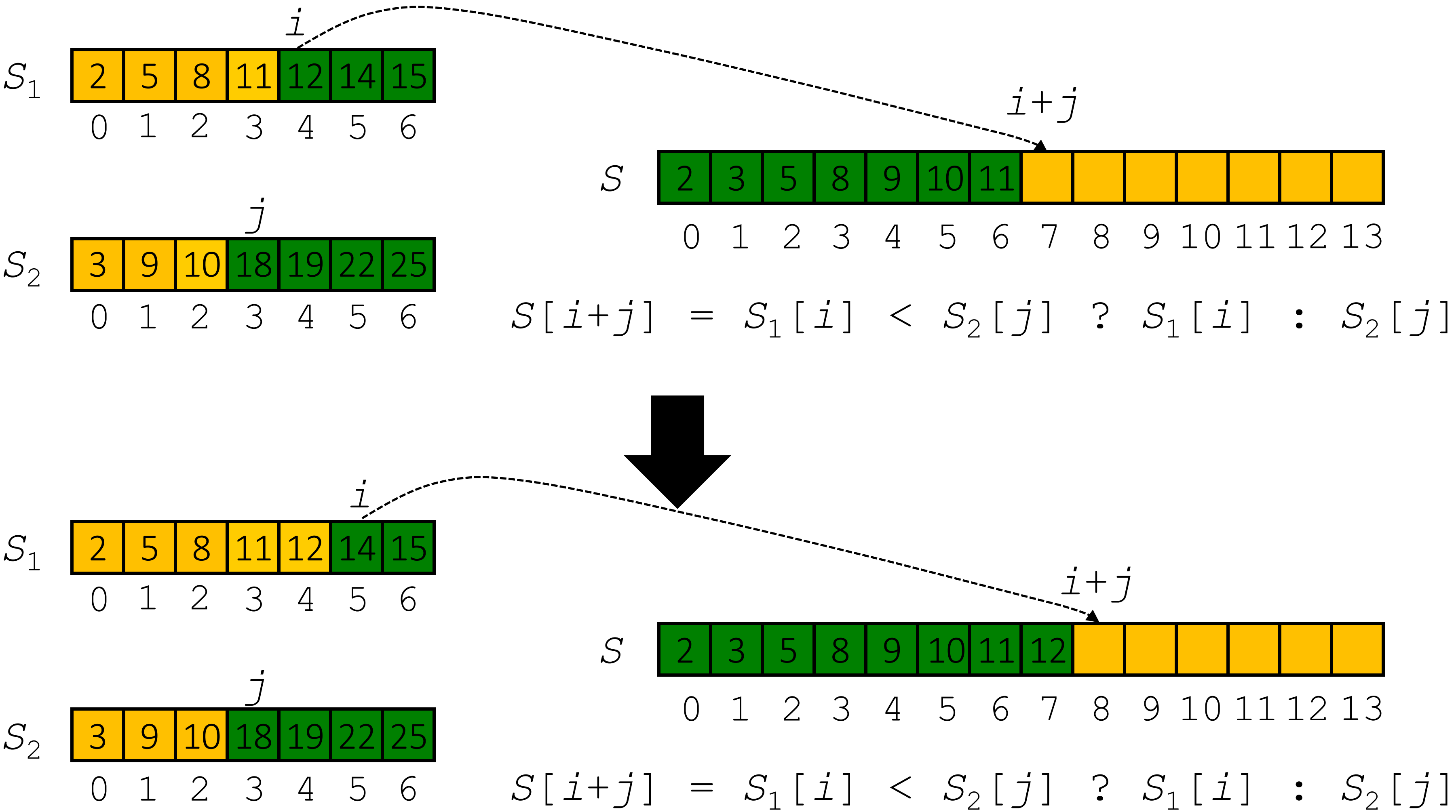
				i			
S_1	2	5	8	11	12	14	15
	0	1	2	3	4	5	6

				j			
S_2	3	9	10	18	19	22	25
	0	1	2	3	4	5	6

							$i+j$							
S	2	3	5	8	9	10								
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

$S[i+j] = S_1[i] < S_2[j]\ ?\ S_1[i] : S_2[j]$

MergeSort.java



	0	1	2	3	4	5	6
S_1	2	5	8	11	12	14	15

	0	1	2	3	4	5	6
S_2	3	9	10	18	19	22	25

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
S	2	3	5	8	9	10	11							

$S[i+j] = S_1[i] < S_2[j]\ ?\ S_1[i] : S_2[j]$

	0	1	2	3	4	5	6
S_1	2	5	8	11	12	14	15

	0	1	2	3	4	5	6
S_2	3	9	10	18	19	22	25

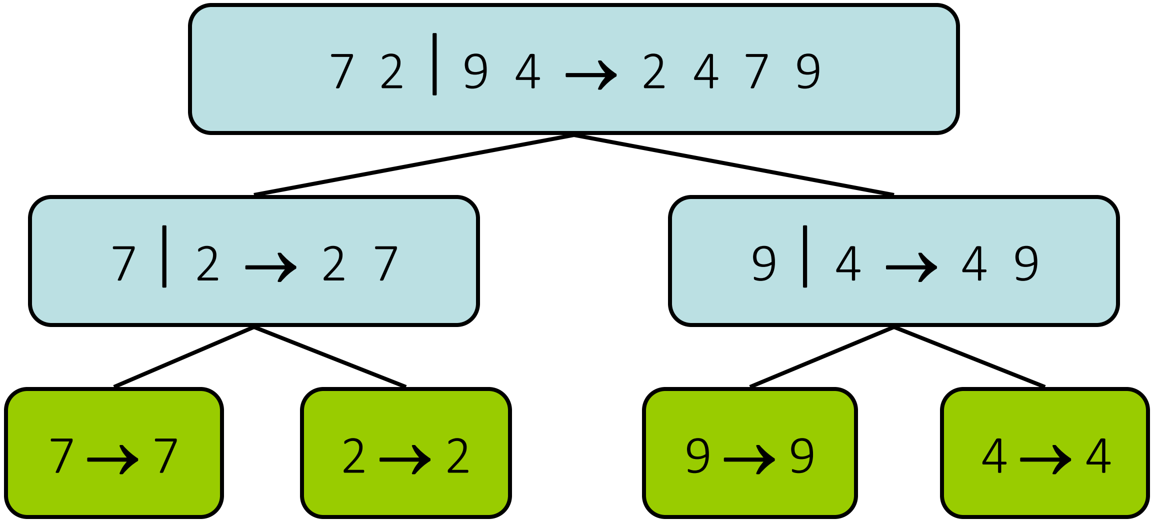
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
S	2	3	5	8	9	10	11	12						

$S[i+j] = S_1[i] < S_2[j]\ ?\ S_1[i] : S_2[j]$

MergeSort.java

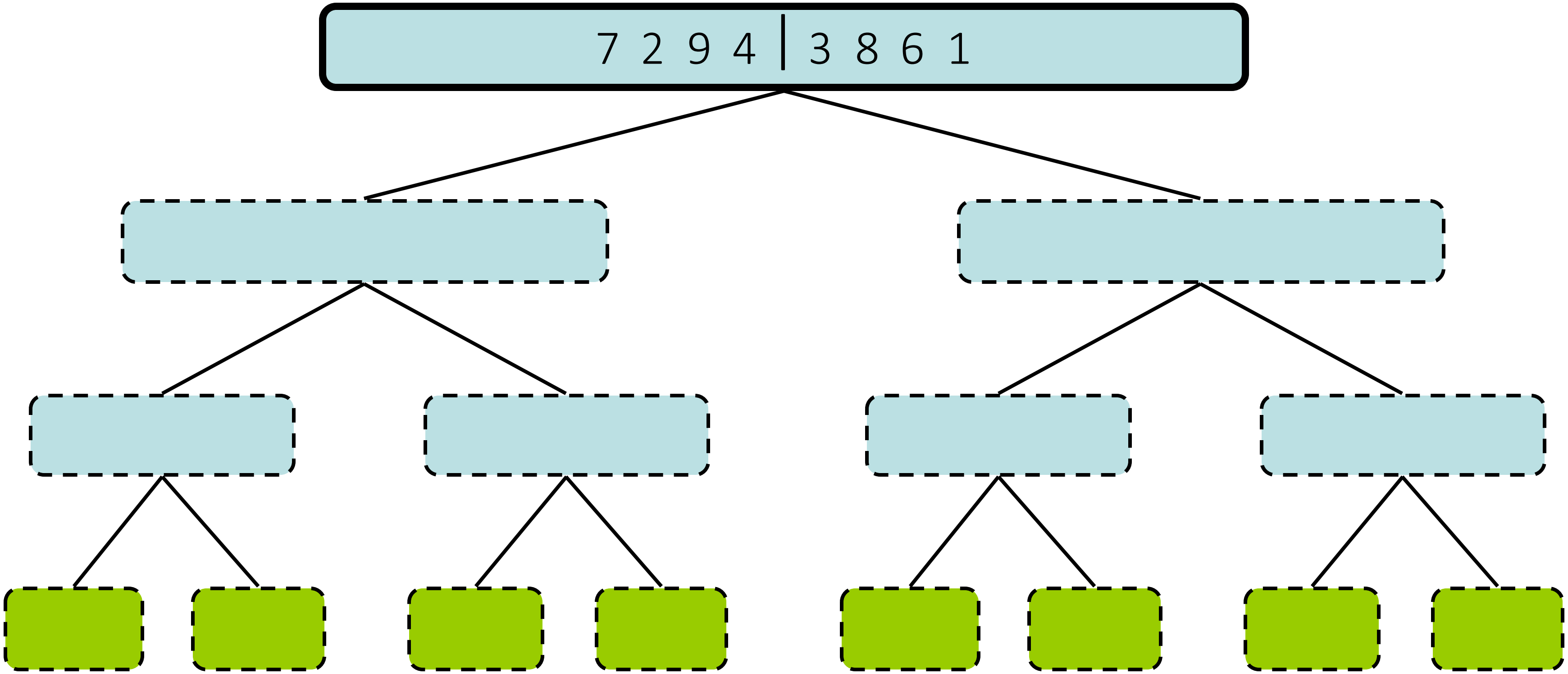
Merge Sort Tree

- 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



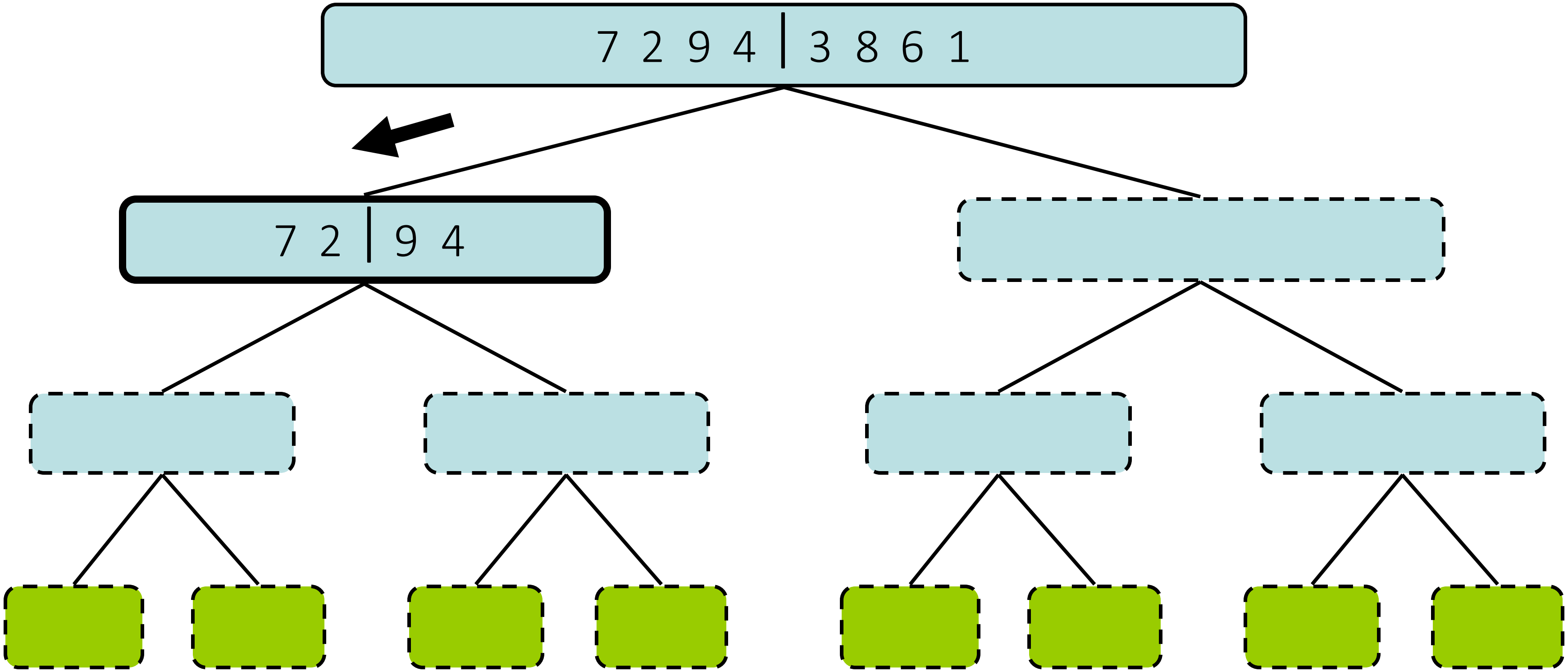
Execution Example

- Partition



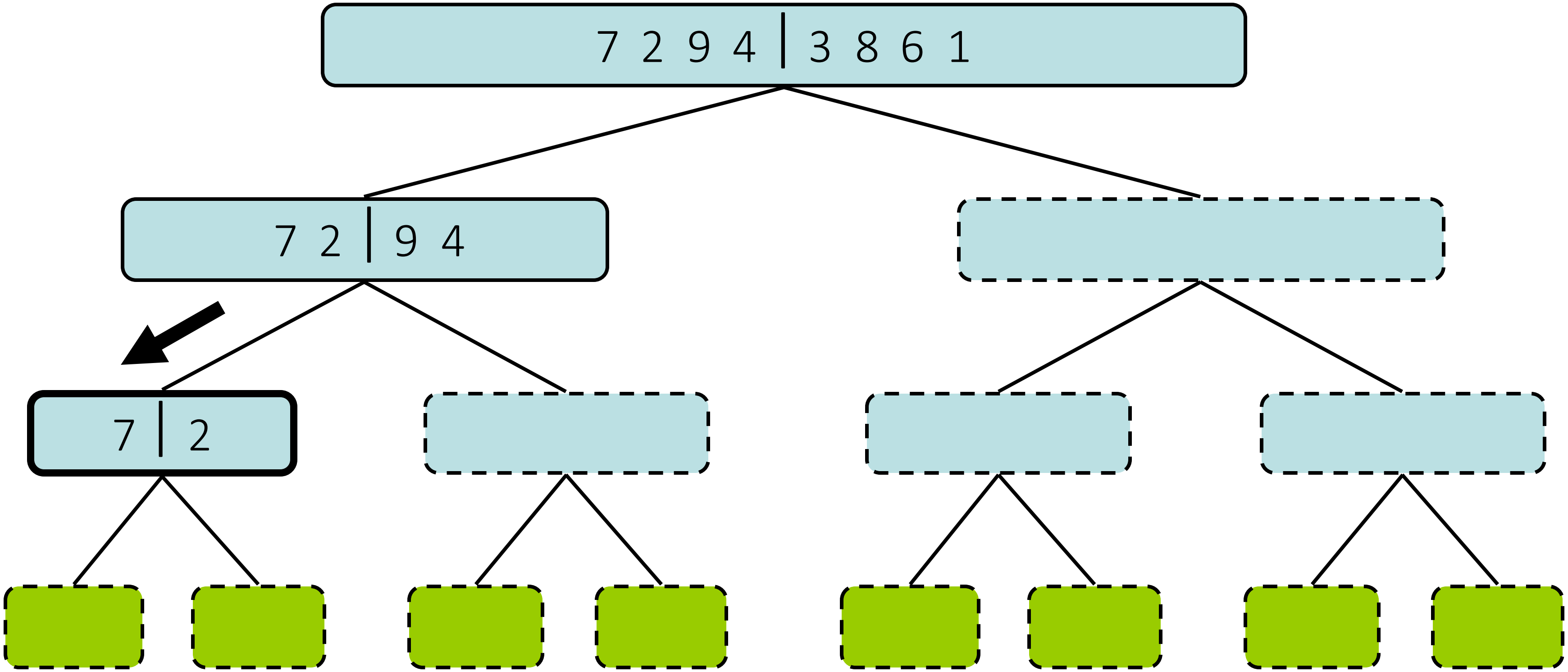
Execution Example (cont'd.)

- Recursive call, partition



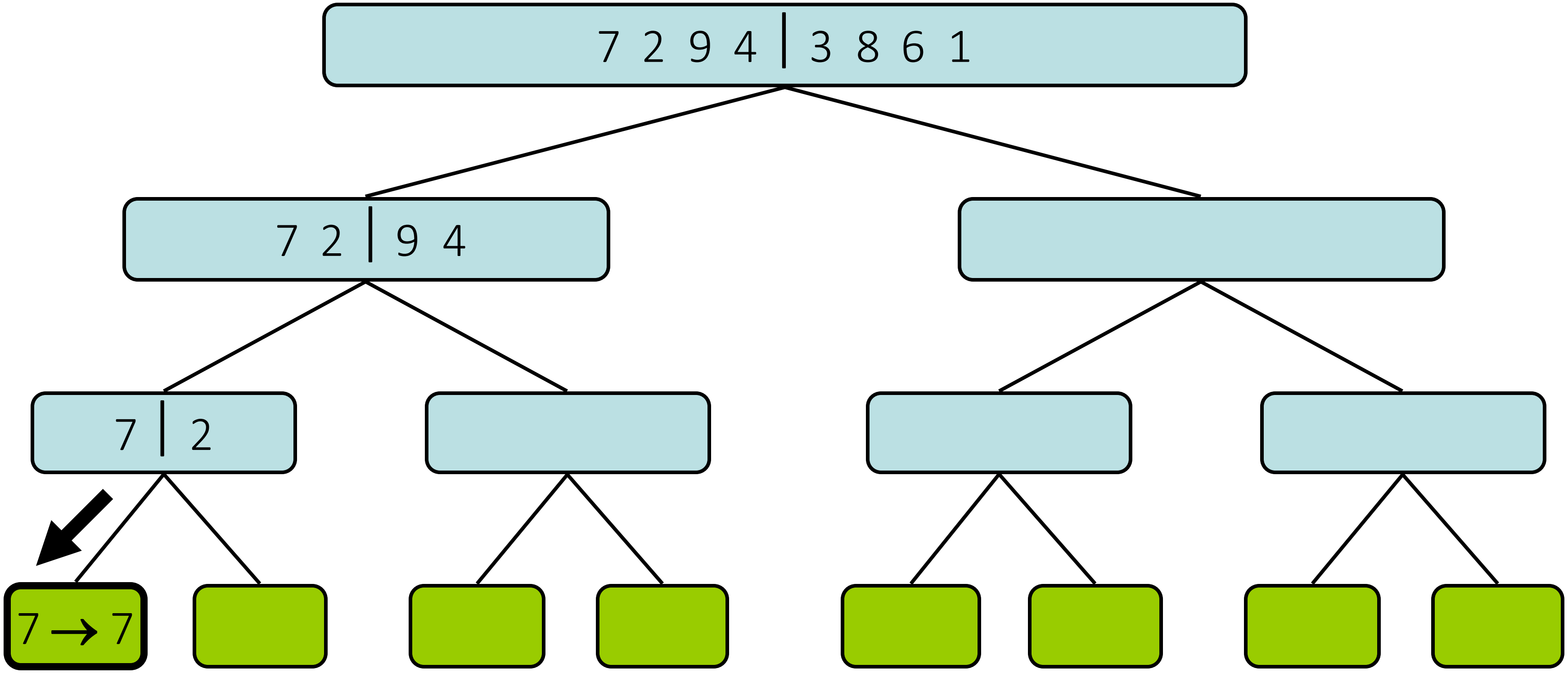
Execution Example (cont’d.)

- Recursive call, partition



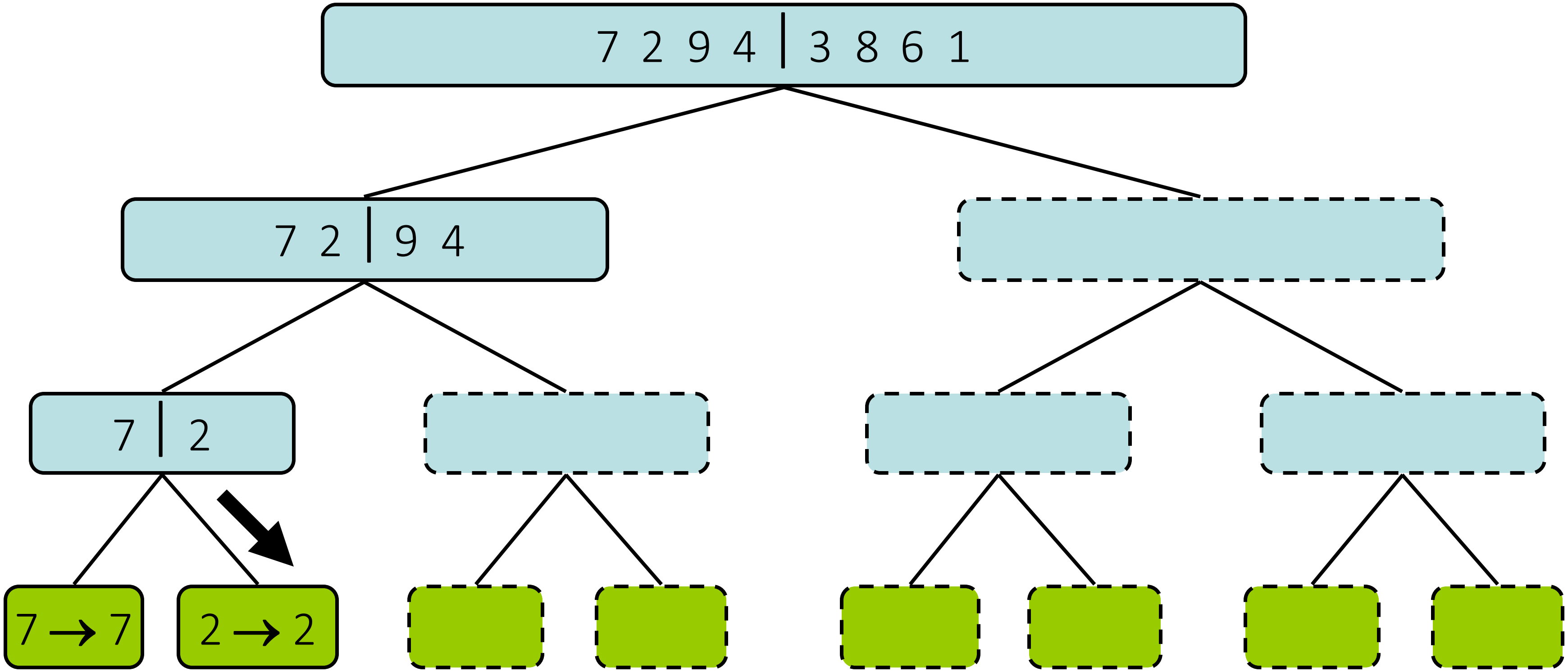
Execution Example (cont'd.)

- Recursive call, base case



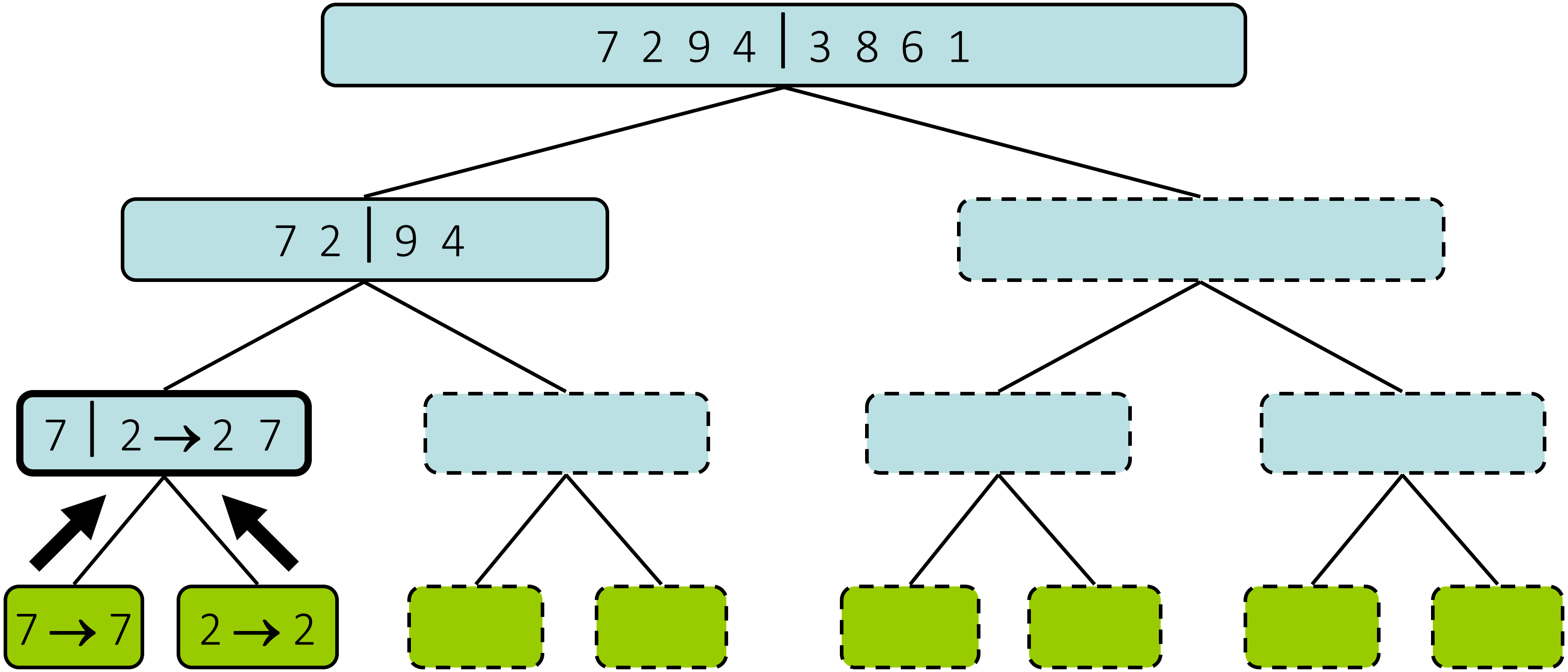
Execution Example (cont'd.)

- Recursive call, base case



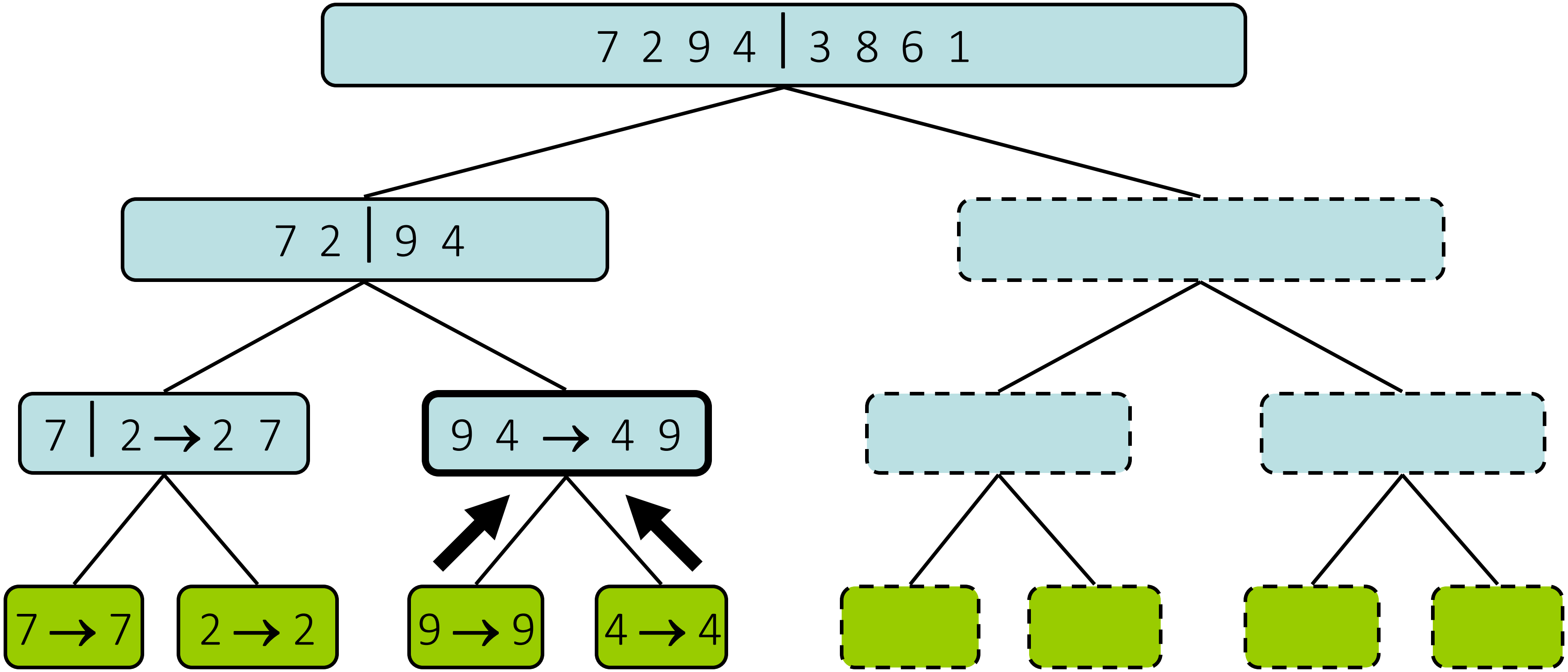
Execution Example (cont'd.)

- Merge



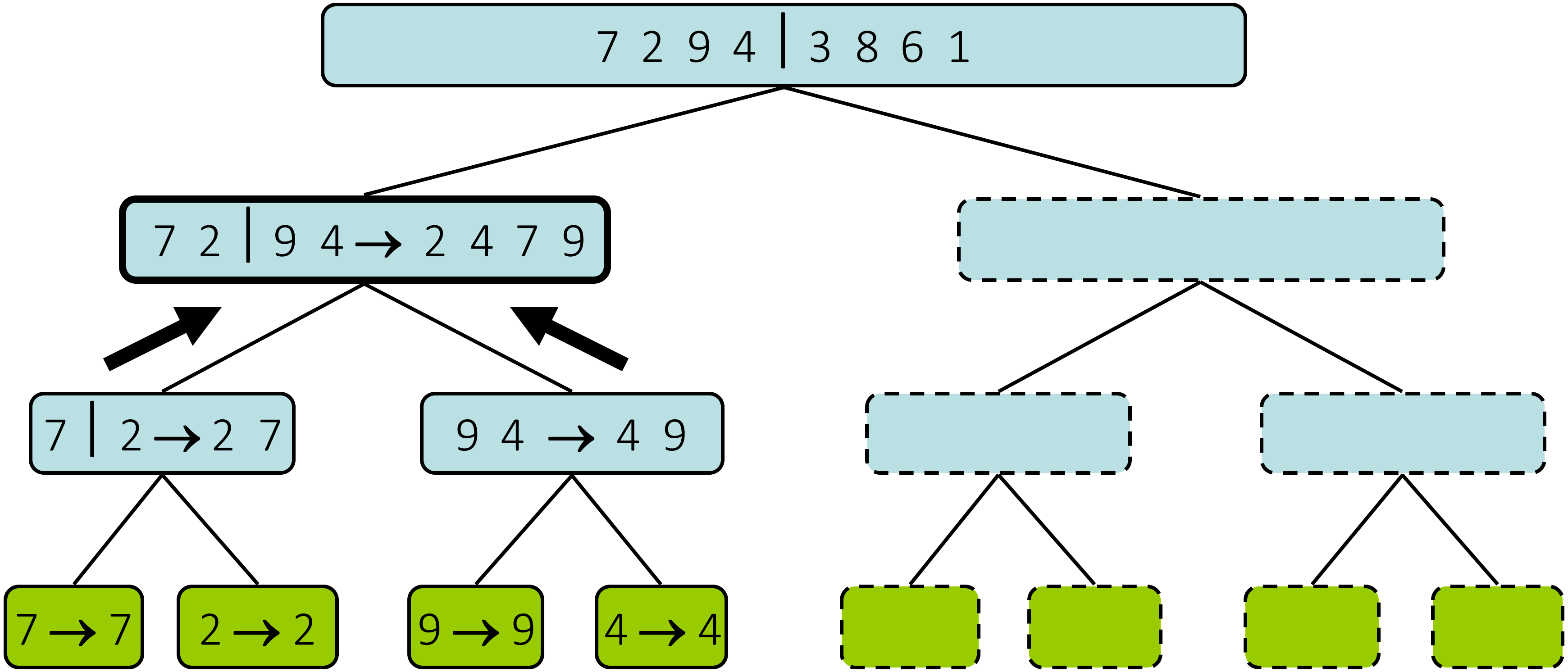
Execution Example (cont'd.)

- Recursive call, …, base case, merge



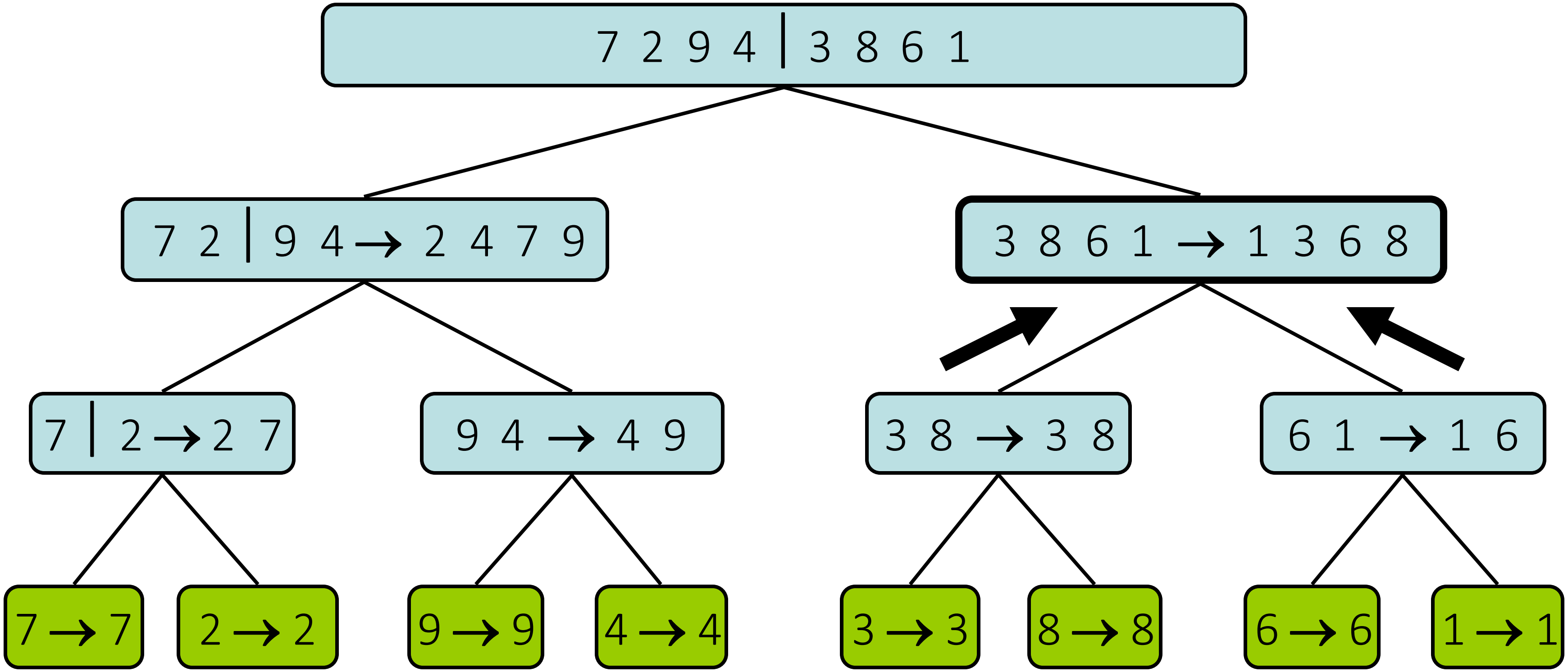
Execution Example (cont'd.)

- Merge



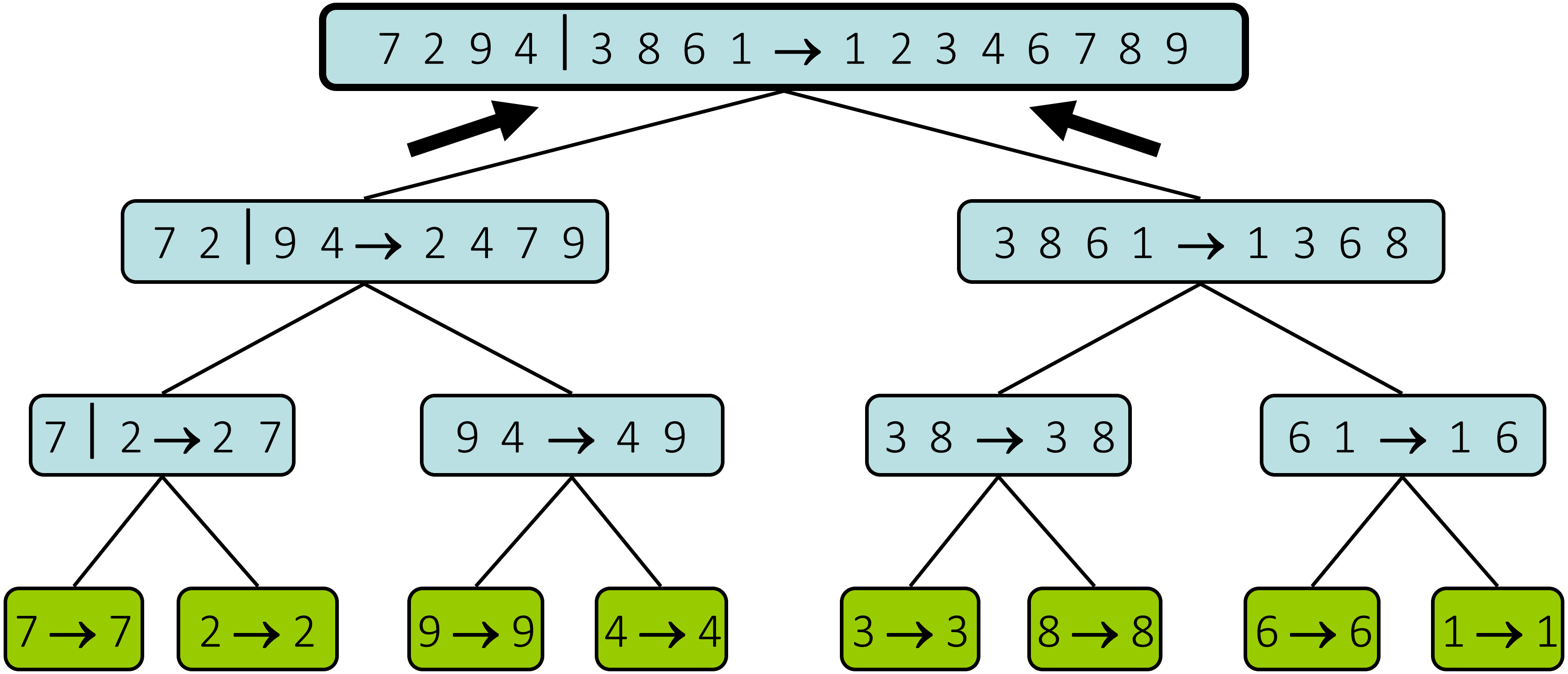
Execution Example (cont'd.)

- Recursive call, …, merge, merge



Execution Example (cont'd.)

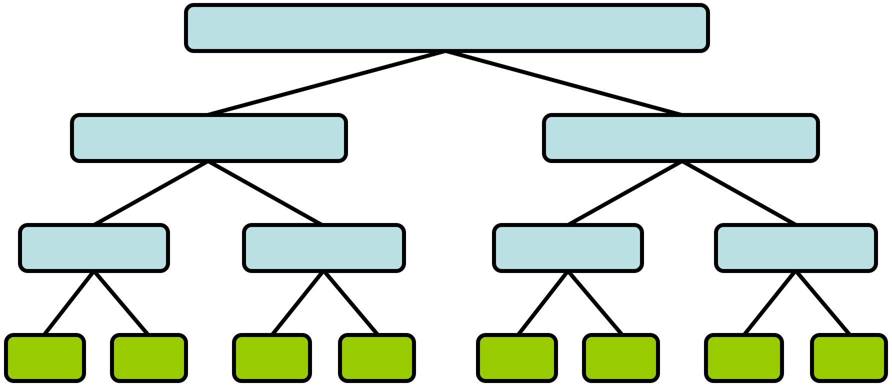
- Merge



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: Merge Sort

- Merge sort:
    - Out-of-place
    - Stable
    - Fast

	Worst-case	Best-case	Average	Space
Merge sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$

Algorithm	Time	Notes
Selection sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs) ▪ stable
Insertion sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs) ▪ stable
Quick sort	$O(n\log n)$ expected	▪ in-place, randomized ▪ fastest (good for large inputs) ▪ not stable
Merge sort	$O(n\log n)$	▪ fast ▪ sequential data access ▪ for huge data sets (> 1M) ▪ stable

Summary

- Basic sorting algorithms:
    - Bubble sort
    - Selection sort
    - Insertion sort
    - Invariants
    - Measures
        - Runtime complexity
        - Space complexity
        - Stable
        - In-place

Summary (cont'd.)

- Advanced sorting algorithms
    - Quick sort: in-place, not stable
        - Divide and conquer
        - Partitioning
        - Three way partition → out of place
        - Two way partition → in-place, but not stable
            - One pointer
            - Two pointers
            - Randomized pivot
        - Dutch national flag problem
    - Merge sort: not in-place, stable
        - Divide and conquer