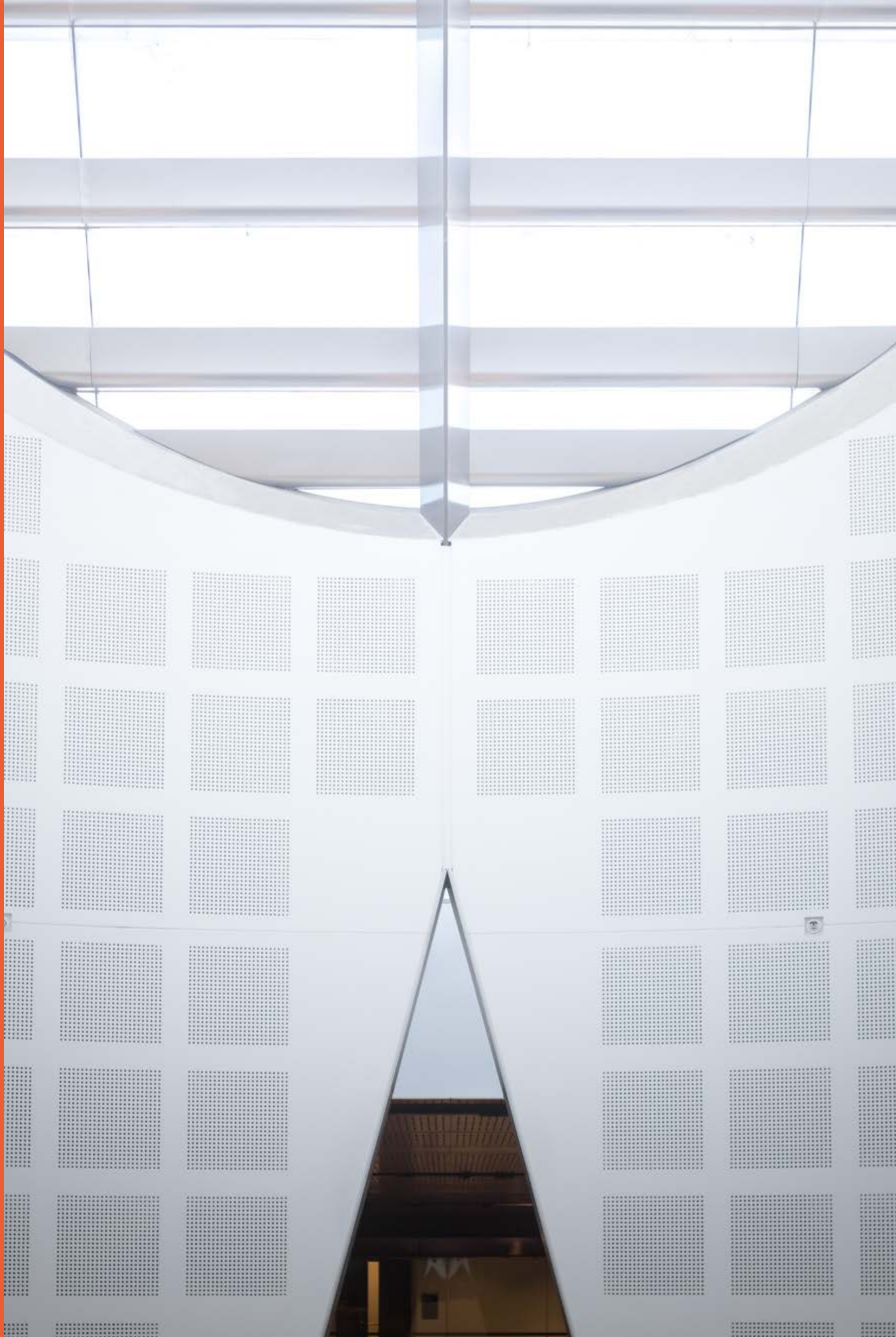
INFO1105/1905
Data Structures
Week 11: Sorting

see textbook section 12.1, 12.2, 12.4

Professor Alan Fekete
Prof Seokhee Hong
School of Information Technologies

using material from the textbook
and A/Prof Kalina Yacef, Dr Taso Viglas

THE UNIVERSITY OF SYDNEY



Copyright warning

- These slides contain material from the textbook (Goodrich, Tamassia & Goldwasser)
  - Data structures and algorithms in Java (5th & 6th edition)
- With modifications and additions from the University of Sydney

- The slides are a guide or overview of some big ideas
  - Students are responsible for knowing what is in the referenced sections of the textbook, not just what is in the slides

Reminder! Quiz 5

- Quiz 5 will take place during lab in week 12
- Done online, over a 20 minutes duration,
  - during the last 30 minutes of the lab period, or as indicated by your tutor
- A few multiple choice questions,
  - covering material from weeks 9, 10, 11
    - graph traversal (BFS, DFS)
    - directed graphs and algorithms
    - weighted graphs and algorithms
    - sorting: bubblesort, mergesort, quicksort
    - costs (run-time and space) of all algorithms above

Asst 2

- Due Friday October 27 (week 12); time extended to 11:59pm
- Discussed in second hour today

Outline

- Sorting algorithms and their costs
  - Review of pq-sorting algorithms: insertion, selection sort, heap-sort
  - In-place sorting
  - **Bubble sort**
  - **Merge-sort**
  - **Quick-sort**

Recall: Priority Queue Sorting (from week 5)

– We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of **insert** operations
     - element is used as key
     - null is the value (never considered, just goes along to fit the priority queue API)
  2. Remove the elements one-by-one with a series of **removeMin** operations
     - elements come out in sorted order

– The running time of this sorting method depends on the priority queue implementation

Algorithm ***PQ-Sort***(***S***, ***C***)
  Input list ***S***, comparator ***C*** for the elements of ***S***
  Output list ***S*** sorted in increasing order according to ***C***

  P ← priority queue with comparator ***C***
  while (!***S.isEmpty*** ())
    e ← ***S.removeFirst*** ()
    P.insert (***e***, **null**)
  while (!***P.isEmpty***())
    e ← ***P.removeMin***().***getKey***()
    S.addLast(***e***)

Recall: PQ-sorts (from week 5)

- PQ implemented as unsorted list
  - called **selection-sort**
  - worst-case runtime cost is **$O(n^2)$**: removeMin $O(n)$
- PQ implemented as sorted list
  - called **insertion-sort**
  - worst-case runtime cost is **$O(n^2)$**: insert $O(n)$
- PQ implemented as heap (either heap-ordered complete tree, or as heap-in-array)
  - called **heap-sort**
  - worst-case runtime cost is **$O(n \log n)$**: insert/removeMin $O(\log n)$

Outline

- Sorting algorithms and their costs
  - Review of pq-sorting algorithms: insertion, selection sort, heap-sort
  - **In-place sorting**
  - Bubble sort
  - Merge-sort
  - Quick-sort

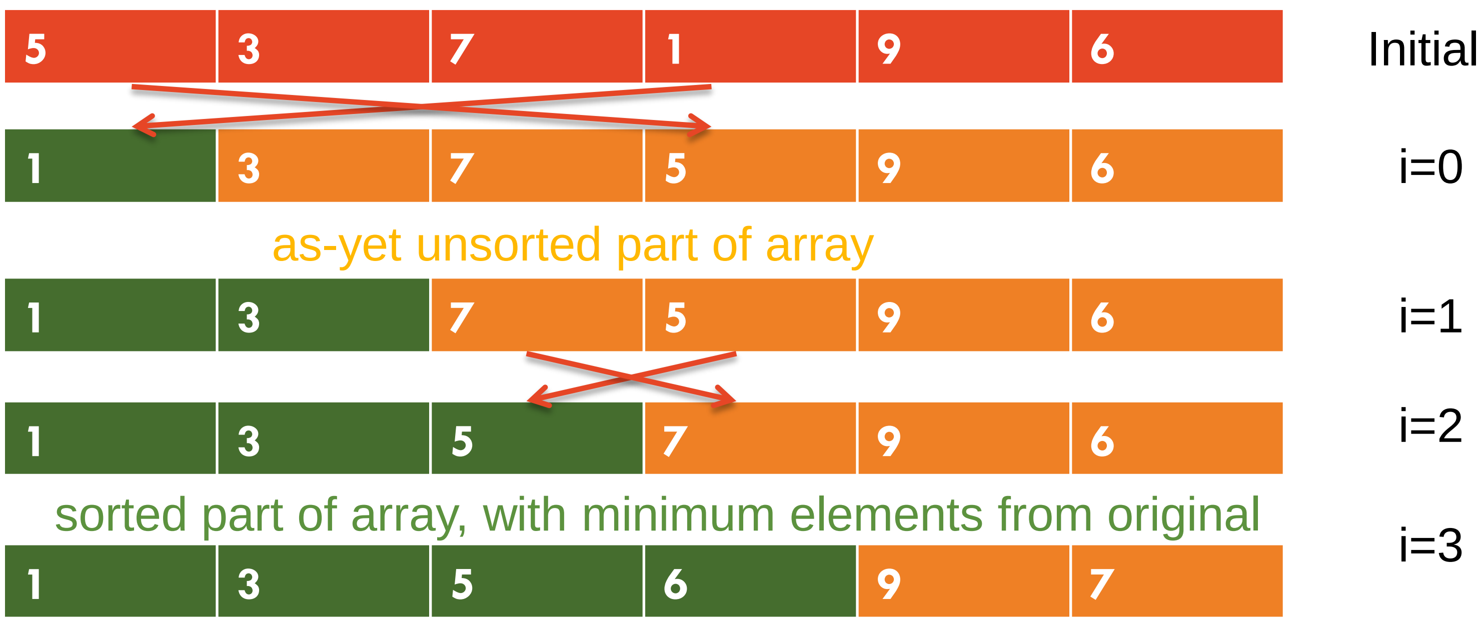
In-place sorting

- the simplest form of each PQ-sort keeps an extra data structure whose space is **O(n)**
  - *in addition to* the space used for the input data itself
  - we move elements into the **pq**, and perhaps move them around in the **pq** as more are inserted, and then move them from the **pq** to the output

- We can code similar algorithmic techniques, so that they move elements only within the array itself, which ends up being sorted
  - it may keep a small amount of extra space, for indices, local maximum, etc; this extra space should be **O(1)**

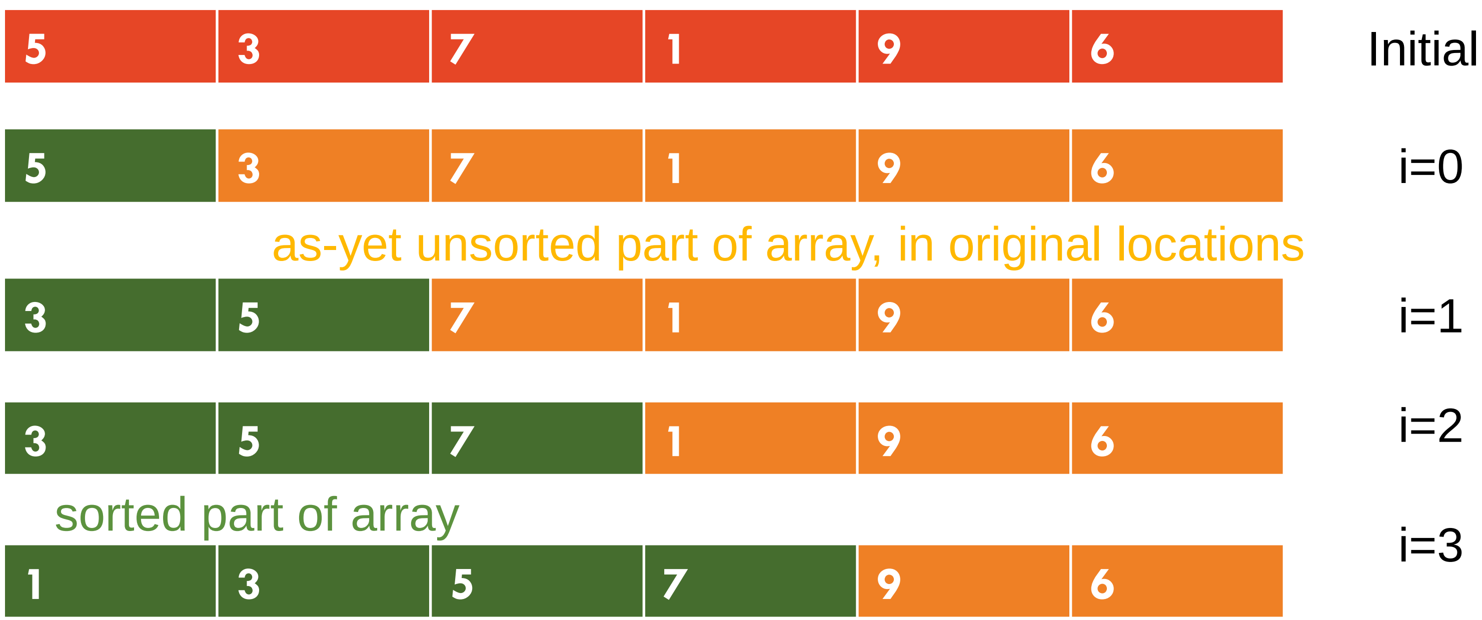
<u>In-place</u> selection sort

- The part of the array from <u>index 0 to i keeps the smallest elements of the original array, sorted</u>
- The part of the array from index i+1 to N-1 keeps the rest of the original contents
  - Each inner loop takes <u>min of unsorted part</u>, and <u>swap</u>s it into index i+1
- At the end of the algorithm I = N-1, so all the array is sorted



In-place insertion sort

– The part of the array from index 0 to i keeps a priority queue as a sorted array, that is a sorted form of the original contents
– The part of the array from index i+1 to N-1 has its original contents
– At the end of the algorithm I = N-1, so all the array is sorted



In-place heapsort

- Clever technique to build the **maxheap-in-array**, working in-place (Section 9.4.2)

- Repeat: removing max among remaining elements and put into index N-1
  - remove the max (which is at index 0)
  - restructure the remaining maxheap-in-array in index 0 to N-1
  - put the removed max in index N-1

- Time complexity: O(n log n)

Outline

- Sorting algorithms and their costs
  - Review of pq-sorting algorithms: insertion, selection sort, heap-sort
  - In-place sorting
  - **Bubble sort**
  - Merge-sort
  - Quick-sort

Bubble-sort

- in-place algorithm for data in an array
- Very easy to code
- Variants can be parallelized well
- But: performance is usually slow, or (even) medium data
- Key idea: To sort a sequence of n comparable elements
  - Scan the sequence n-1 times
  - In each step of a scan, compare the current element with the next and swap them if they are out of order
    - note that if a swap occurs, the next step will compare the larger element in its new position, with its successor in the array
    - by the end of the scan, the largest element has reached the last position
    - so the next scan works on a slightly shorter part of the array

Example Bubble-sort

__: comparison

swap

sorted

First Pass:

(5 1 4 2 8) → (1 5 4 2 8)

(1 5 4 2 8) → (1 4 5 2 8)

(1 4 5 2 8) → (1 4 2 5 8)

(1 4 2 5 8) → (1 4 2 5 8)

Second Pass:

(1 4 2 5 8) → (1 4 2 5 8)

(1 4 2 5 8) → (1 2 4 5 8)

(1 2 4 5 8) → (1 2 4 5 8)

Third Pass:

(1 2 4 5 8) → (1 2 4 5 8)

(1 2 4 5 8) → (1 2 4 5 8)

Fourth Pass:

(1 2 4 5 8) → (1 2 4 5 8)

Bubble-sort

array elements[1..N]
for *j:= 1* **to** *N-1* <u>*do*</u>
    for *k := 1* **to** *N-j* <u>*do*</u>
        if *elements[k] > elements[k+1]*
        then *swap(k, k+1, elements)*

Iterations of bubble-sort

- Each full iteration through the array (inner for loop) will place at least one element at its final resting position
  - The last element

- Therefore in the next iteration we do not need to check if the last element needs to be swapped
  - Inner for loop stops at N-j in the j-th iteration

Bubble-sort running time

outer loop executes *N-1* times: *O(N)*

array elements[1..N]

```
for j:= 1 to N-1 do
    for k := 1 to N-j do
        if elements[k] > elements[k+1]
        then swap(k, k+1, elements)
```

inner loop executes a number of times
that depends on j
worst case is to execute *N-1* times: *O(N)*

inside the inner loop:
time doesn't grow no matter
how much data in array
cost: O(1)

total worst-case running time: $O(N)*O(N)*O(1)=$ $\mathbf{O(N^2)}$

Variant bubble-sort

// may stop early as soon as one pass finds no swaps needed

```
array elements[1..N]
swapDone = true

while swapDone do
   swapDone = false
   for k := 1 to N-1 do
    if elements[k] > elements[k+1] then
         swap(k,k+1, elements)
         swapDone = true
```

Variant Bubble-sort running time bound

- Is this a faster solution than nested for loops ?
  - In many cases, it takes less time, because it ends early
- But what is the worst-case running time?
- General analysis: Each complete iteration of the inner loop will place at least one new element in its final position
  - A maximum of N iterations of the while loop are needed
  - This implies a bound of no more than $O(n^2)$
- Is there a lower bound?
  - or, does the variant still need all N-1 iterations of the inner loop in its worst-case?
- What input makes the algorithm use all N-1 iterations of inner loop?
  - When the input is in reverse sorted order
  - Eg 8,7,6,5,4,3,2,1
  - Indeed any input where the smallest element is at the end
- Variant bubble-sort has **$O(n^2)$ worst-case runtime**

Outline

- Sorting algorithms and their costs
  - Review of pq-sorting algorithms: insertion, selection sort, heap-sort
  - In-place sorting
  - Bubble sort
  - **Merge-sort**
  - Quick-sort

Divide-and-Conquer Algorithm

- 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
- $O(n \log n)$ running time
  - It does not use an auxiliary priority queue
  - It accesses data in a sequential manner (suitable to sort data stored 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 a comparator function)

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

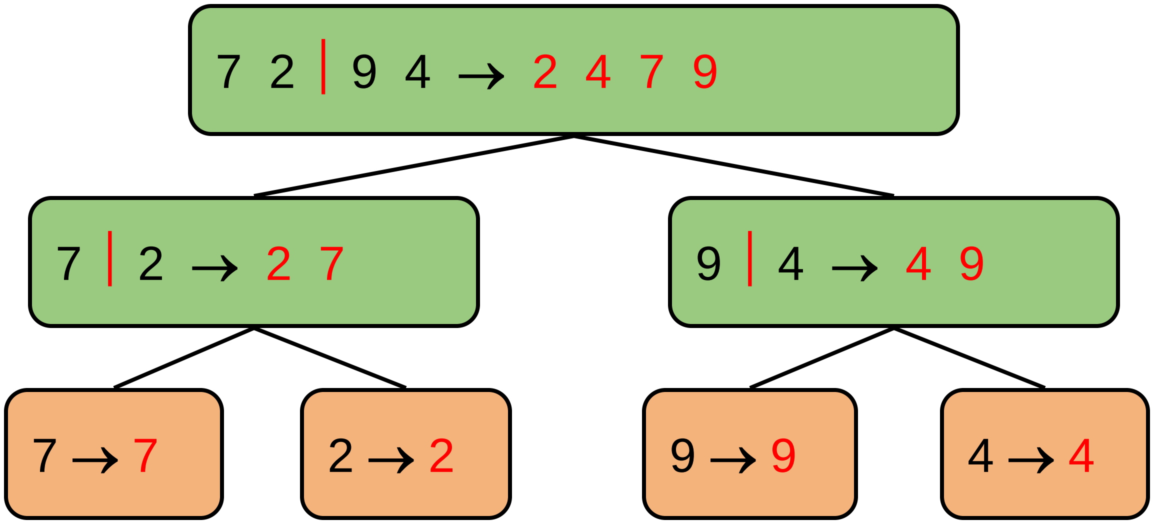
- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B

- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

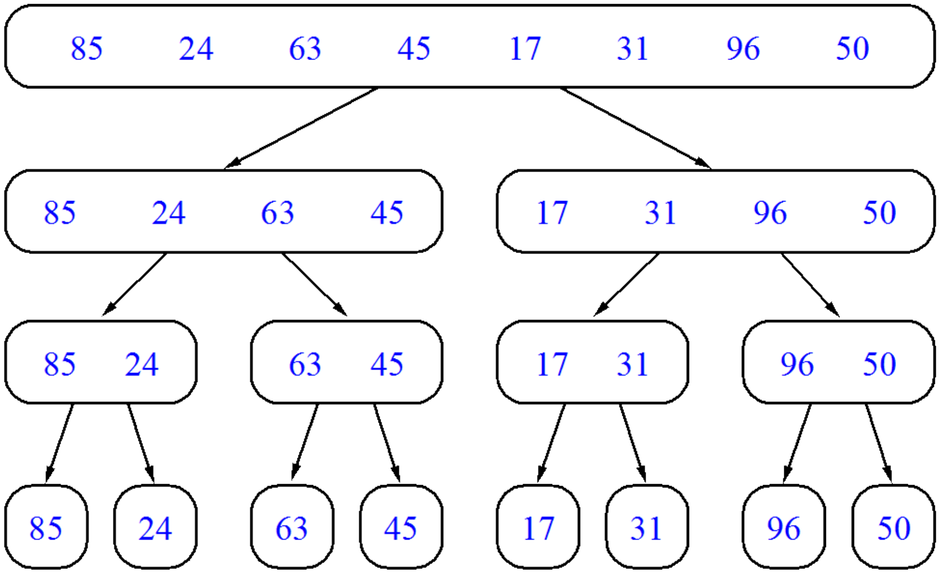
```
Algorithm merge(A, B)
    Input sorted sequences A, B with n/2 elements each
    Output sorted sequence S of A ∪ B

    S ← empty sequence
    while !A.isEmpty()  && !B.isEmpty()
        if A.first().element() < B.first().element()
            S.addLast(A.remove(A.first()))
        else
            S.addLast(B.remove(B.first()))

    while !A.isEmpty()
        S.addLast(A.remove(A.first()))
    while !B.isEmpty()
        S.addLast(B.remove(B.first()))
    return S
```

Merge-Sort Tree

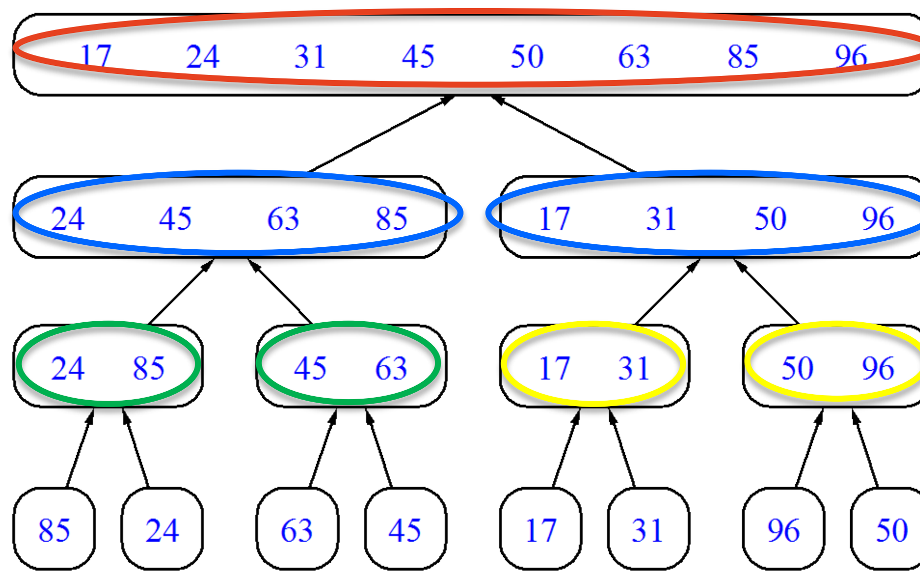
- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences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



Merge sort trees (input and output sequences)



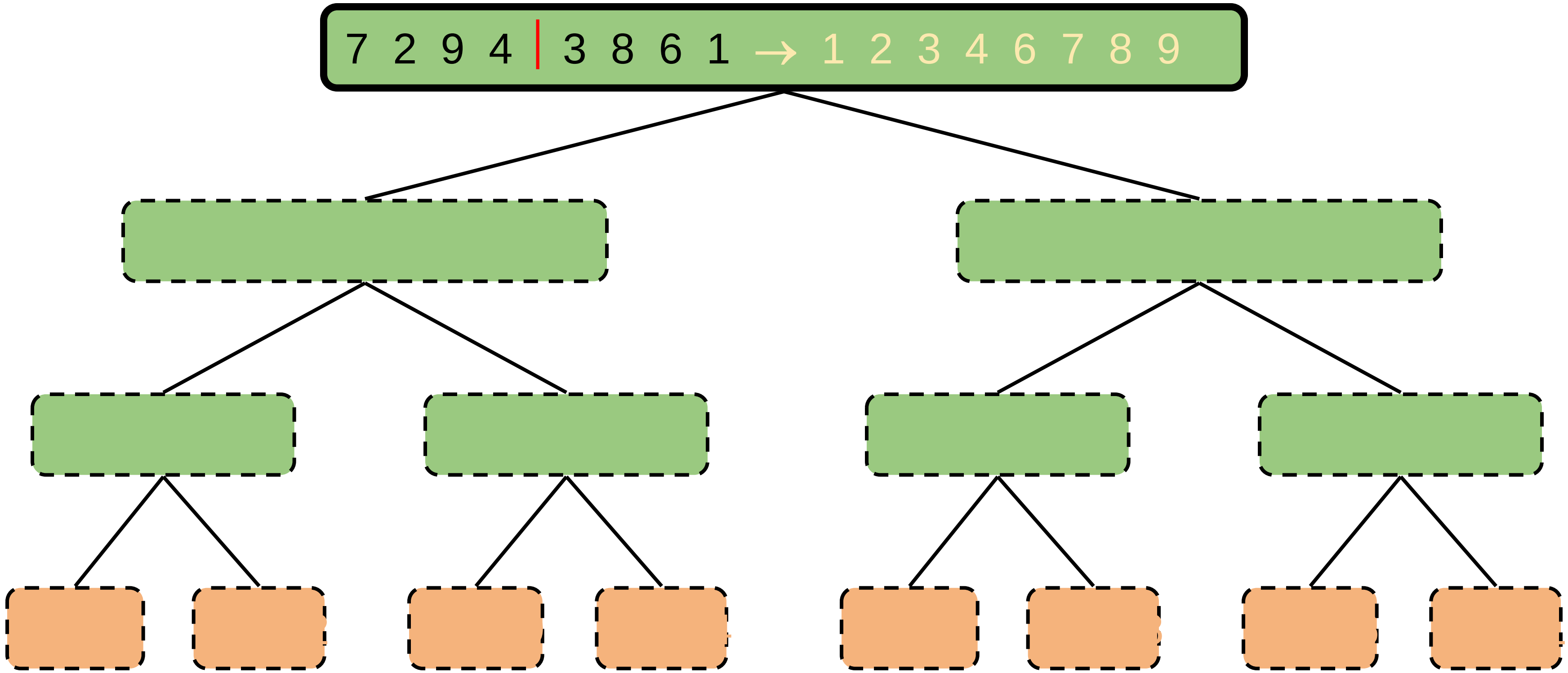
Divide



Merge

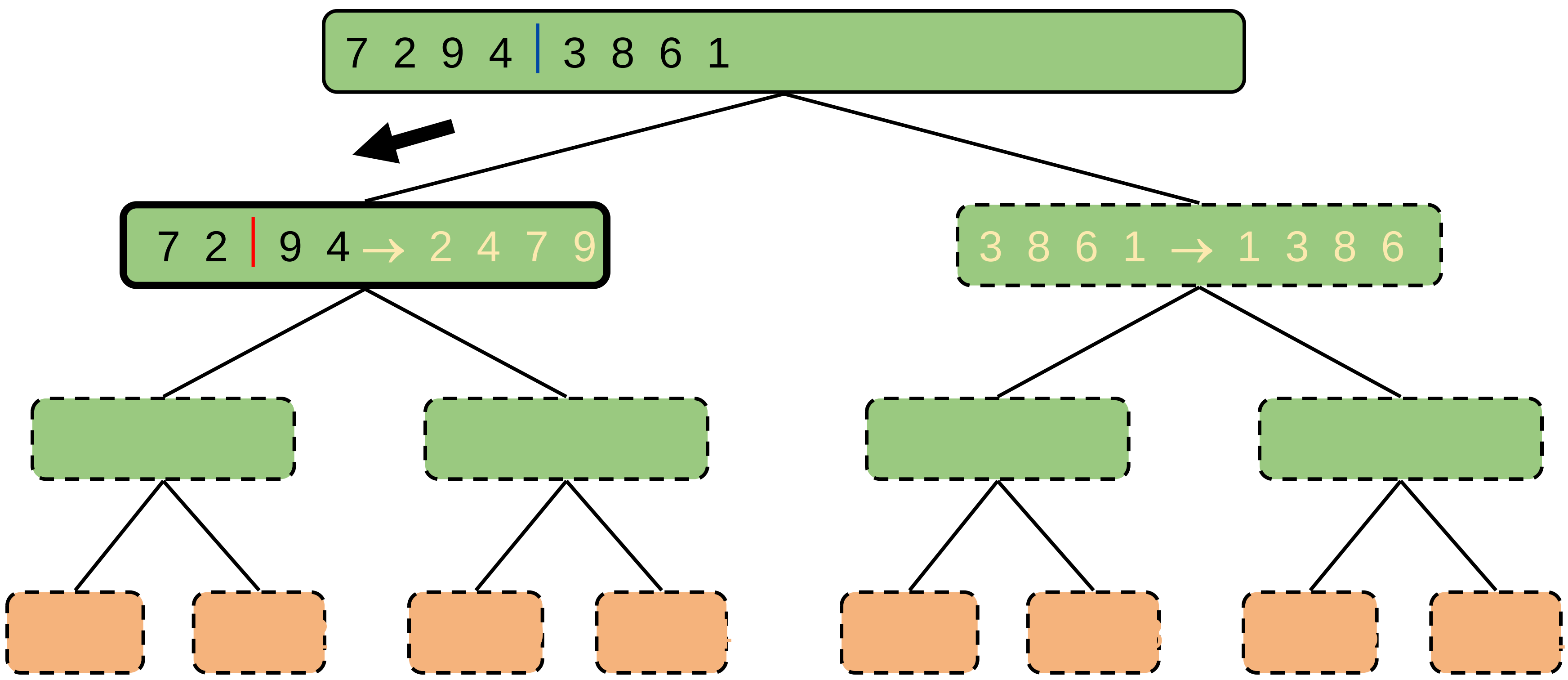
Execution Example

– Partition



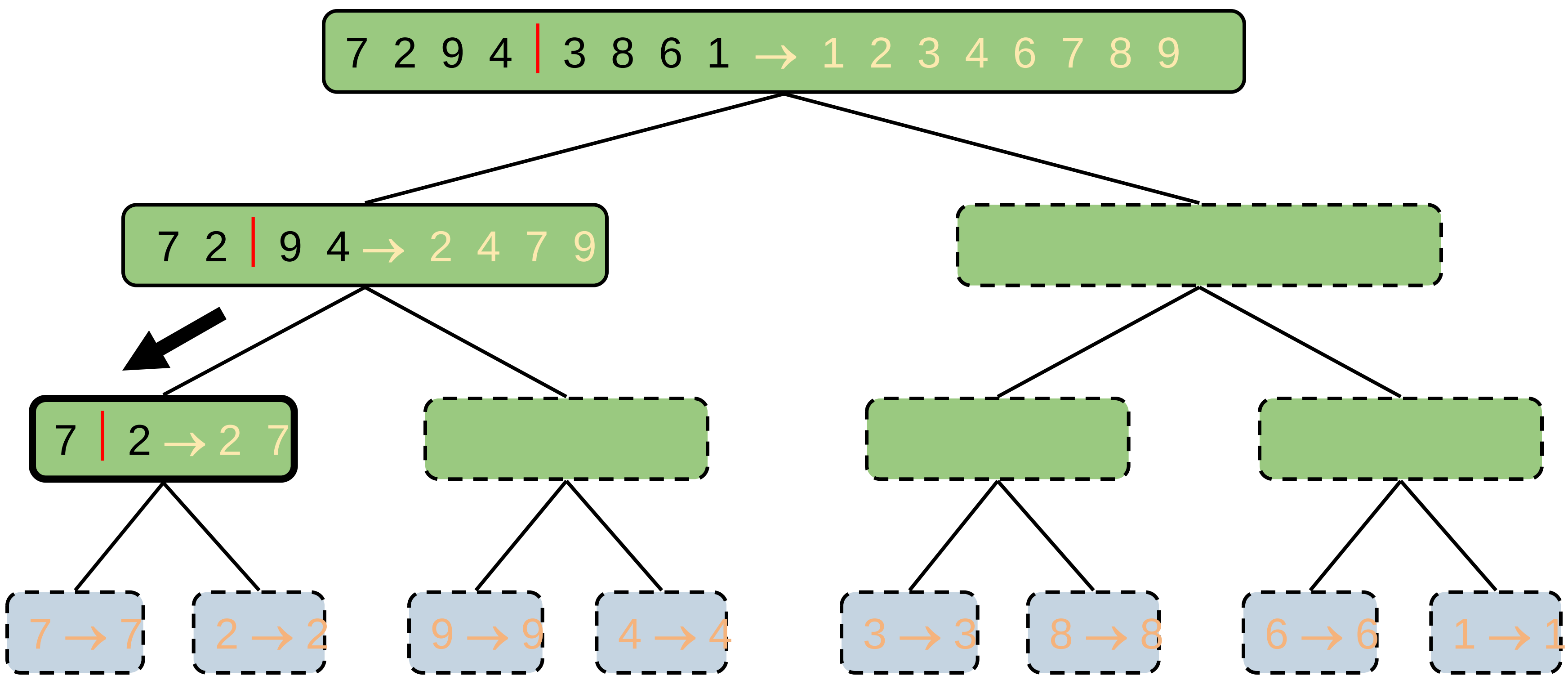
Execution Example (cont.)

– Recursive call, partition



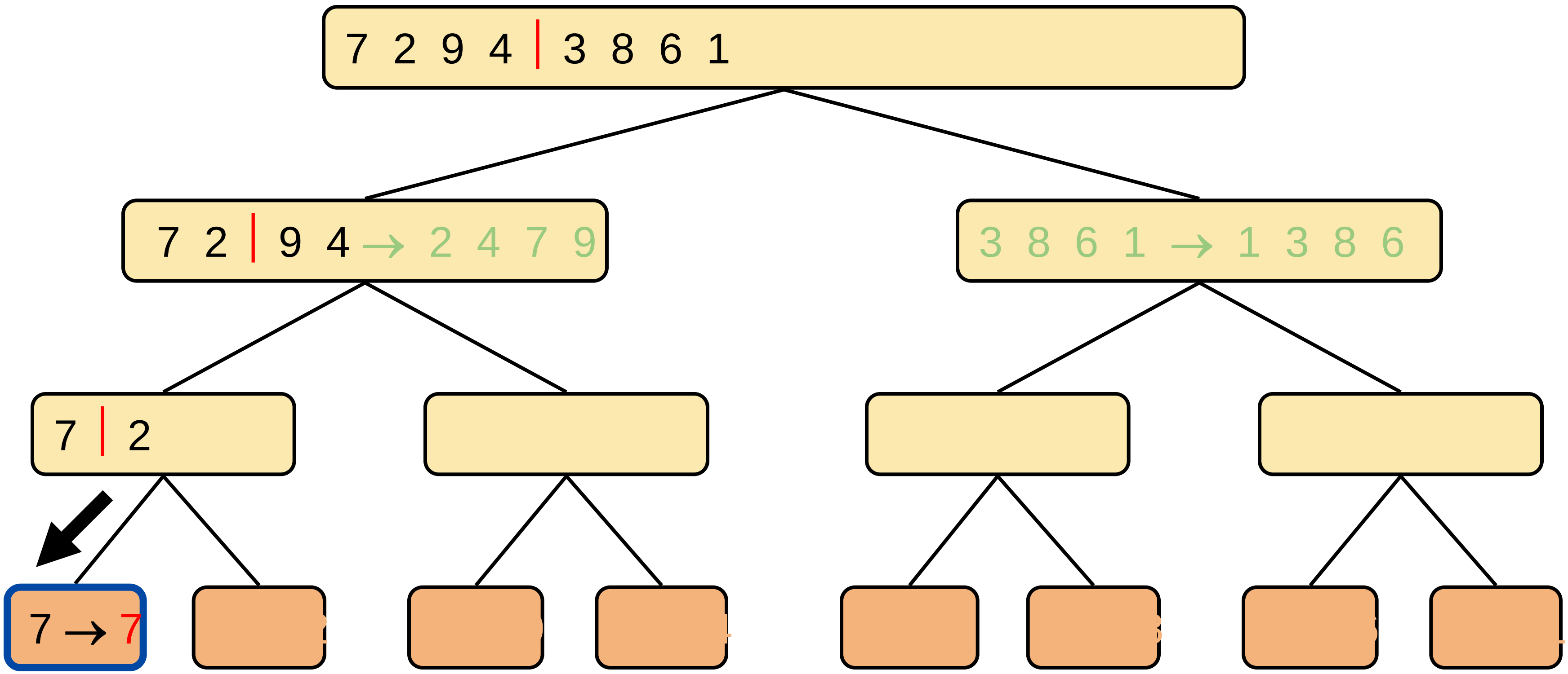
Execution Example (cont.)

– Recursive call, partition



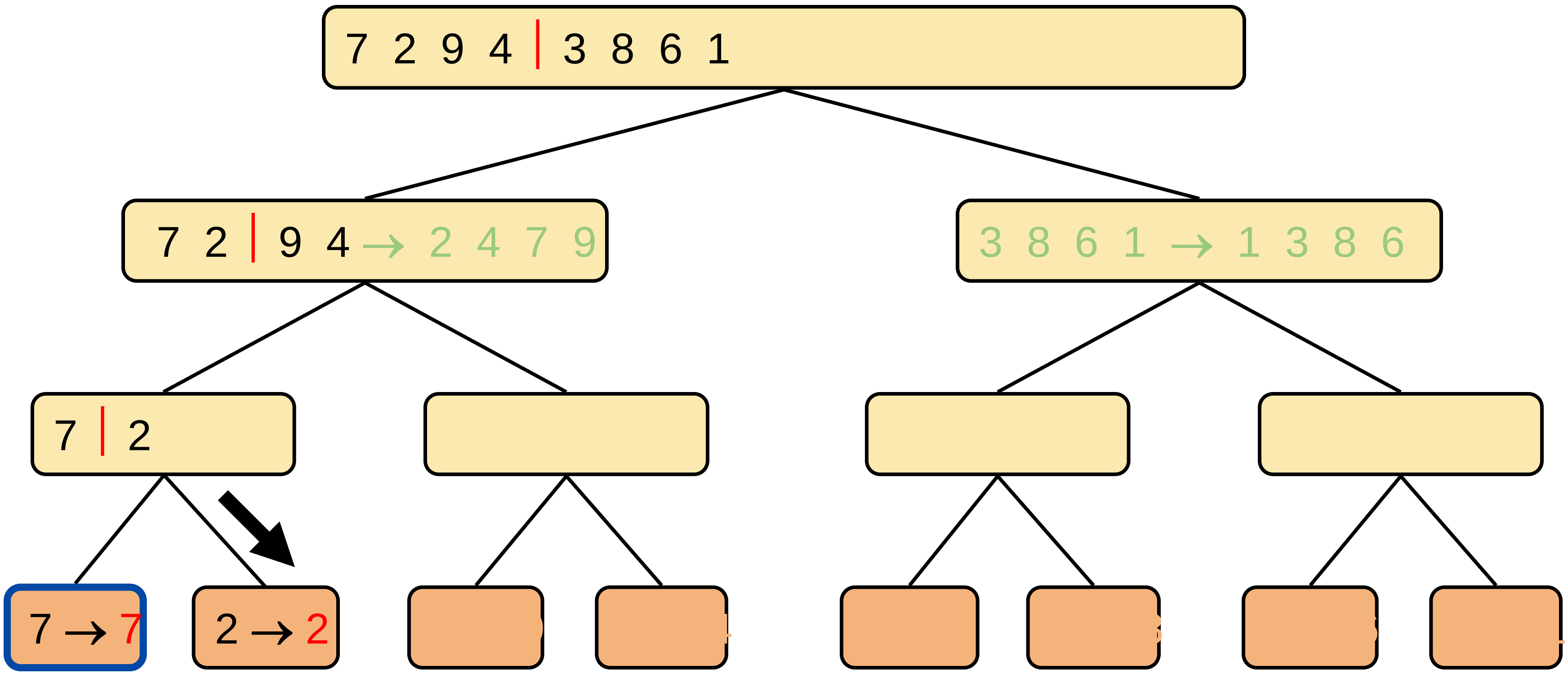
Execution Example (cont.)

– Recursive call, base case



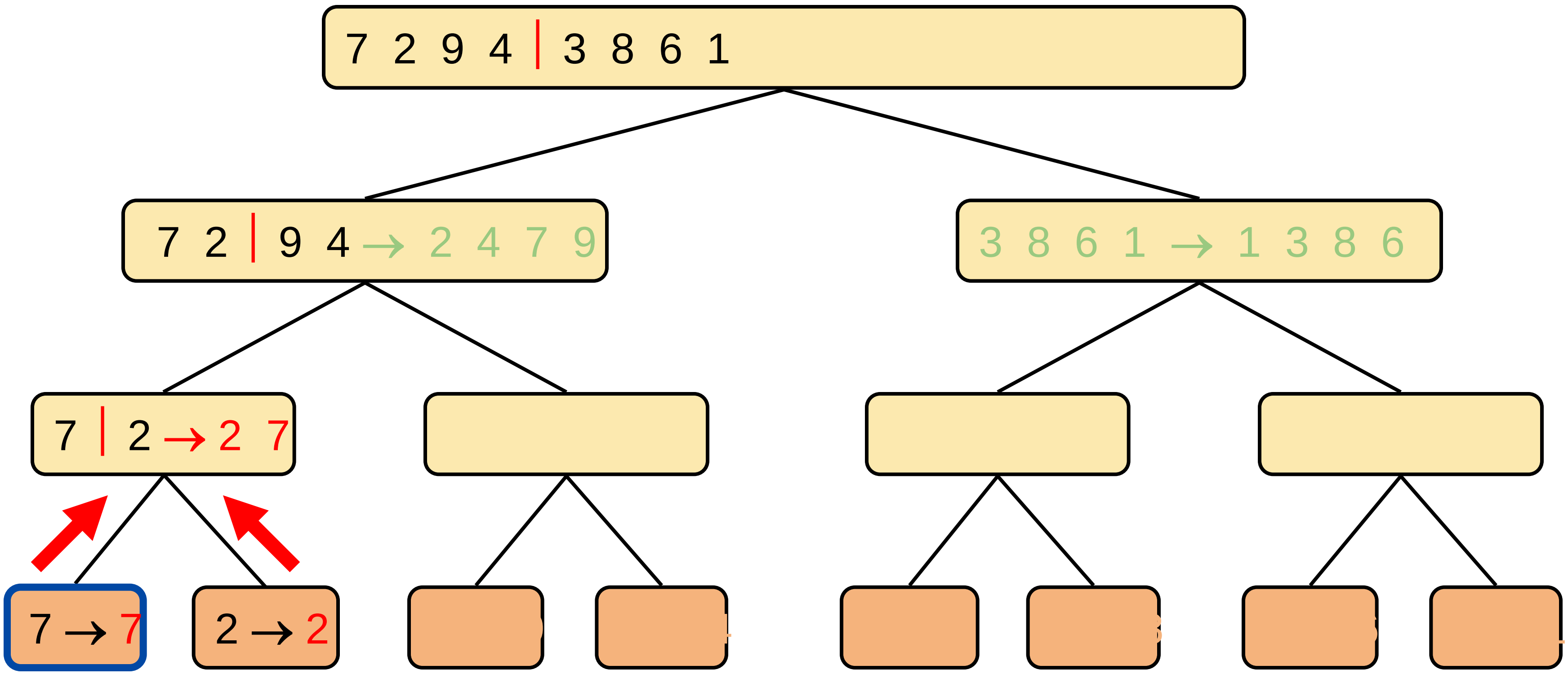
Execution Example (cont.)

– Recursive call, base case



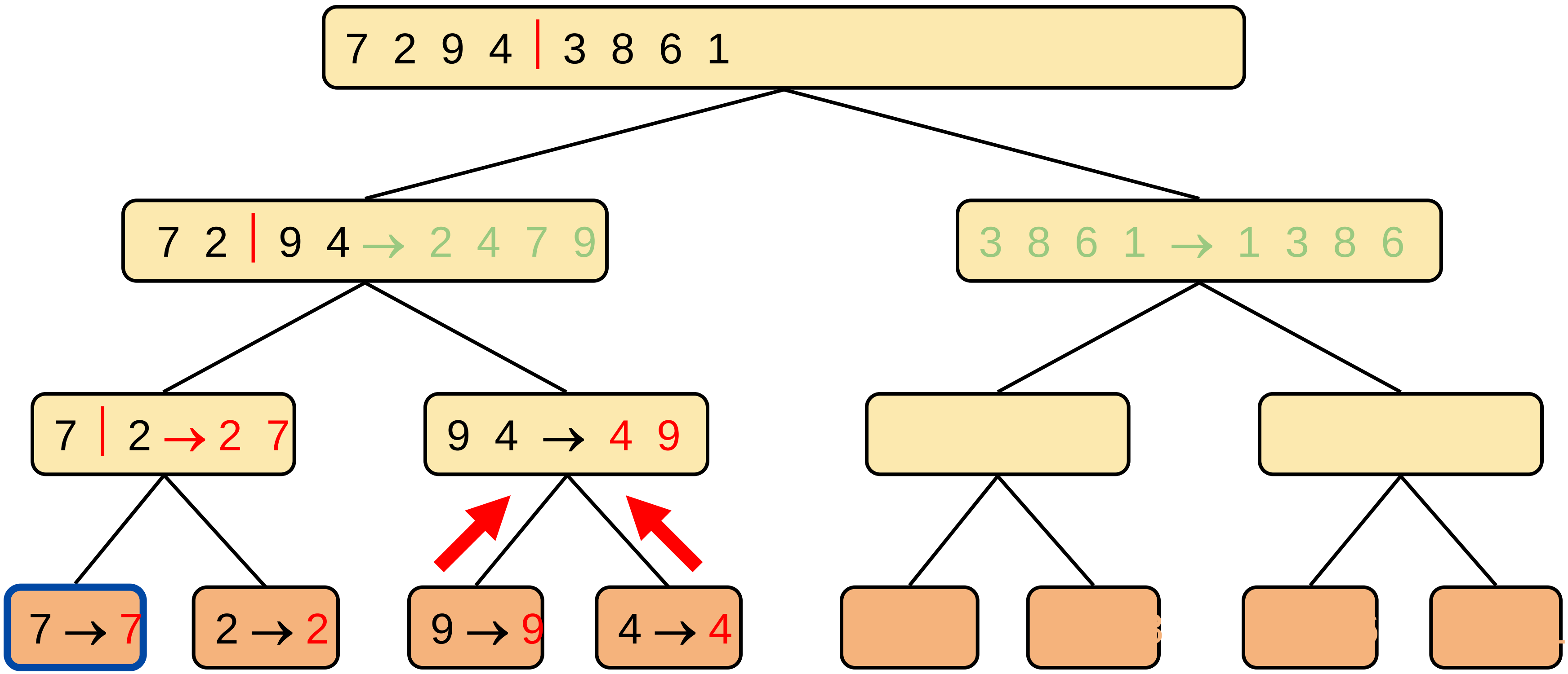
Execution Example (cont.)

– Merge



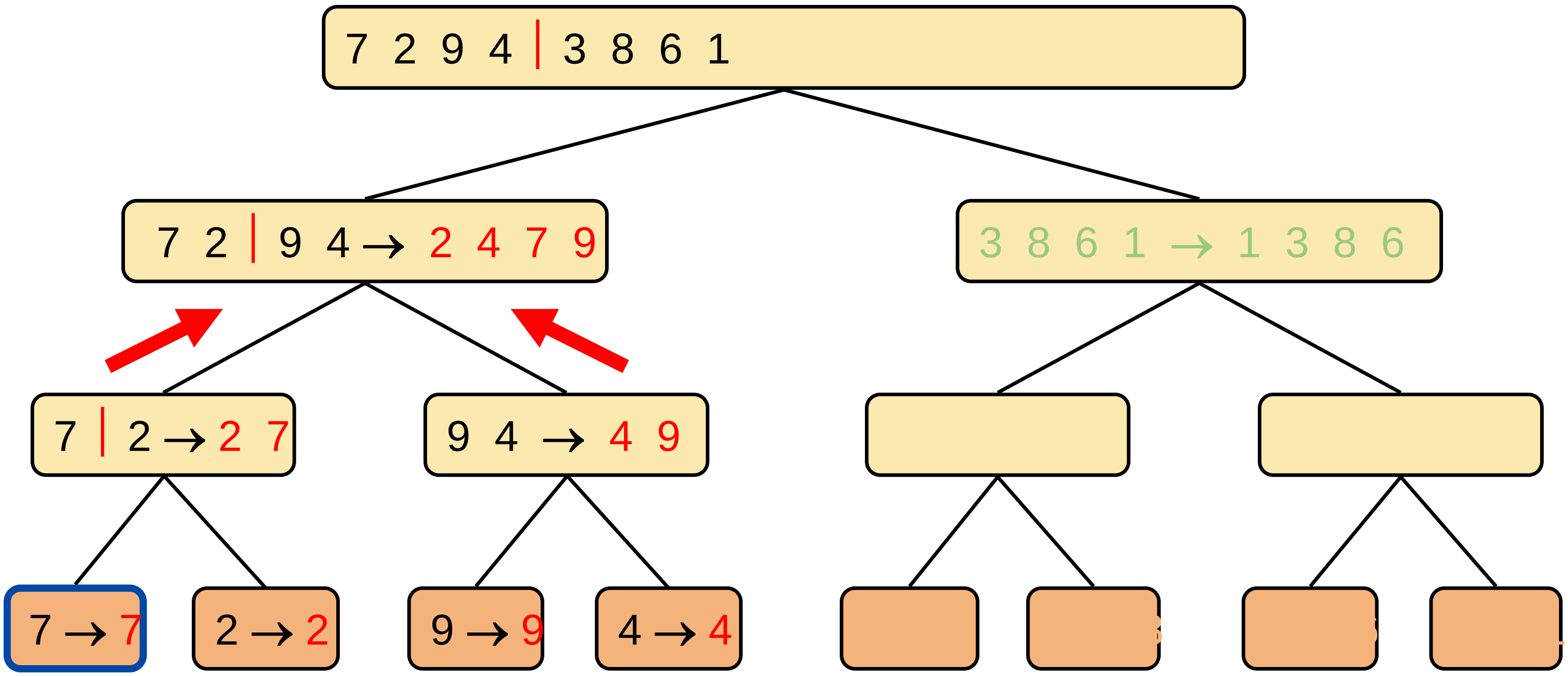
Execution Example (cont.)

– Recursive call, …, base case, merge



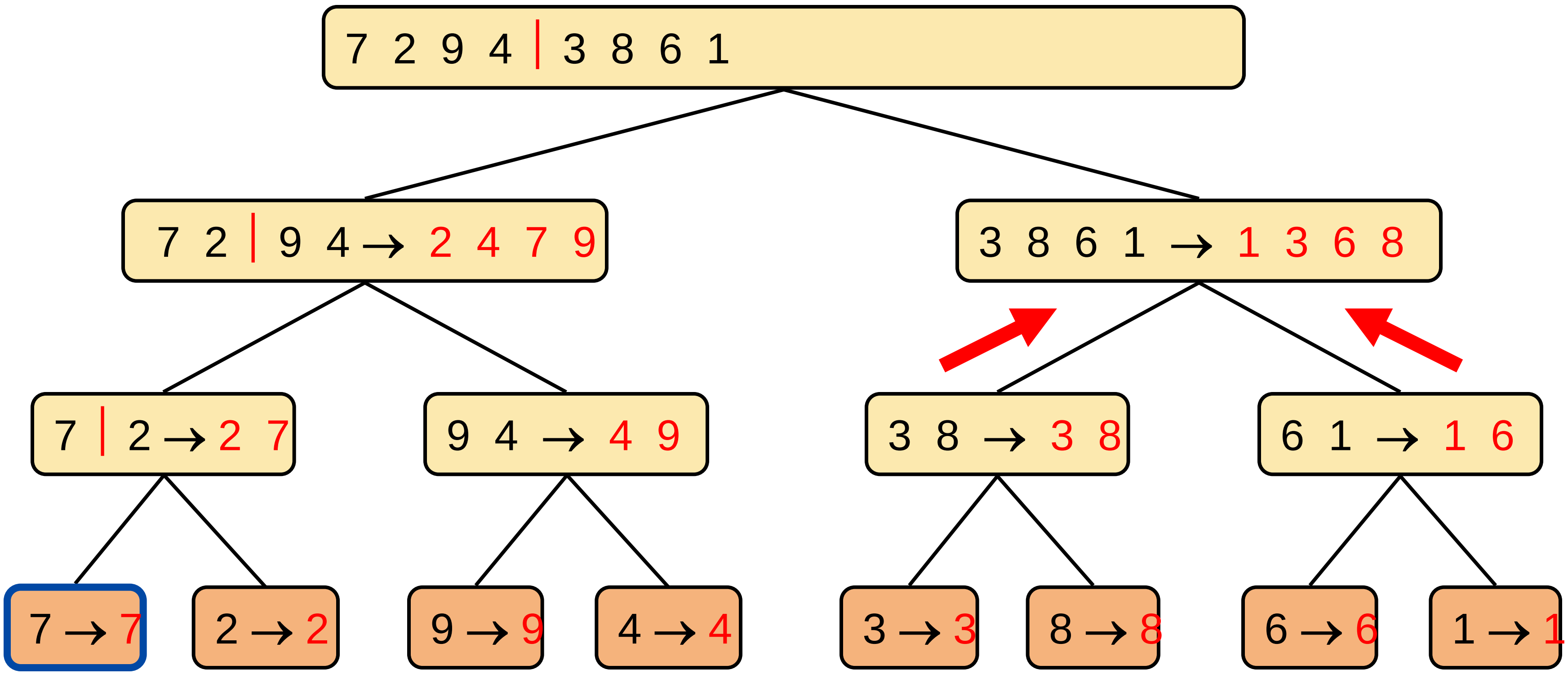
Execution Example (cont.)

– Merge



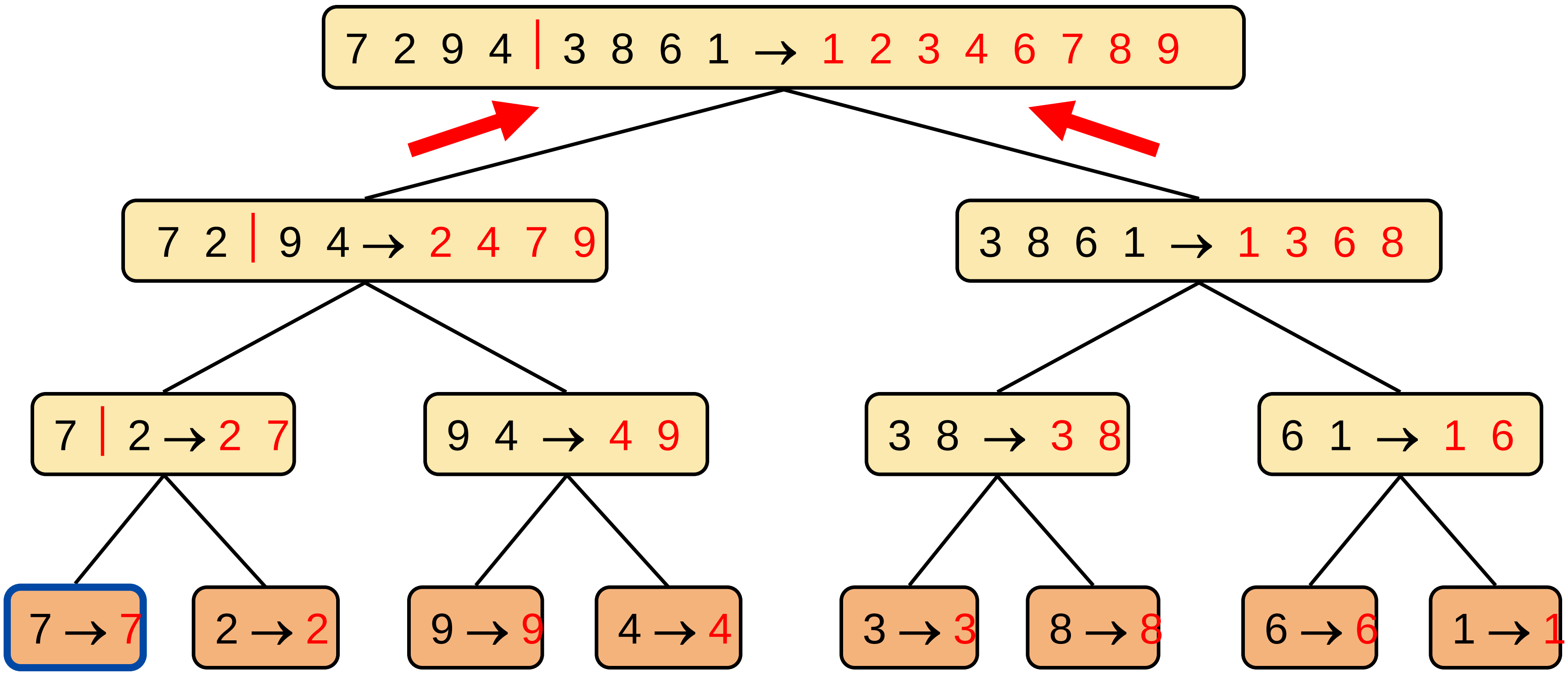
Execution Example (cont.)

– Recursive call, …, merge, merge



Execution Example (cont.)

– Recursive call, …, merge, merge



Recall: Recurrence Analysis of Mergesort (from week 8)

MergeSort(list):

If the list has one or less elements, return the list	O(1)
Otherwise, **Divide** the list into two halves A and B	**O(n)**
List sortedA = MergeSort(A)	T(n/2)
List sortedB = MergeSort(B)	T(n/2)
Merge the sorted lists sortedA and sorted	**O(n)**
Return the merged list	

Let T(n) be the time to run MergeSort on a list of size n

Then **T(n) = 2*T(n/2) + O(n)**

Solve recurrence for worst-case runtime of MergeSort

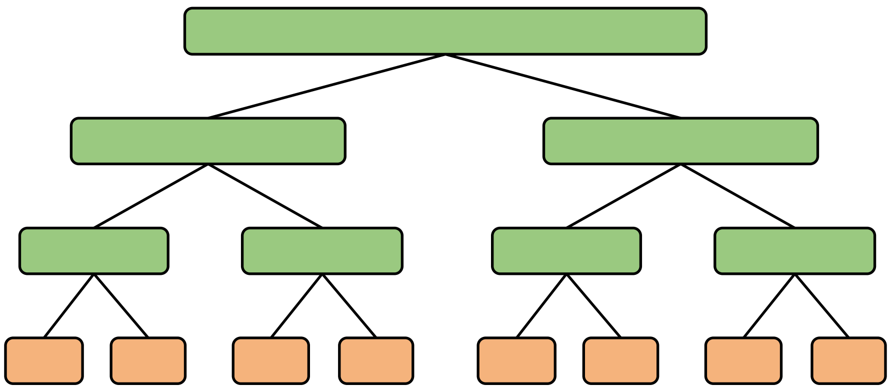
– We saw: **T(n) = 2*T(n/2) + O(n)**

– From week 8, we know this has solution **T(n) = O(n log n)**

Direct Analysis of Merge-Sort

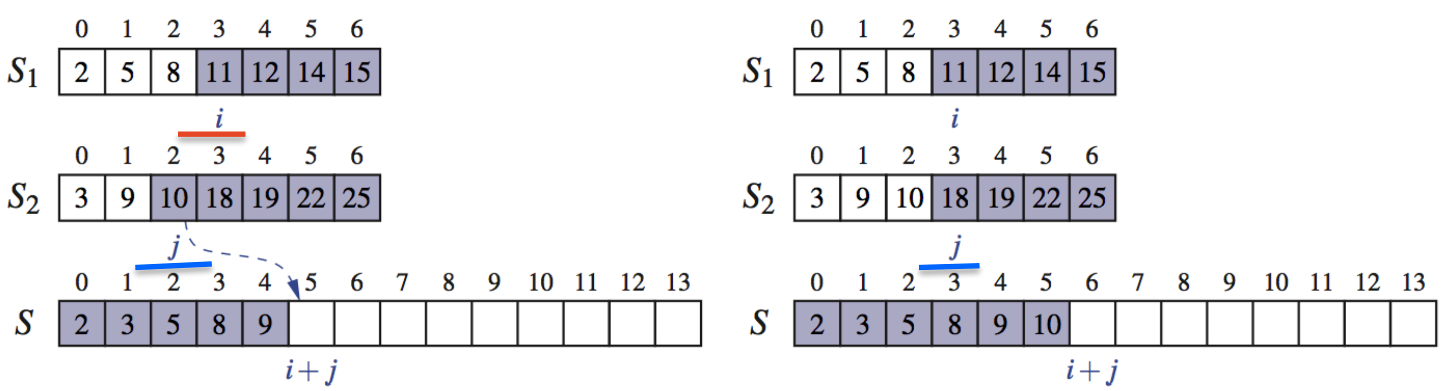
- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time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

depth	#seqs	size	Time per level
0	1	n	$O(n)$
1	2	$n/2$	$O(n)$
i	2^i	$n/2^i$	$O(n)$
…	…	…	$O(n)$



Java Merge Implementation (using arrays)

```
/** Merge contents of arrays S1 and S2 into properly sized array S. */
public static <K> void merge(K[ ] S1, K[ ] S2, K[ ] S, Comparator<K> comp) {
  int i = 0, j = 0;
  while (i + j < S.length) {
    if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))
      S[i+j] = S1[i++];                    // copy ith element of S1 and increment i
    else
      S[i+j] = S2[j++];                    // copy jth element of S2 and increment j
  }
}
```



Java Merge-Sort Implementation

```
/** Merge-sort contents of array S. */
public static <K> void mergeSort(K[ ] S, Comparator<K> comp) {
  int n = S.length;
  if (n < 2) return;                              // array is trivially sorted
  // divide
  int mid = n/2;
  K[ ] S1 = Arrays.copyOfRange(S, 0, mid);        // copy of first half
  K[ ] S2 = Arrays.copyOfRange(S, mid, n);        // copy of second half
  // conquer (with recursion)
  mergeSort(S1, comp);                            // sort copy of first half
  mergeSort(S2, comp);                            // sort copy of second half
  // merge results
  merge(S1, S2, S, comp);               // merge sorted halves back into original
}
```

In-place mergesort

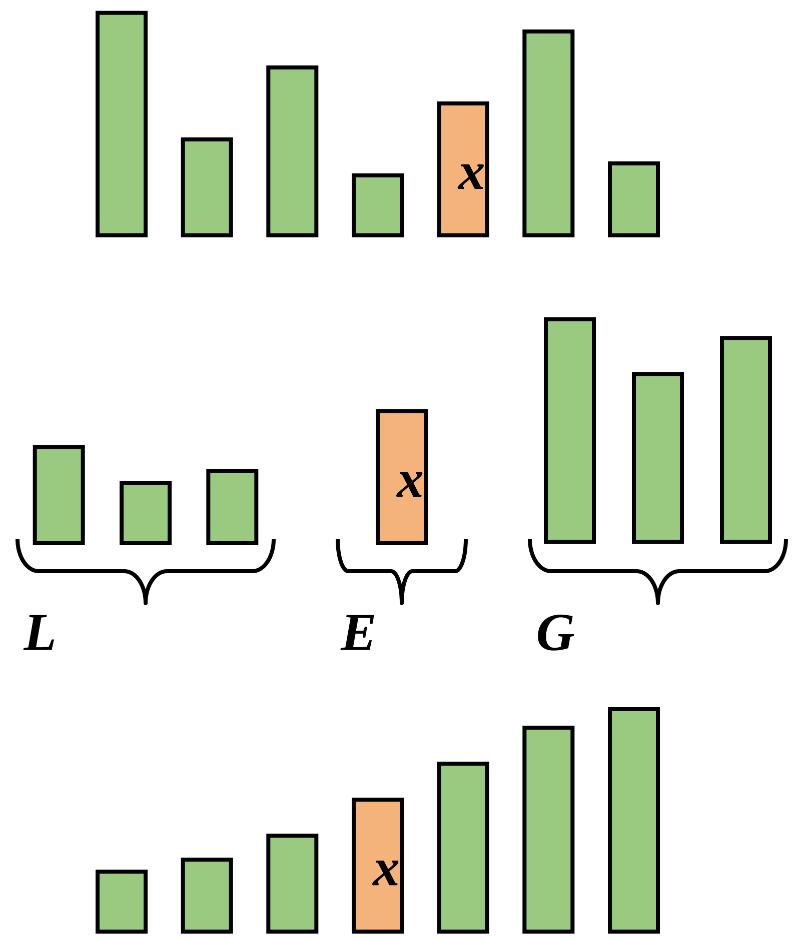
– The algorithm we just saw is not in-place
– It is easy to partition S in-place
– It is not easy to merge two sorted lists in-place, but there are complicated ways to get an algorithm that does so, allowing one to mergesort an array in-place

Outline

- Sorting algorithms and their costs
  - Review of pq-sorting algorithms: insertion, selection sort, heap-sort
  - In-place sorting
  - Bubble sort
  - Merge-sort
  - **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
- Unlike merge-sort, hard work done *before* the recursive calls



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

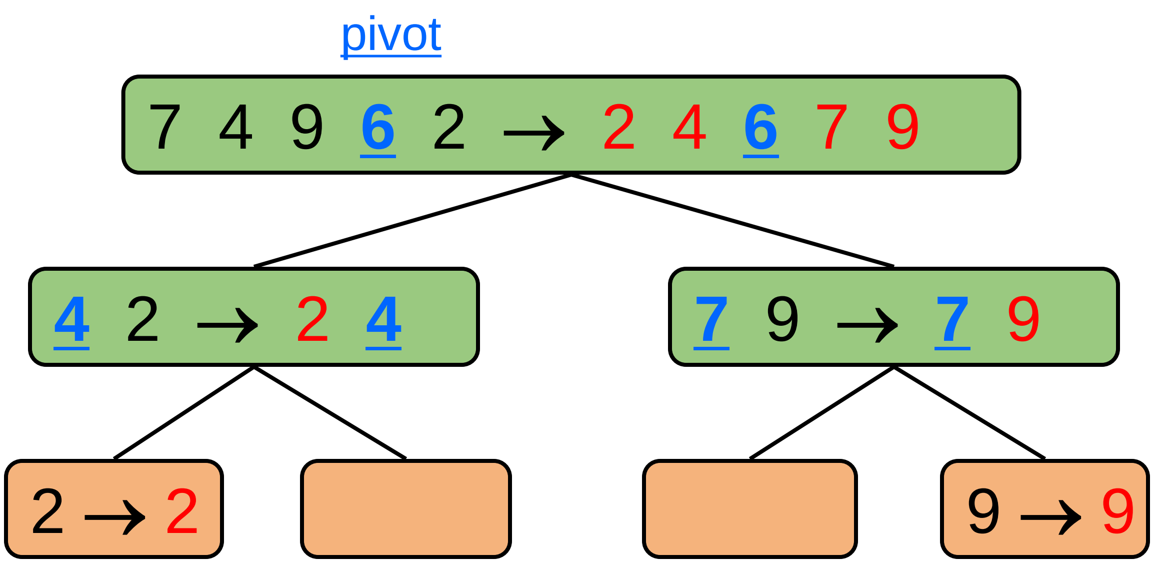
Algorithm *partition*(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the elements of S less than, equal to, or greater than the pivot, resp.
    $L, E, G \leftarrow$ empty sequences
    $x \leftarrow S.remove(p)$
    while $\neg S.isEmpty()$
        $y \leftarrow S.remove(S.first())$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

Java Implementation

```
/** Quick-sort contents of a queue. */
public static <K> void quickSort(Queue<K> S, Comparator<K> comp) {
  int n = S.size();
  if (n < 2) return;                             // queue is trivially sorted
  // divide
  K pivot = S.first();                           // using first as arbitrary pivot
  Queue<K> L = new LinkedQueue<>();
  Queue<K> E = new LinkedQueue<>();
  Queue<K> G = new LinkedQueue<>();
  while (!S.isEmpty()) {                         // divide original into L, E, and G
    K element = S.dequeue();
    int c = comp.compare(element, pivot);
    if (c < 0)                                   // element is less than pivot
      L.enqueue(element);
    else if (c == 0)                             // element is equal to pivot
      E.enqueue(element);
    else                                         // element is greater than pivot
      G.enqueue(element);
  }
  // conquer
  quickSort(L, comp);                            // sort elements less than pivot
  quickSort(G, comp);                            // sort elements greater than pivot
  // concatenate results
  while (!L.isEmpty())
    S.enqueue(L.dequeue());
  while (!E.isEmpty())
    S.enqueue(E.dequeue());
  while (!G.isEmpty())
    S.enqueue(G.dequeue());
}
```

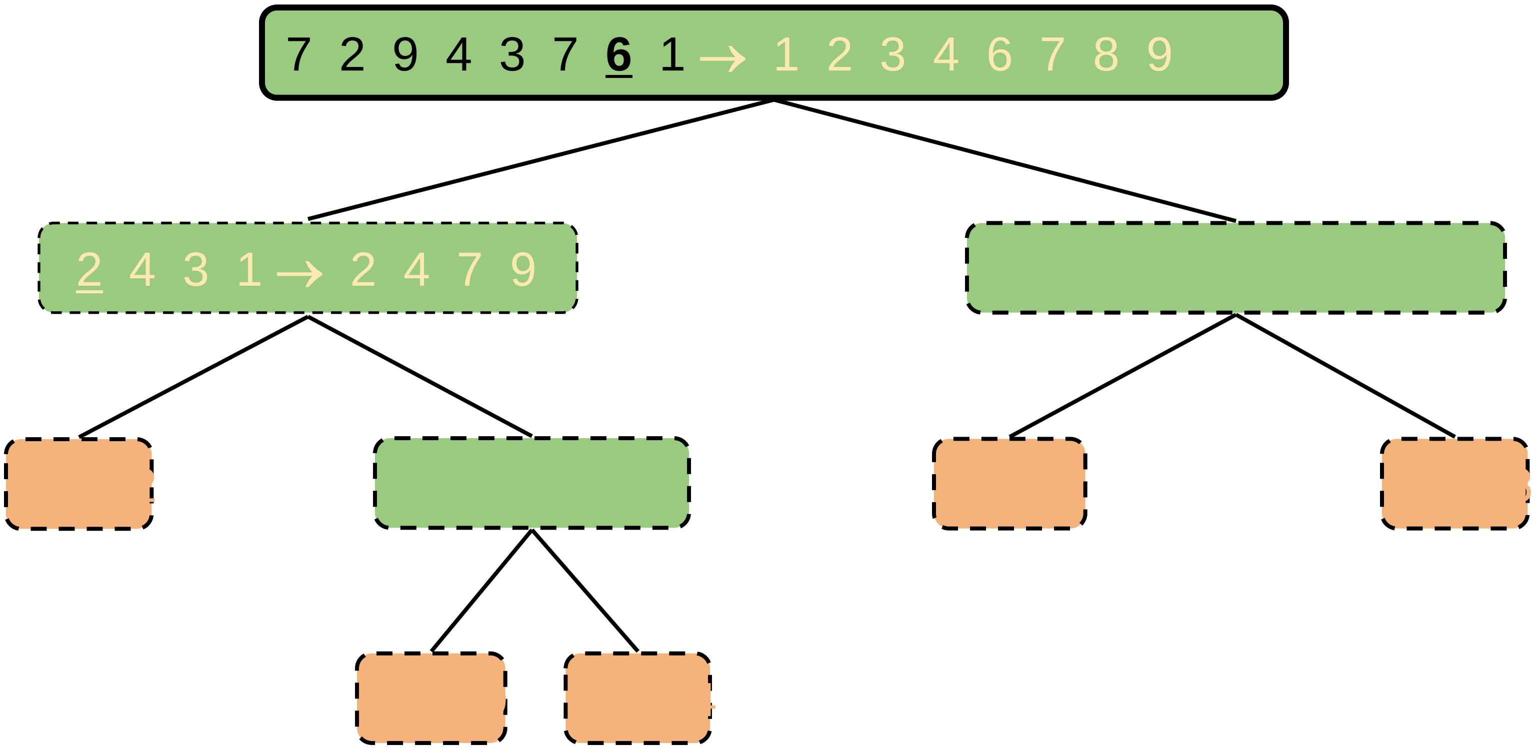
Quick-Sort Tree

- 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



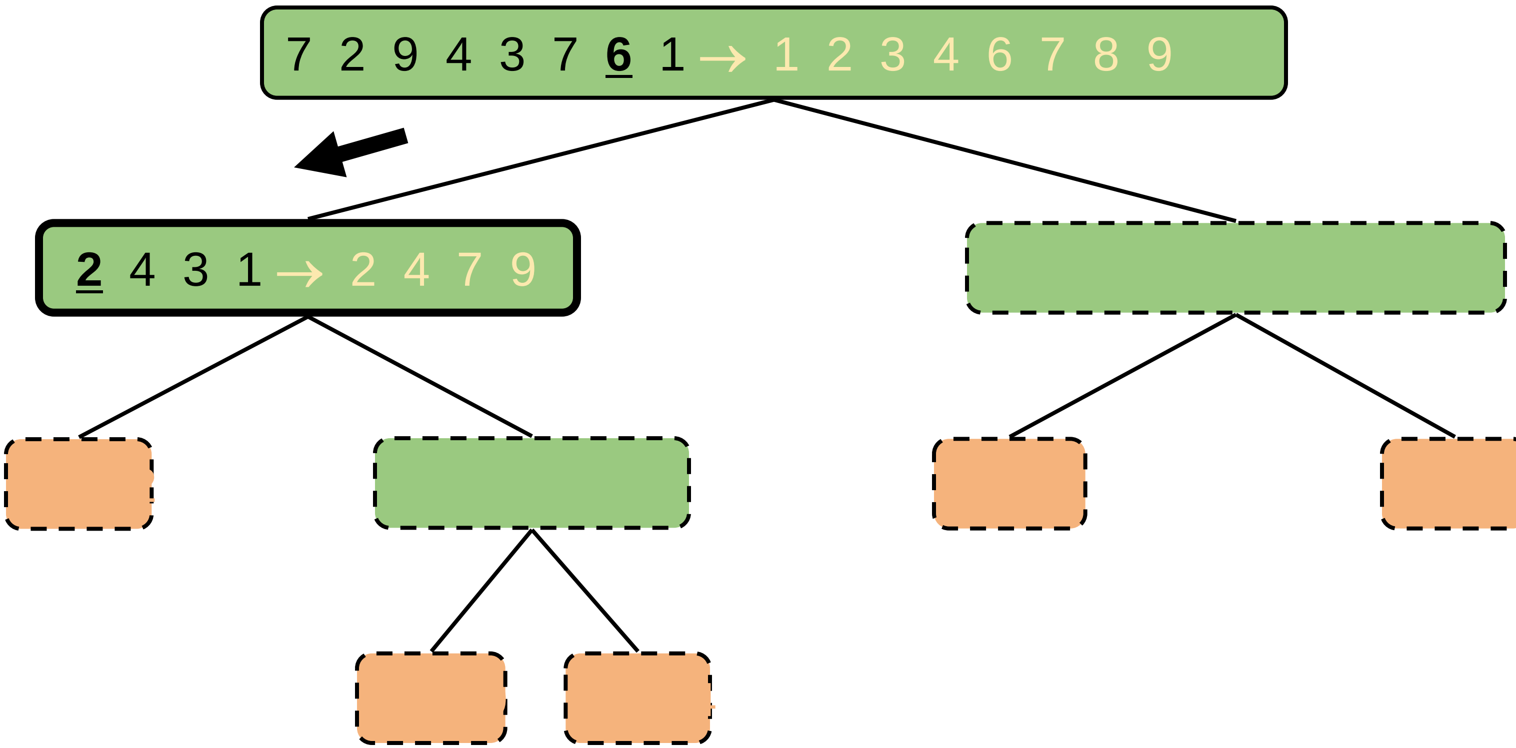
Execution Example

– Pivot selection



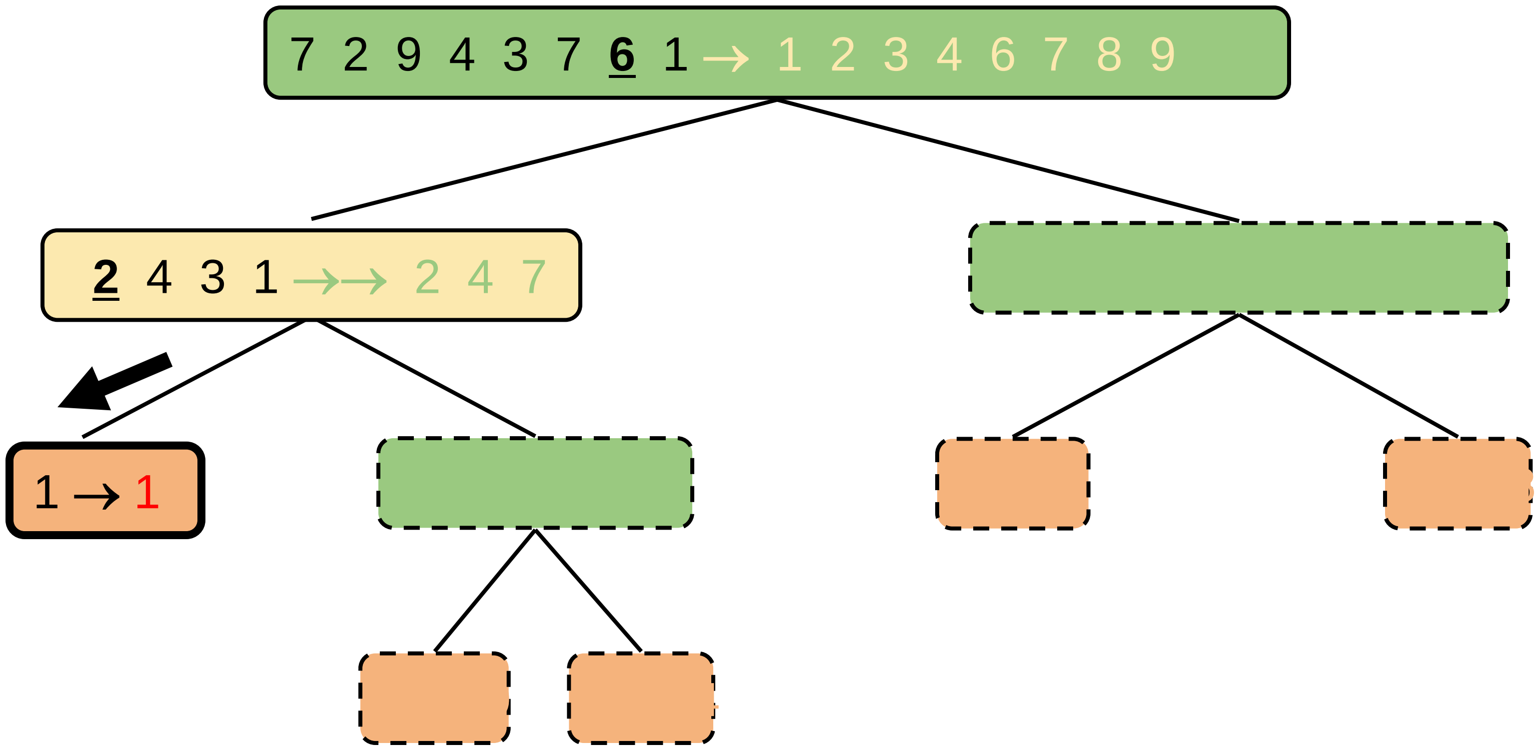
Execution Example (cont.)

– Partition, recursive call, pivot selection



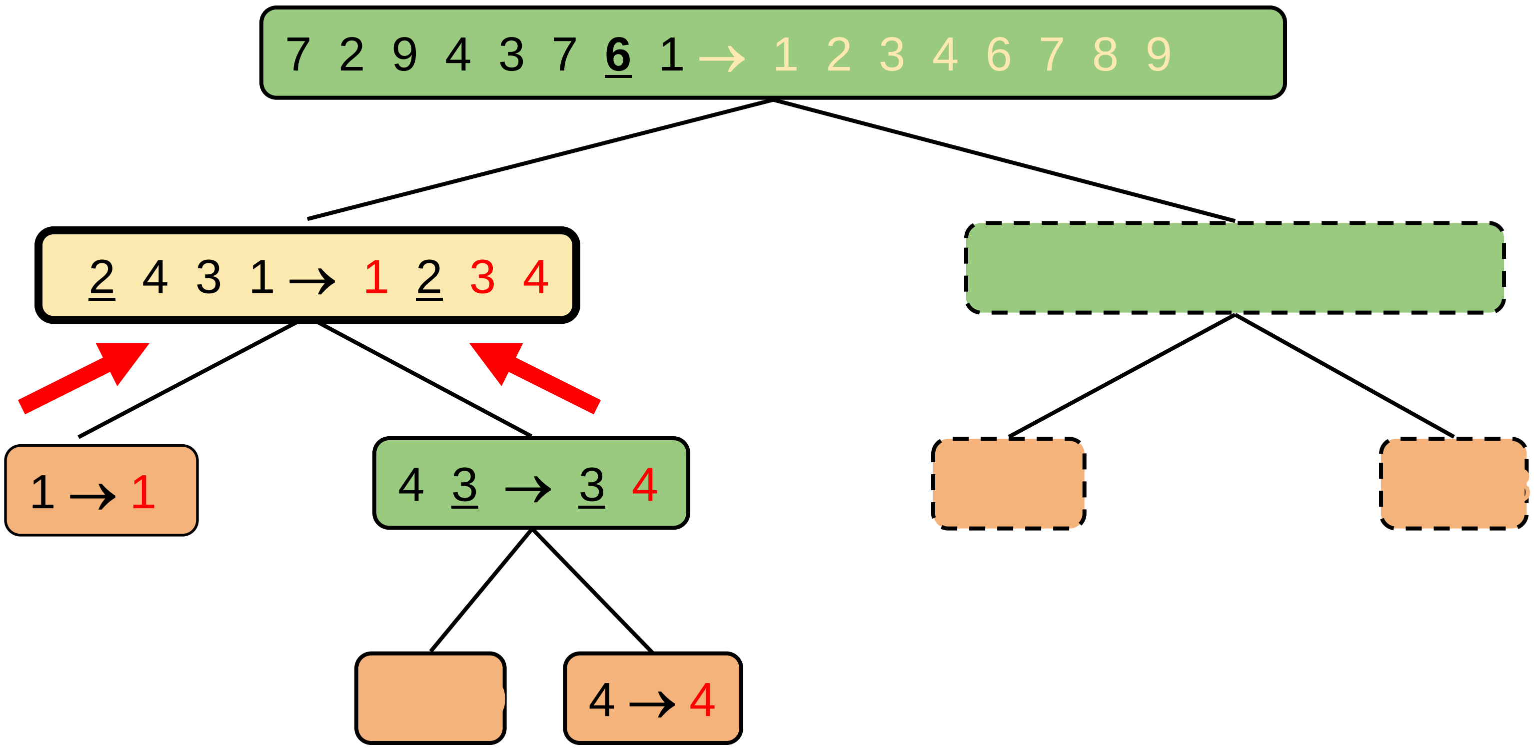
Execution Example (cont.)

– Partition, recursive call, base case



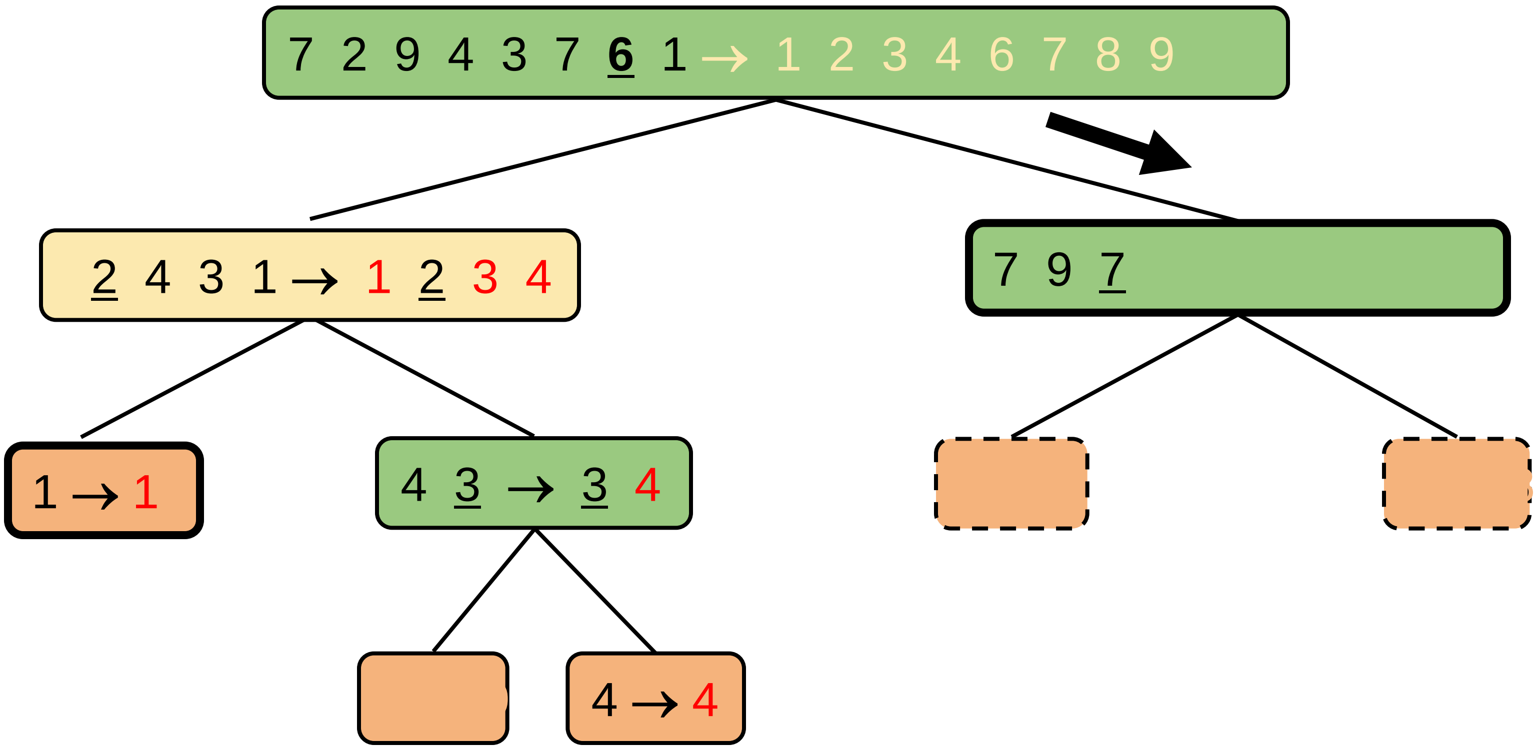
Execution Example (cont.)

– Recursive call, ..., base case, join



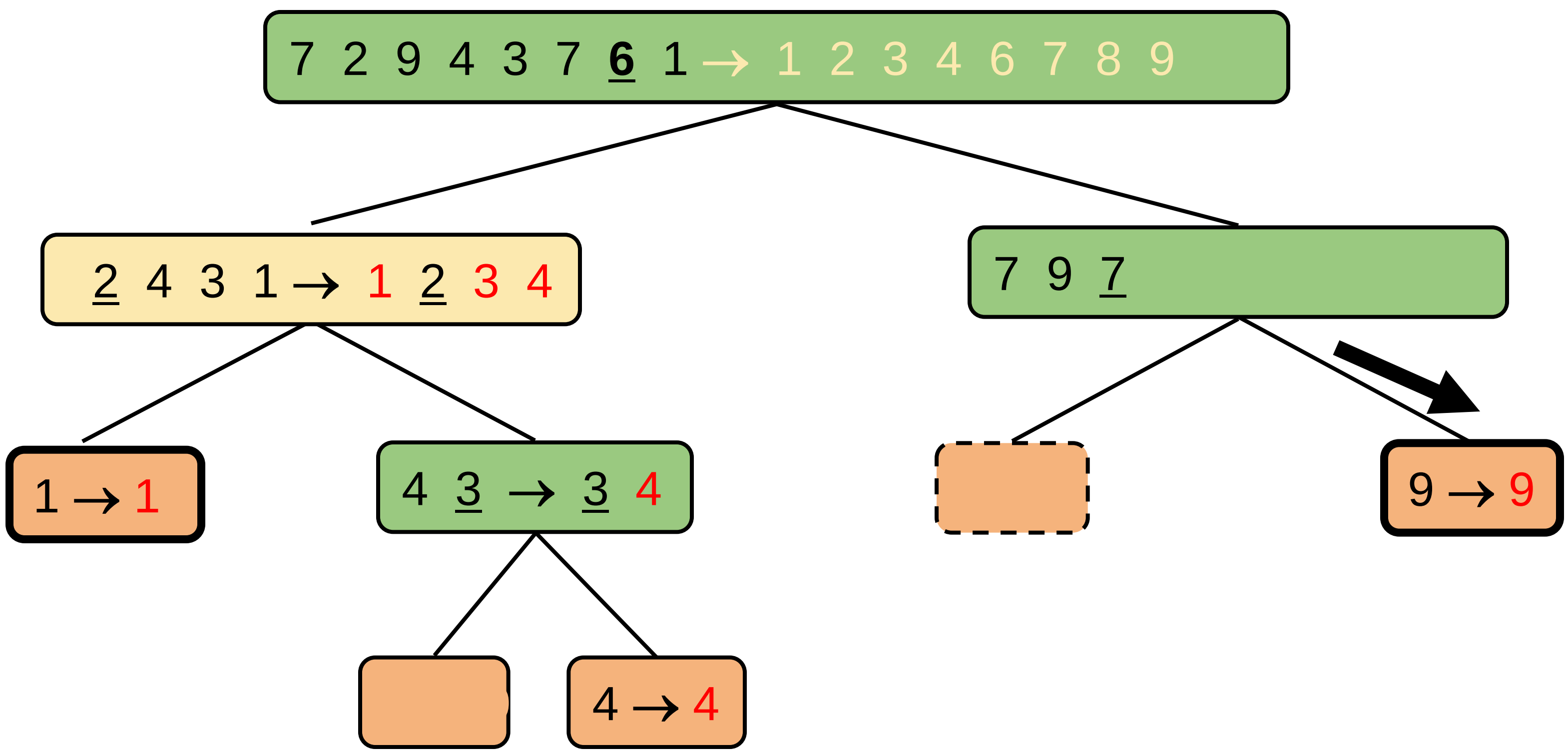
Execution Example (cont.)

– Recursive call, ..., base case, join



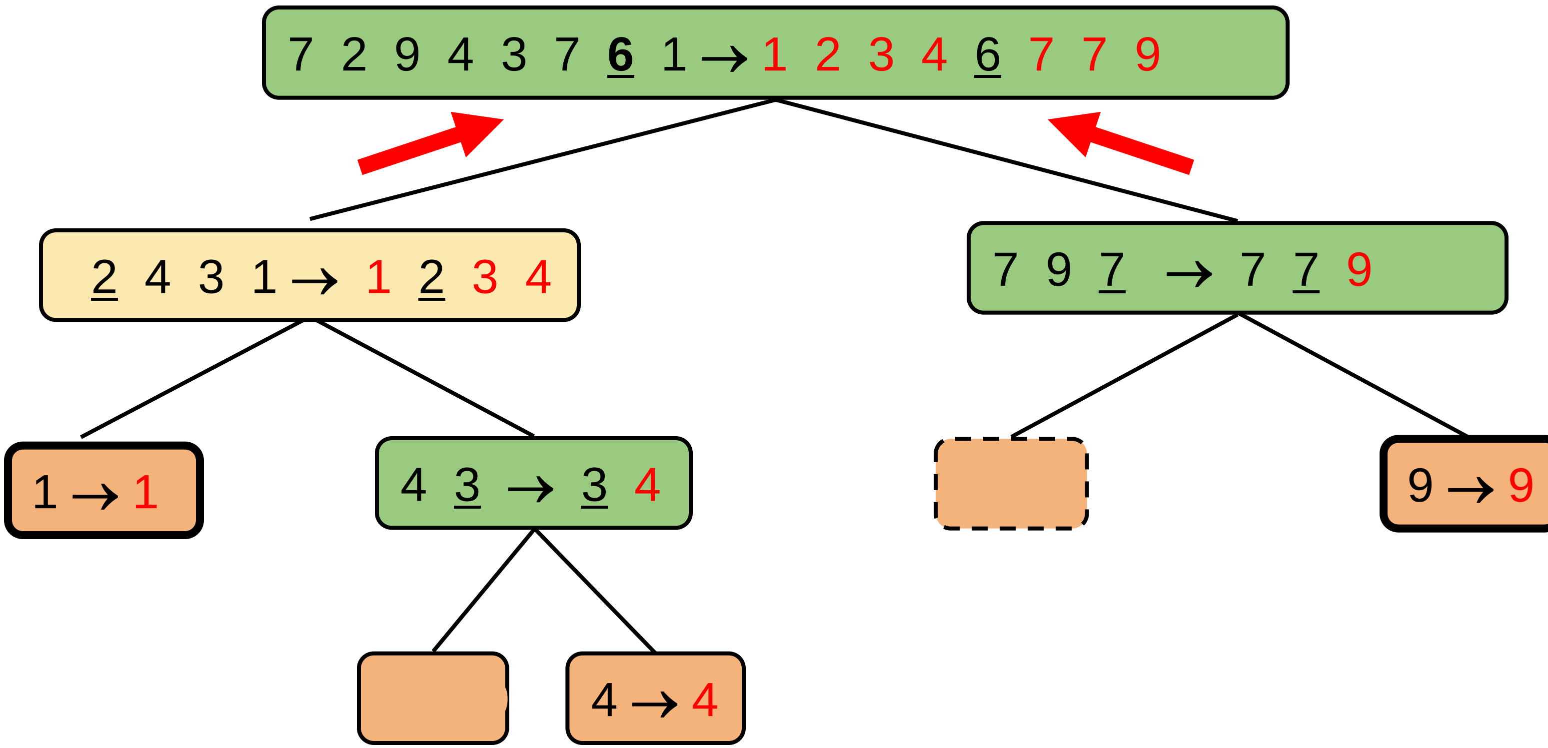
Execution Example (cont.)

– Partition, ..., recursive call, base case

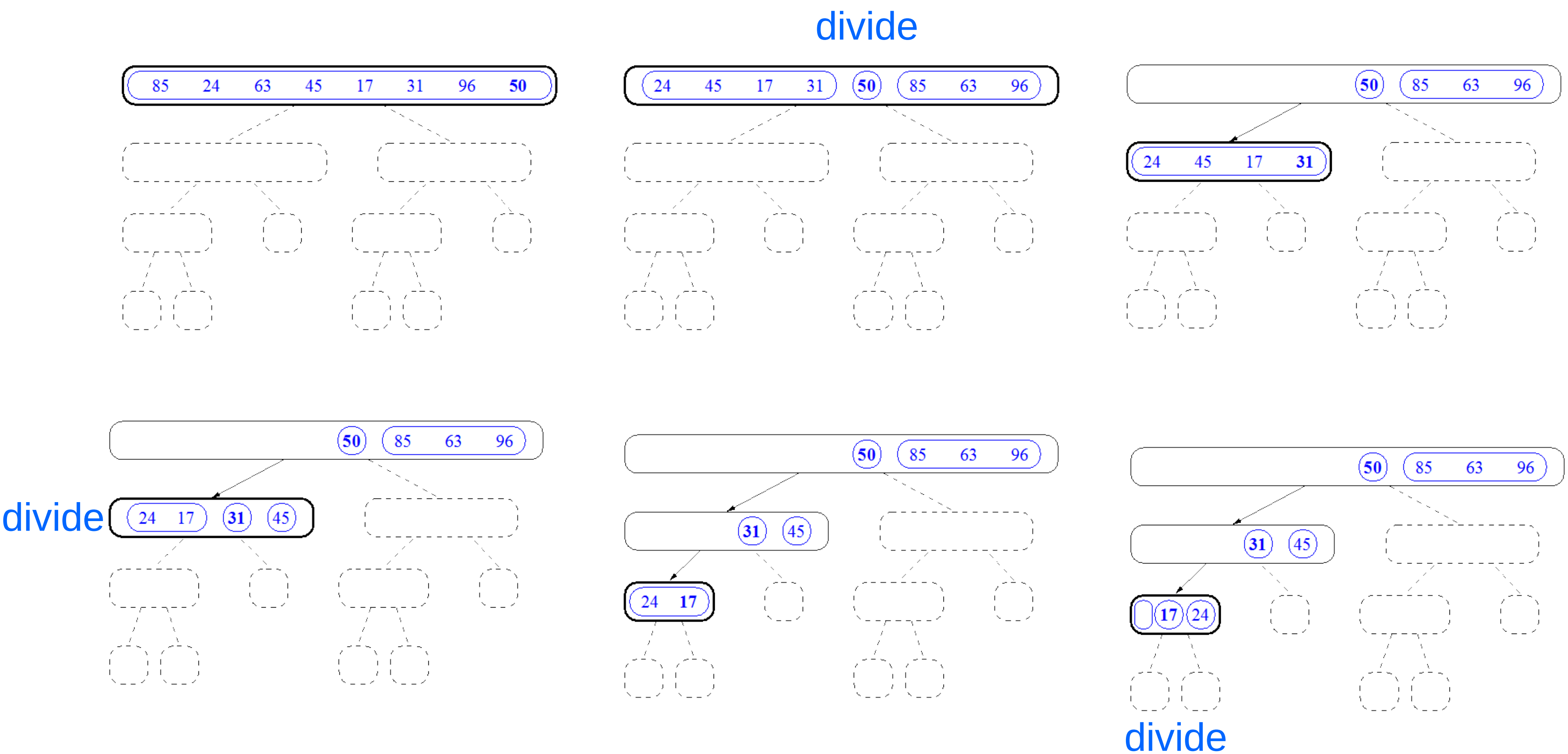


Execution Example (cont.)

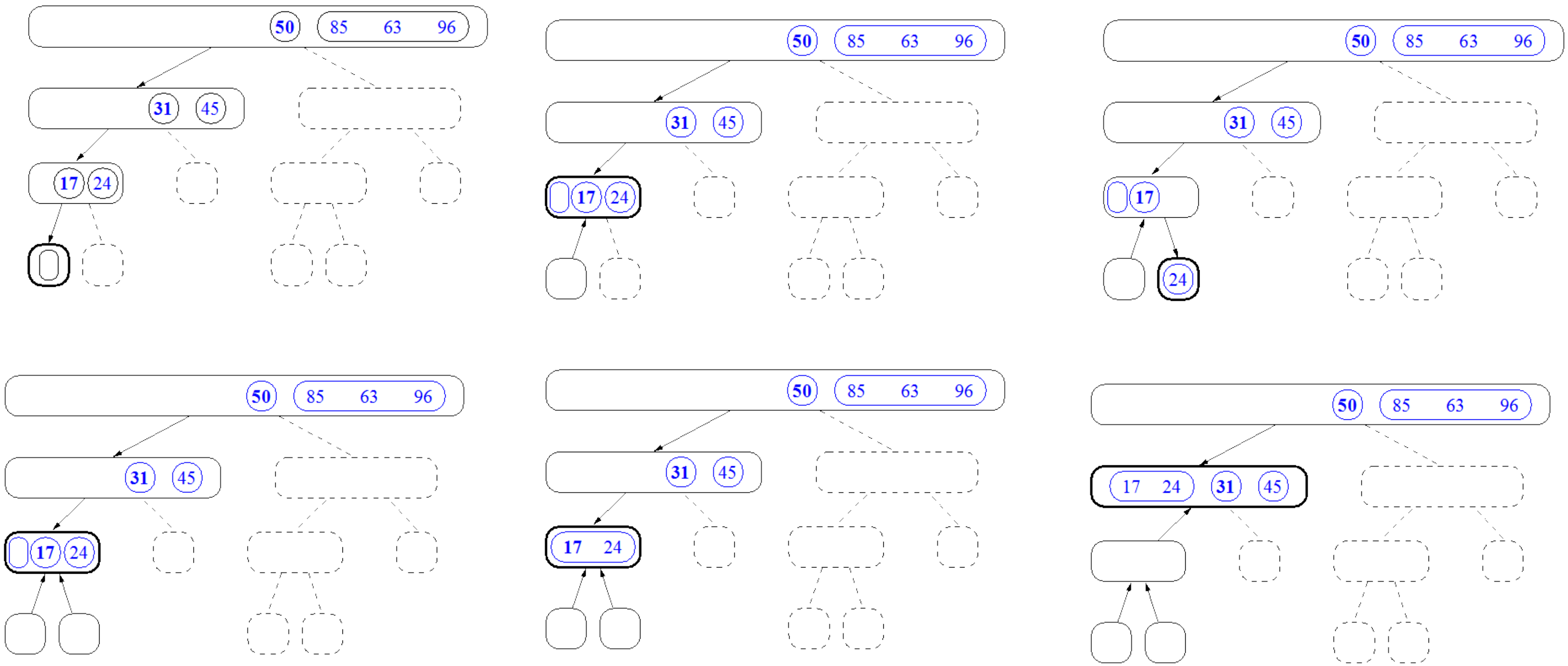
– Join, join



Example 2 (pivot is the last element)

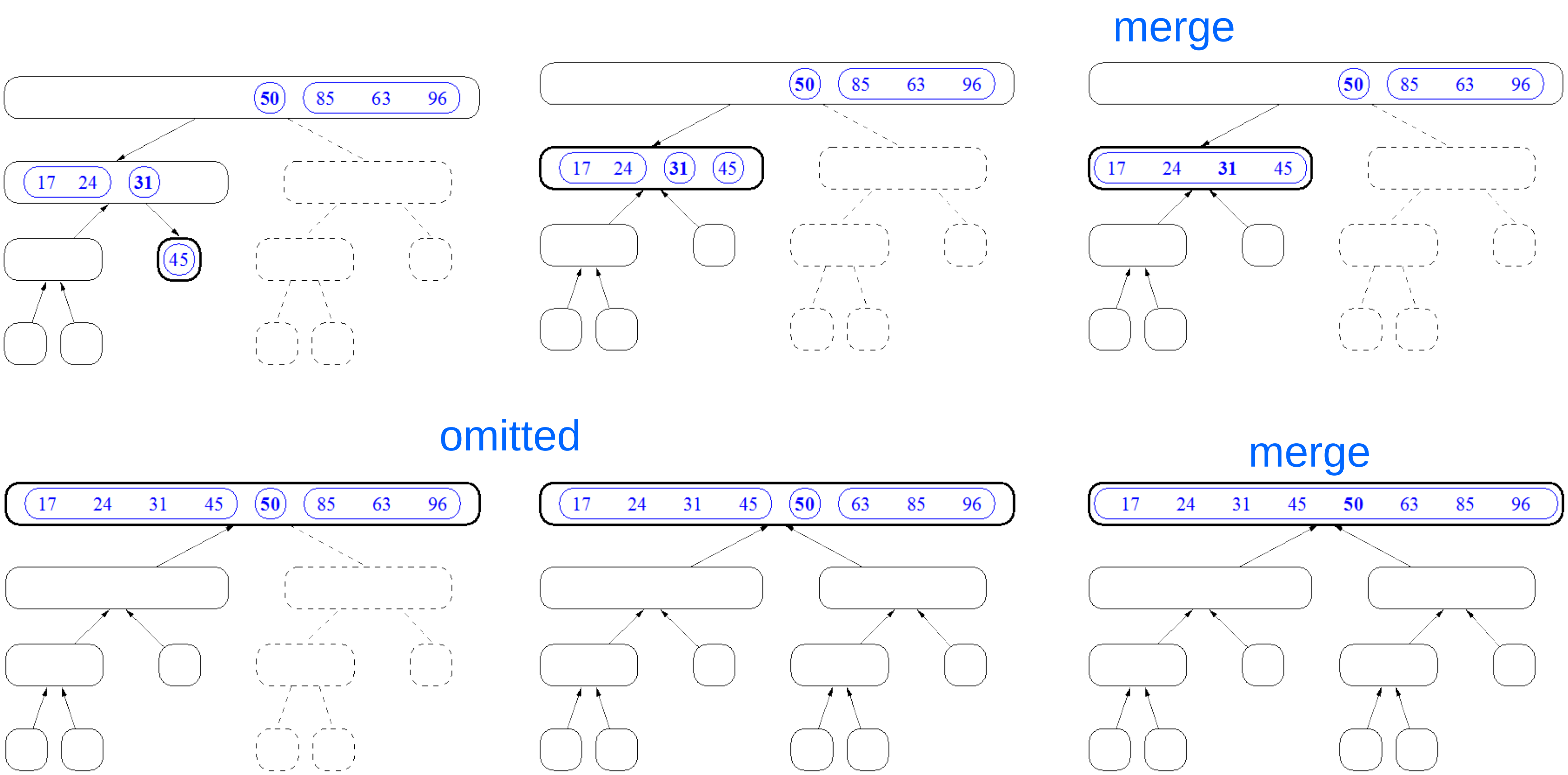


Example 2 (pivot is the last element) cont..



merge

Example 2 (pivot is the last element) cont..



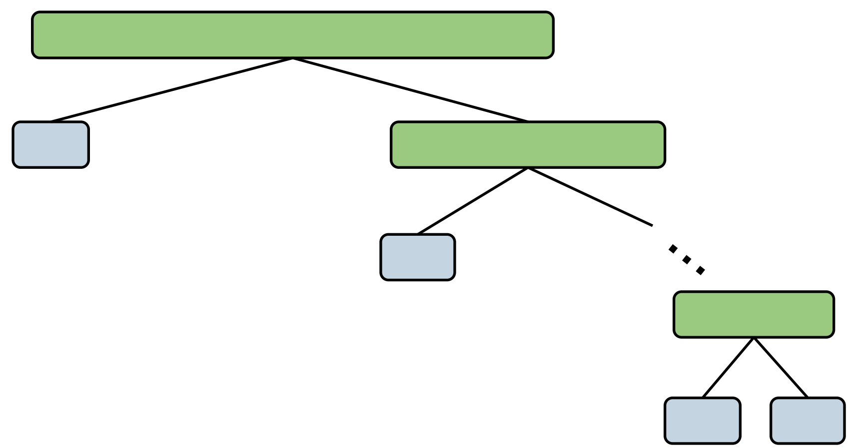
Quick-sort Worst-case Running Time

- 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



Quick-sort Average-case running time

- While the worst-case for quick-sort is $O(n^2)$, many (most) executions run a lot faster
  - when the pivot is closer to the middle value of the input for that step, then the two recurrence steps are close to half the size
    - similar to what happens in merge-sort (n/2 each)
    - n/4 and 3n/4: height O(logn)

- Fact: the average running time is O(n log n)
  - take the running time of many different executions, with pivots chosen randomly, and average these running times
  - Expected running time: over all possible random choices (independent from input distribution)

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).

l *r*

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 **6** 9

(pivot x = 6)

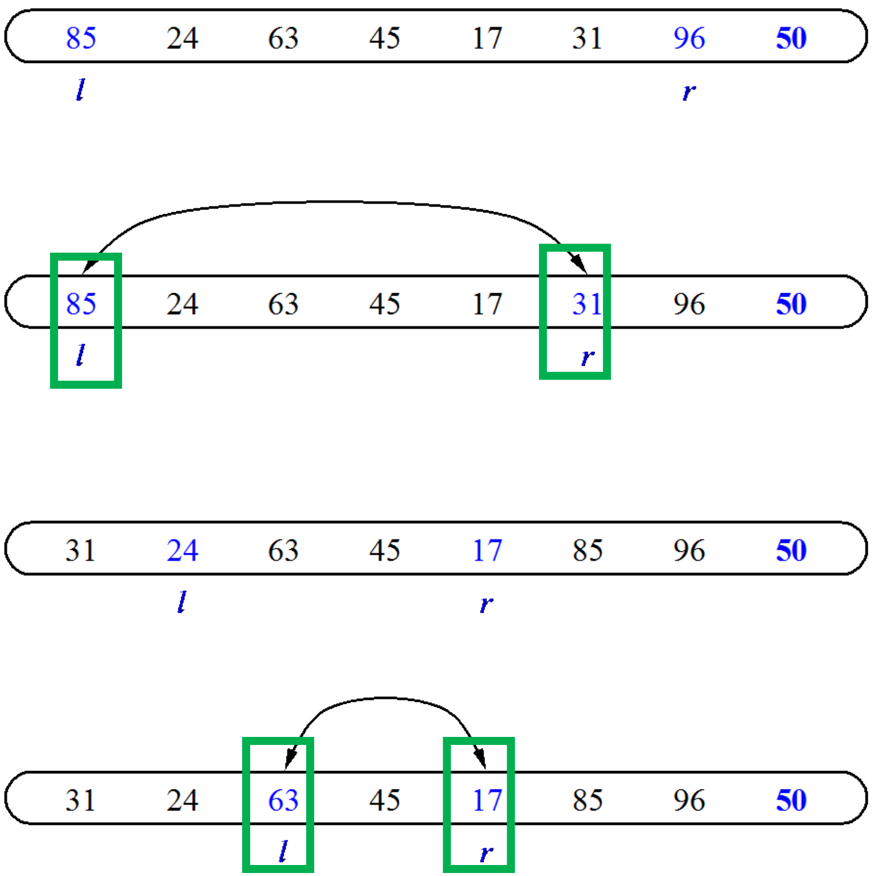
– Repeat until *l* and *r* cross:
  – Scan *l* to the right until finding an element ≥ x.
  – Scan *r* to the left until finding an element < x.
  – Swap elements at indices *l* and r

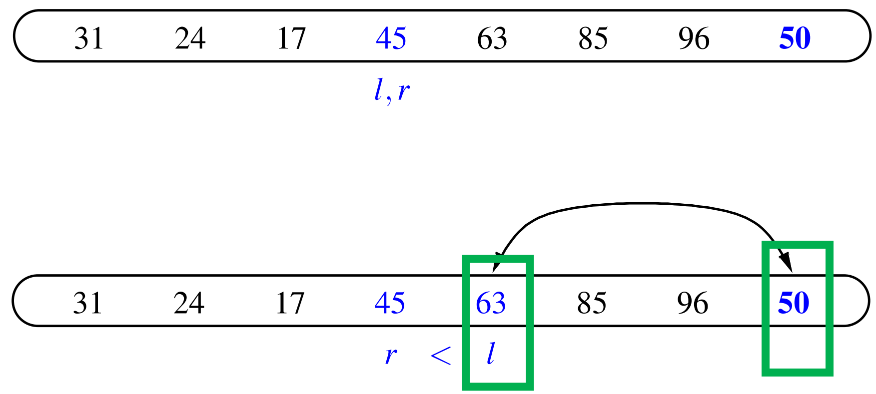
l *r*

3 2 5 1 0 $\boxed{7}$ 3 5 9 $\boxed{2}$ 7 9 8 9 7 **6** 9

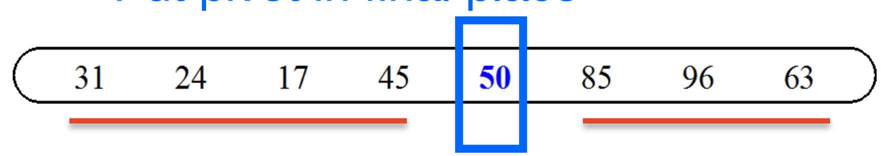
swap

In-Place : divide step





Put pivot in final place



Make recursive calls…

Java Implementation

```
/** Sort the subarray S[a..b] inclusive. */
private static <K> void quickSortInPlace(K[ ] S, Comparator<K> comp,
                                                              int a, int b) {
  if (a >= b) return;            // subarray is trivially sorted
  int left = a;
  int right = b-1;
  K pivot = S[b];
  K temp;                         // temp object used for swapping
  while (left <= right) {
    // scan until reaching value equal or larger than pivot (or right marker)
    while (left <= right && comp.compare(S[left], pivot) < 0) left++;
    // scan until reaching value equal or smaller than pivot (or left marker)
    while (left <= right && comp.compare(S[right], pivot) > 0) right--;
    if (left <= right) {         // indices did not strictly cross
      // so swap values and shrink range
      temp = S[left]; S[left] = S[right]; S[right] = temp;
      left++; right--;
    }
  }
  // put pivot into its final place (currently marked by left index)
  temp = S[left]; S[left] = S[b]; S[b] = temp;
  // make recursive calls
  quickSortInPlace(S, comp, a, left - 1);
  quickSortInPlace(S, comp, left + 1, b);
}
```

Summary of sorting algorithms

Sort algorithm	Time cost	Comments
Selection-sort	$O(n^2)$	can be done in-place
Insertion-sort	$O(n^2)$	can be done in-place
Heap-sort	$O(n \log n)$	can be done in-place
Bubble-sort	$O(n^2)$	can be done in-place sequential access, so works well with data on disk
Merge-sort	$O(n \log n)$	can be done in-place sequential access, so works well with data on disk
Quick-sort	worst: $O(n^2)$ Average/expected: $O(n \log n)$	can be done in-place requires randomization

Lower bound of comparison-based sorting algorithm: $\underline{\Omega}(n \log n)$ necessary
Linear-time sorting: Bucket sort, Radix sort (special assumption on input)

Summary

- Sorting algorithms and their costs
  - Review of pq-sorting algorithms: insertion, selection sort, heap-sort
  - In-place sorting
  - **Bubble sort**
  - **Merge-sort (section 13.1)**
  - **Quick-sort (section 13.2)**