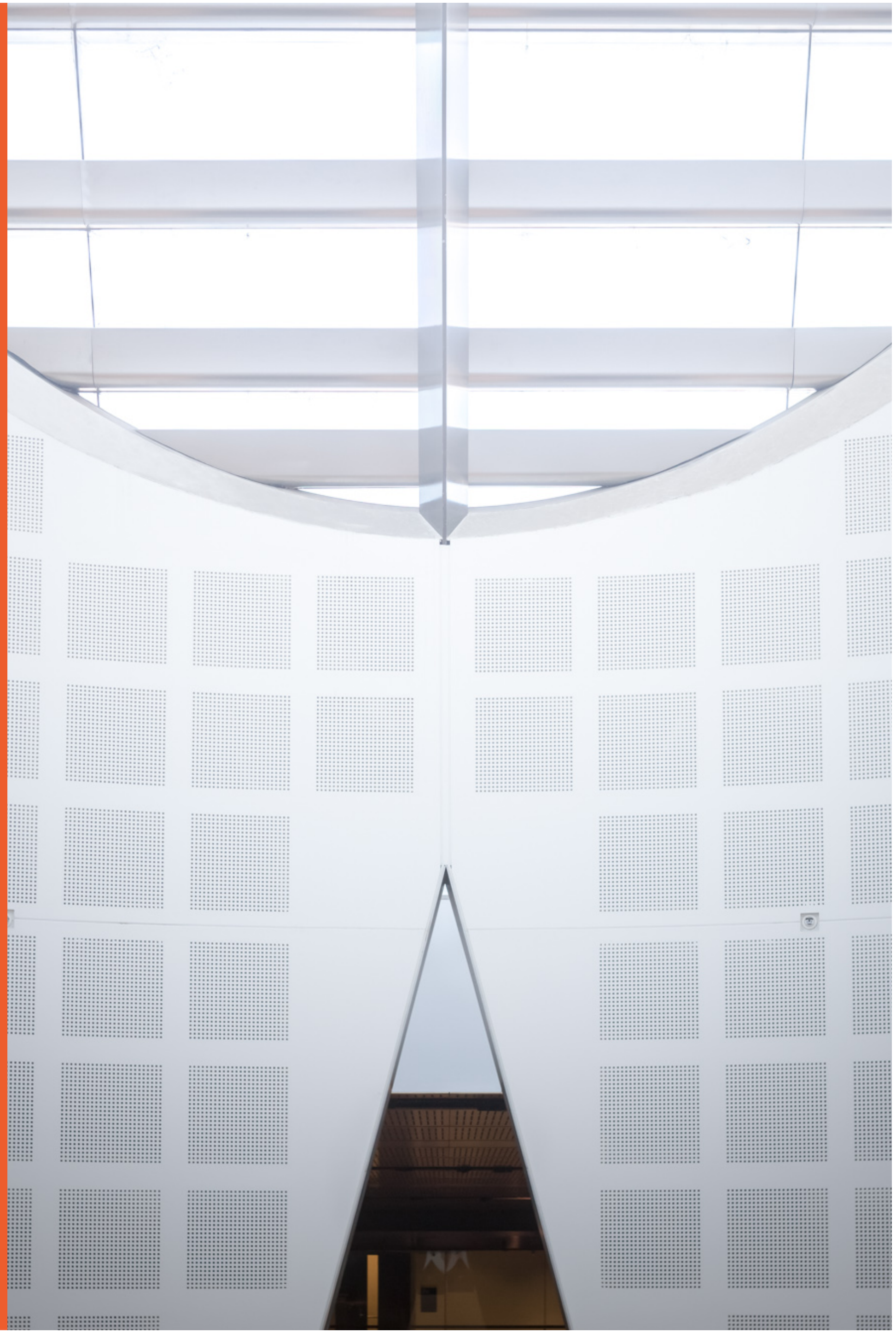
INFO1105/1905
Data Structures
Week 11: Sorting
see textbook sections 12.1, 12.2

Professor Alan Fekete
Dr John Stavrakakis
School of Information Technologies

using material from the textbook
and A/Prof Kalina Yacef

THE UNIVERSITY OF SYDNEY



Copyright warning

- These slides contain material from the textbook (Goodrich, Tamassia & Goldwasser)
  - Data structures and algorithms in Java (5^{th} & 6^{th} 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 lectures of weeks 9, 10, and 11 (labs 10 and 11)
    - hash function and properties
    - separate chaining hashtable
    - open addressing hashtable (linear probing, quadratic probing, double hashing)
    - trie
    - sorting algorithms and their costs

Reminder: Asst 2

- Asst 2 has been released
- Due date postponed to Monday Oct 24 (9pm)
- You must write your own code that implements the interface, *using the data structure described in the instructions*
  - Do not use any Map from other libraries
  - You may use some List types from JCF

Outline

- Sorting algorithms
  - Elementary sorting algorithms based on priority queue (review):
    - insertion sort, selection sort, heapsort
  - Bubblesort
  - Merge-sort
  - Quick-sort
  - Bucket-sort
  - Radix-sort

Recall: Priority Queue Sorting

- 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
  2. Remove the elements in sorted order with a series of removeMin operations
- The running time of this sorting method depends on the priority queue implementation

Algorithm *PQ-Sort*(*S*, *C*)
    Input sequence *S*, comparator *C* for the elements of *S*
    Output sequence *S* sorted in increasing order according to *C*
    $P \leftarrow$ priority queue with
        comparator C
    while $\neg$ *S.isEmpty* ()
        $e \leftarrow$ *S.removeFirst* ()
        P.insert (e, $\emptyset$)
    while $\neg$ *P.isEmpty*()
        $e \leftarrow$ *P.removeMin*().*getKey*()
        S.addLast(e)

Sequence-based Priority Queue

- Implementation with an unsorted list

- Performance:
  - insert takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list

- Performance:
  - insert takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin and min take $O(1)$ time, since the smallest key is at the beginning

Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an *unsorted* sequence
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with n insert operations takes $O(n)$ time
  2. Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

$$1 + 2 + \ldots + n$$

- Selection-sort runs in $O(n^2)$ time
- Selection is the bottleneck computation

Selection-Sort Example

		Sequence S	Priority Queue P
Input:		(7,4,8,2,5,3,9)	()
Phase 1			
(a)		(4,8,2,5,3,9)	(7)
(b)		(8,2,5,3,9)	(7,4)
..	..	..	
(g)		()	(7,4,8,2,5,3,9)
Phase 2			
(a)		(2)	(7,4,8,5,3,9)
(b)		(2,3)	(7,4,8,5,9)
(c)		(2,3,4)	(7,8,5,9)
(d)		(2,3,4,5)	(7,8,9)
(e)		(2,3,4,5,7)	(8,9)
(f)		(2,3,4,5,7,8)	(9)
(g)		(2,3,4,5,7,8,9)	()

Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with n insert operations takes time proportional to

$$1 + 2 + \ldots + n$$

  2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time
- Insertion is the bottleneck computation

Insertion-Sort Example

	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
.. ..	..	
(g)	(2,3,4,5,7,8,9)	()

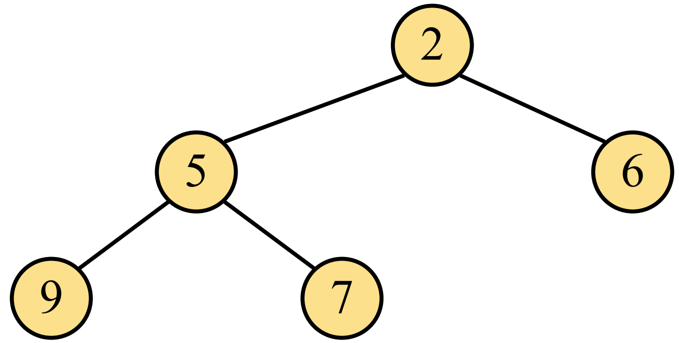
Heap-Sort

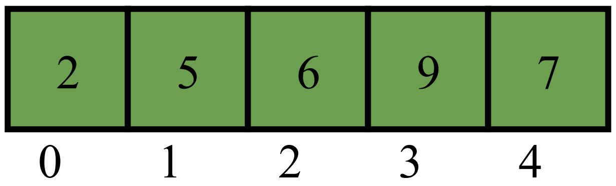
- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods insert and removeMin take $O(\log n)$ time
  - methods size, isEmpty, and min take time $O(1)$ time

- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

Array-based Heap Implementation

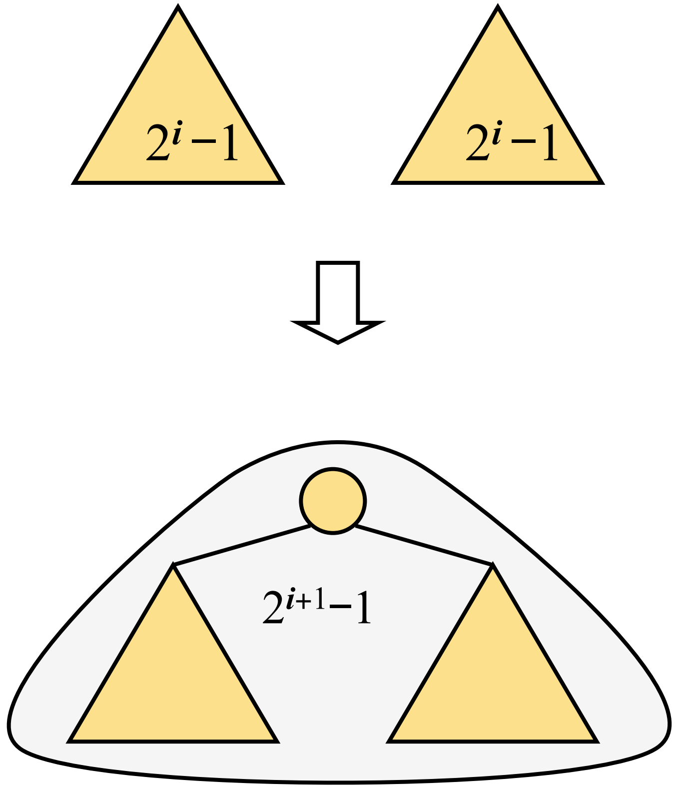
- We can represent a heap with n keys by means of an array of length $n + 1$
- For the node at rank i
  - the left child is at rank $2i+1$
  - the right child is at rank $2i + 2$
- Links between nodes are not explicitly stored
- Operation insert corresponds to inserting at rank $n + 1$
- Operation removeMin corresponds to removing at rank n
- Yields in-place heap-sort





Bottom-up Heap Construction

- Instead of inserting each element one by one into the heap
- We can construct a heap storing n given keys in using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys



Bubble-sort

- A simple sorting algorithm that is easy to code
- To sort a sequence of n comparable elements
  - Scan the sequence n-1 times
  - At each step in a scan, compare the current element with the next and swap them if they are out of order
- Each scan moves the largest remaining element to the end of the sequence
  - the next scan is over a sequence that is one element shorter

Example Bubble-sort

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 algorithm

```
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)
```

big-Oh Run-time analysis for Bubble-sort

```
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)
```

outer loop: n iterations

inner loop: at worst n iterations

body of inner loop: constant steps

So, total runtime is at worst $O(n*n*1) = O(n^2)$

we can do more careful analysis of the inner loop
(which is often a lot less than n iterations
cost is $C*\{(n-1)+(n-2)+\ldots\ \_2+1\}$ but this is still $O(n^2)$

Summary of Sorting Algorithms so far

Algorithm	Time	Notes
selection-sort insertion-sort Bubble-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
heap-sort	$O(n \log n)$	▪ fast ▪ in-place ▪ for large data sets (1K — 1M)

In-place: uses a small amount of memory in addition to that needed to store the objects being sorted

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

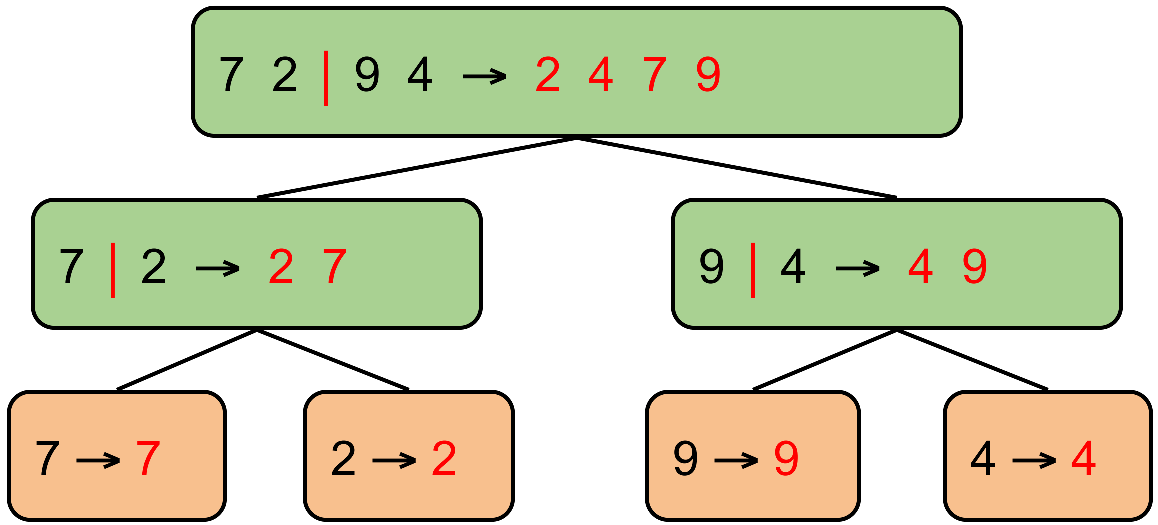
Merging Two Sorted Sequences

- 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

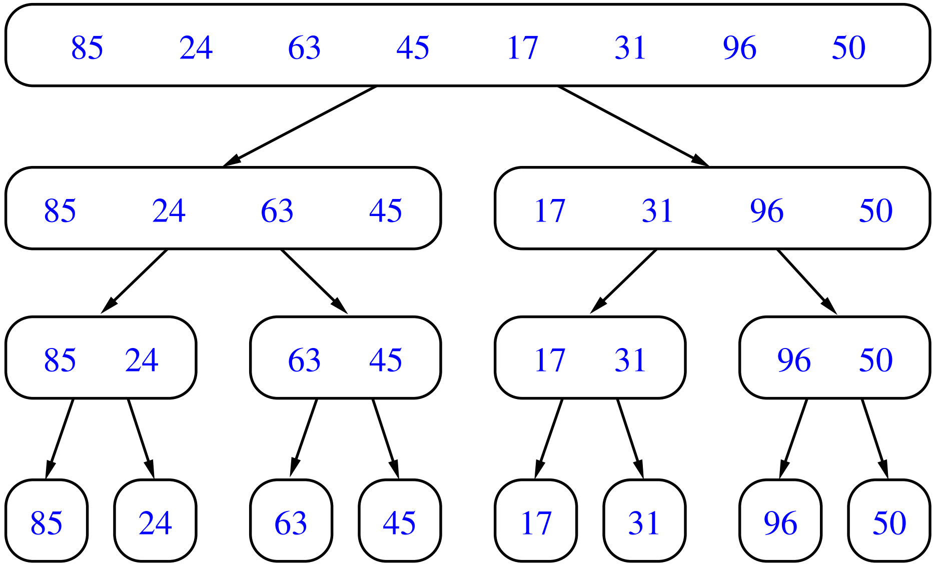
```
Algorithm merge(A, B)
    Input sequences A and B with
        n/2 elements each
    Output sorted sequence of A ∪ B

    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
```

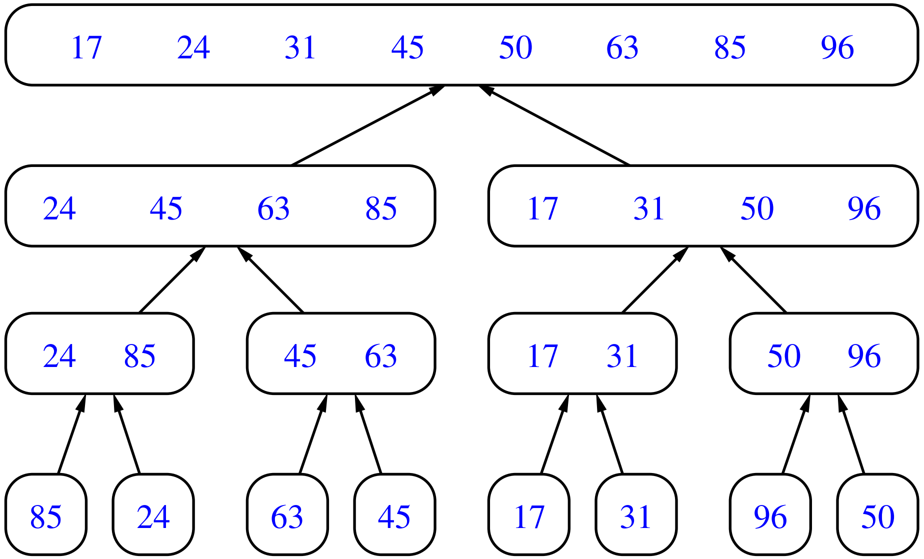
Merge-Sort Tree

- 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



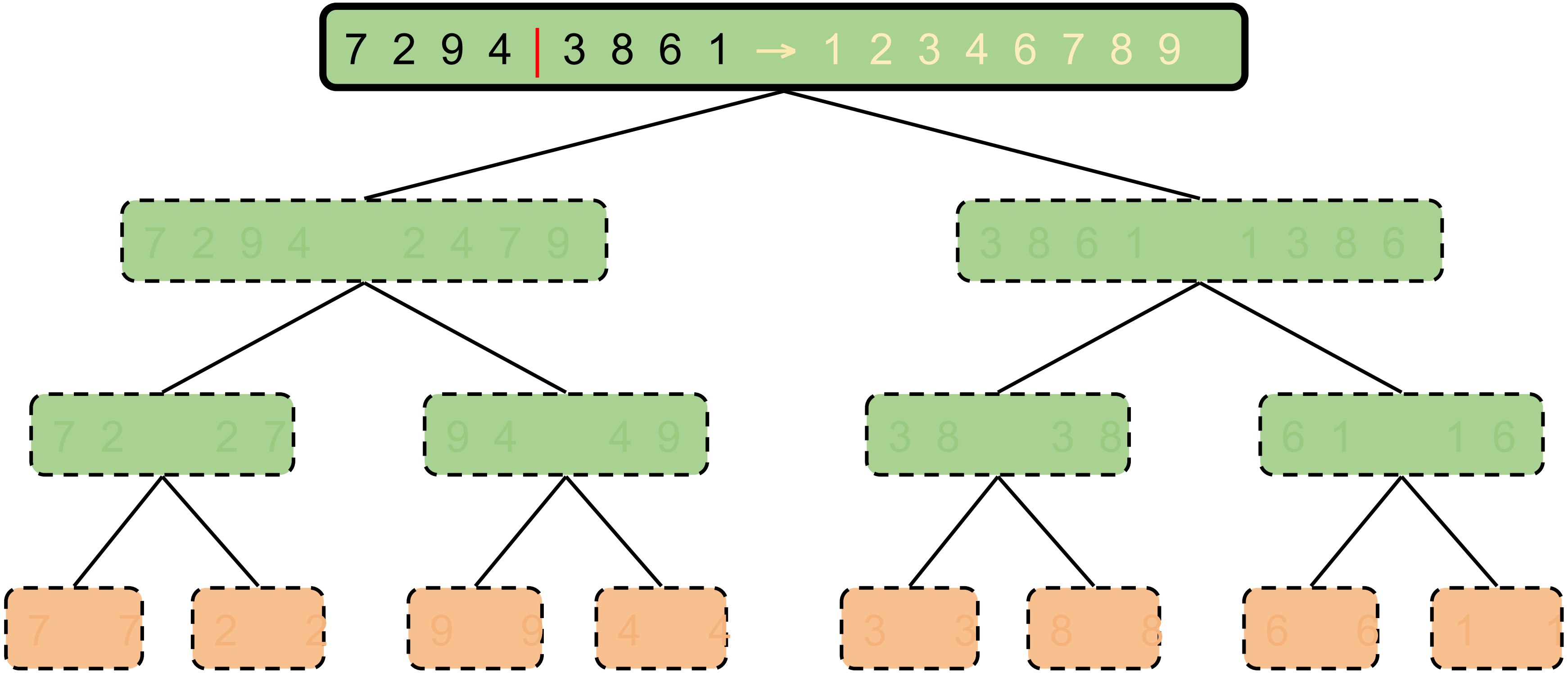
Merge sort trees (input and output sequences)





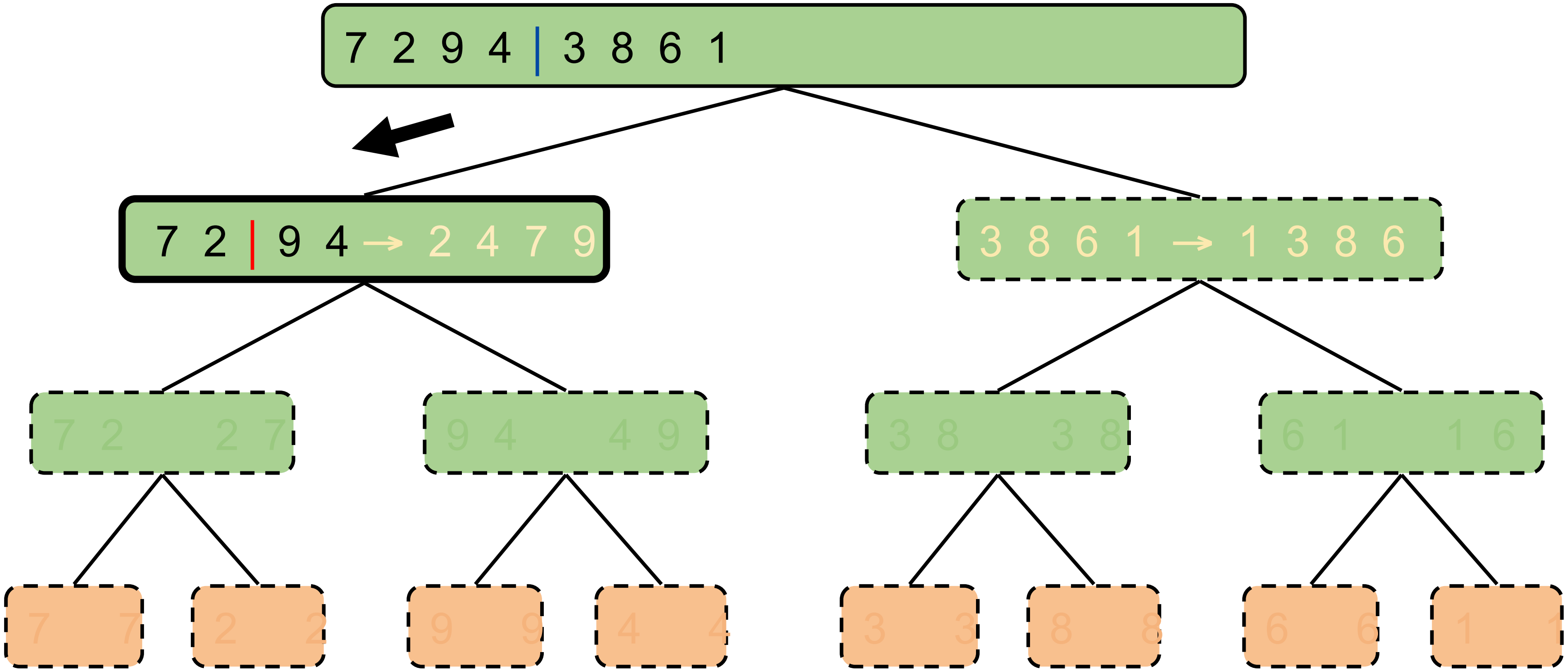
Execution Example

– Partition



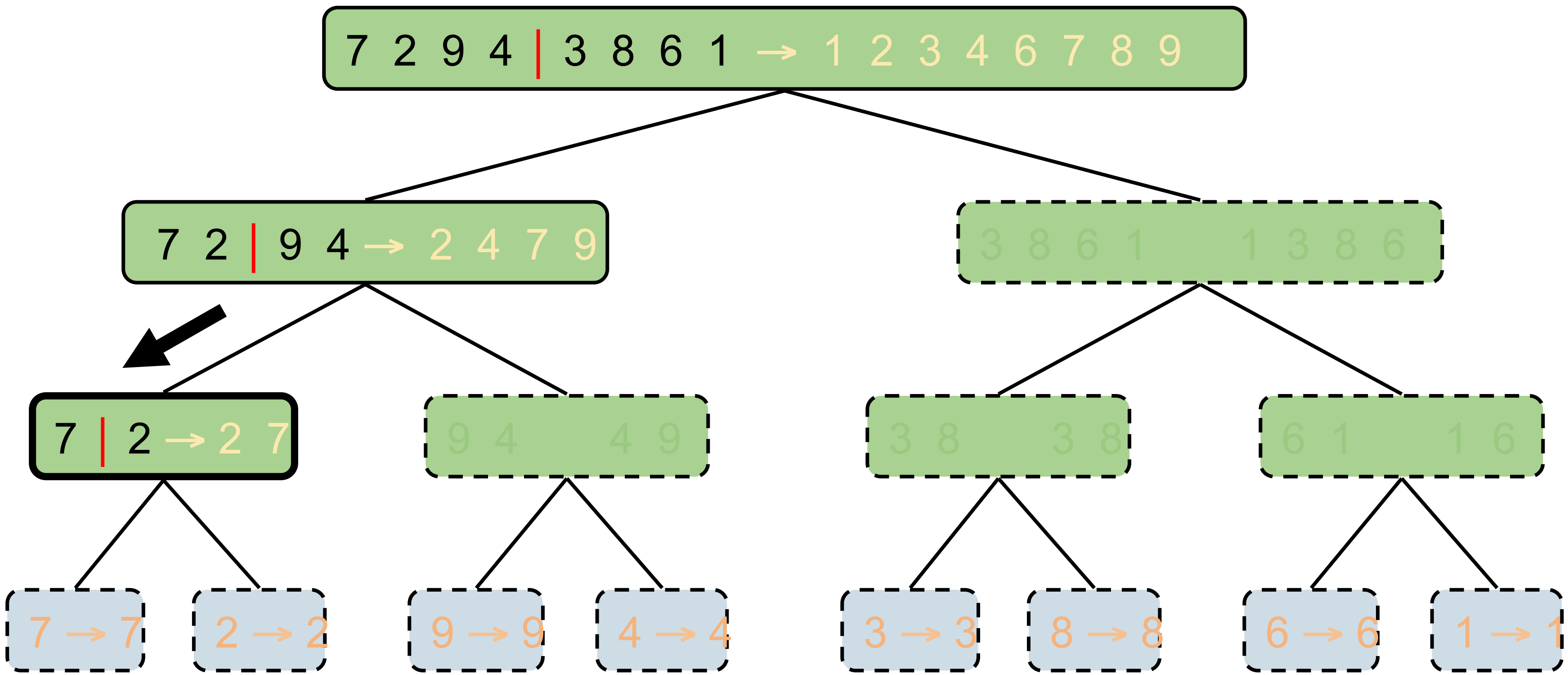
Execution Example (cont.)

– Recursive call, partition



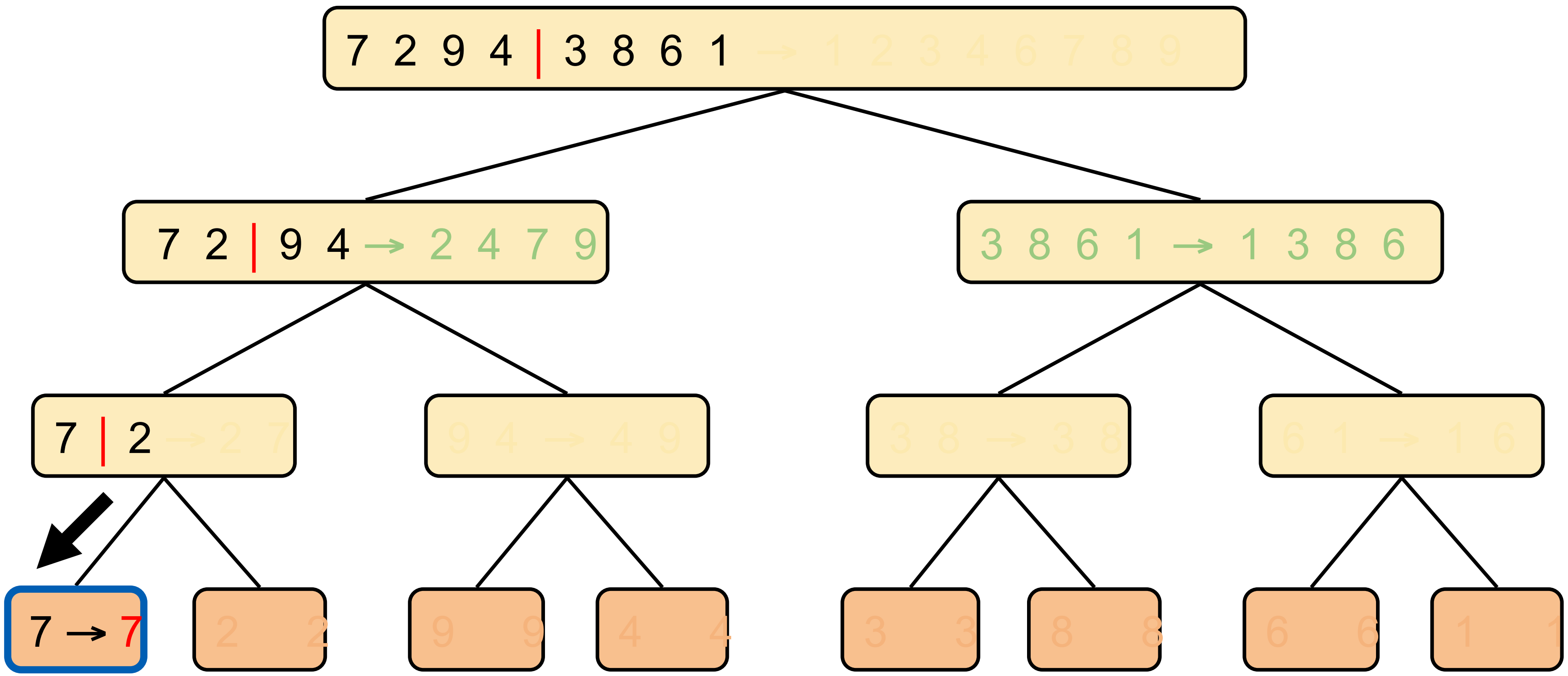
Execution Example (cont.)

– Recursive call, partition



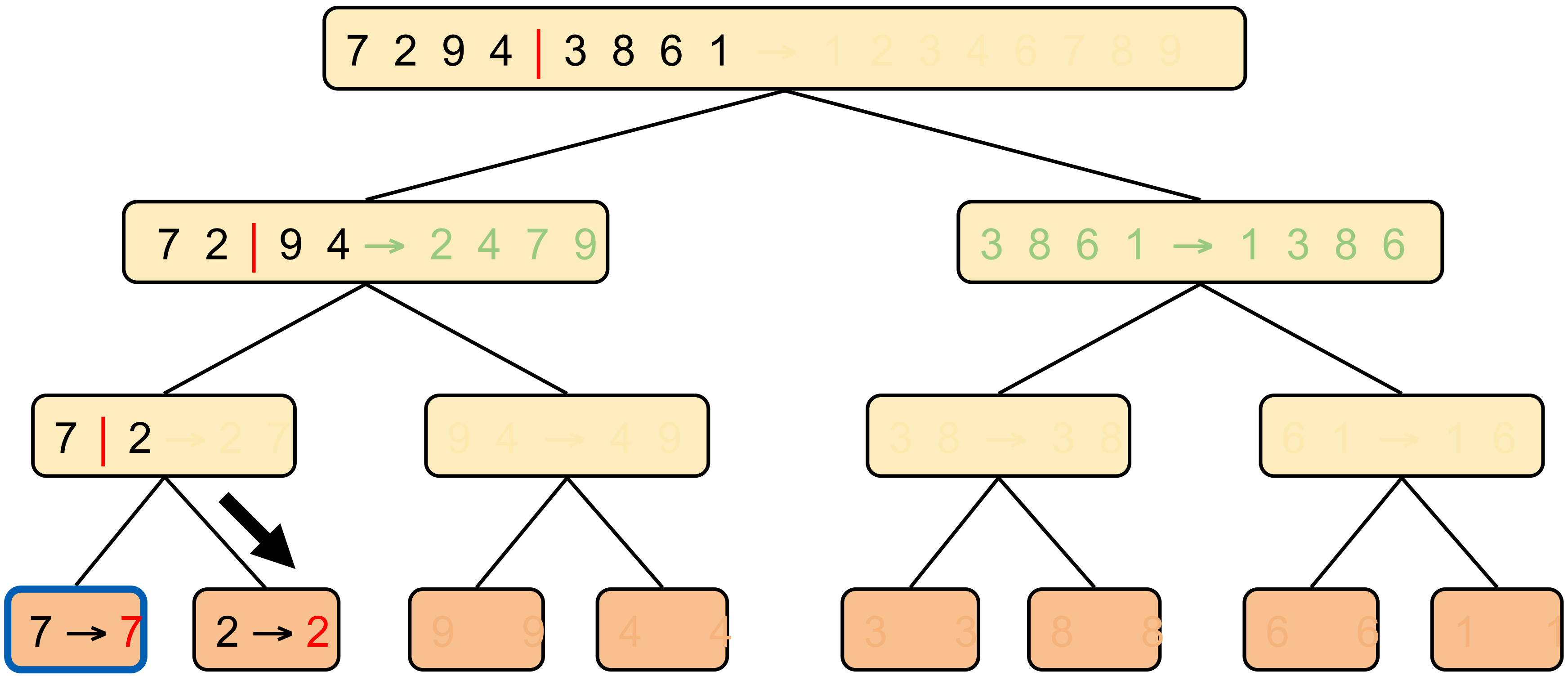
Execution Example (cont.)

– Recursive call, base case



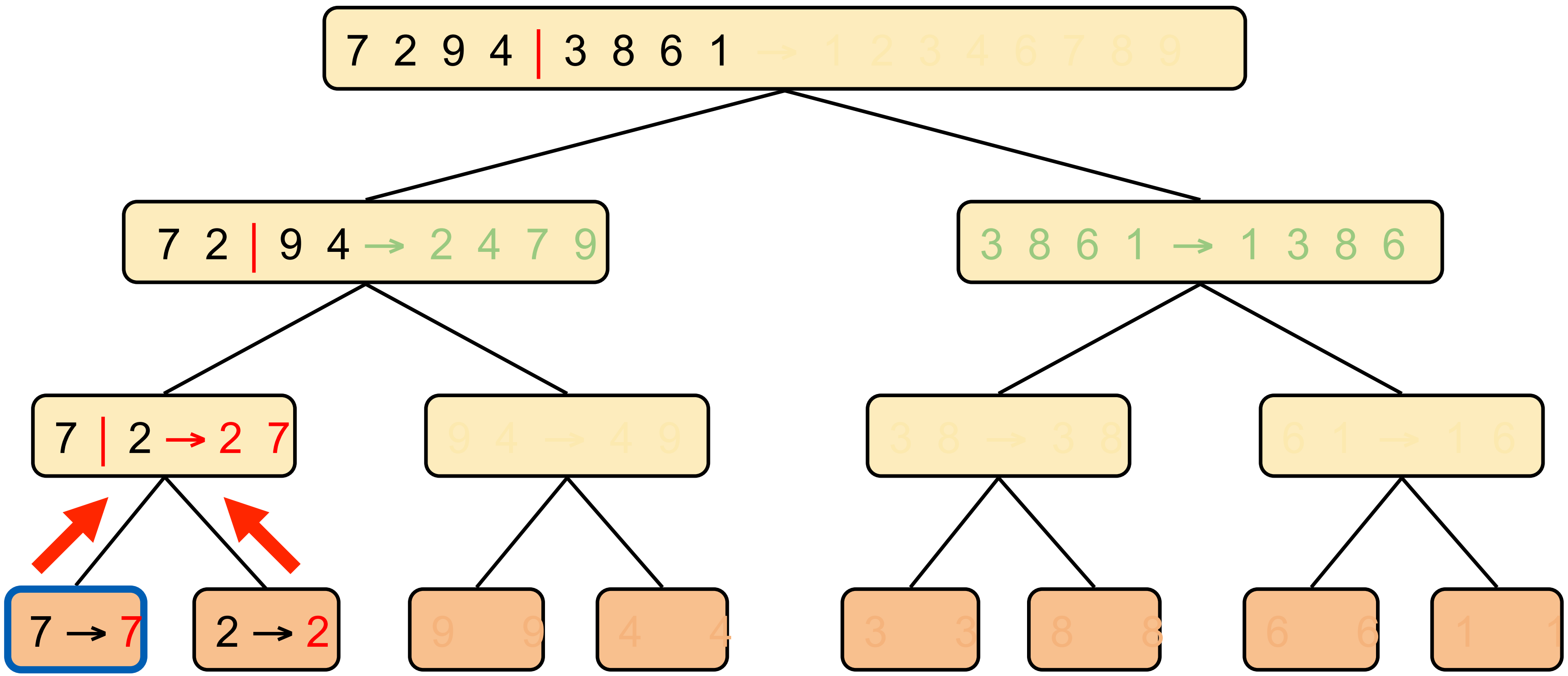
Execution Example (cont.)

– Recursive call, base case



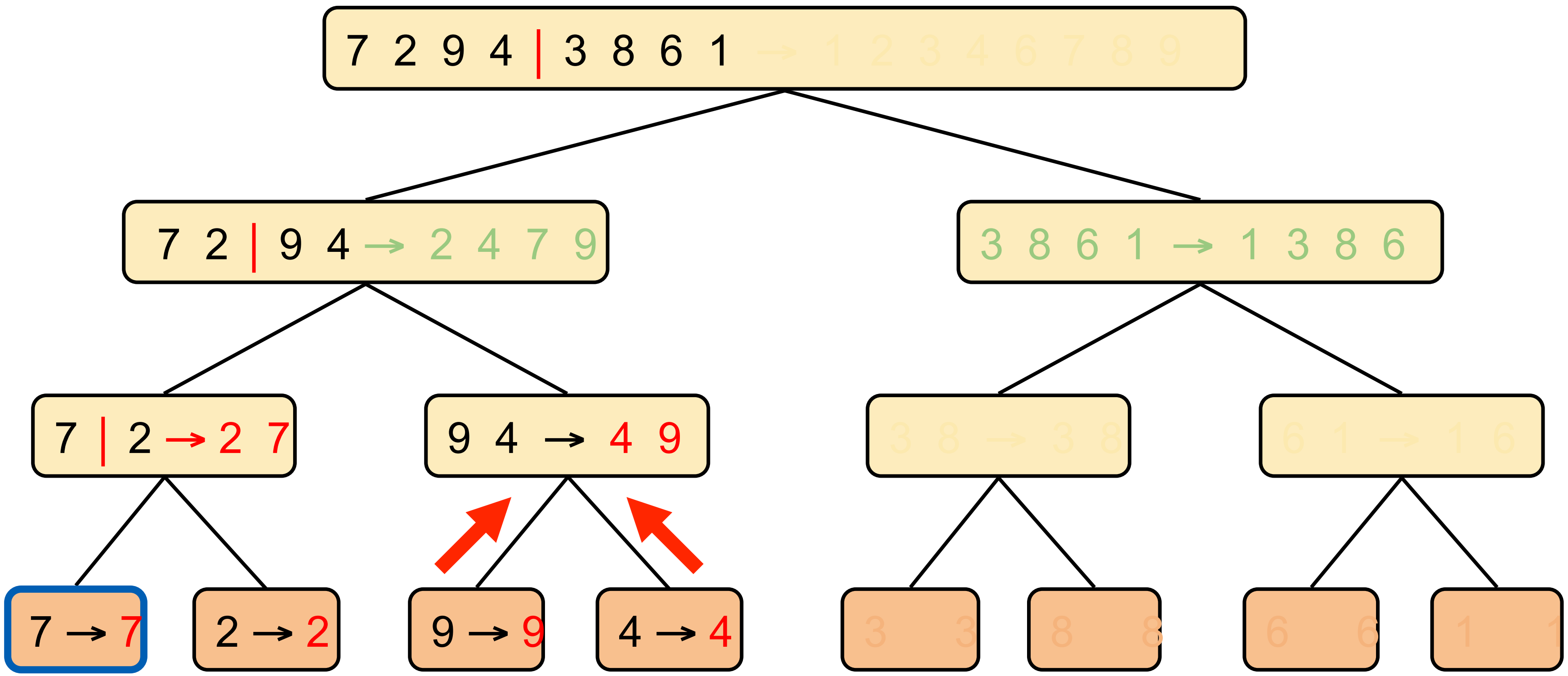
Execution Example (cont.)

- Merge



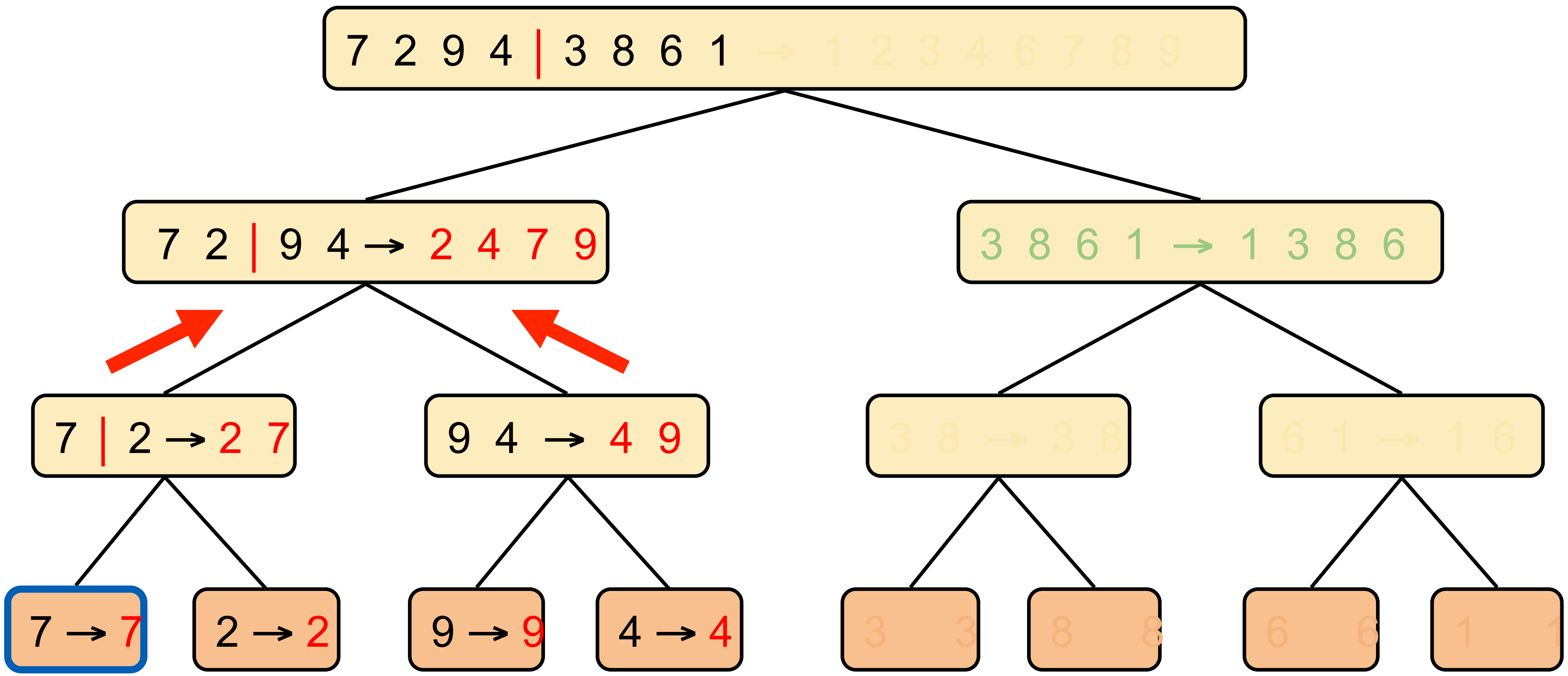
Execution Example (cont.)

– Recursive call, ..., base case, merge



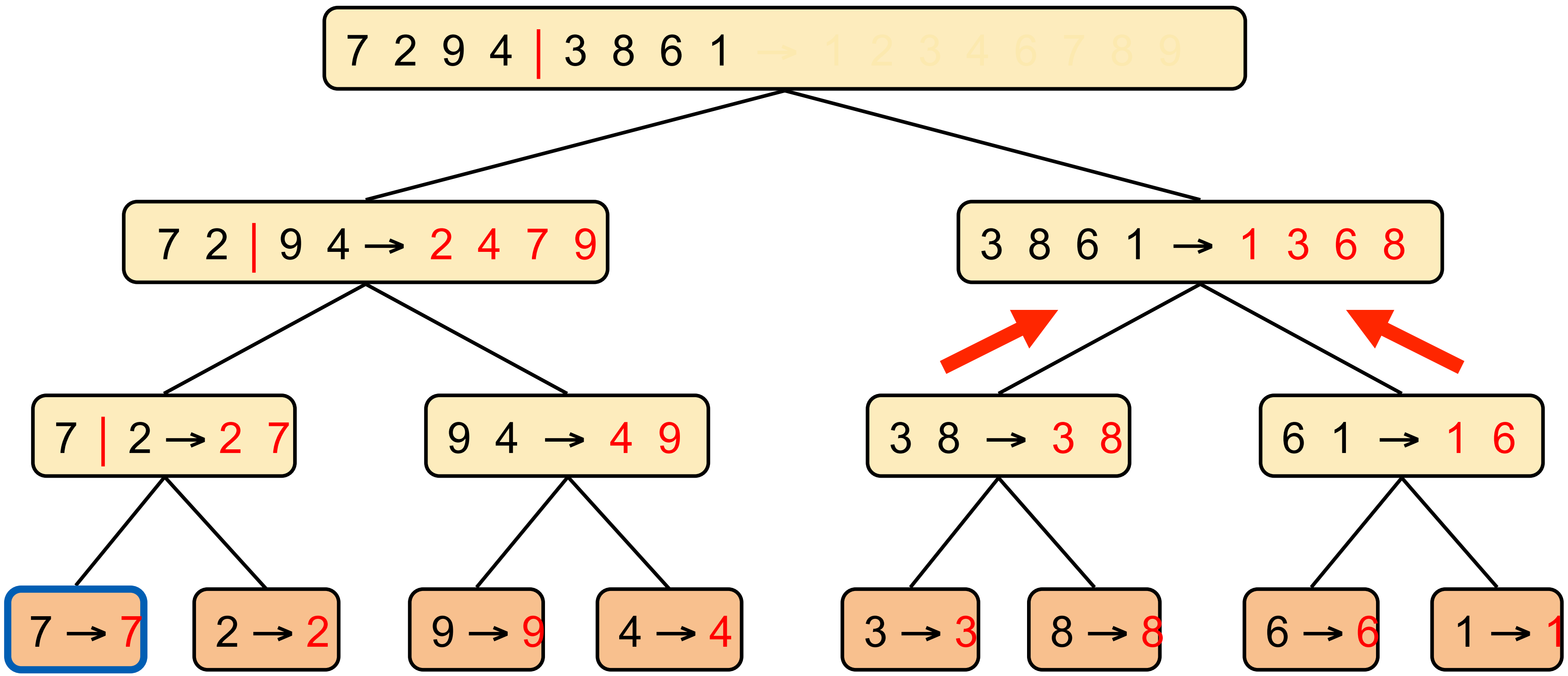
Execution Example (cont.)

- Merge



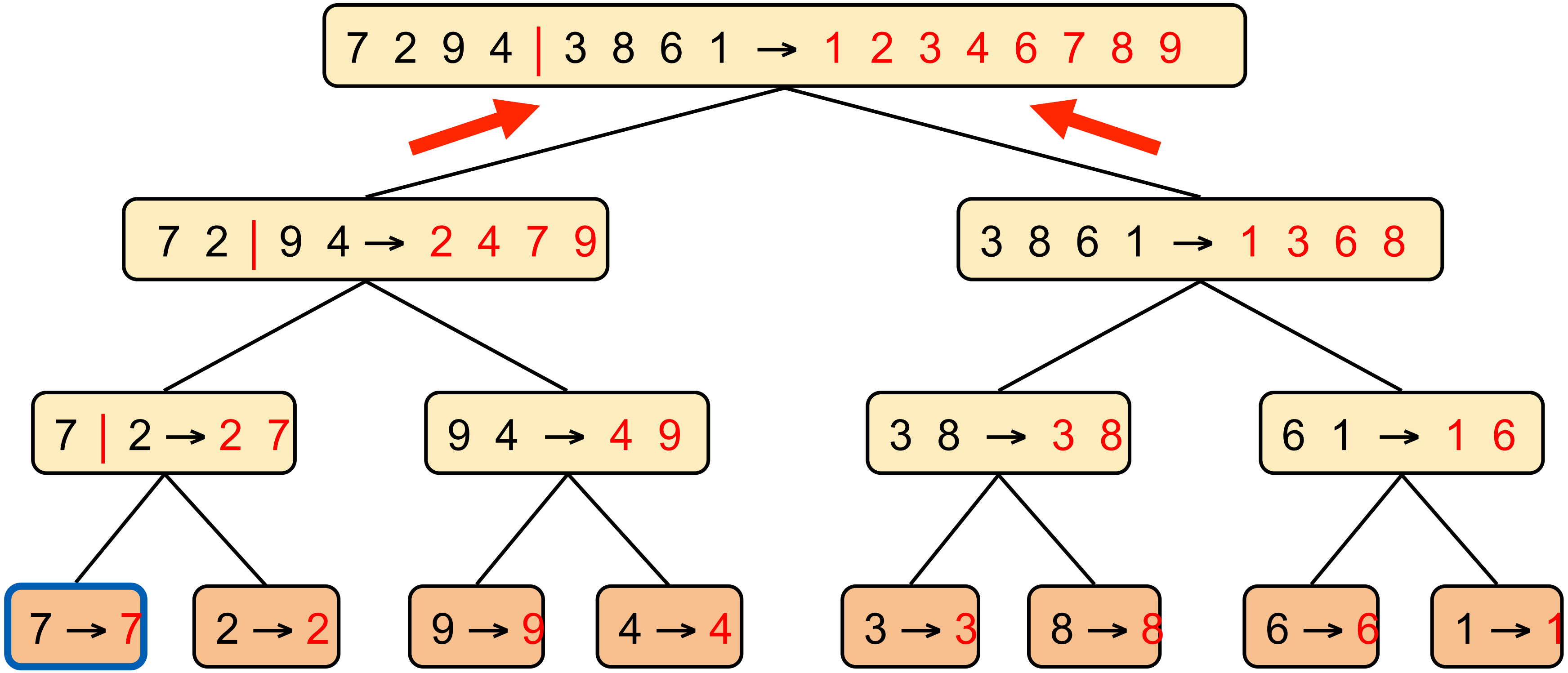
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

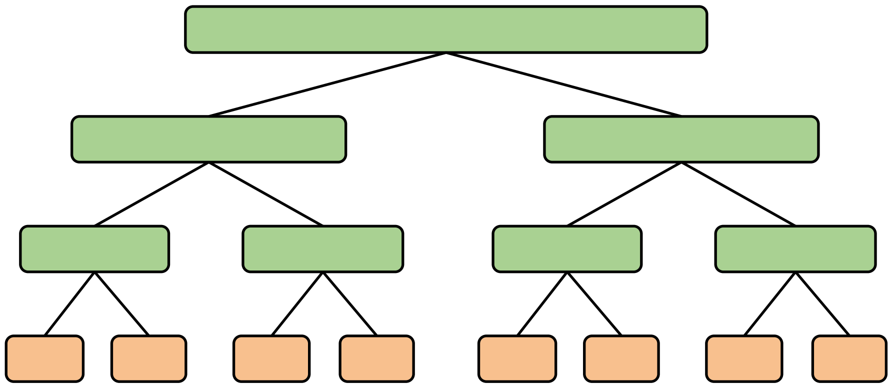
– Recursive call, …, merge, merge



Analysis of Merge-Sort

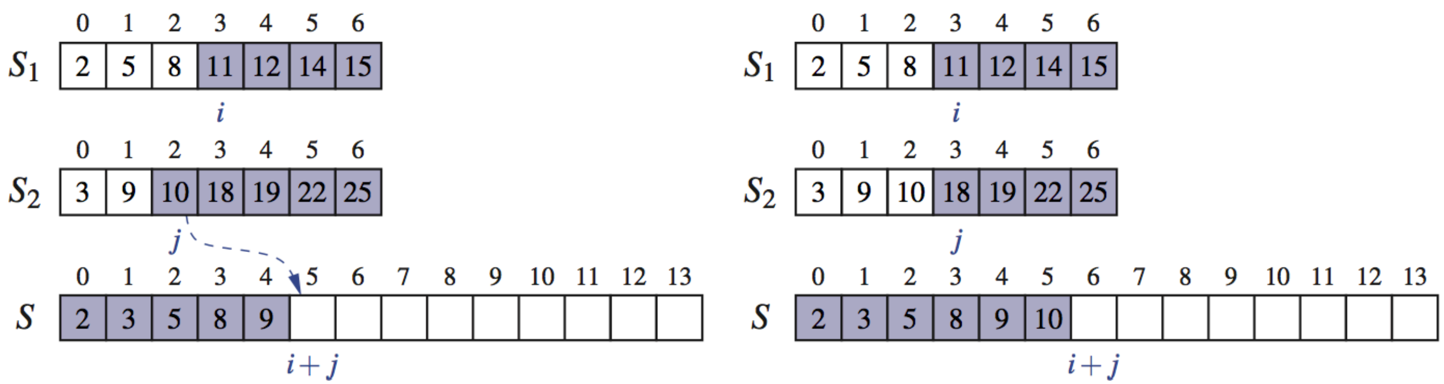
- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time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

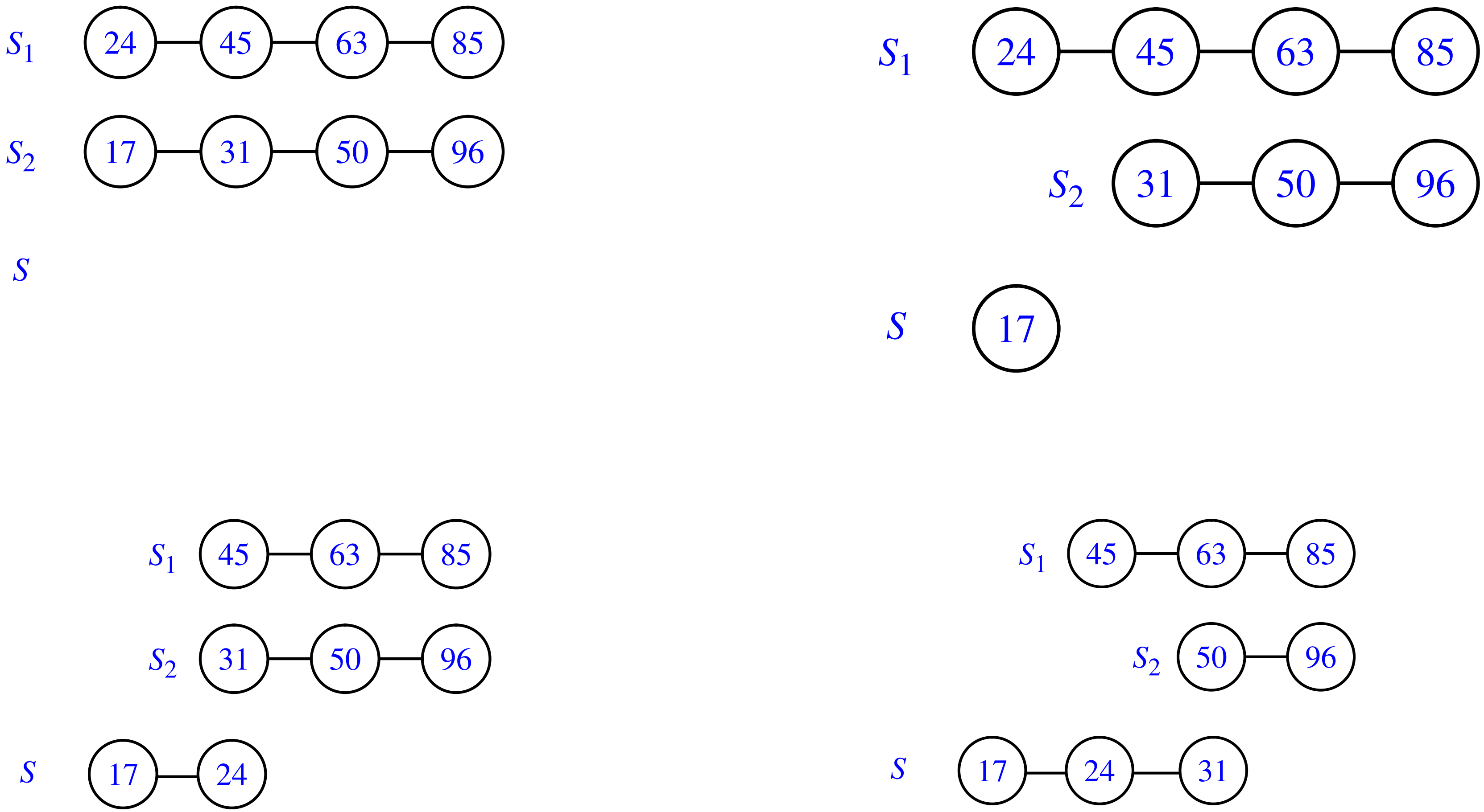
```
/** Merge contents of arrays S1 and S2 into properly sized array S. */
public static <K> void merge(K[ ] S1, K[ ] S2, K[ ] S, Comparator<K> comp) {
  int i = 0, j = 0;
  while (i + j < S.length) {
    if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))
      S[i+j] = S1[i++];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



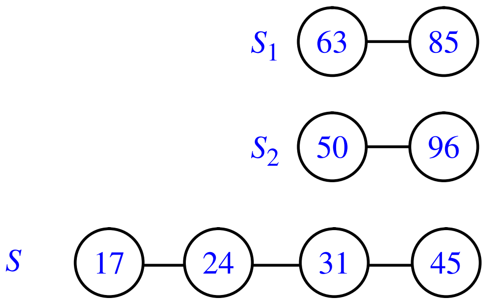
Java Merge-Sort Implementation

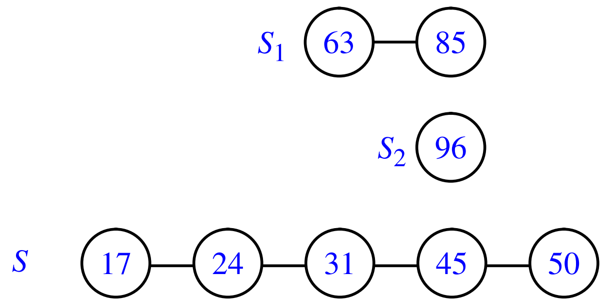
```
/** Merge-sort contents of array S. */
public static <K> void mergeSort(K[ ] S, Comparator<K> comp) {
  int n = S.length;
  if (n < 2) return;                                    // array is trivially sorted
  // divide
  int mid = n/2;
  K[ ] S1 = Arrays.copyOfRange(S, 0, mid);              // copy 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
}
```

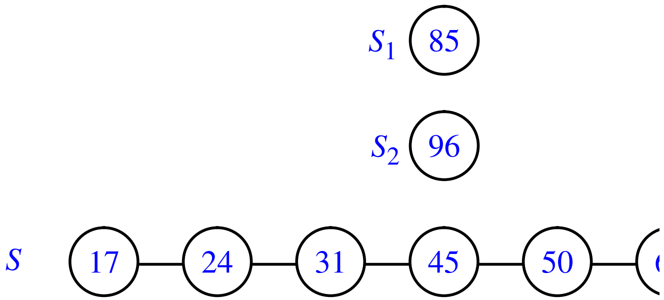
Alternative implementation: Linked Lists

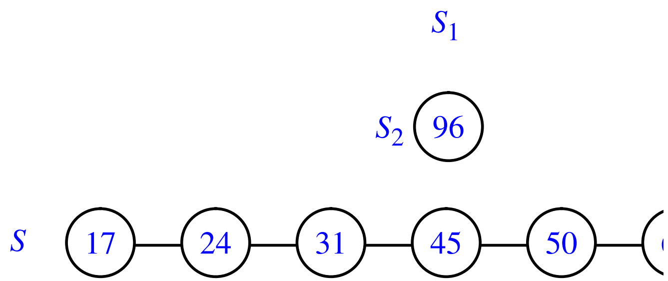


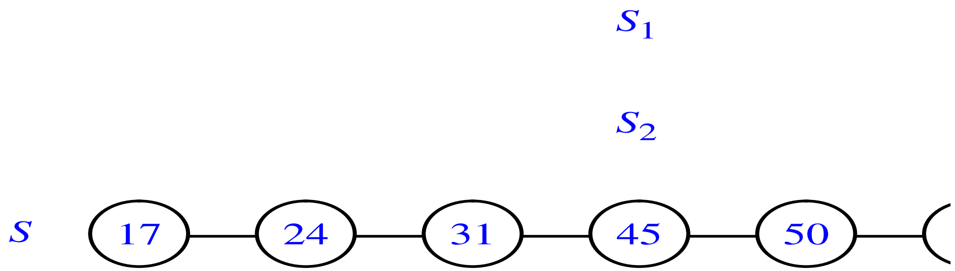
Alternative implementation: Linked Lists









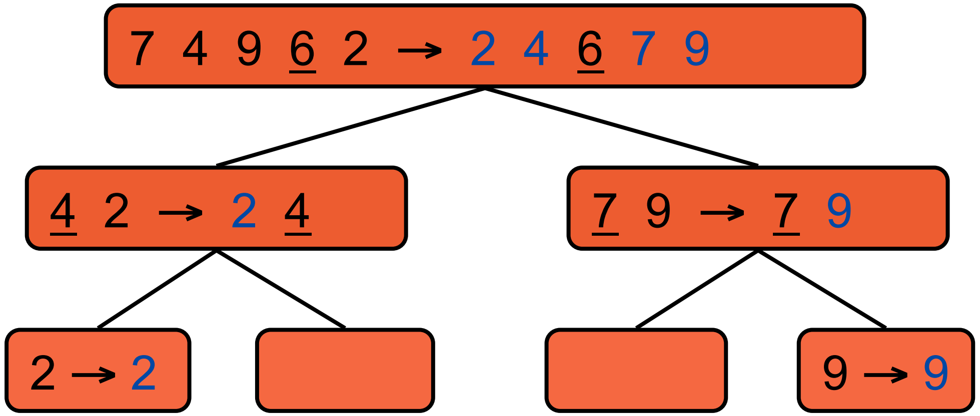


Summary of Sorting Algorithms so far

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

Advantages of merge sort over heap-sort?

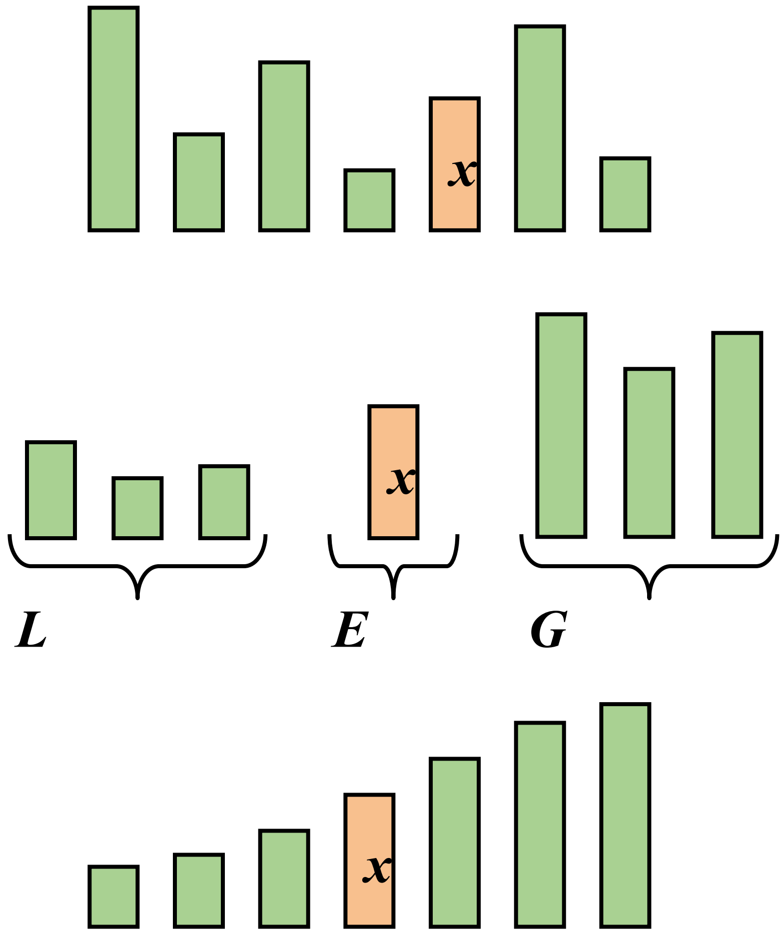
MergeSort parallelises well, stable algorithm



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

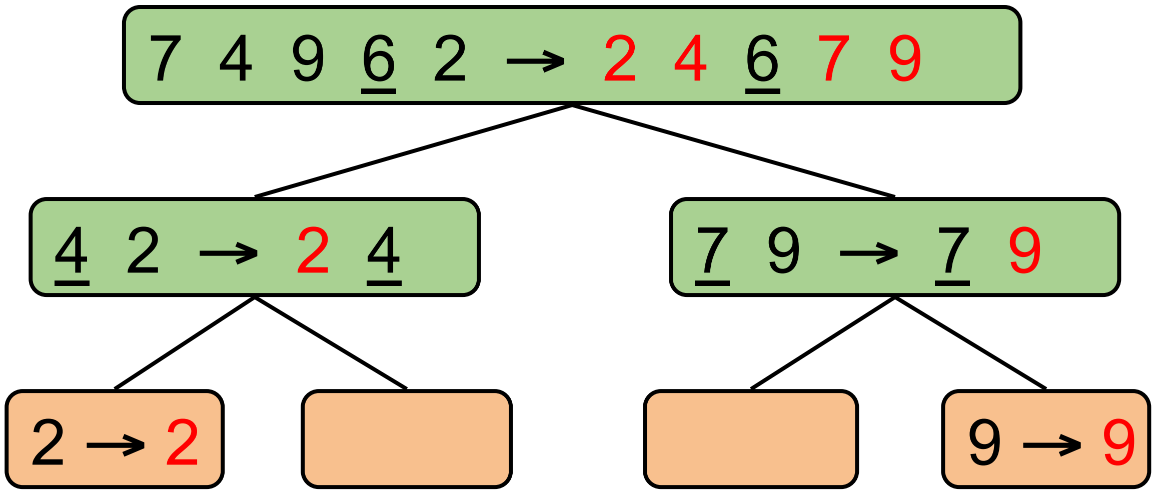
```
Algorithm partition(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the
        elements of S less than, equal to,
        or greater than the pivot, resp.
    L, E, G ← empty sequences
    x ← S.remove(p)
    while ¬S.isEmpty()
        y ← S.remove(S.first())
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

Java Implementation

```java
/** 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());
}
```

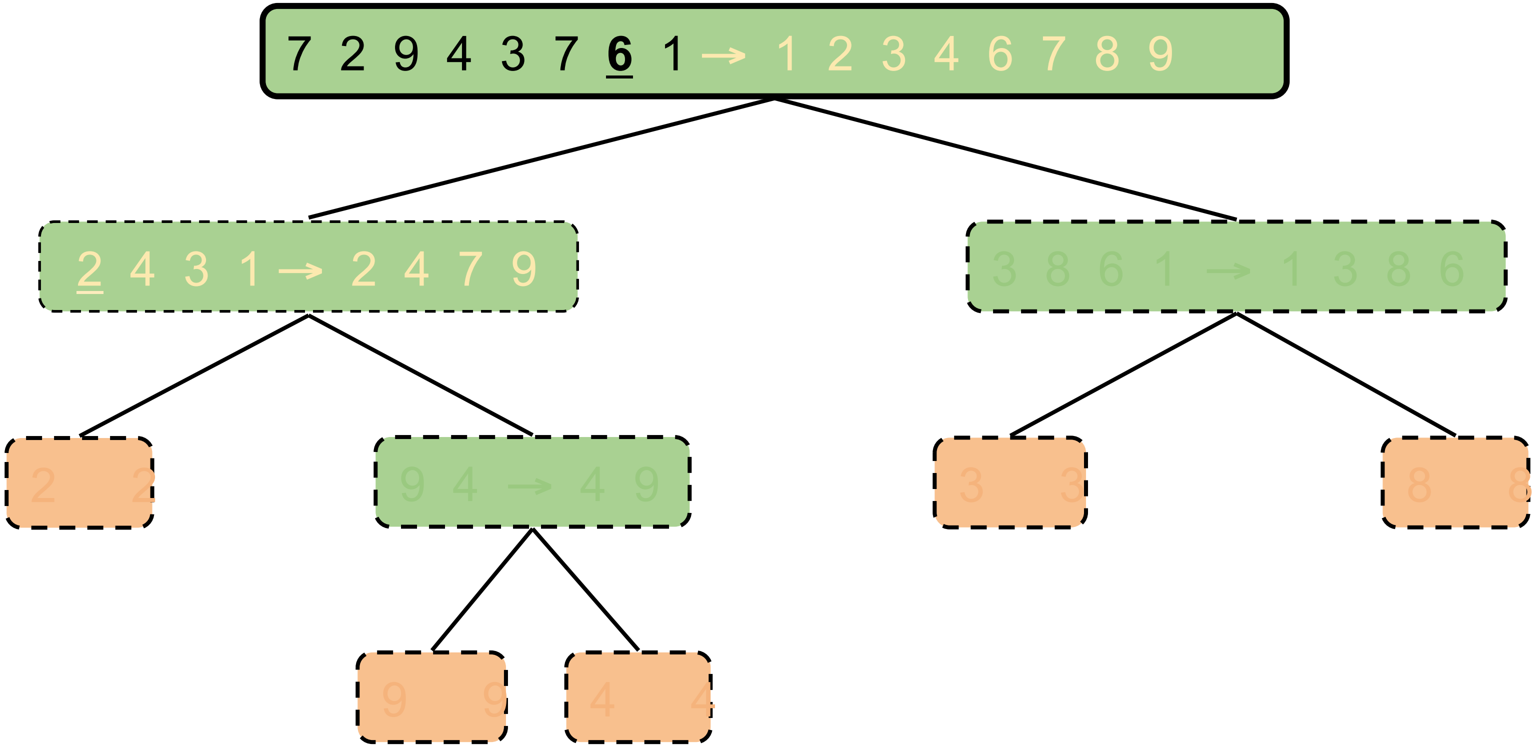
Quick-Sort Tree

- 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



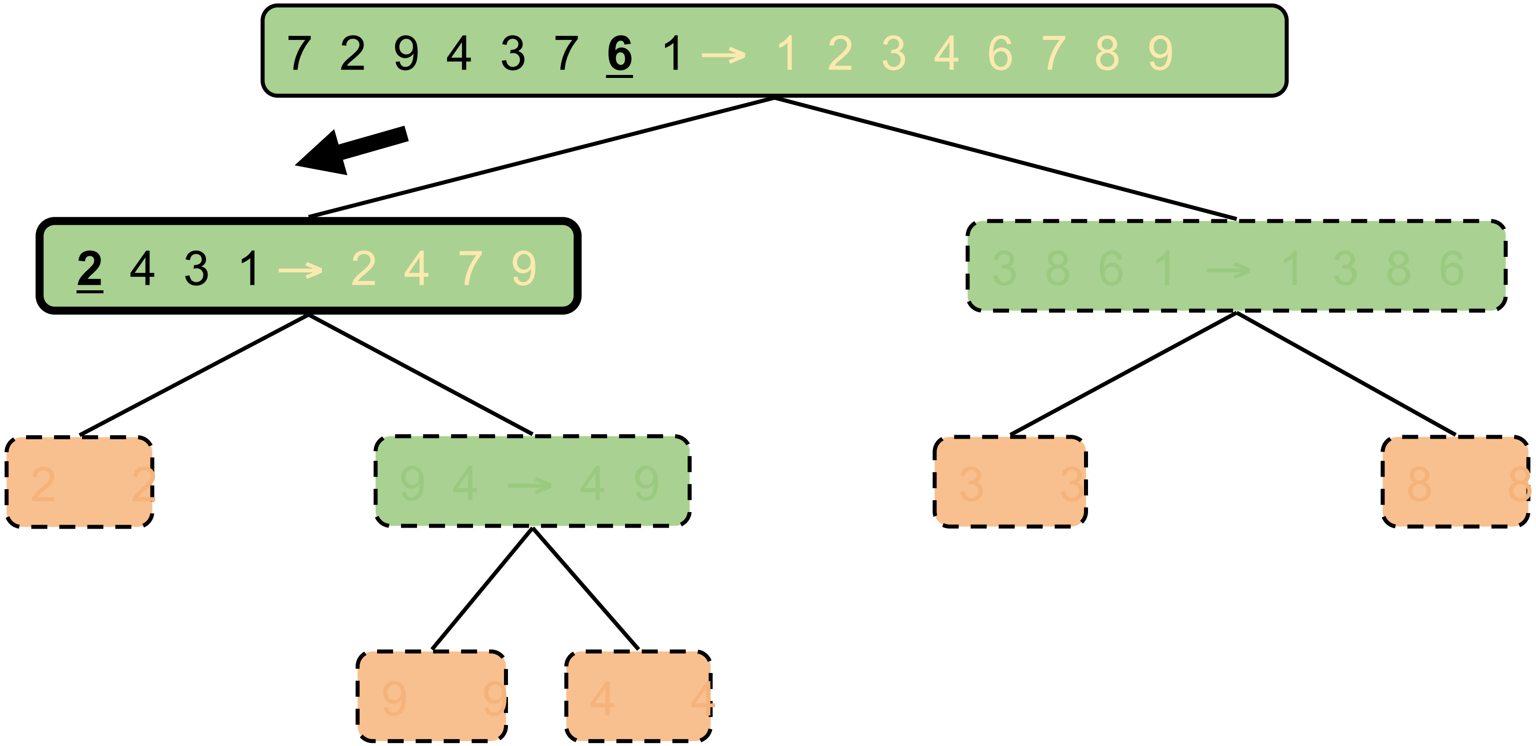
Execution Example

– Pivot selection



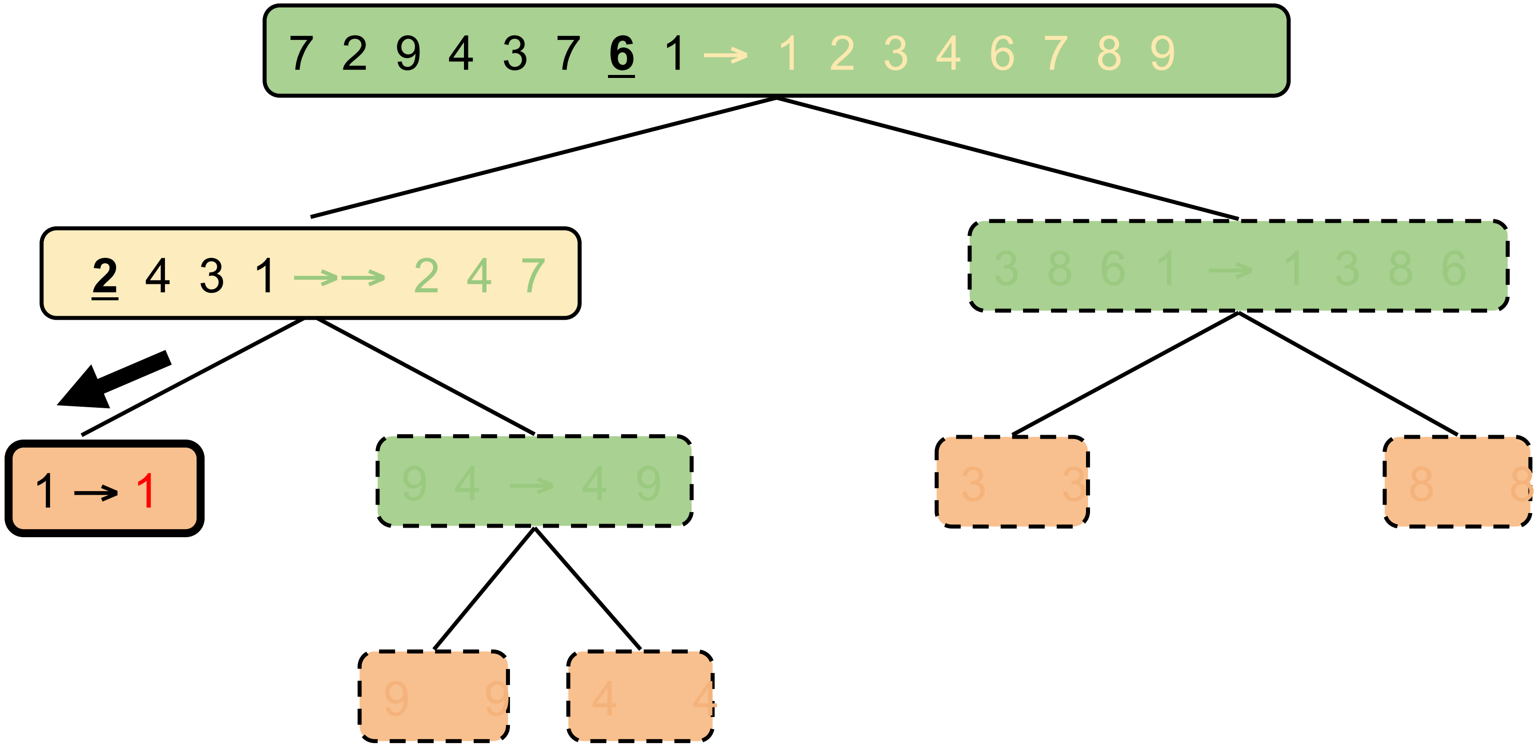
Execution Example (cont.)

– Partition, recursive call, pivot selection



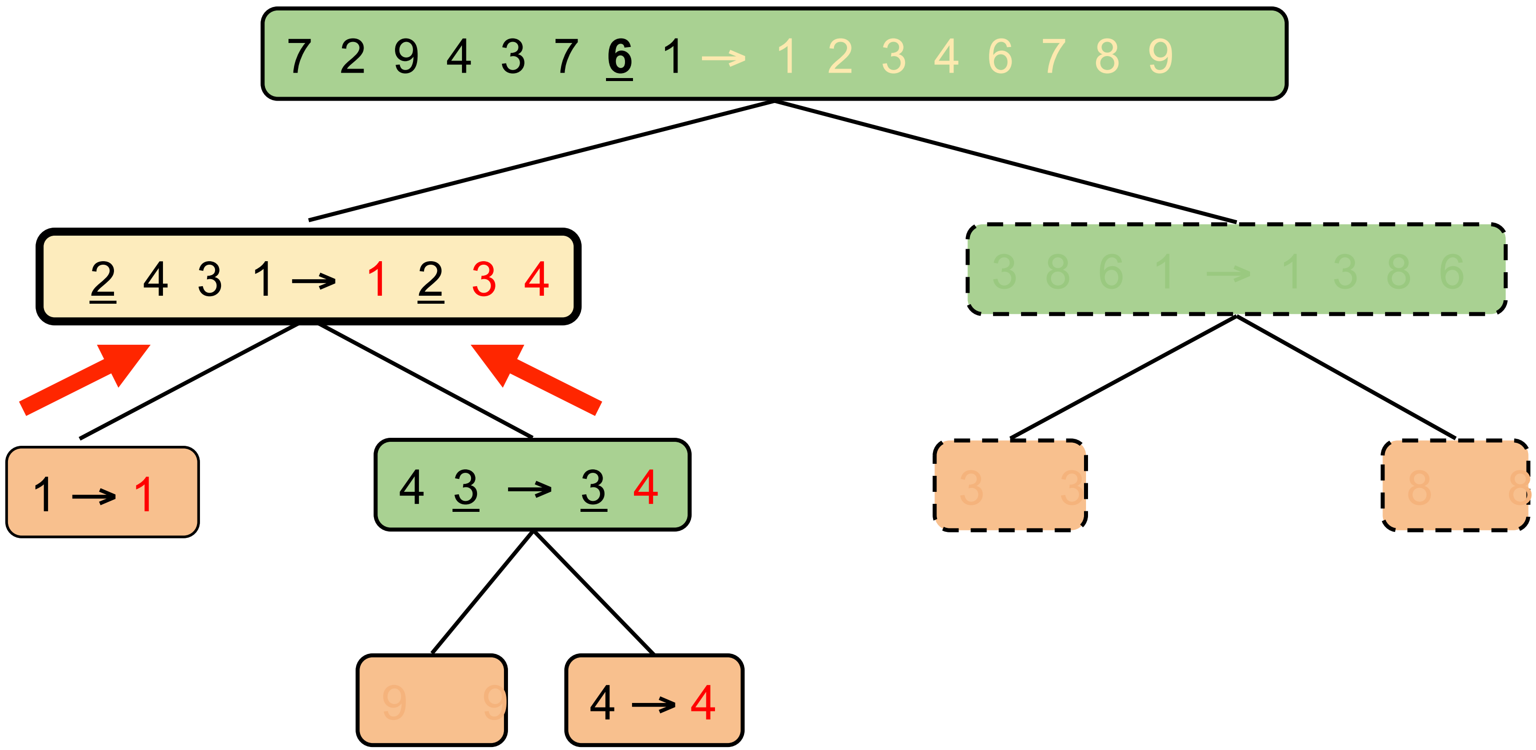
Execution Example (cont.)

- Partition, recursive call, base case



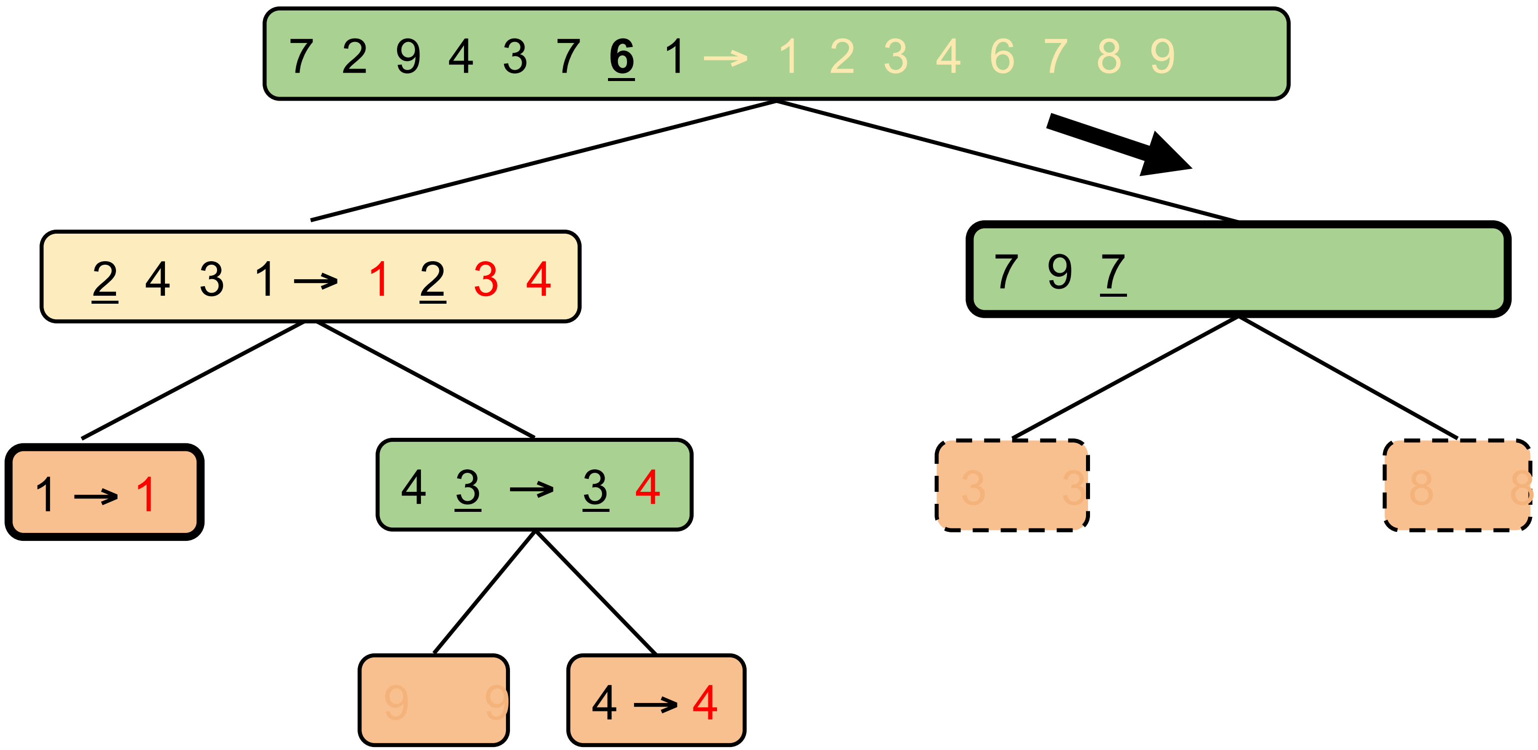
Execution Example (cont.)

- Recursive call, …, base case, join



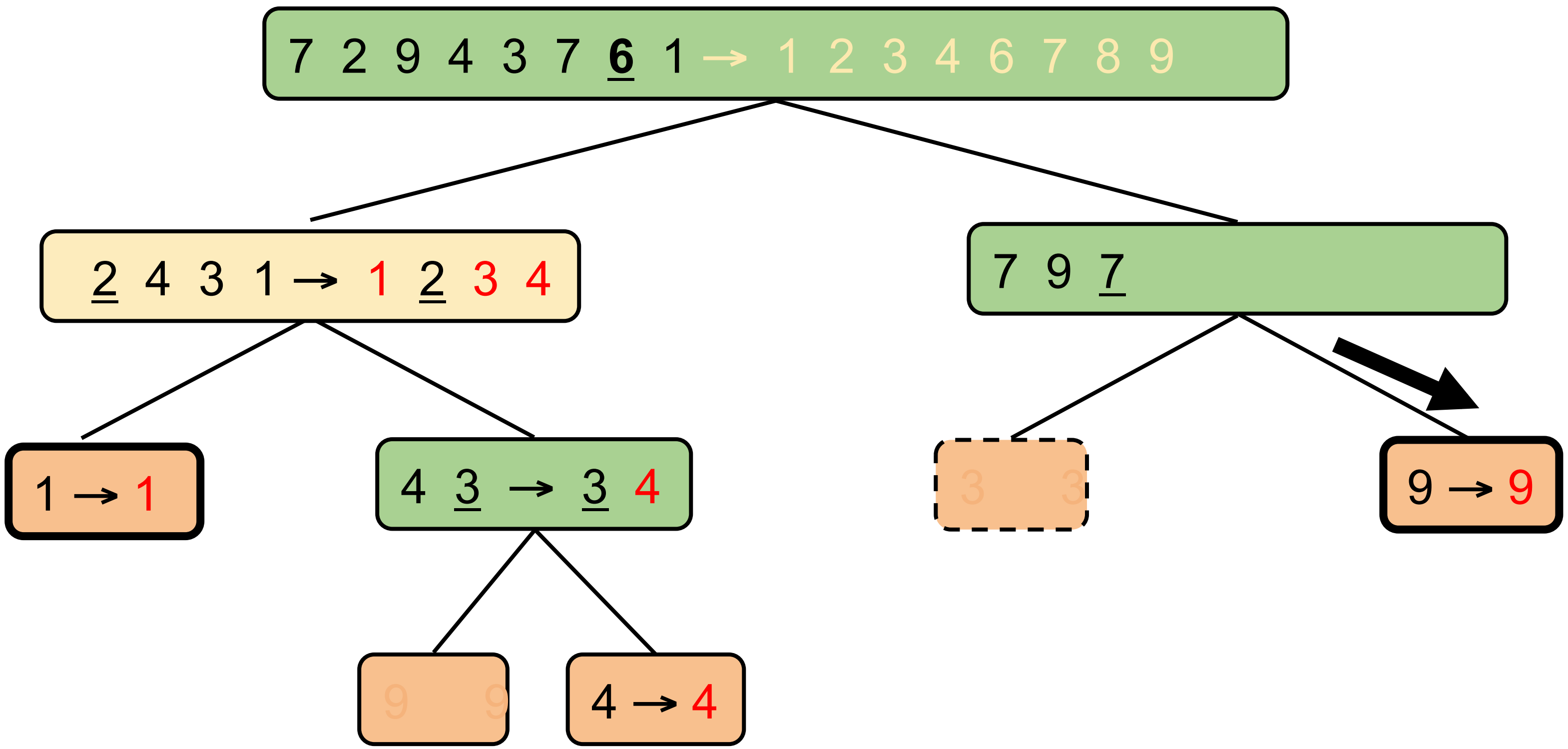
Execution Example (cont.)

- Recursive call, ..., base case, join



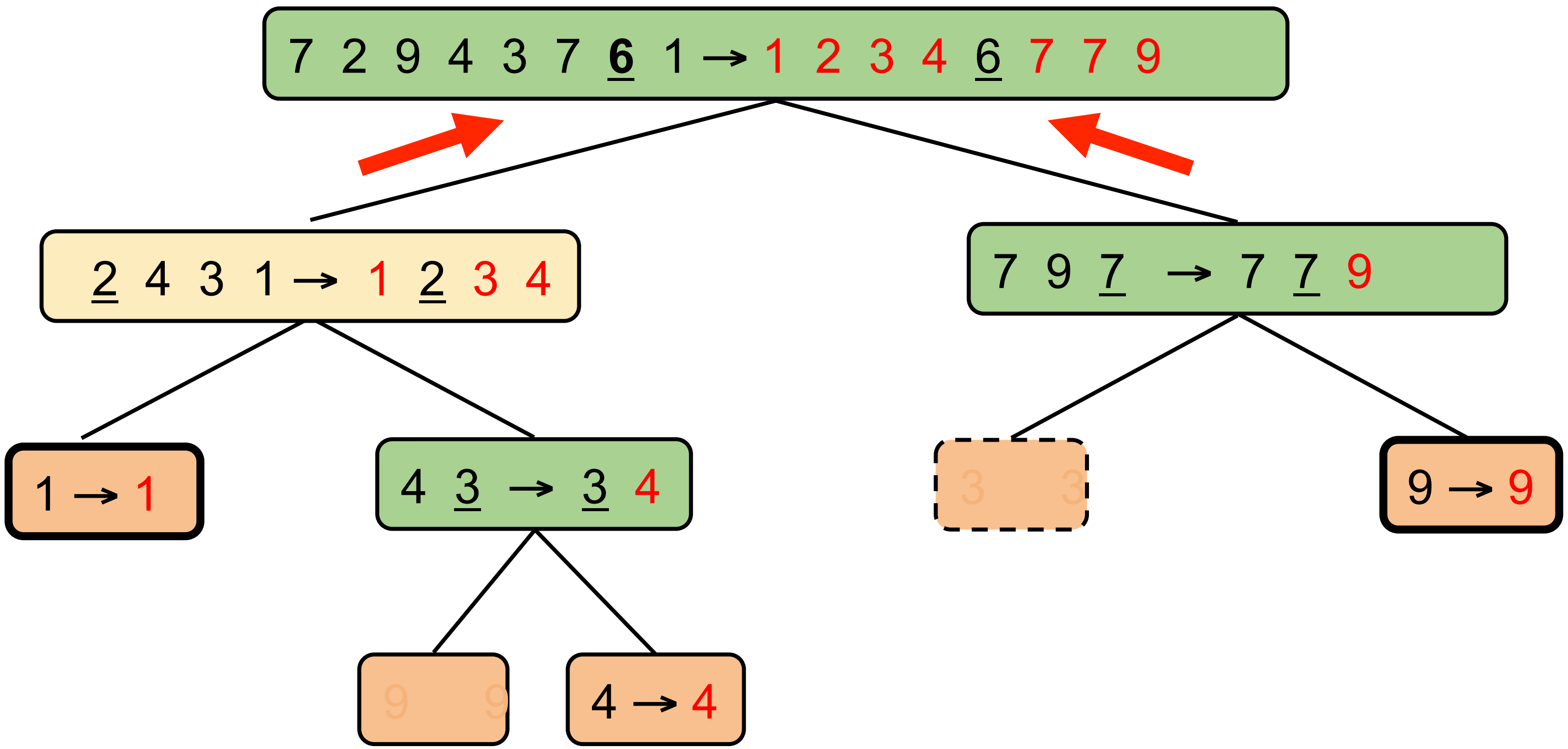
Execution Example (cont.)

- Partition, …, recursive call, base case

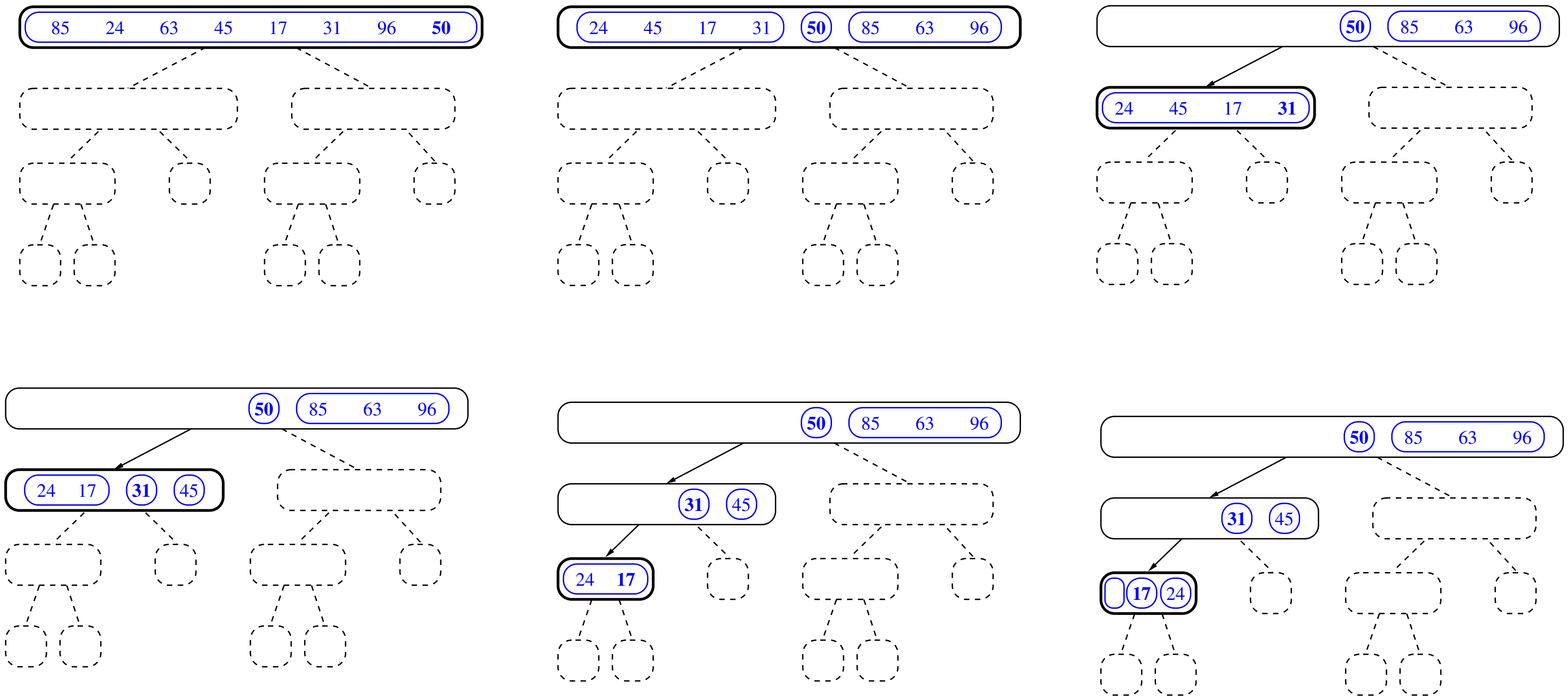


Execution Example (cont.)

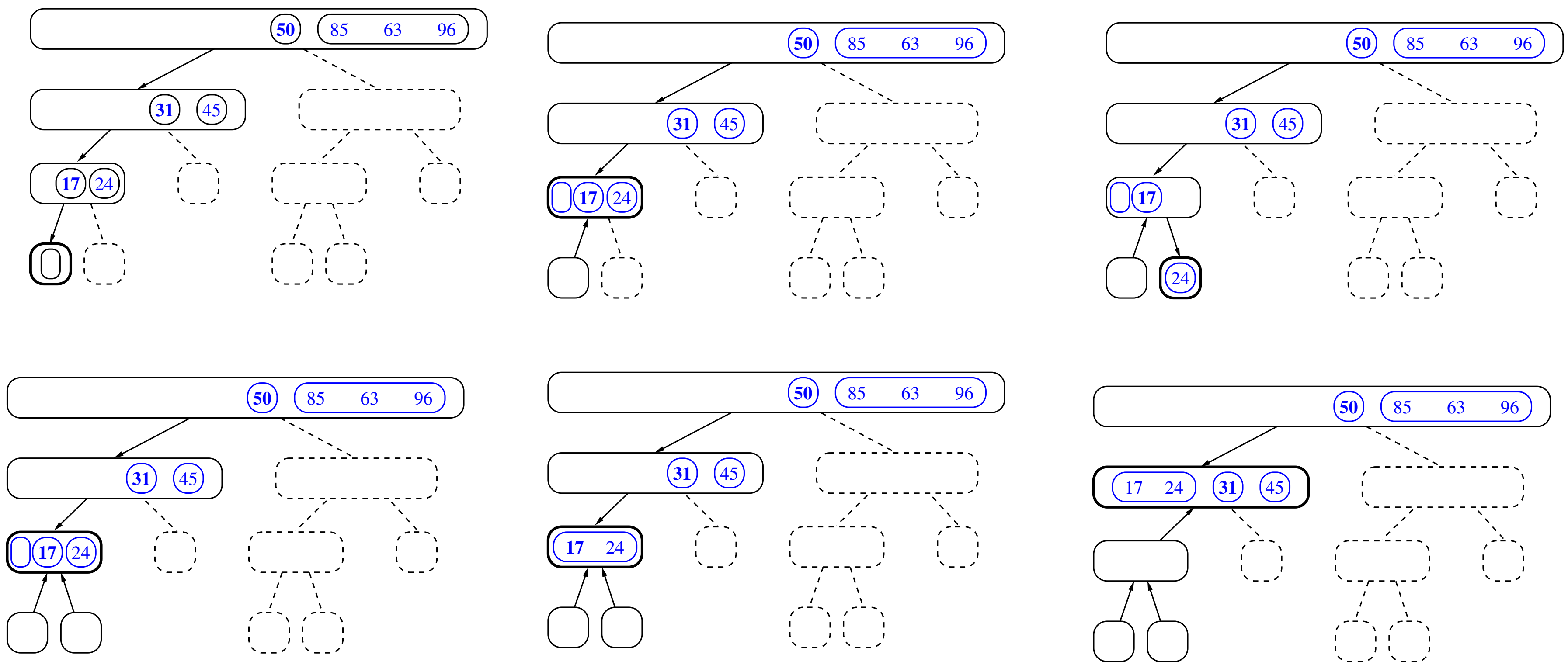
- Join, join



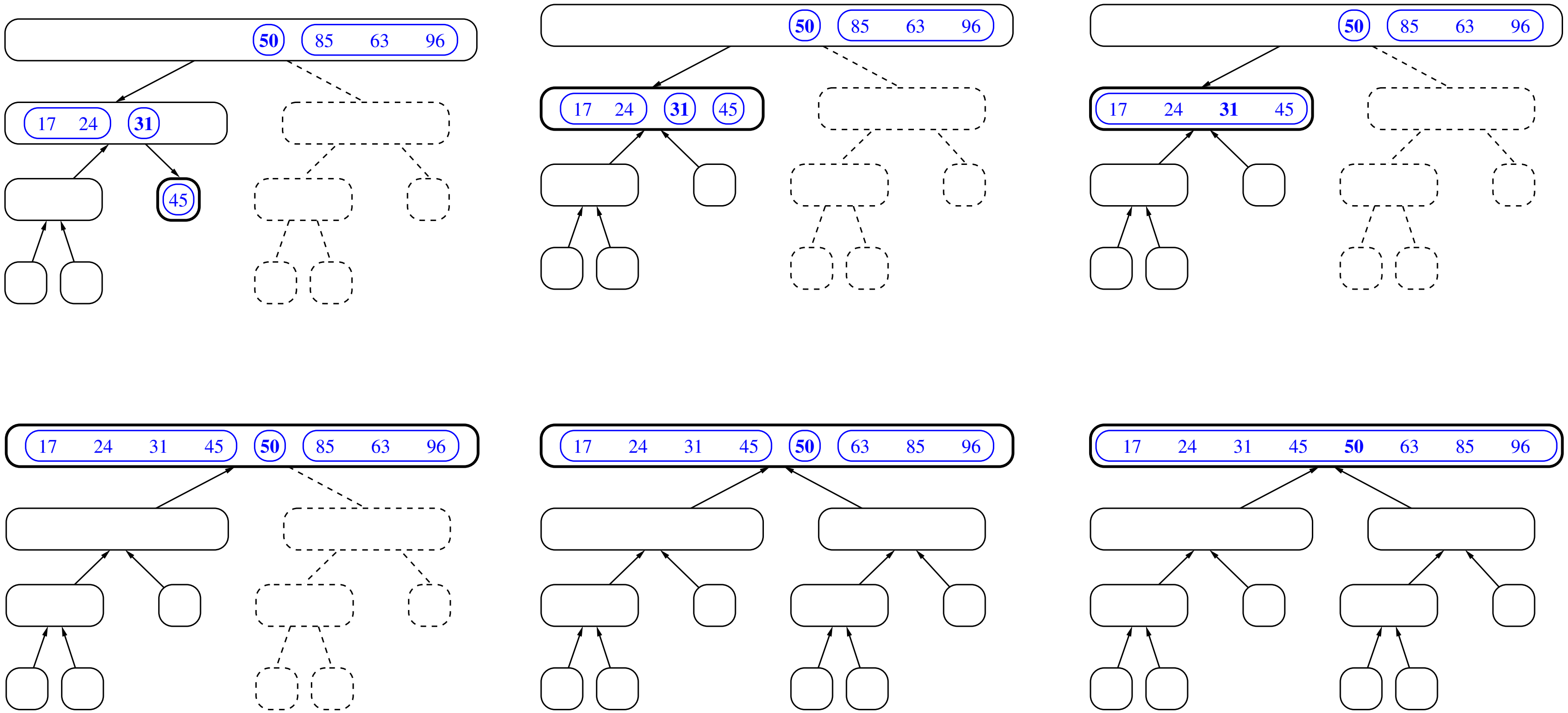
Example 2 (pivot is the last element)



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



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



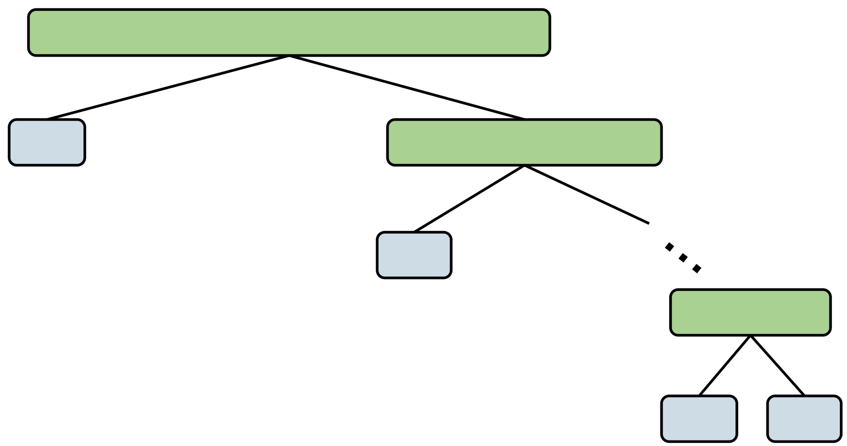
Worst-case Running Time

- 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

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

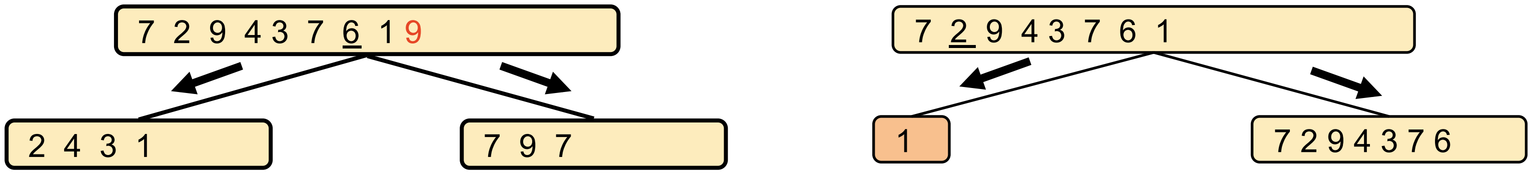
- Thus, the worst-case running time of quick-sort is $O(n^2)$

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



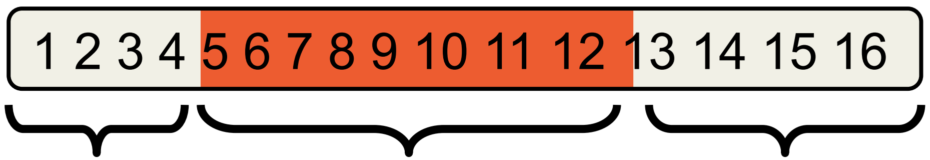
Expected Running Time

- 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:

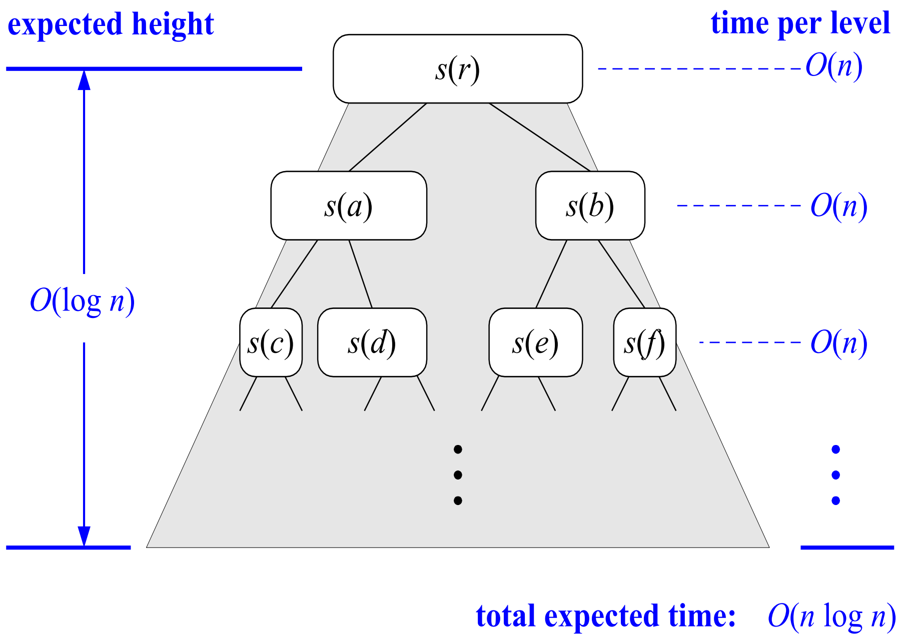


Bad pivots **Good pivots** **Bad pivots**

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 (probability is that every second call is a good call)
  - 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 $O(n \log n)$
- Randomised quick-sort: picks pivot randomly



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 $\geq$ 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 7 3 5 9 2 7 9 8 9 7 **6** 9

In-Place : divide step

85	24	63	45	17	31	96	**50**
l						r	

85	24	63	45	17	31	96	**50**
l					r		

31	24	63	45	17	85	96	**50**
	l			r			

31	24	63	45	17	85	96	**50**
		l		r			

31	24	17	45	63	85	96	**50**
			l, r				

31	24	17	45	63	85	96	**50**
			r $<$	l			

Put pivot in final place

31	24	17	45	**50**	85	96	63

Make recursive calls…

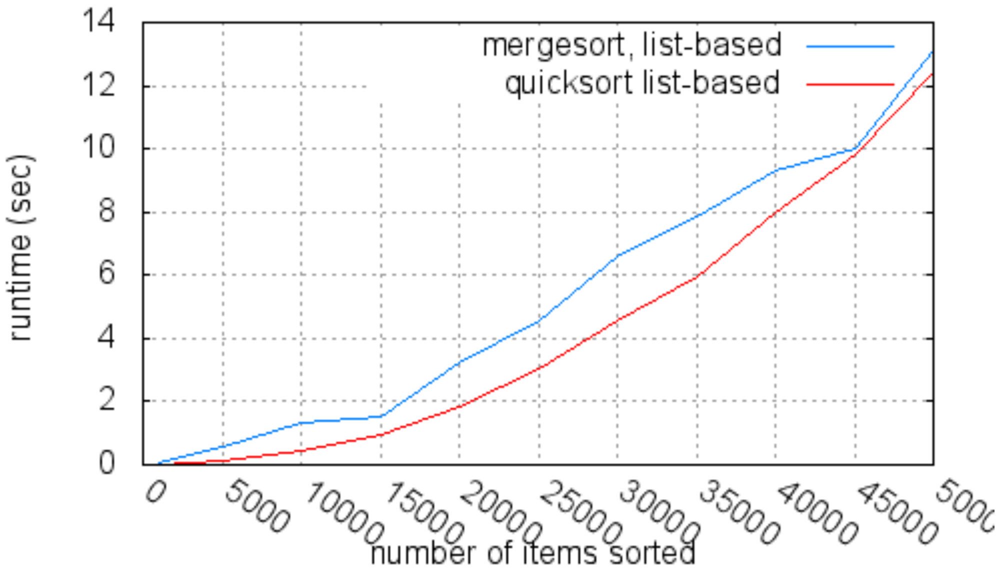
Java Implementation

```java
/** 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);
}
```

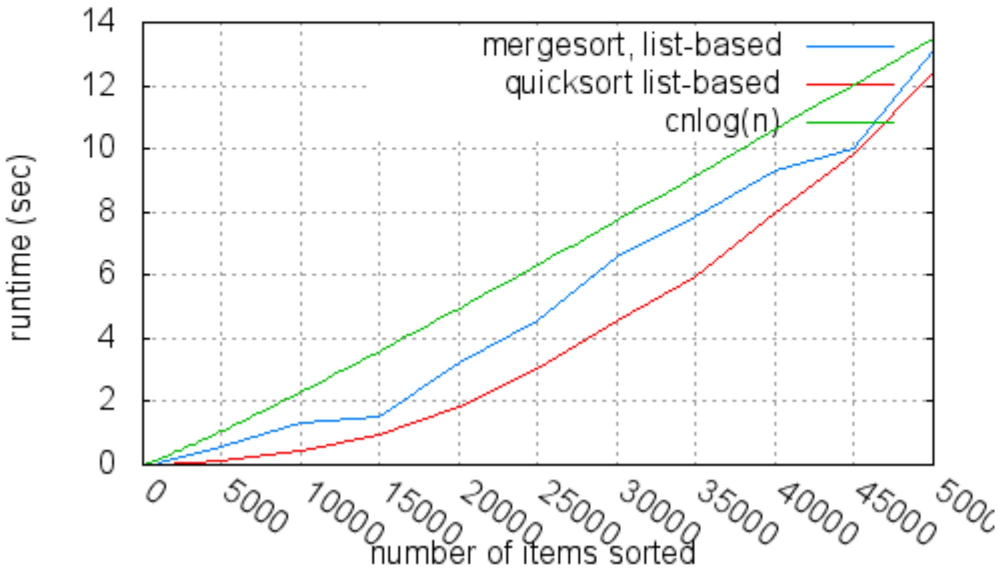
Runtimes (experimental)

- Comparisons of mergesort and quicksort (100,000 items)
  - Times (in seconds), List-Based
    - MSort: 55.47
    - QSort: 50.698

mergesort versus quicksort



mergesort versus quicksort



Running times, cpu time

Times (in seconds)

- Array-Based
  - MSort: 0.027 CPU time: 0.0156001
  - QSort: 0.129 CPU time: 0.1248008
- List-Based
  - MSort: 0.732 CPU time: 0.0936006
  - QSort: 0.436 CPU time: 0.0468003

Summary of Sorting Algorithms in this lecture

Algorithm	Time	Notes
selection-sort insertion-sort Bubble-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	▪ in-place, randomized ▪ fast (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)

Summary

- Read sections 12.1 and 12.2 of textbook (and review section 9.4)
- Sorting algorithms and their costs
  - selection-sort
  - insertion-sort
  - heap-sort
  - bubble-sort
  - merge sort
  - quick sort