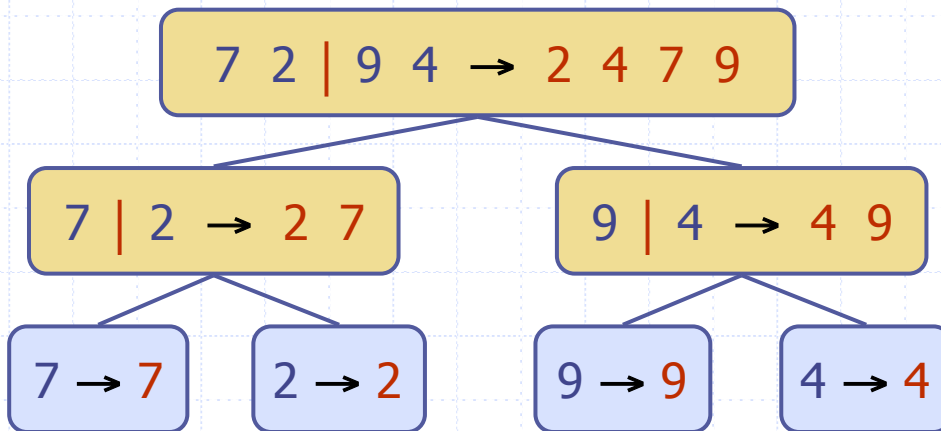


# Recursive Sorting



# 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  $S_2$
  - **Recurse**: 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

# 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
  - **Recurse**: recursively sort  $S_1$  and  $S_2$
  - **Conquer**: merge  $S_1$  and  $S_2$  into sorted sequence

```
mergeSort(S)  
  if  $S.size() \leq 1$   
    return  
  
   $(S_1, S_2) = partition(S, 2)$   
  mergeSort(S_1)  
  mergeSort(S_2)  
   $S = merge(S_1, S_2)$ 
```

# 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 takes  $O(n)$  time

*merge(A, B)*

$S$  = array of size  $A.length + B.length$

sIndex = 0

aIndex = 0

bIndex = 0

**while** aIndex <  $A.length$  and bIndex <  $B.length$

**if**  $A[aIndex] < B[bIndex]$

$S[sIndex++] = A[aIndex++]$

**else**

$S[sIndex++] = B[bIndex++]$

**while** aIndex <  $A.length$

$S[sIndex++] = A[aIndex++]$

**while** bIndex <  $B.length$

$S[sIndex++] = B[bIndex++]$

# 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

*merge(A, B)*

$S = \text{ArrayList of size } A.\text{size}() + B.\text{size}()$

**while**  $A.\text{isEmpty}() == \text{false}$  and  $B.\text{isEmpty}() == \text{false}$

**if**  $A.\text{get}(0) < B.\text{get}(0)$

$S.\text{add}(A.\text{remove}(0))$

**else**

$S.\text{add}(B.\text{remove}(0))$

**while**  $A.\text{isEmpty}() == \text{false}$

$S.\text{add}(A.\text{remove}(0))$

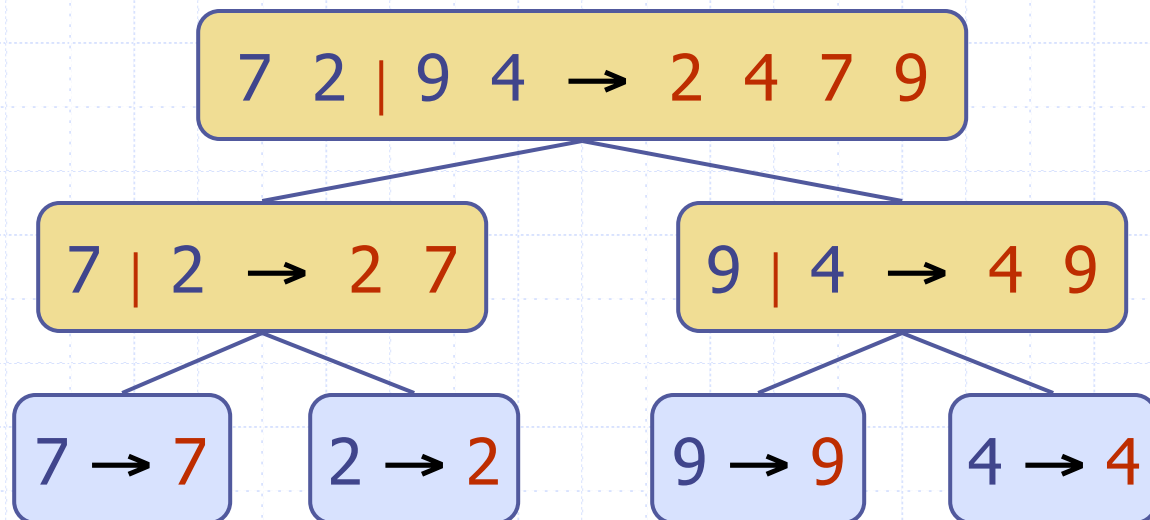
**while**  $B.\text{isEmpty}() == \text{false}$

$S.\text{add}(B.\text{remove}(0))$

**return**  $S$

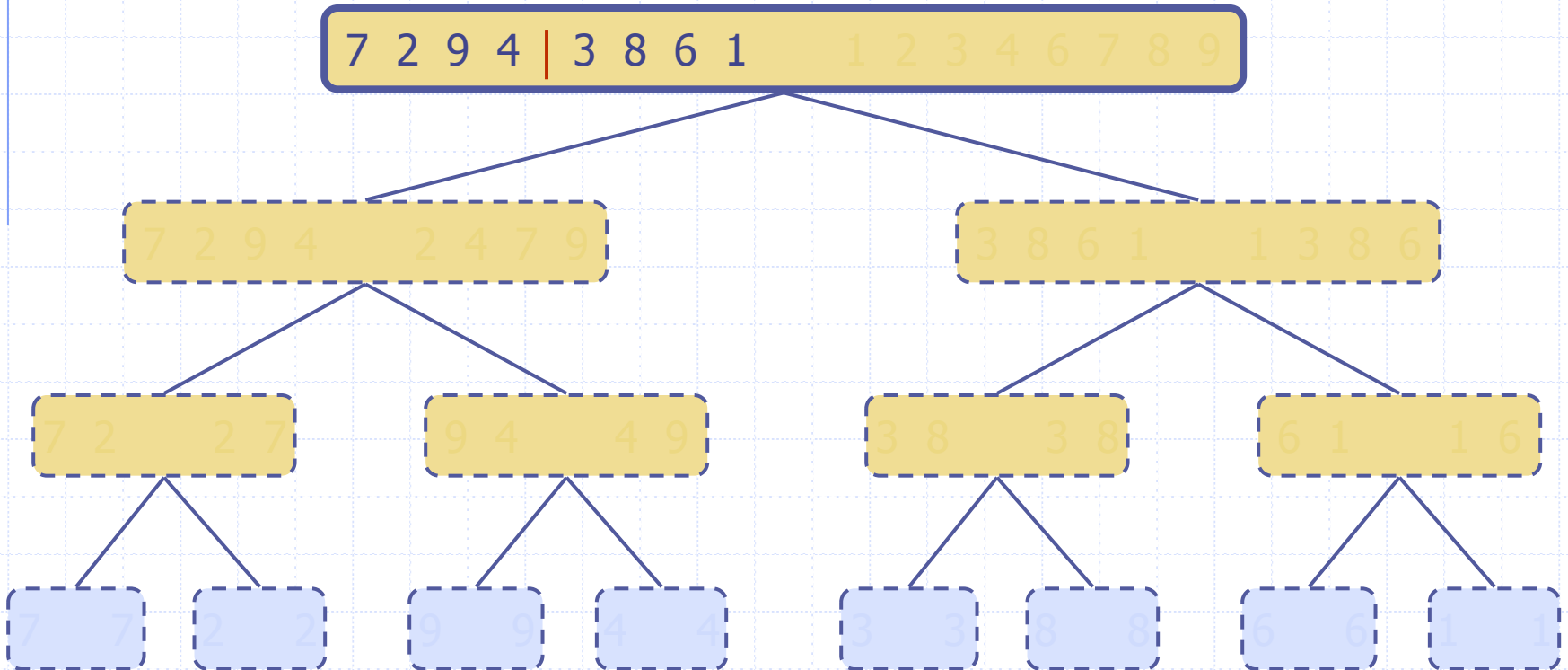
# Merge-Sort Tree

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



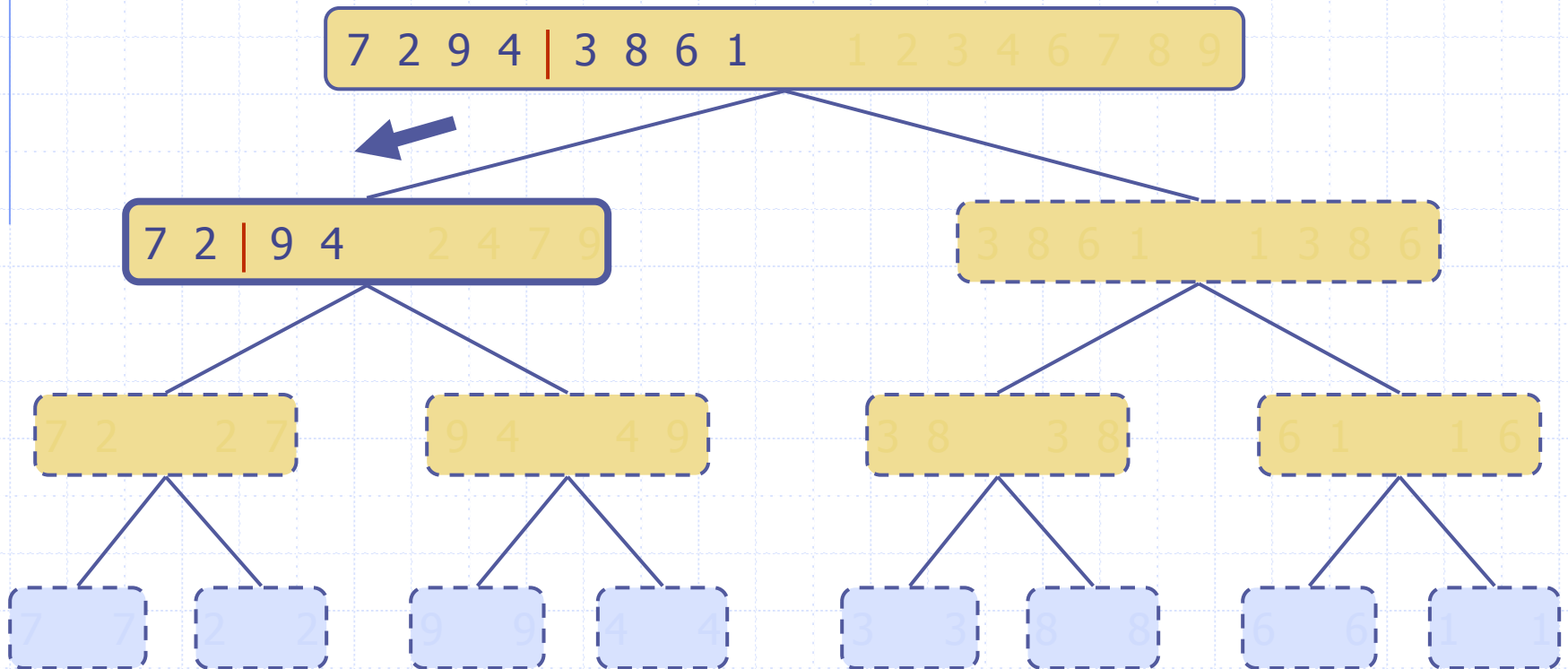
# Execution Example

## ◆ Partition



# Execution Example (cont.)

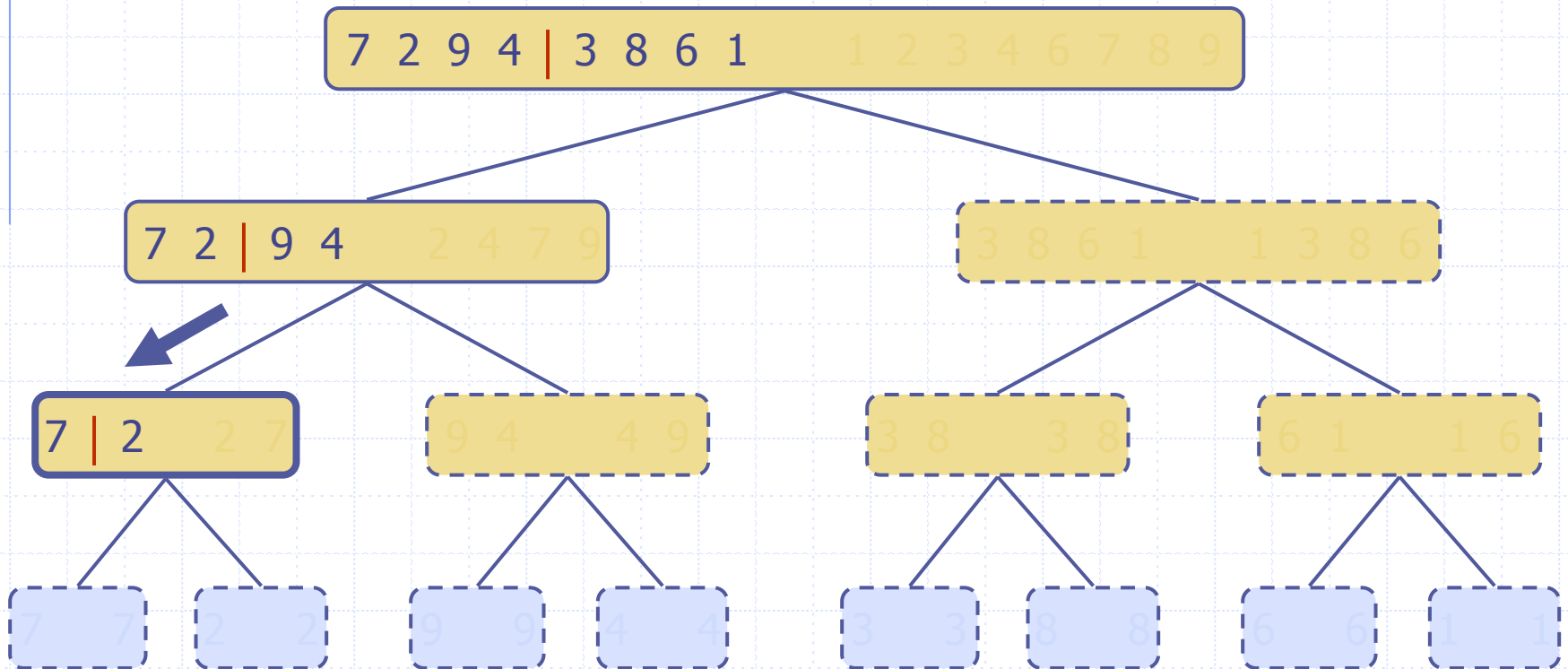
◆ Recursive call, partition





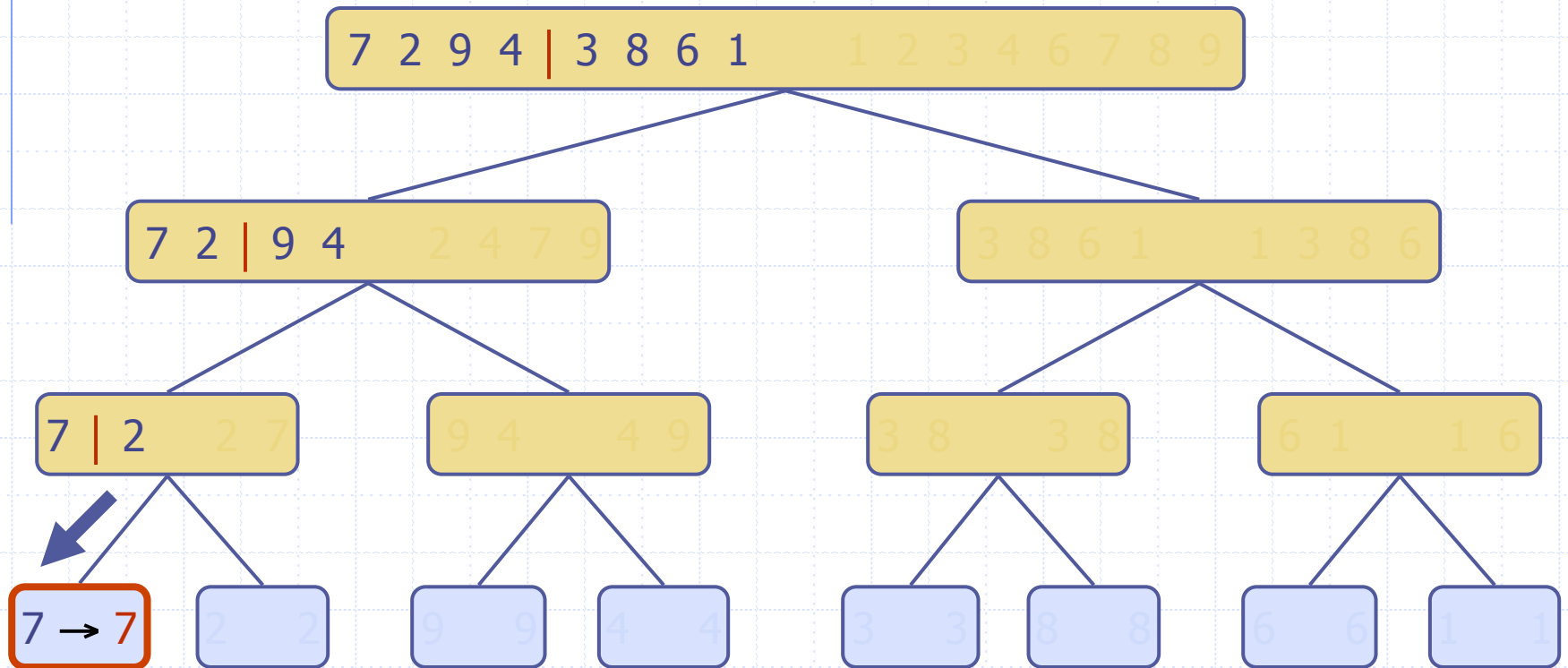
# Execution Example (cont.)

## ◆ Recursive call, partition



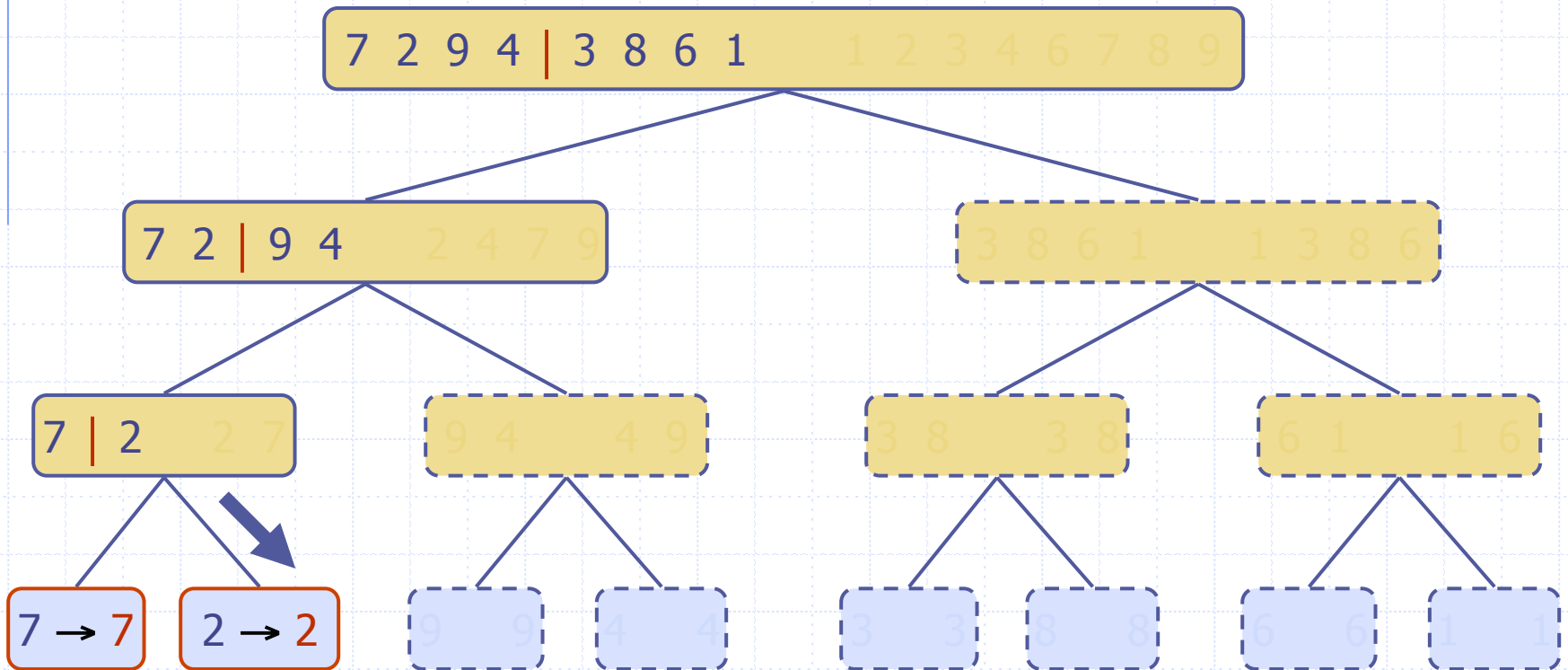
# Execution Example (cont.)

◆ Recursive call, base case



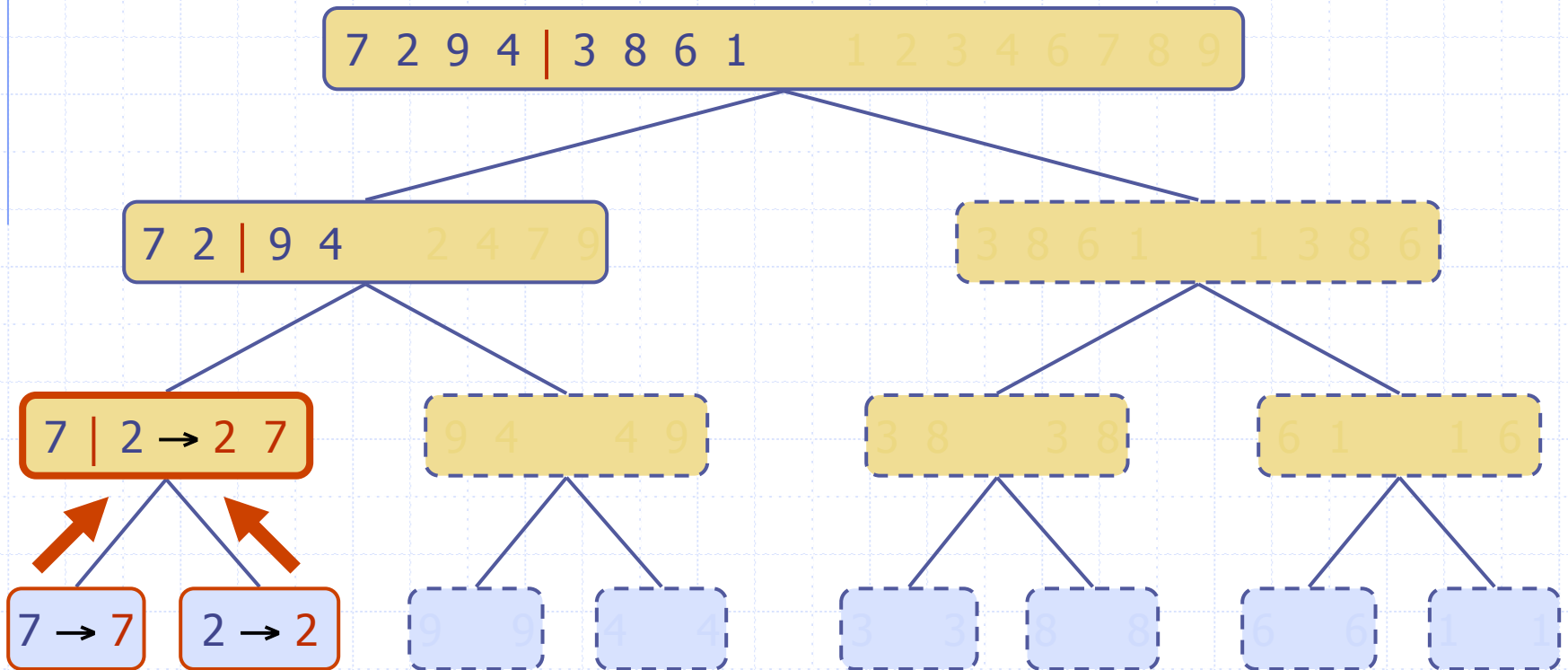
# Execution Example (cont.)

◆ Recursive call, base case



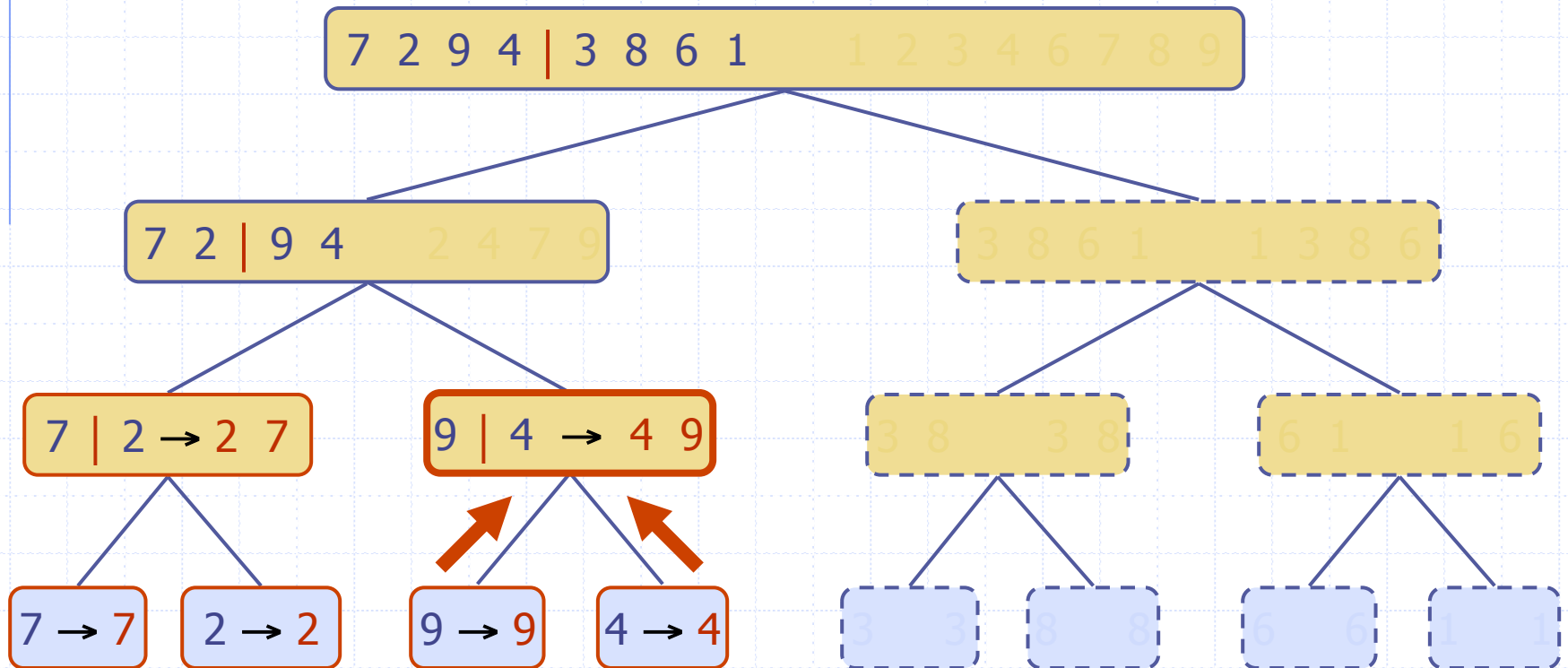
# Execution Example (cont.)

## ◆ Merge



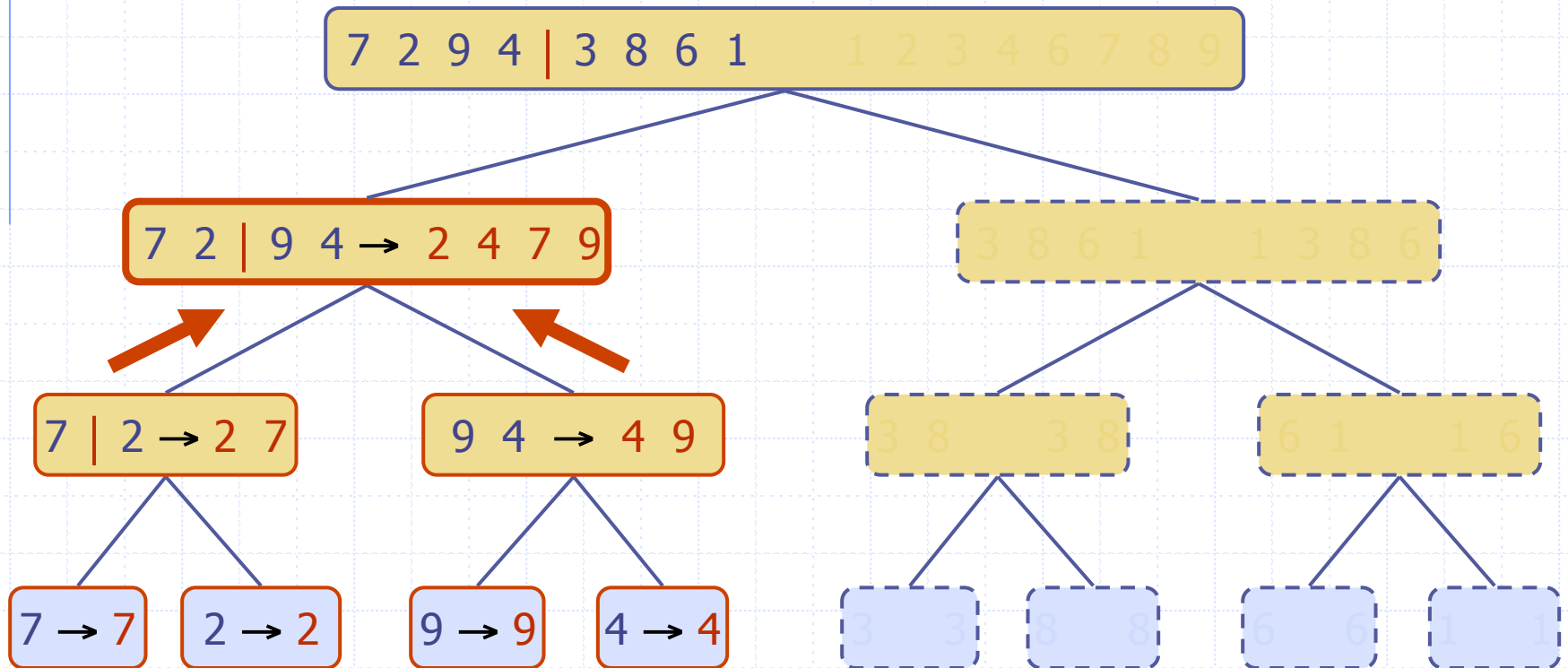
# Execution Example (cont.)

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



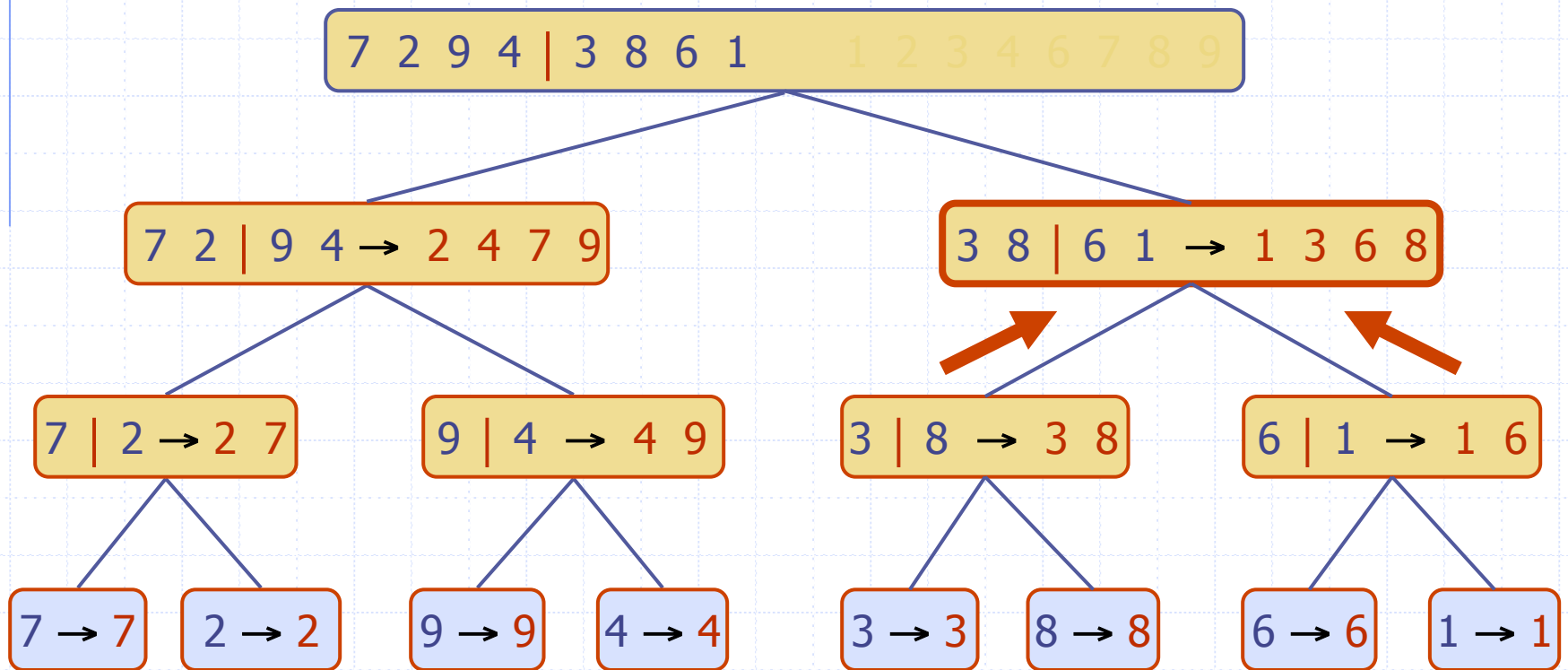
# Execution Example (cont.)

## ◆ Merge



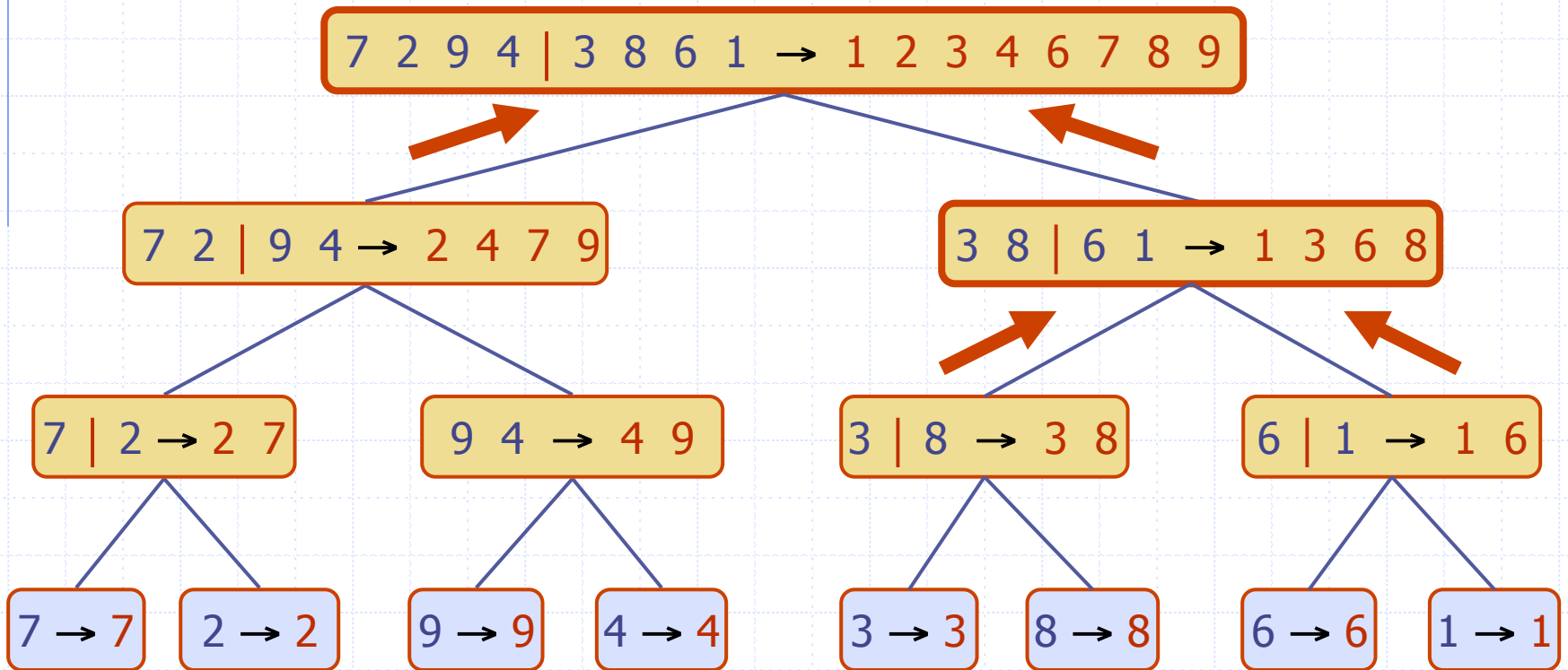
# Execution Example (cont.)

◆ Recursive call, ..., merge, merge



# Execution Example (cont.)

## ◆ Merge





# Non-Recursive Merge-Sort

merge runs of length 2, then 4, then 8, and so on

```
public static void mergeSort(Object[] orig, Comparator c) { // nonrecursive
    Object[] in = new Object[orig.length]; // make a new temporary array
    System.arraycopy(orig,0,in,0,in.length); // copy the input
    Object[] out = new Object[in.length]; // output array
    Object[] temp; // temp array reference used for swapping
    int n = in.length;
    for (int i=1; i < n; i*=2) { // each iteration sorts all length-2*i runs
        for (int j=0; j < n; j+=2*i) // each iteration merges two length-i pairs
            merge(in,out,c,j,i); // merge from in to out two length-i runs at j
        temp = in; in = out; out = temp; // swap arrays for next iteration
    }
    // the "in" array contains the sorted array, so re-copy it
    System.arraycopy(in,0,orig,0,in.length);
}
```

merge two runs in the in array to the out array

```
protected static void merge(Object[] in, Object[] out, Comparator c, int start,
    int inc) { // merge in[start..start+inc-1] and in[start+inc..start+2*inc-1]
    int x = start; // index into run #1
    int end1 = Math.min(start+inc, in.length); // boundary for run #1
    int end2 = Math.min(start+2*inc, in.length); // boundary for run #2
    int y = start+inc; // index into run #2 (could be beyond array boundary)
    int z = start; // index into the out array
    while ((x < end1) && (y < end2))
        if (c.compare(in[x],in[y]) <= 0) out[z++] = in[x++];
        else out[z++] = in[y++];
    if (x < end1) // first run didn't finish
        System.arraycopy(in, x, out, z, end1 - x);
    else if (y < end2) // second run didn't finish
        System.arraycopy(in, y, out, z, end2 - y);
}
```

Merge Sort

# Visualizations

## Sorting Algorithms

# Efficiency?

- ◆ Can't just count loop iterations!
- ◆ How many levels of recursion?
- ◆ How much non-recursive work done at each level?
- ◆ Need to solve a "recurrence equation"