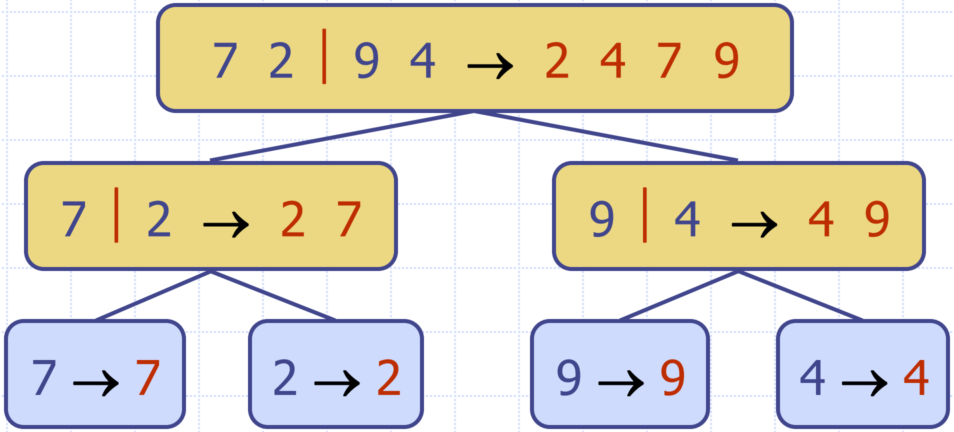
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Merge Sort



Aim and Learning Objectives

- To be able to *understand* and *describe* the merge-sort algorithm
- To be able to *analyze* the complexity of the merge-sort algorithm
- To be able to *implement* the merge-sort algorithm and *apply* it to solve problems

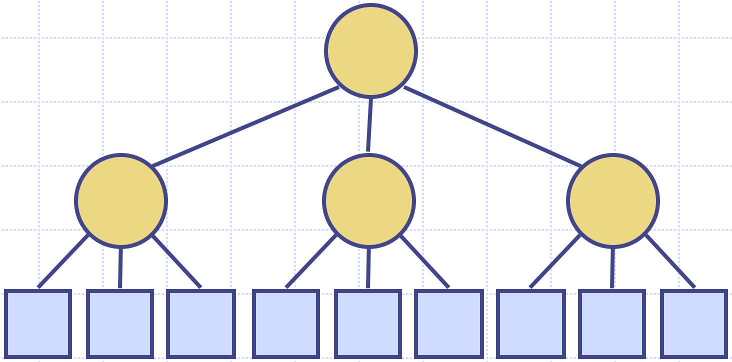
Reading

M. T. Goodrich, R. Tamassia and M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th Edition, 2014.

- **Chapter 13. Sorting and Selection**

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 subproblems recursively
  - Combine: combine the solutions for S_1, S_2, …, into a solution for S
- The base case for the recursion are subproblems of constant size



在分治法中，主要有三个步骤：
Divide（分解）：将输入数据 S 分成两个或多个不重叠的子集 S_1, S_2, \dots，这个步骤是将问题规模逐步缩小的过程。**Conquer（解决）**：递归地解决这些子问题。在这
个步骤中，每个子问题通常会继续分解，直到子问题足够简单，可以直接解决。
Combine（合并）：将各个子问题的解合并成最终问题的解。合并步骤是将分解后的部分结果结合成一个完整的答案。
递归的基本情况是：当子问题的规模足够小，无法继续分解时，算法会停止递归，并直接解决这些最小的子问题。

优先队列中的键可以是任意对象：在优先队列中，键（key）不一定是基本数据类型，它可以是任意对象，只要这些对象上定义了顺序关系。
不同的条目可以有相同的键：在优先队列中，不同的条目可以有相同的键，这意味着这些条目在优先队列中的优先级是相同的。
总序关系的数学概念：
可比性属性（Comparability property）：对于任意两个元素 x 和 y，必须满足其中一个关系：x \leq y 或 y \leq x，即任何两个元素总是可以比较大小。

Total Order Relations

反对称属性（Antisymmetric property）：如果 x \leq y 且 y \leq x，则必须有 x = y。也就是说，如果两个元素互相比较小且相等，那么它们实际上是同一个元素。

- Keys in a priority queue can be arbitrary objects on which an order is defined.
- Two distinct entries in a priority queue can have the same key.

- Mathematical concept of total order relation $\leq$
  - Comparability property: either $x \leq y$ or $y \leq x$
  - Antisymmetric property: $x \leq y$ and $y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y$ and $y \leq z \Rightarrow x \leq z$

传递性属性（Transitive property）：如果 x \leq y 且 y \leq z，那么必然有 x \leq z。即如果第一个元素小于第二个，第二个小于第三个，那么第一个元素一定小于第三个。

Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation.
- A generic priority queue uses an auxiliary comparator.
- The comparator is external to the keys being compared.
- When the priority queue needs to compare two keys, it uses its comparator.

- Primary method of the Comparator ADT
- compare(a, b): returns an integer i such that
  - i < 0 if a < b,
  - i = 0 if a = b,
  - i > 0 if a > b.
  - An error occurs if a and b cannot be compared.

Example Comparator

- Lexicographic comparison of 2D points:

```
/** Comparator for 2D points under the
    standard lexicographic order. */
public class Lexicographic implements
    Comparator {
  int xa, ya, xb, yb;
  public int compare(Object a, Object b)
   throws ClassCastException {
    xa = ((Point2D) a).getX();
    ya = ((Point2D) a).getY();
    xb = ((Point2D) b).getX();
    yb = ((Point2D) b).getY();
    if (xa != xb)
          return (xb - xa);
    else
          return (yb - ya);
  }
}
```

- Point objects:

```
/** Class representing a point in the
    plane with integer coordinates */
public class Point2D {
   protected int xc, yc; // coordinates
   public Point2D(int x, int y) {
     xc = x;
     yc = y;
   }
   public int getX() {
        return xc;
   }
   public int getY() {
        return yc;
   }
}
```

归并排序的步骤：
分解（Divide）：将输入序列 S 分成两个子序列 S_1 和 S_2，每个子序列大约包含 n/2 个元素。
递归（Recur）：递归地对这两个子序列 S_1 和 S_2 进行排序。
合并（Conquer）：将排好序的 S_1 和 S_2 合并成一个唯一的排序序列。

Merge-Sort

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

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

C is a comparator.

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

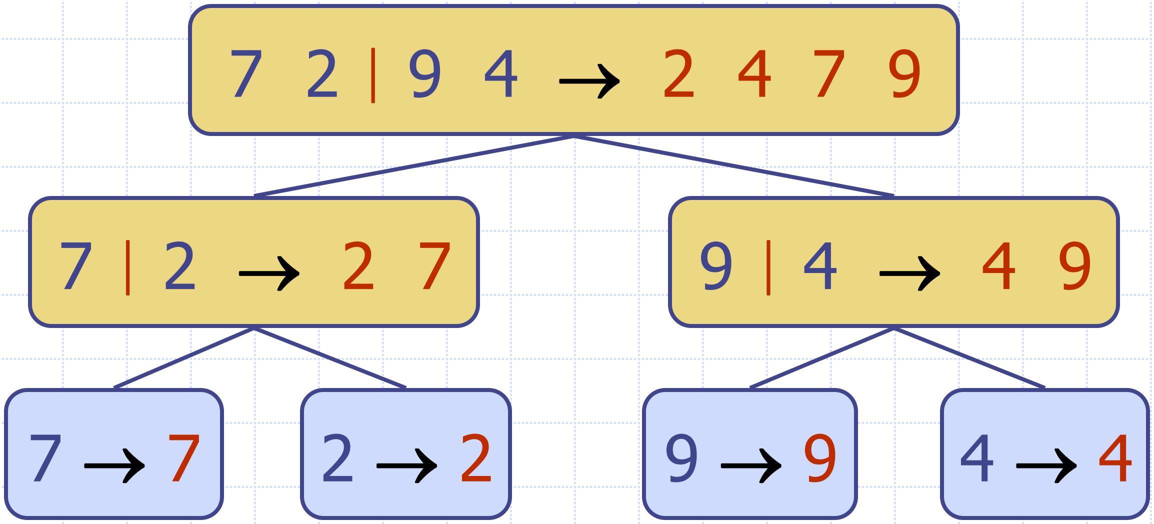
Algorithm *merge*(A, B)
 Input sequences A and B with
  $n/2$ elements each
 Output sorted sequence of $A \cup B$

 $S \leftarrow$ empty sequence
 while $\neg A.isEmpty() \land \neg B.isEmpty()$
  if $A.first().element() < B.first().element()$
   $S.addLast(A.remove(A.first()))$
  else
   $S.addLast(B.remove(B.first()))$
 while $\neg A.isEmpty()$
  $S.addLast(A.remove(A.first()))$
 while $\neg B.isEmpty()$
  $S.addLast(B.remove(B.first()))$
 return S

Merge-Sort Tree

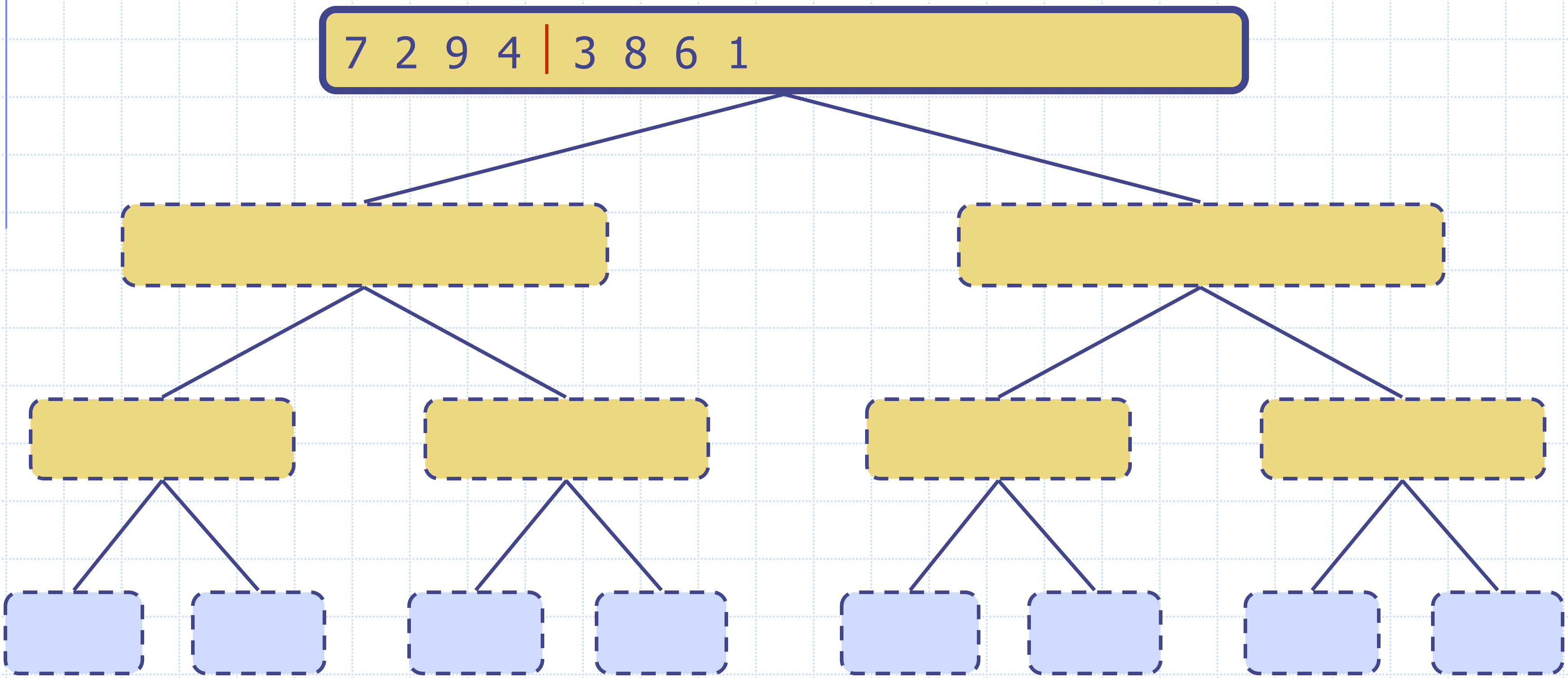
- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences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

未排序的序列：在执行之前，节点上会显示未排序的序列和该序列的分割。
排序后的序列：执行完后，节点上会显示排序后的序列。



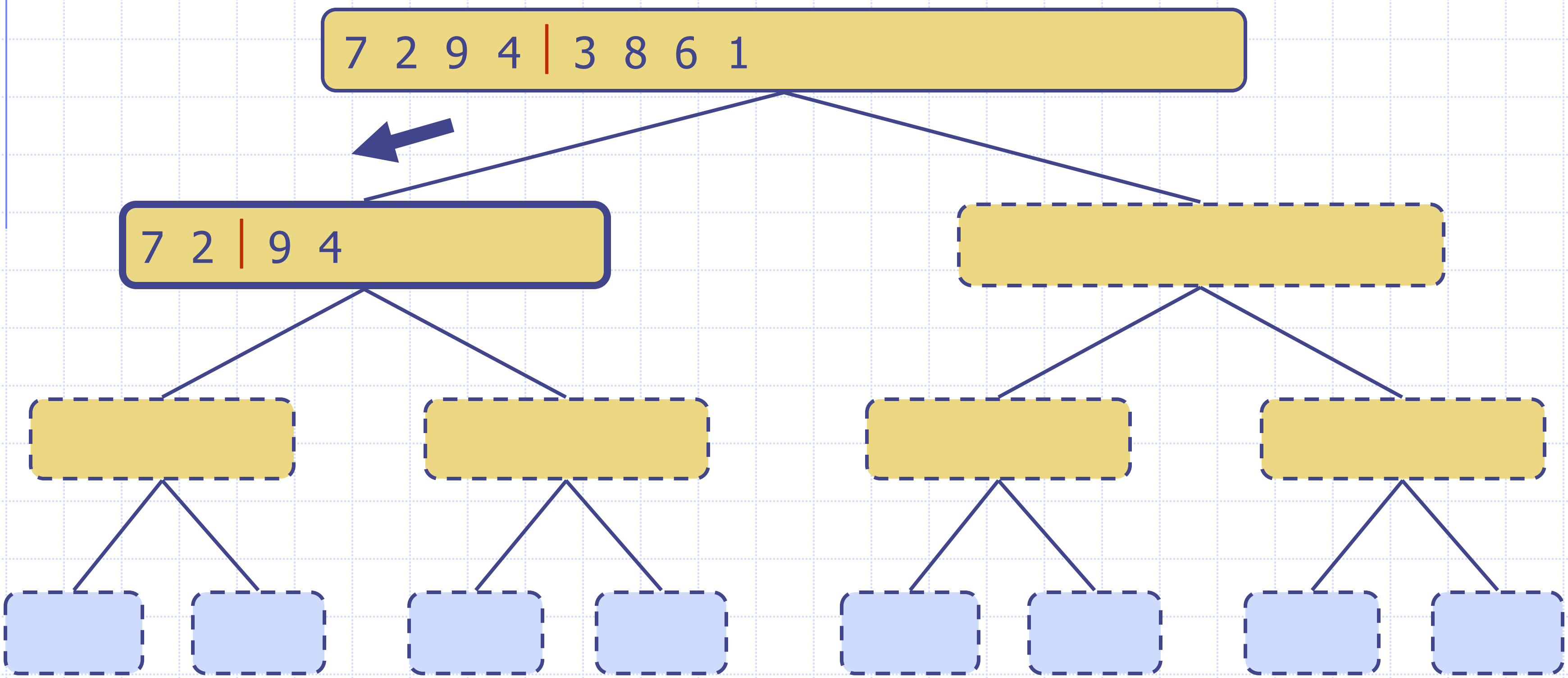
Execution Example

- Partition



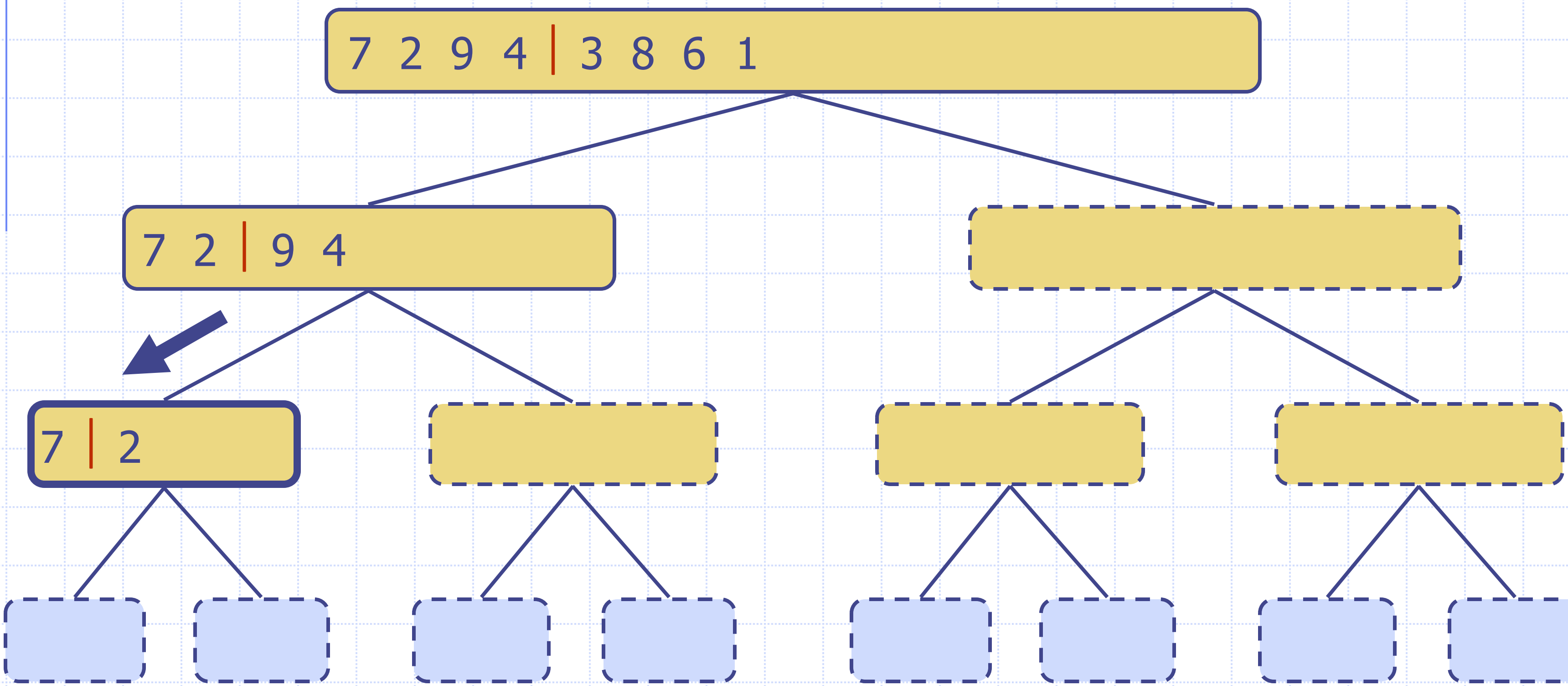
Execution Example (cont.)

- Recursive call, partition



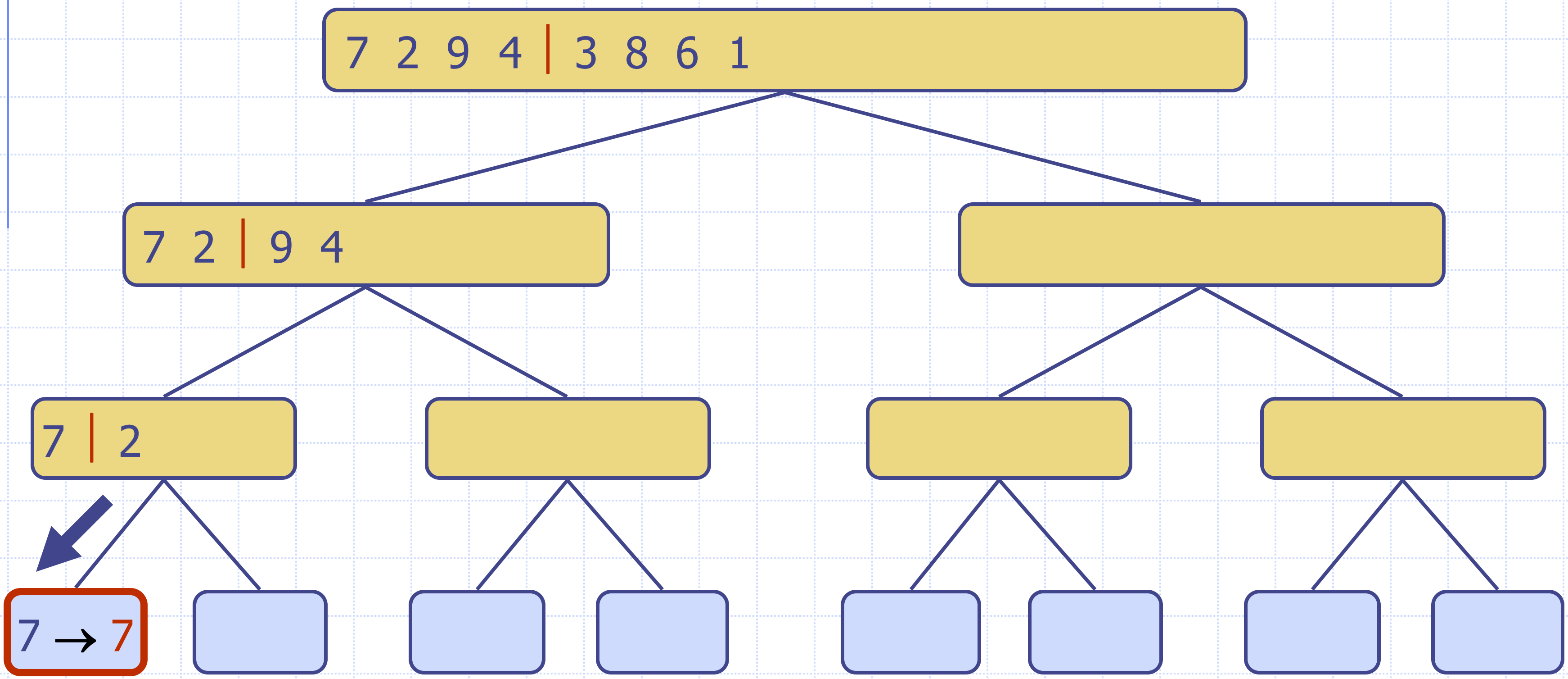
Execution Example (cont.)

- Recursive call, partition



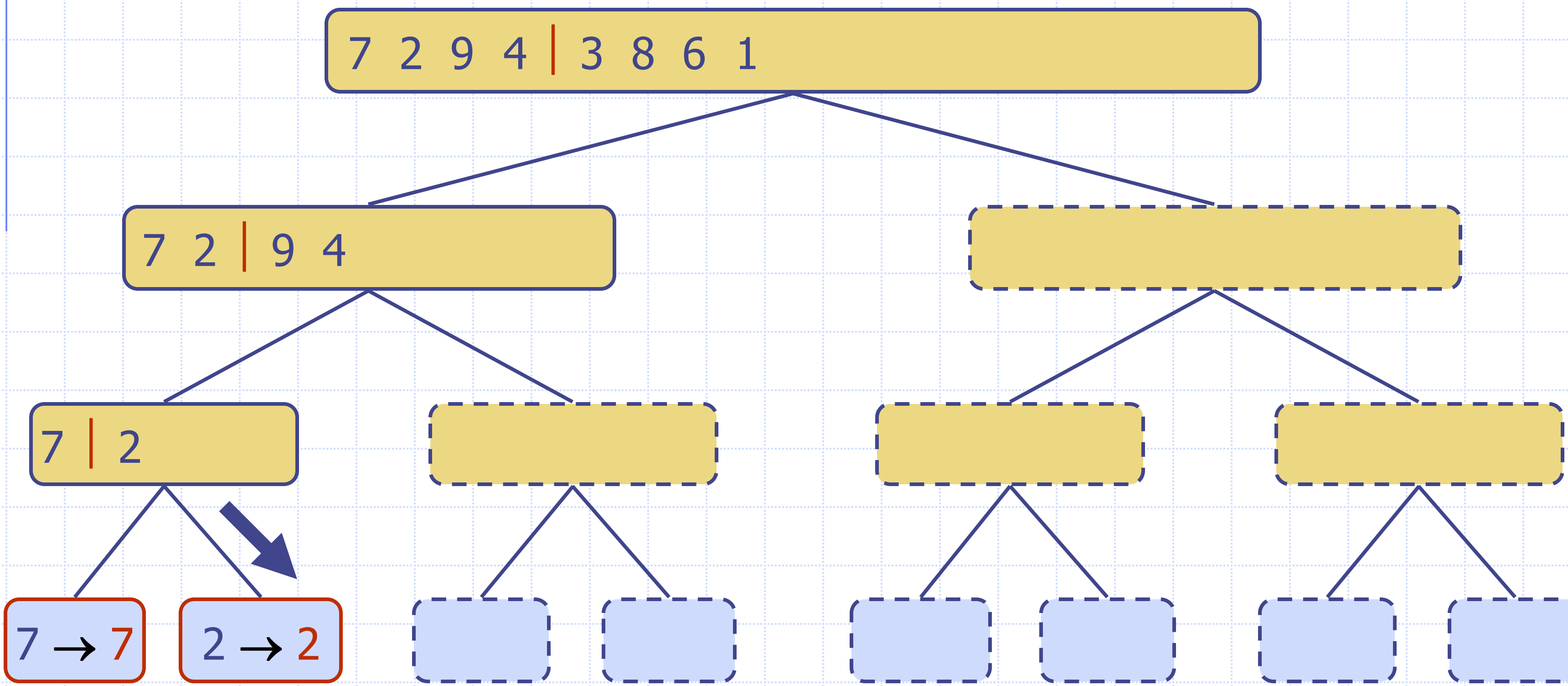
Execution Example (cont.)

- Recursive call, base case



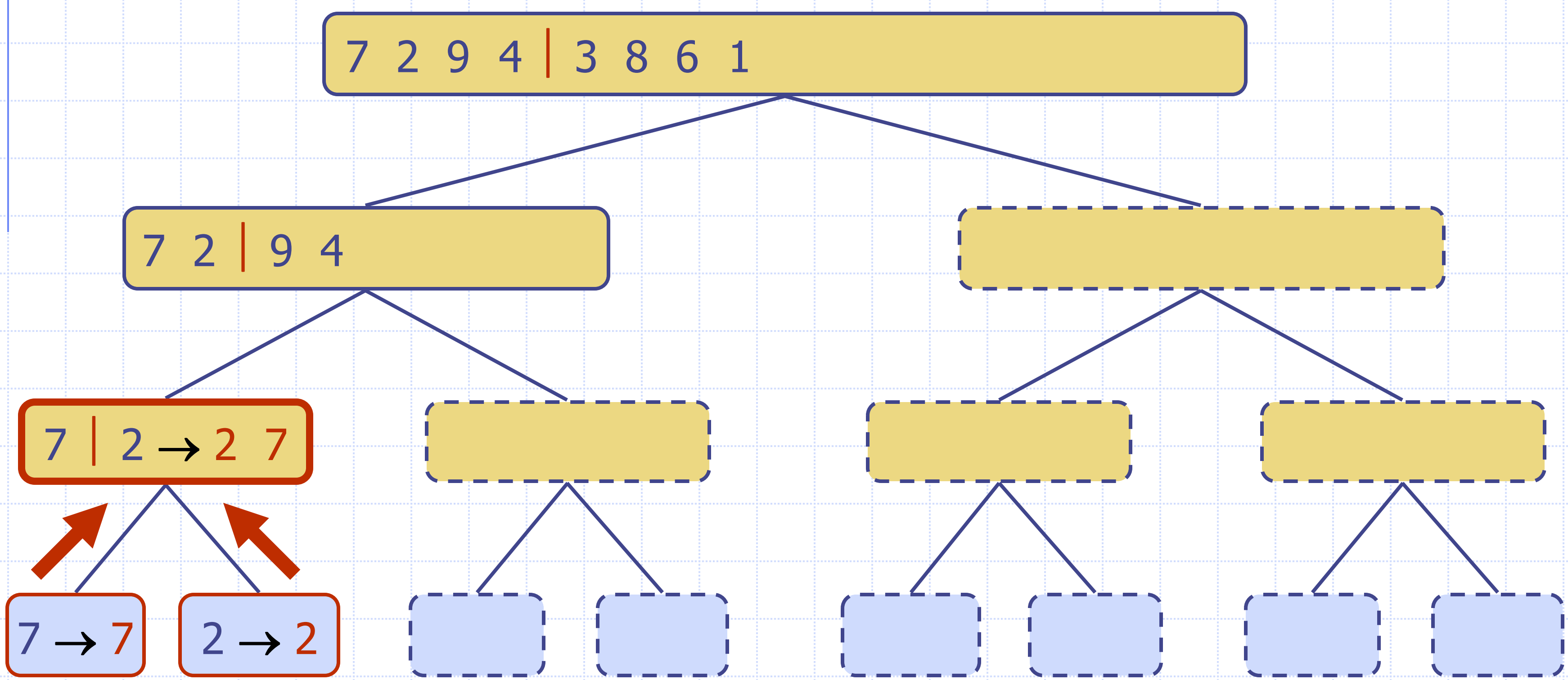
Execution Example (cont.)

- Recursive call, base case



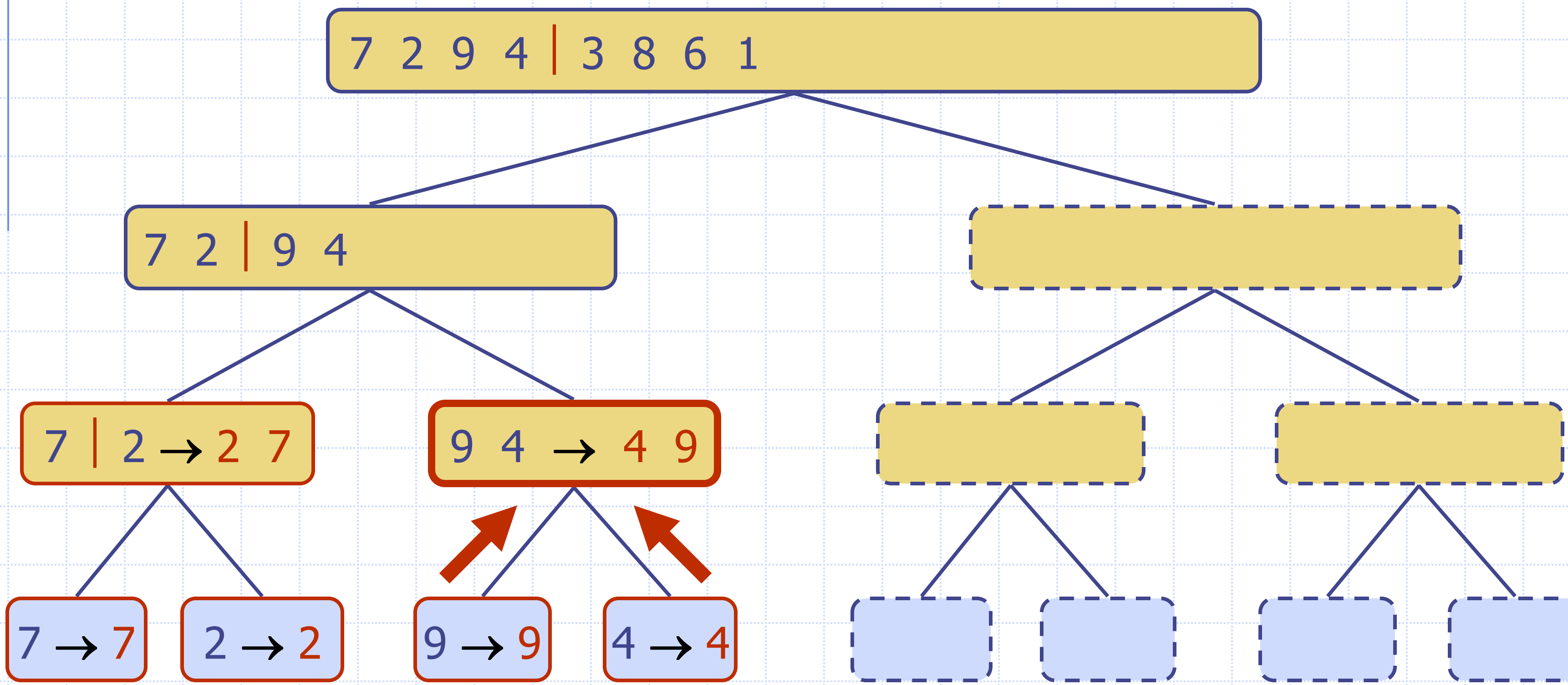
Execution Example (cont.)

- Merge



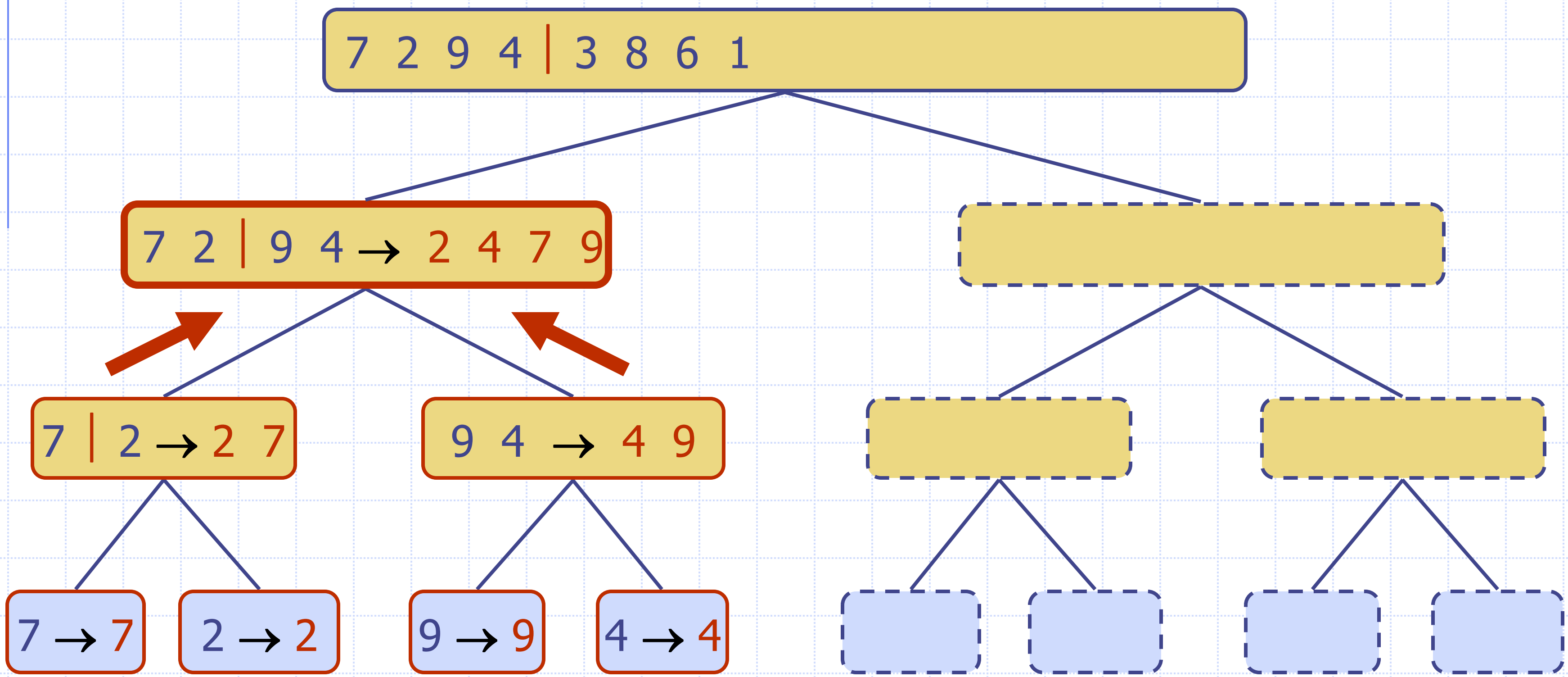
Execution Example (cont.)

- Recursive call, ..., base case, merge



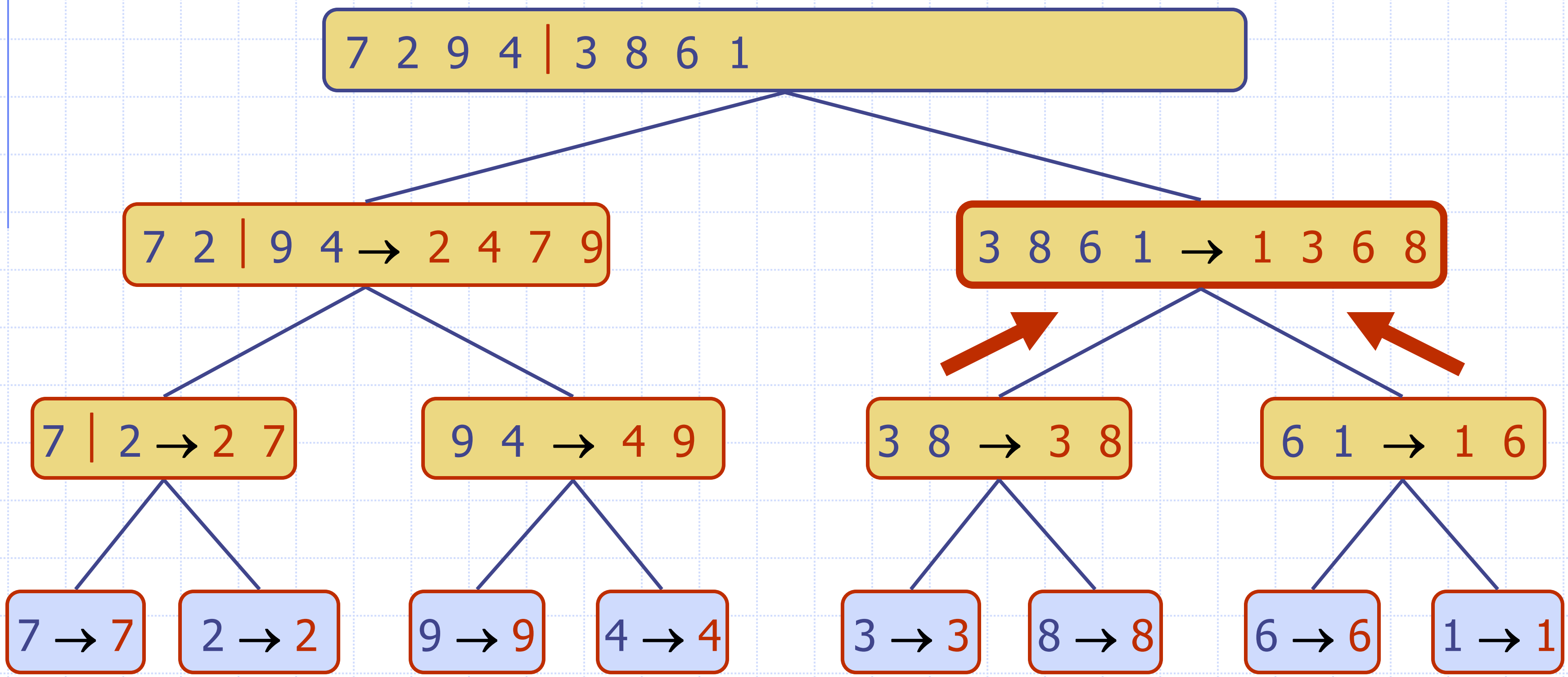
Execution Example (cont.)

- Merge



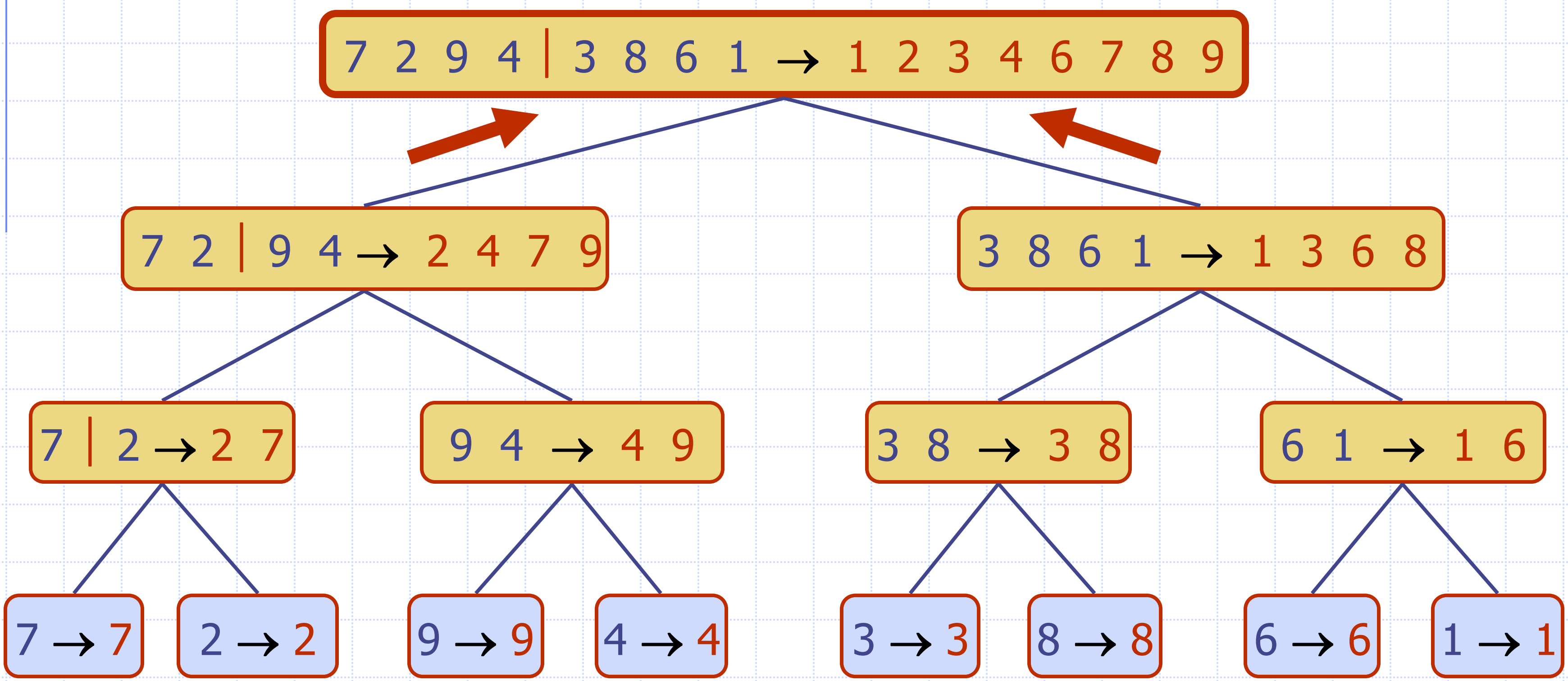
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

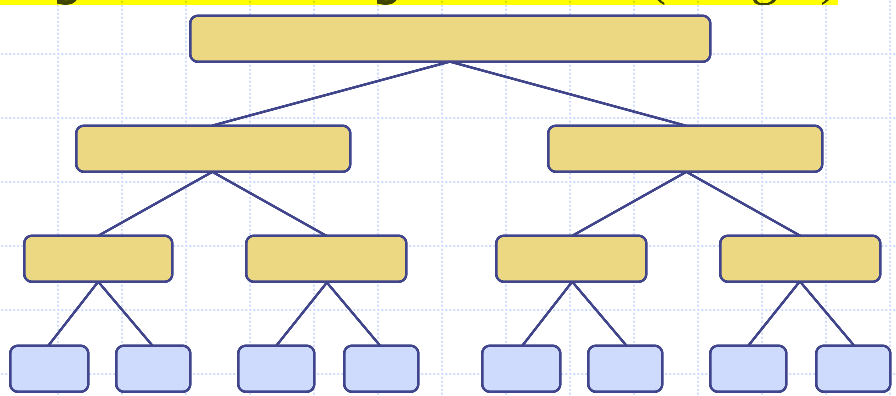
- Merge



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
  - Merge runs in $O(n)$
  - Partition 2^i partitions , $i \le h \le \log n$, $2^i \le 2^h \le 2^{\log n} = n$
- Thus, the total running time of merge-sort is $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
insertion-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)

Reading

M. T. Goodrich, R. Tamassia and M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th Edition, 2014.

- **Chapter 13. Sorting and Selection**