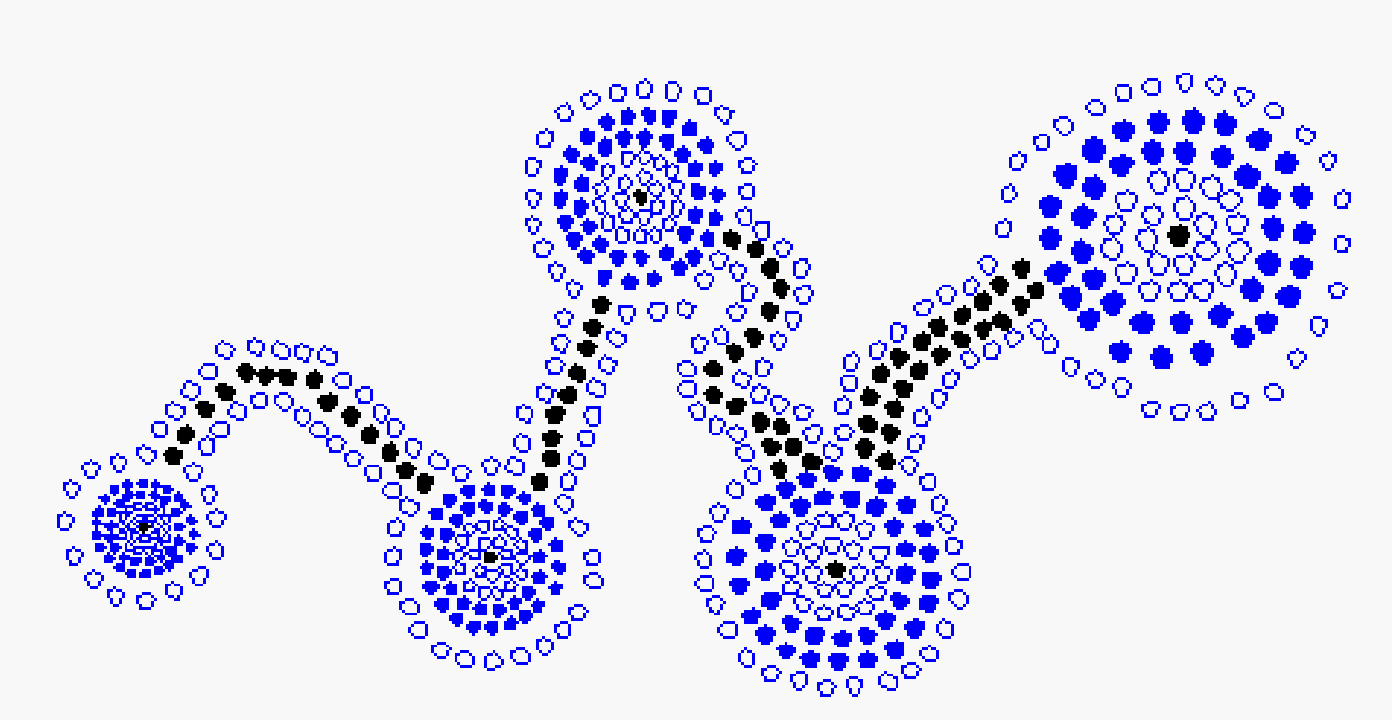
Merge Sort

- Review of Sorting
- Merge Sort



Sorting Algorithms

- Selection Sort uses a priority queue P implemented with an unsorted sequence:
  - **Phase 1**: the insertion of an item into P takes $O(1)$ time; overall $O(n)$ time for inserting n items.
  - **Phase 2**: removing an item takes time proportional to the number of elements in P, which is $O(n)$: overall $O(n^2)$
  - Time Complexity**:** $O(n^2)$

Sorting Algorithms (cont.)

- Insertion Sort is performed on a priority queue P which is a sorted sequence:
  - **Phase 1**: the first insertItem takes $O(1)$, the second $O(2)$, until the last insertItem takes $O(n)$: overall $O(n^2)$
  - **Phase 2:** removing an item takes $O(1)$ time; overall $O(n)$.
  - Time Complexity: $O(n^2)$
- Heap Sort uses a priority queue K which is a heap.
  - **insertItem** and **removeMin** each take $O(\log k)$, k being the number of elements in the heap at a given time.
  - **Phase 1**: n elements inserted: $O(n \log n)$ time
  - **Phase 2**: n elements removed: $O(n \log n)$ time.
  - Time Complexity: $O(n \log n)$

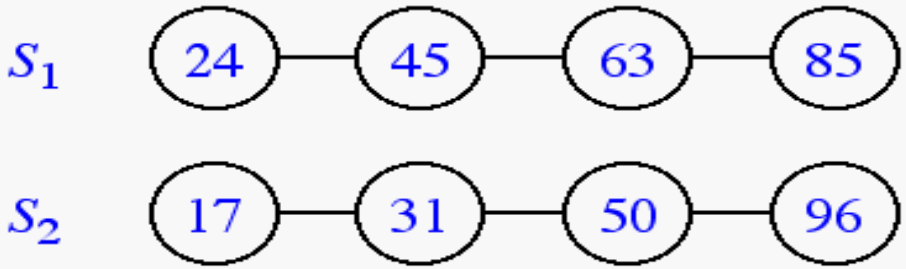
Merging Two Sorted Sequences

```
void mergeArrays(int arr1[], int arr2[], int n, int m)
{
   int i = 0, j = 0, k = 0;
   int arr3[m+n] ;
   // Traverse both array
   while (i < n and j < m) {
      if (arr1[i] < arr2[j]) {
         arr3[k] = arr1[i];
         i = i + 1 ;       }
      else{
         arr3[k] = arr2[j];
         j = j + 1;
      }
      k = k + 1;    }
   while (i < n){
      arr3[k] = arr1[i];
      i = i + 1;
      k = k + 1;    }
   while (j < m){
      arr3[k] = arr2[j];
      j = j + 1;
      k = k + 1;     }
return arr3;
}
```

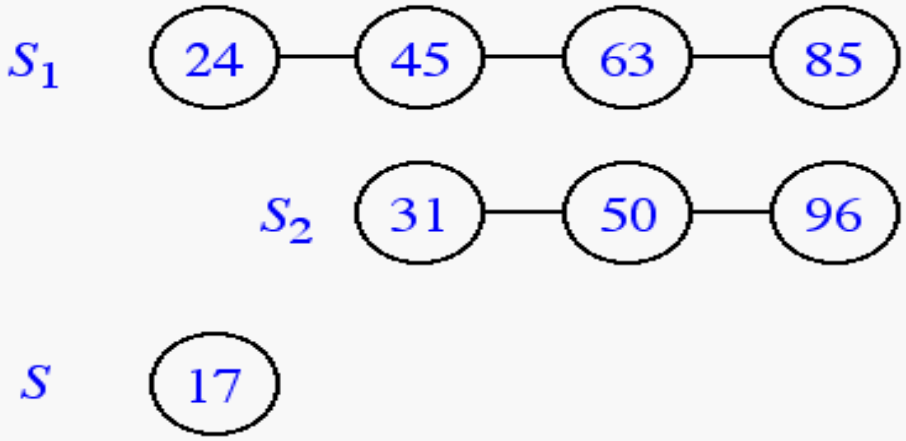
Merging Two Sequences (cont.)

- Some pictures:

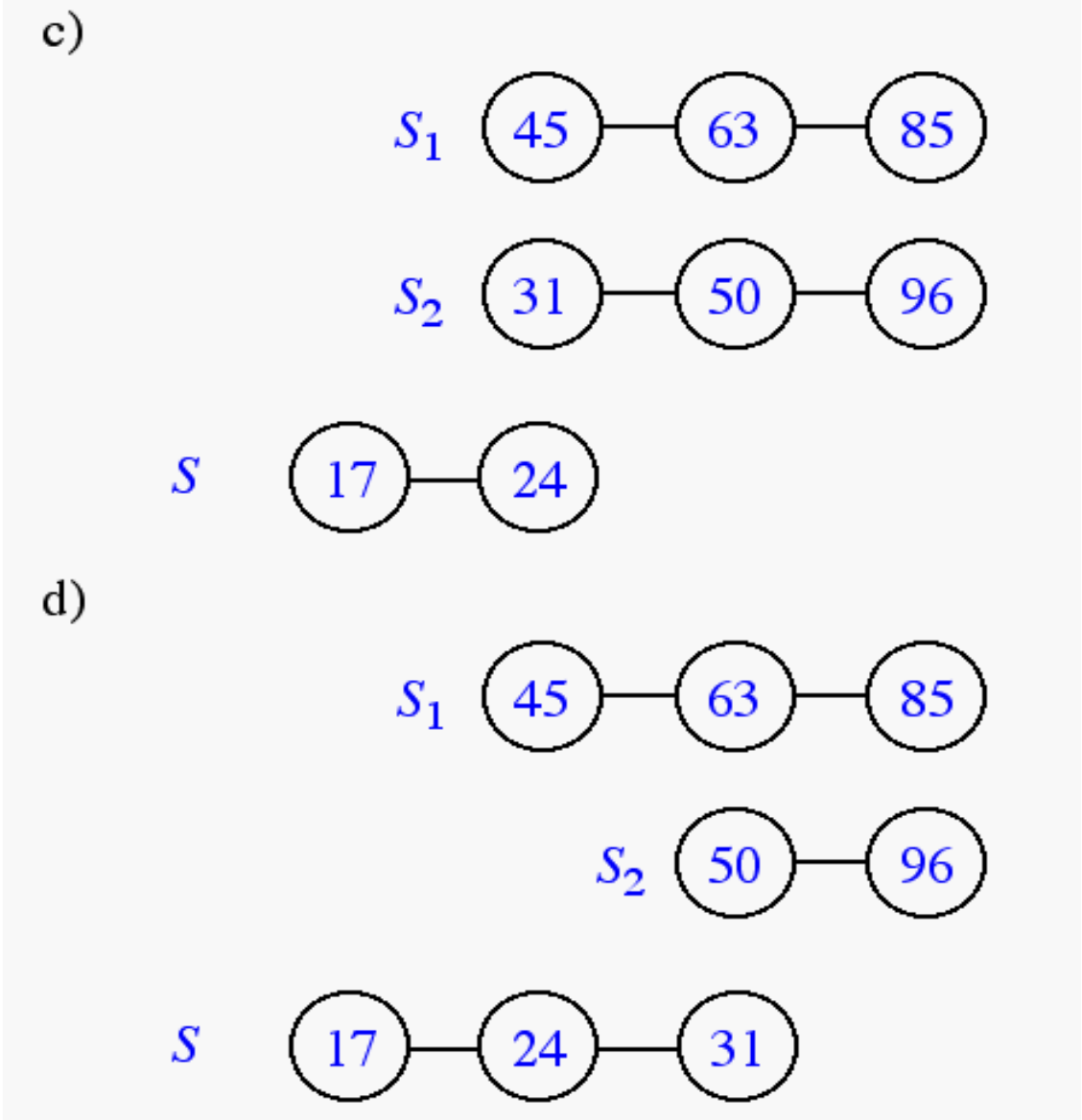
a)



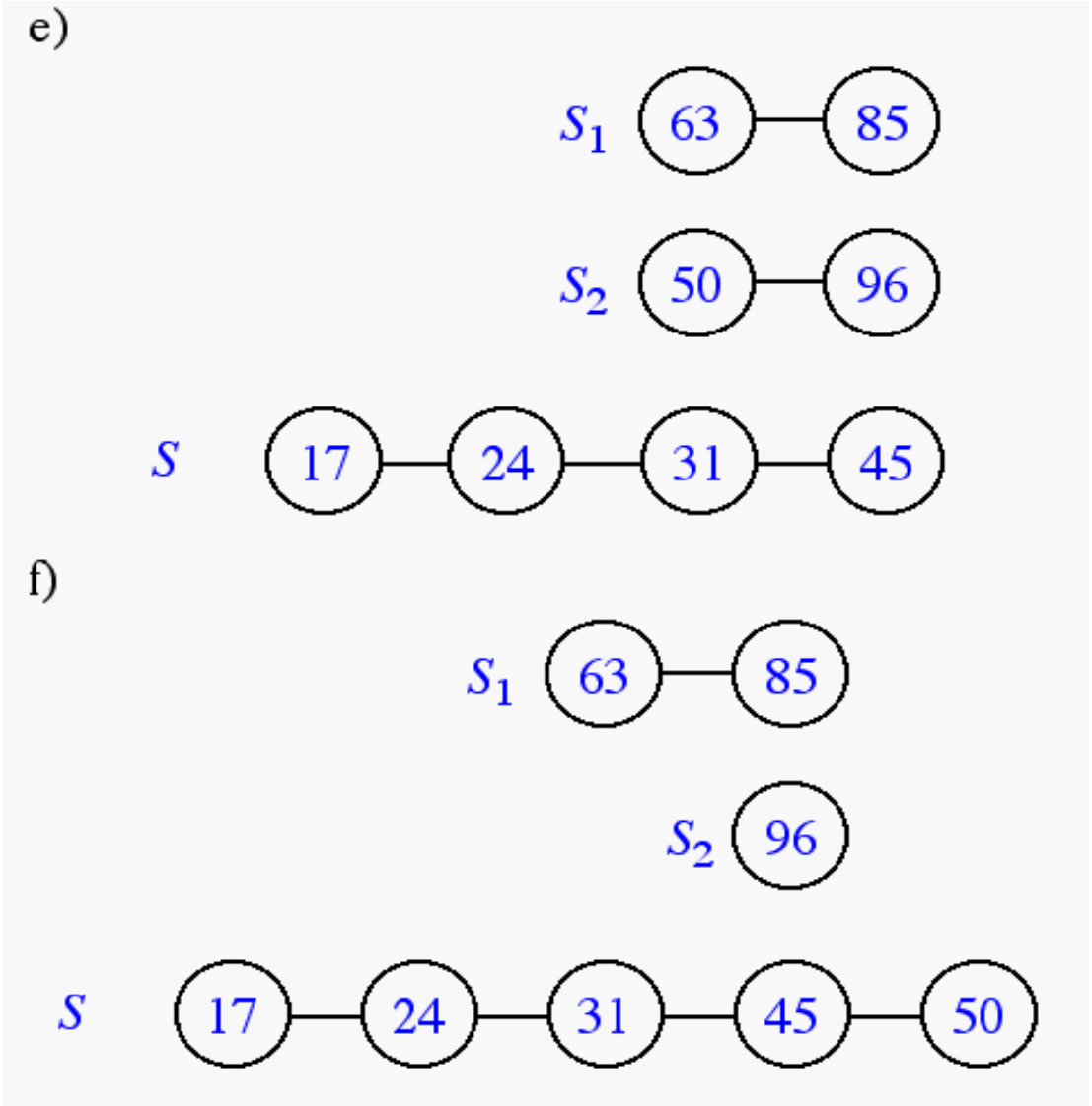
b)



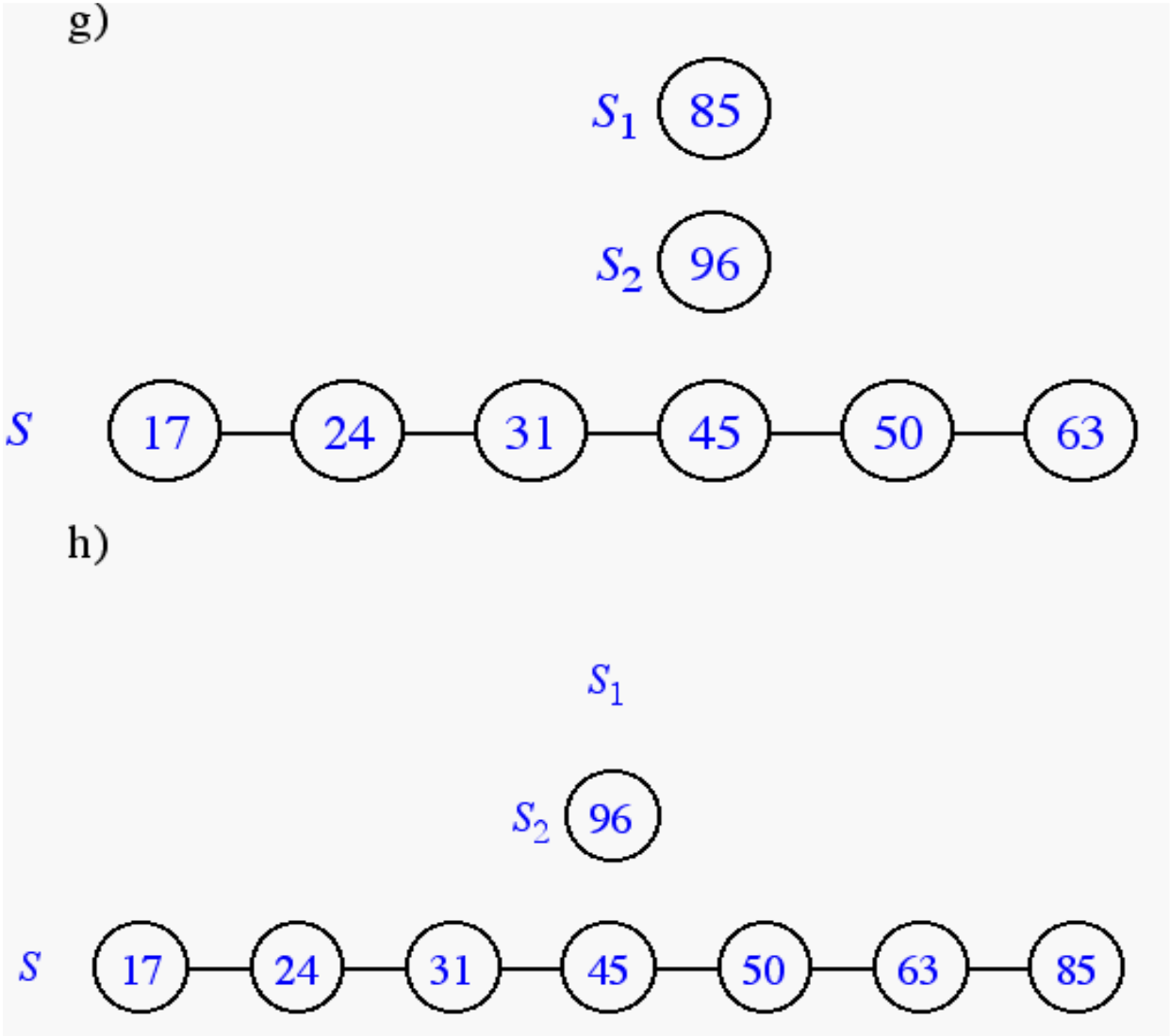
Merging Two Sequences (cont.)



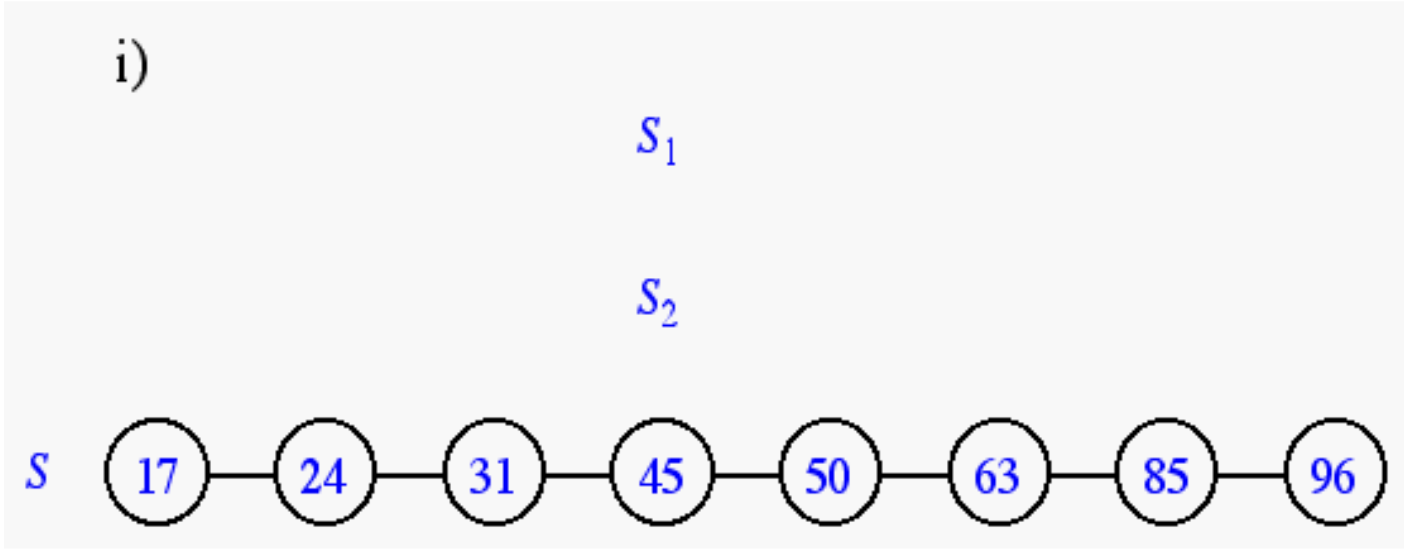
Merging Two Sequences (cont.)



Merging Two Sequences (cont.)



Merging Two Sequences (cont.)



Complexity Analysis

Time Complexity: O(n+m)

Space Complexity: O(n+m)

Divide-and-Conquer

- ***Divide and Conquer*** is more than just a military strategy; it is also a method of algorithm design that has created such efficient algorithms as Merge Sort.
- In terms or algorithms, this method has three distinct steps:
  - **Divide**: If the input size is too large to deal with in a straightforward manner, divide the data into two or more disjoint subsets.
  - **Recur**: Use divide and conquer to solve the subproblems associated with the data subsets.
  - **Conquer**: Take the solutions to the subproblems and "merge" these solutions into a solution for the original problem.

Merge-Sort

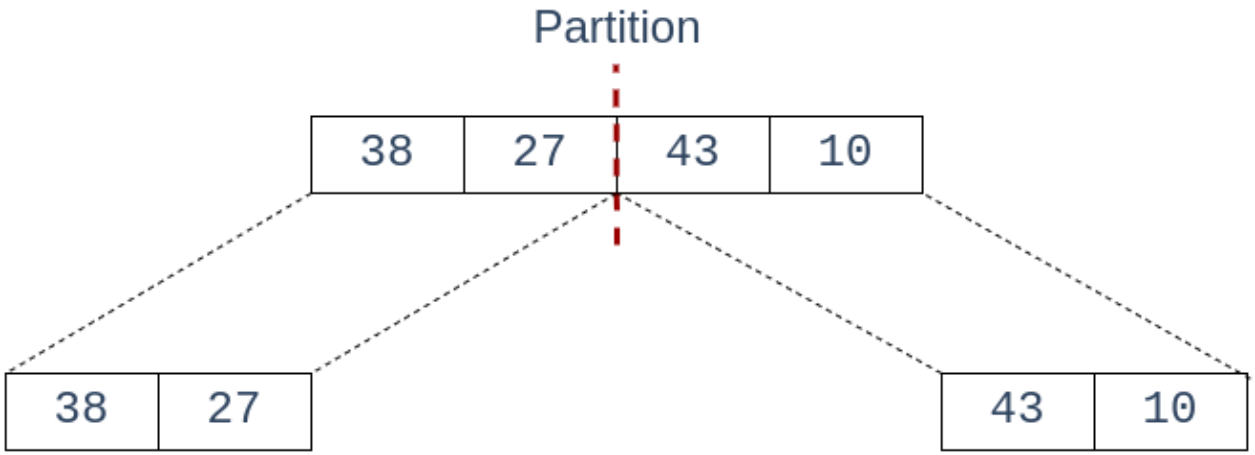
- **Algorithm**:
  - **Divide**: If S has at least two elements (nothing needs to be done if S has zero or one elements), remove all the elements from S and put them into two sequences, S_1 and S_2, each containing about half of the elements of S. (i.e. S_1 contains the first $\lceil n/2 \rceil$ elements and S_2 contains the remaining $\lfloor n/2 \rfloor$ elements.
  - **Recur**: Recursive sort sequences S_1 and S_2.
  - **Conquer**: Put back the elements into S by merging the sorted sequences S_1 and S_2 into a unique sorted sequence.
- Merge Sort Tree:
  - Take a binary tree T
  - Each node of T represents a recursive call of the merge sort algorithm.
  - We associate with each node v of T a the set of input passed to the invocation v represents.
  - The external nodes are associated with individual elements of S, upon which no recursion is called.

Merge Sort

Merge Sort: Divide the array into two halves

STEP 01

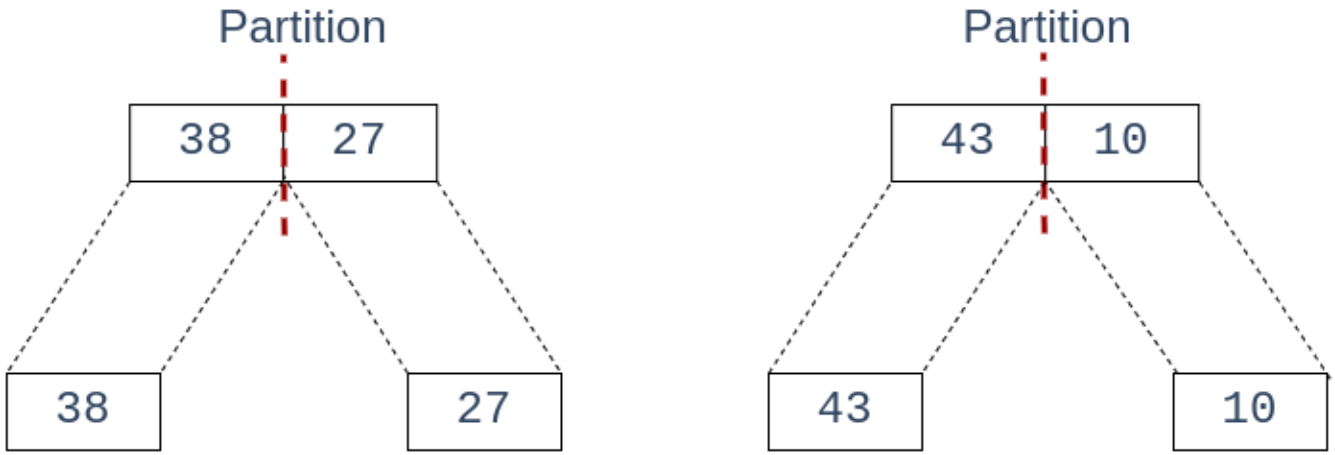
Splitting the Array into two equal halves



Merge Sort

These subarrays are further divided into two halves. Now they become array of unit length that can no longer be divided and array of unit length are always sorted.

STEP 02 — Splitting the subarrays into two halves

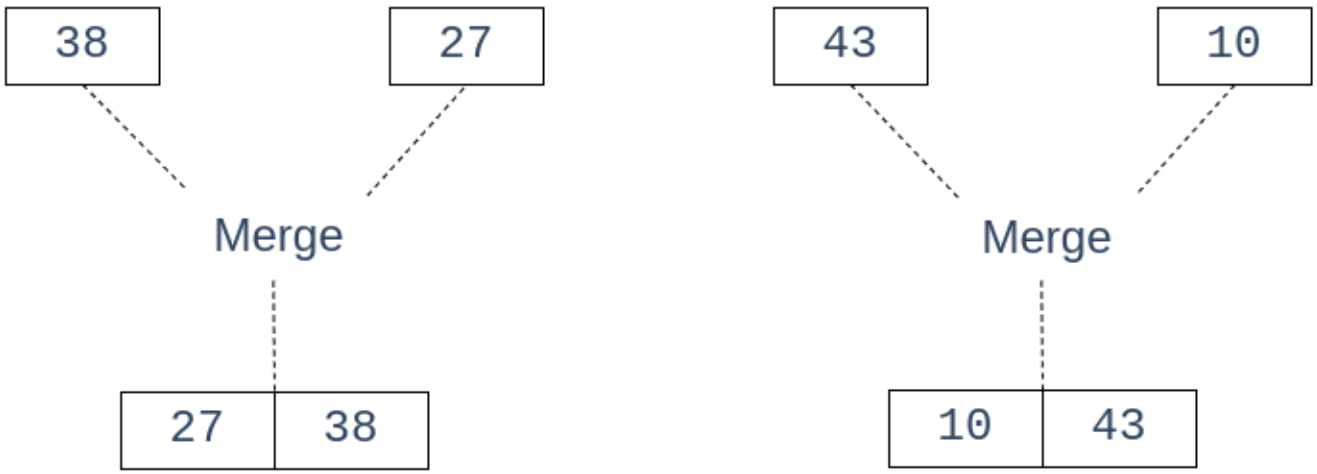


*Merge Sort: Divide the subarrays into two halves (**unit length subarrays here**)*

Merge Sort

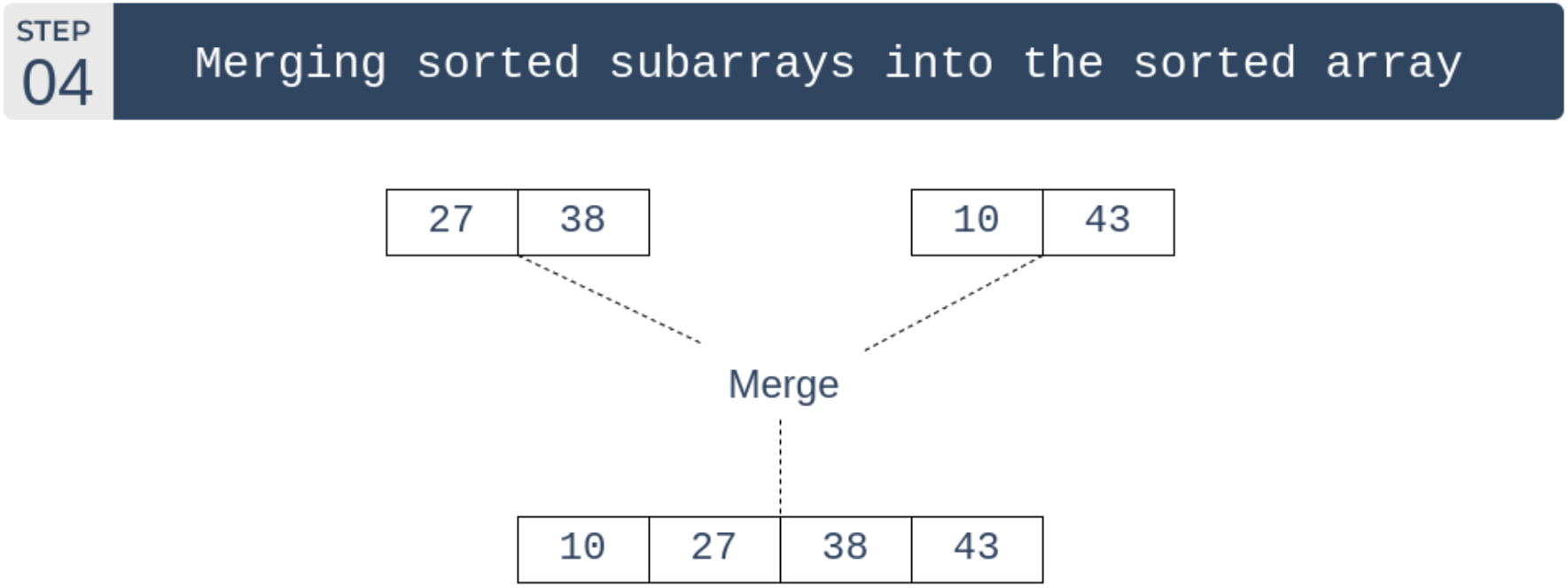
These sorted subarrays are merged together, and we get bigger sorted subarrays.

STEP 03 — Merging unit length cells into sorted subarrays



Merge Sort

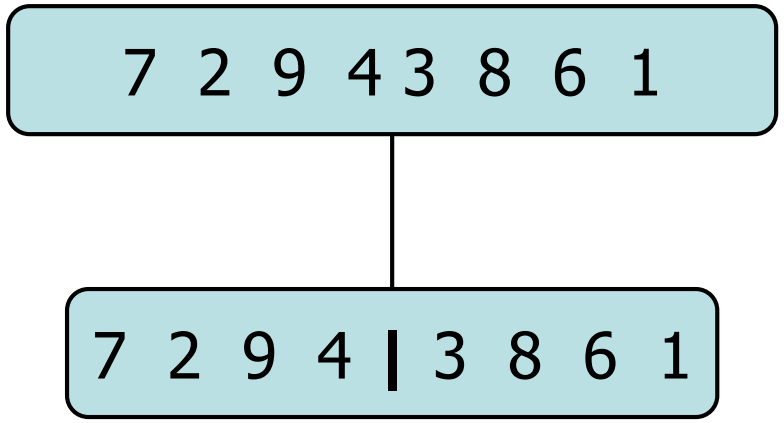
This merging process is continued until the sorted array is built from the smaller subarrays.



Merge Sort - Example

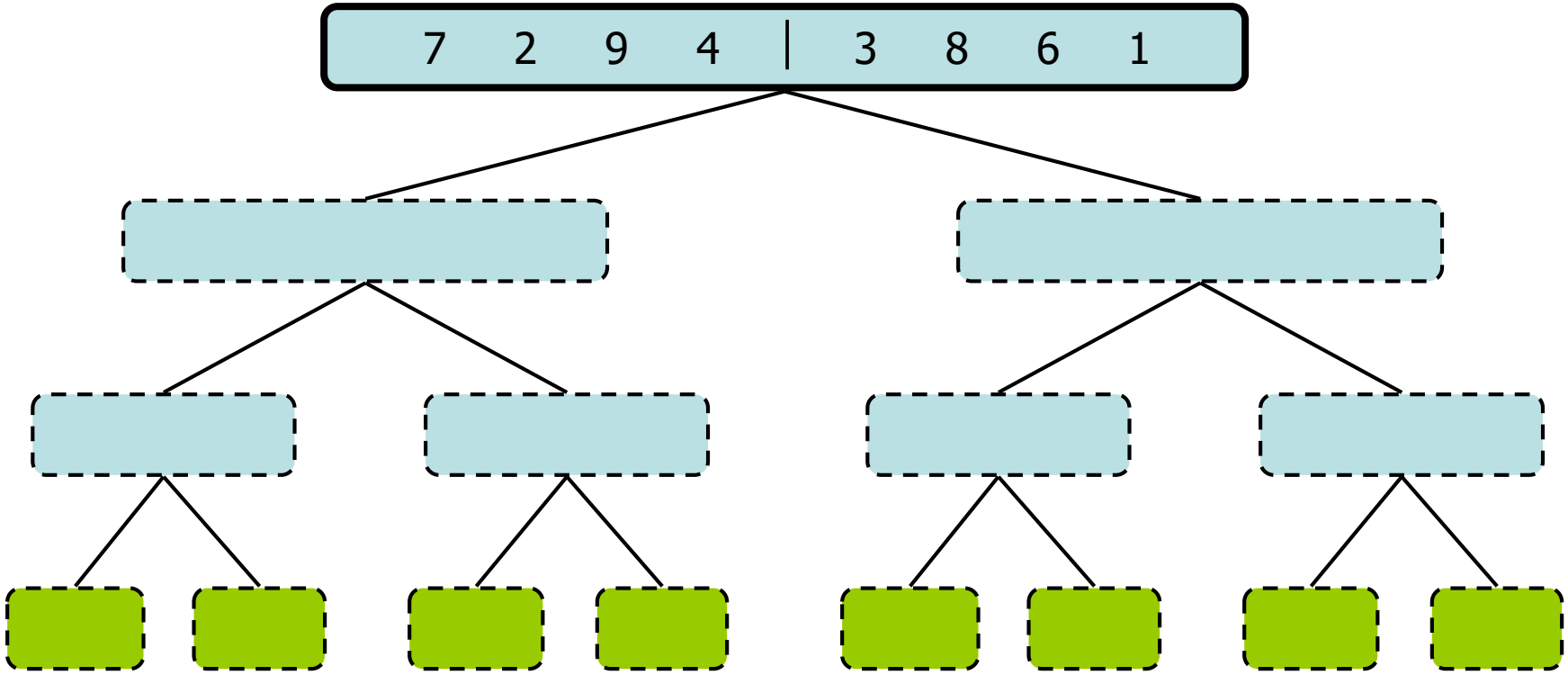
Merge-Sort Tree

- 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



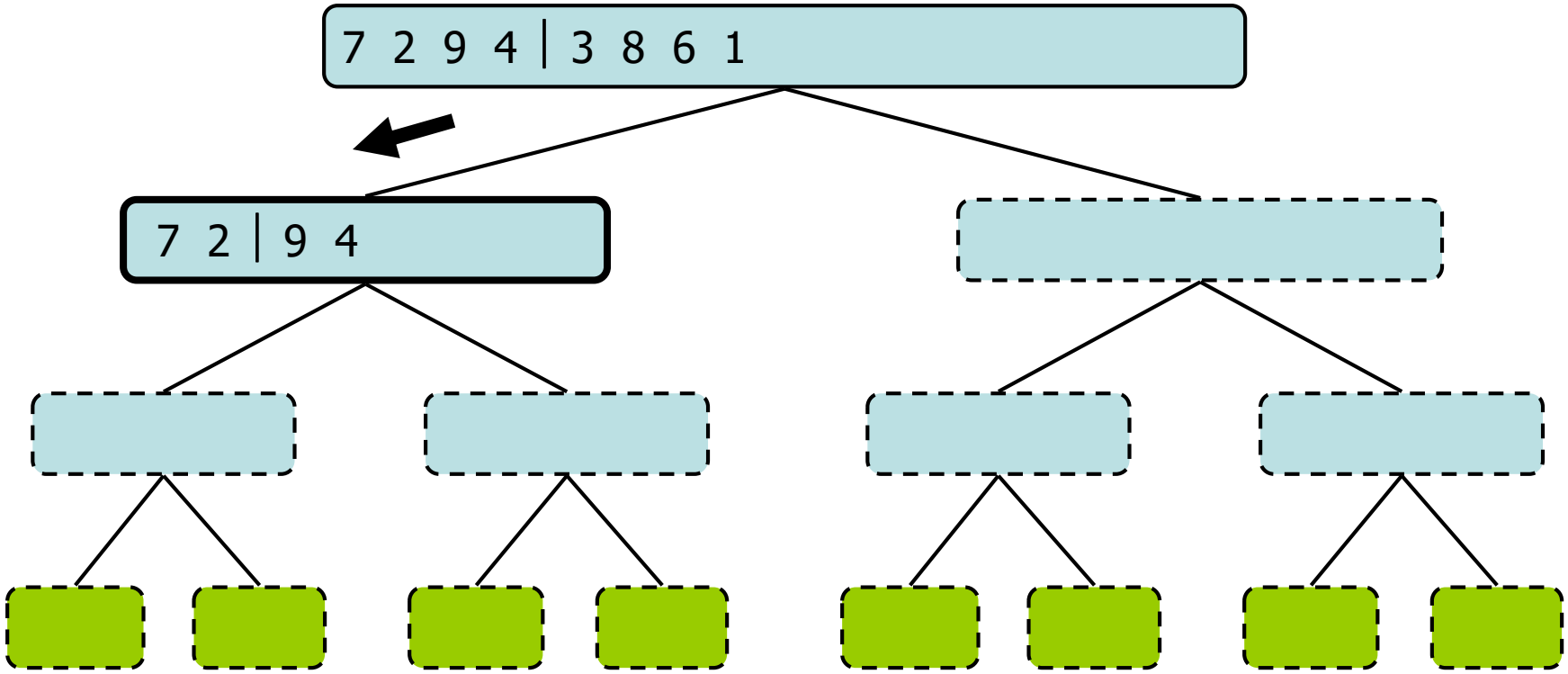
Execution Example

- Partition



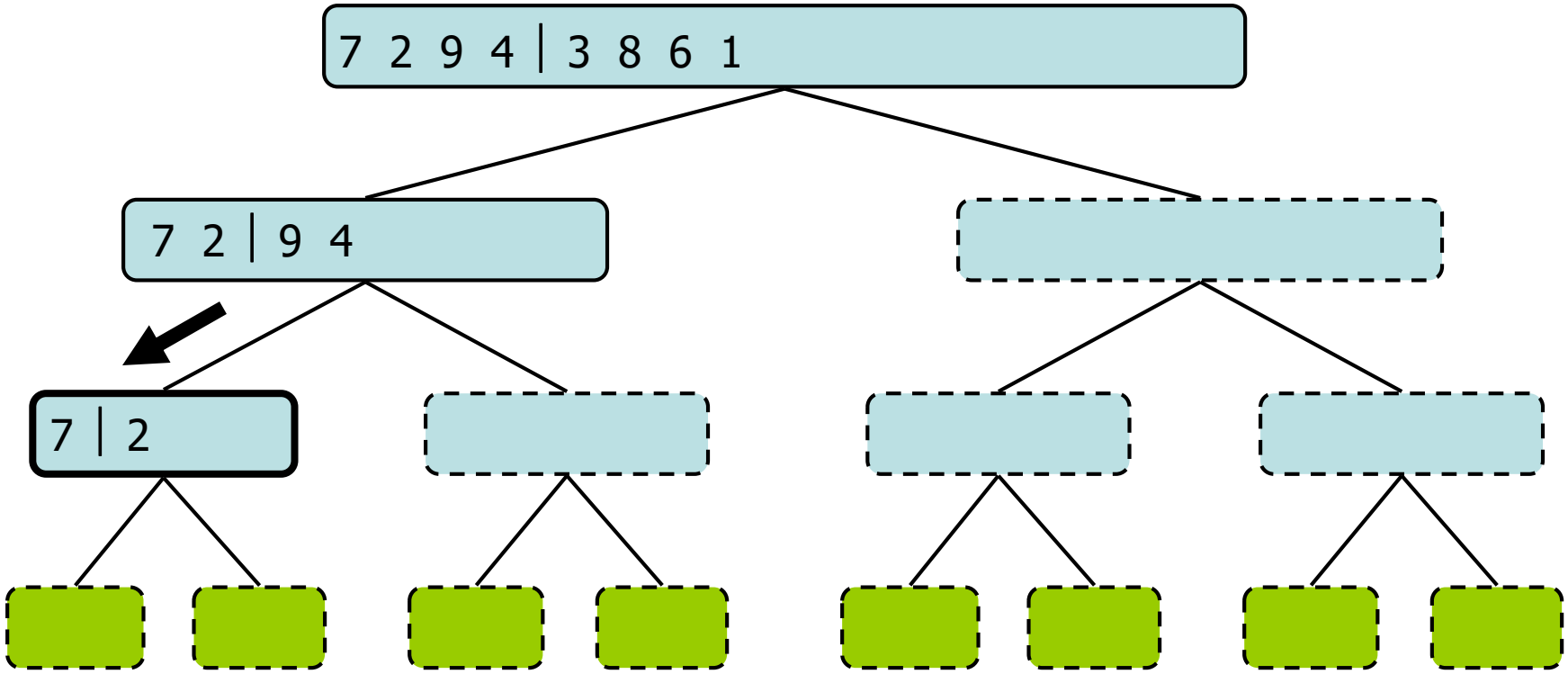
Execution Example (cont.)

- Recursive call, partition



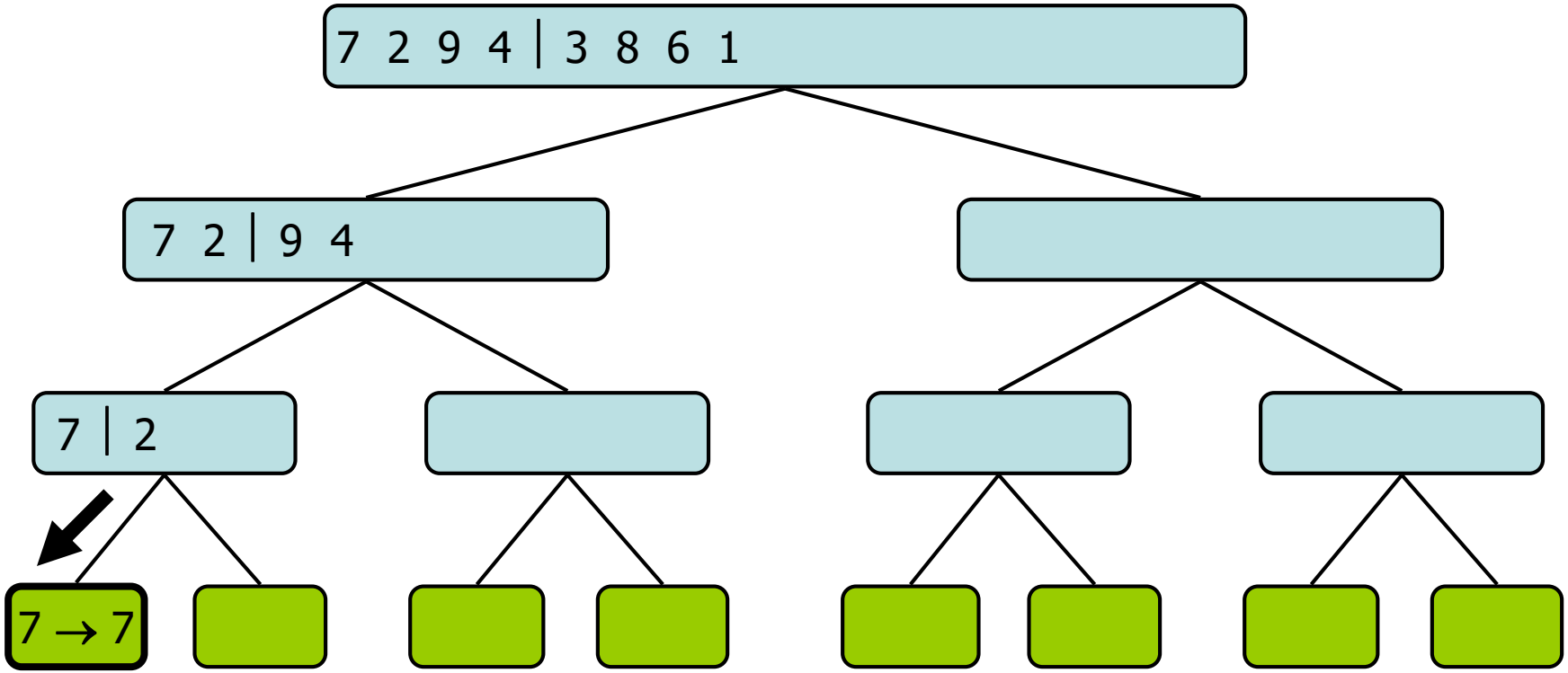
Execution Example (cont.)

- Recursive call, partition



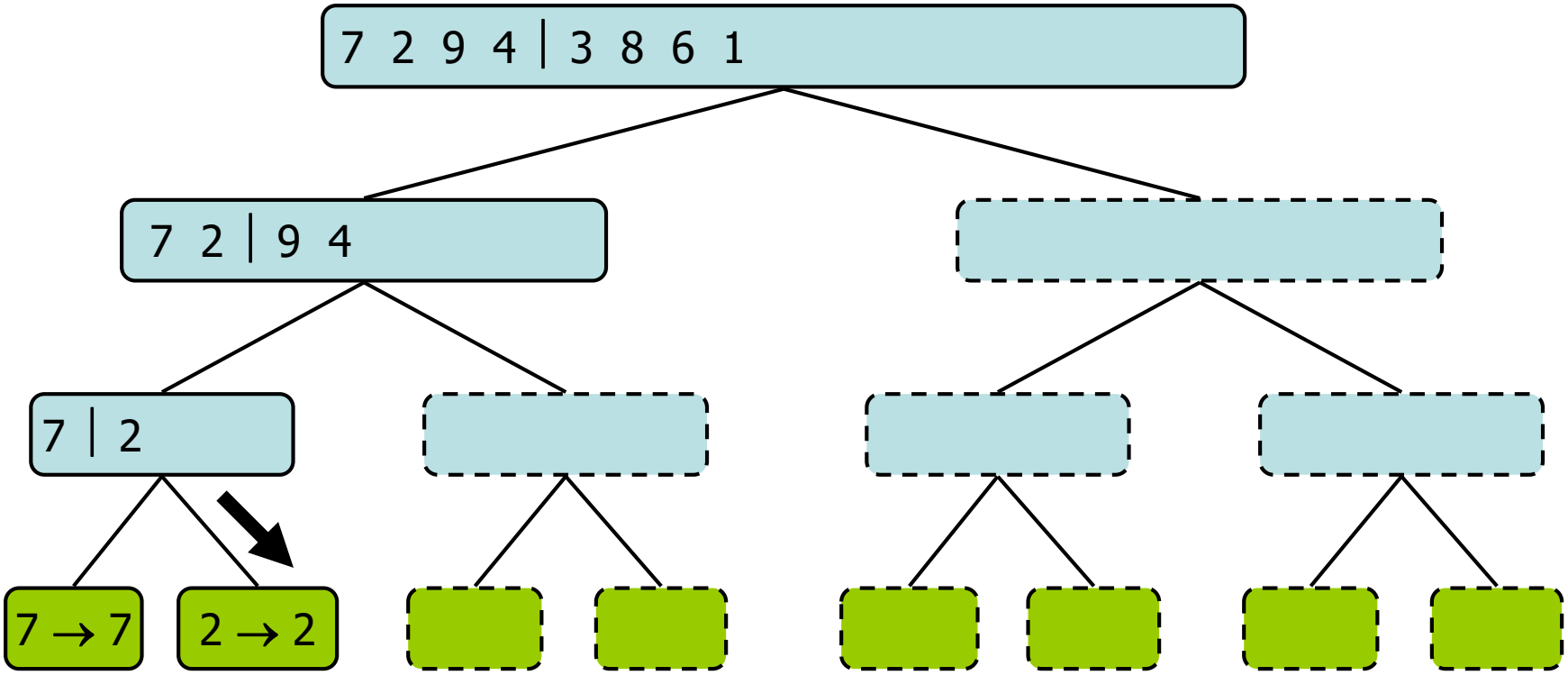
Execution Example (cont.)

- Recursive call, base case



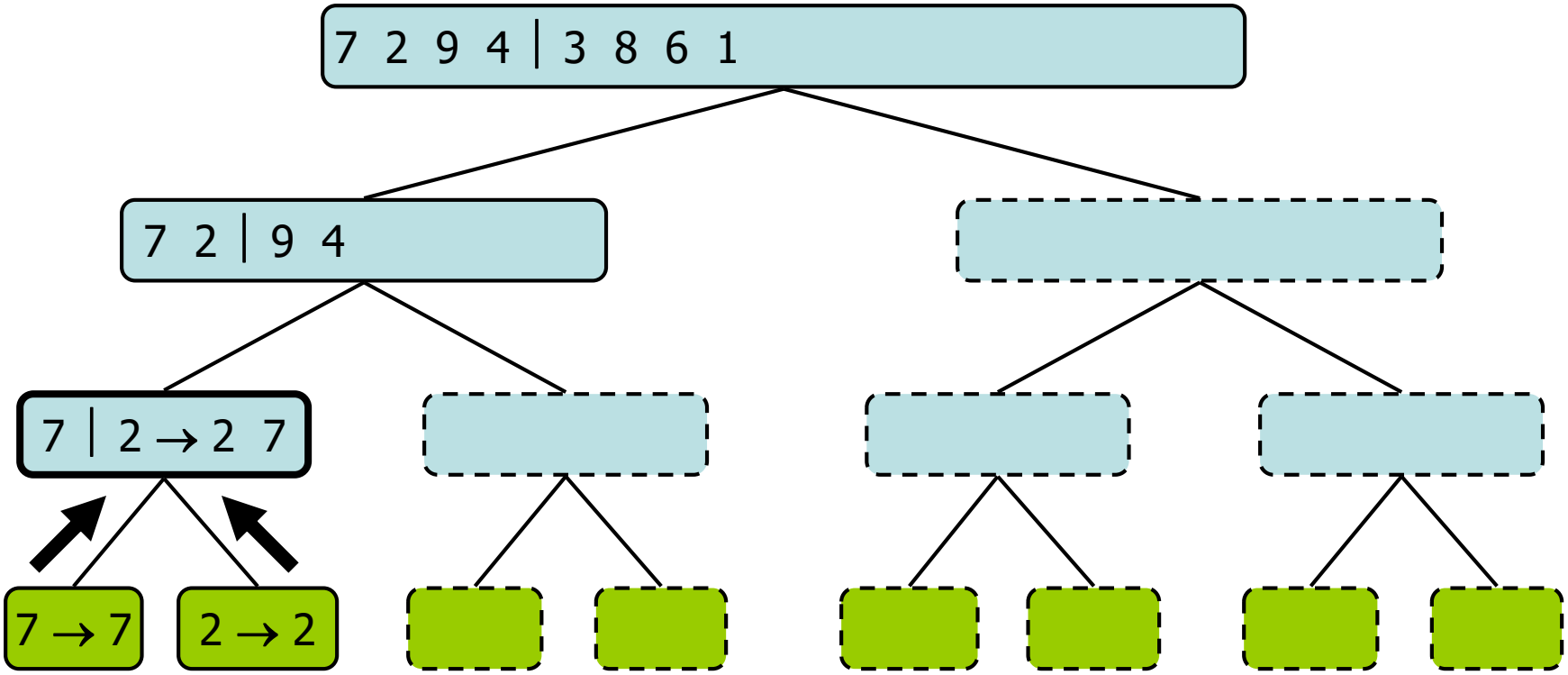
Execution Example (cont.)

- Recursive call, base case



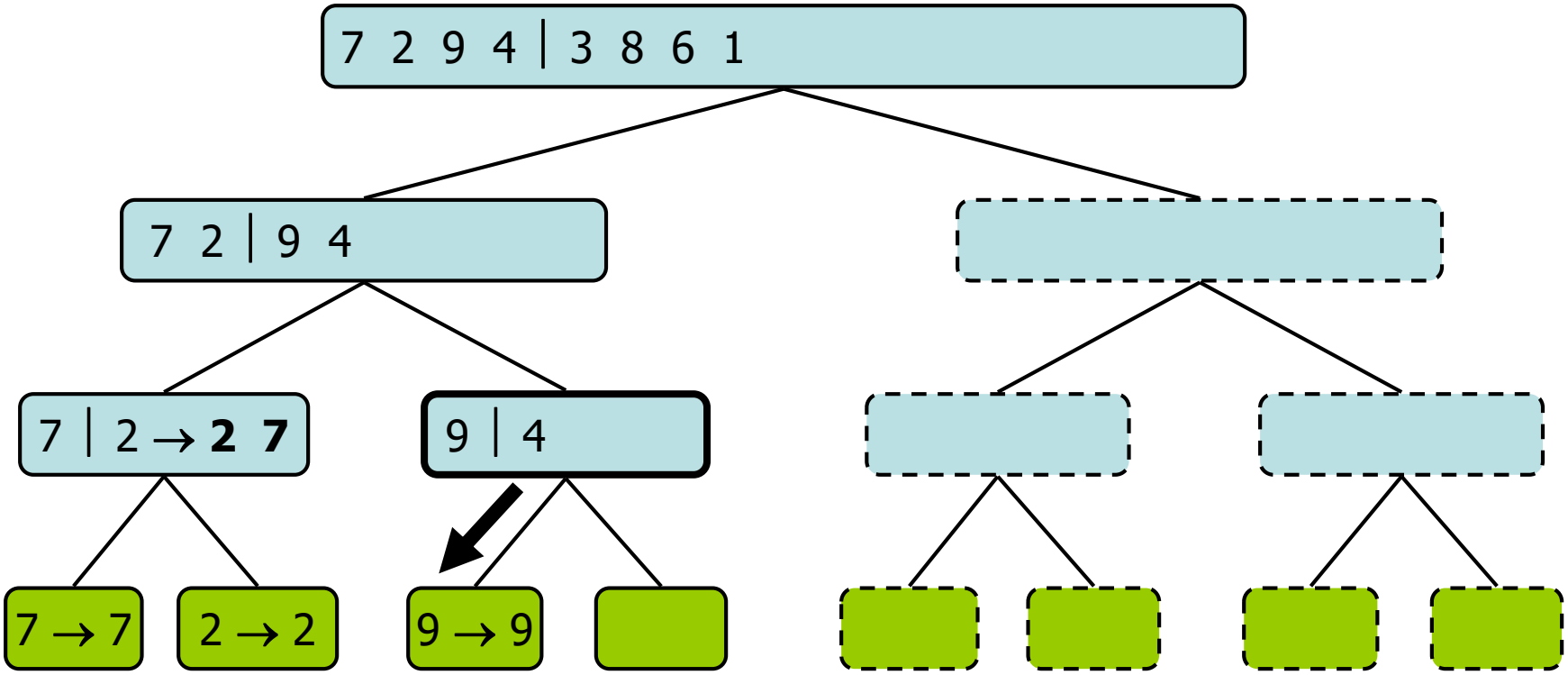
Execution Example (cont.)

- Merge



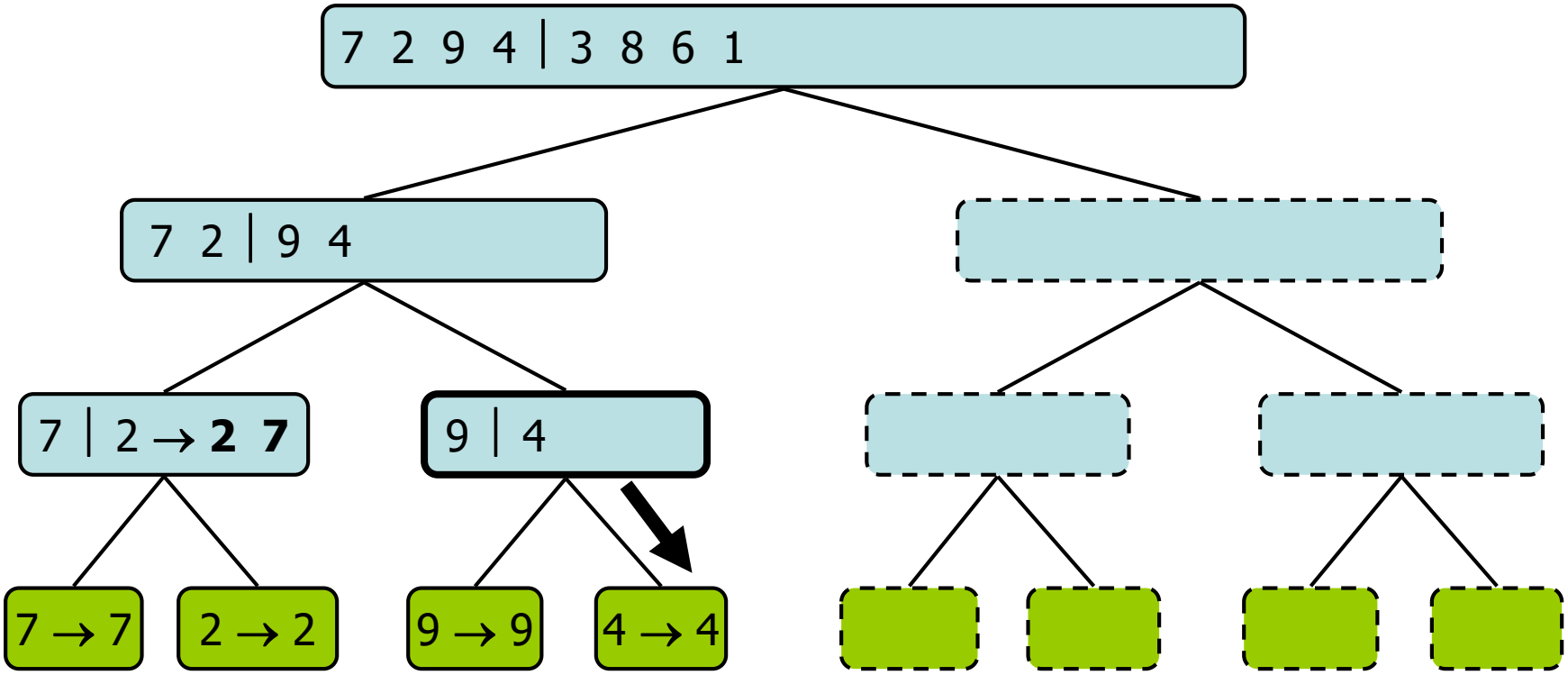
Execution Example (cont.)

- Recursive call, …, base case, merge



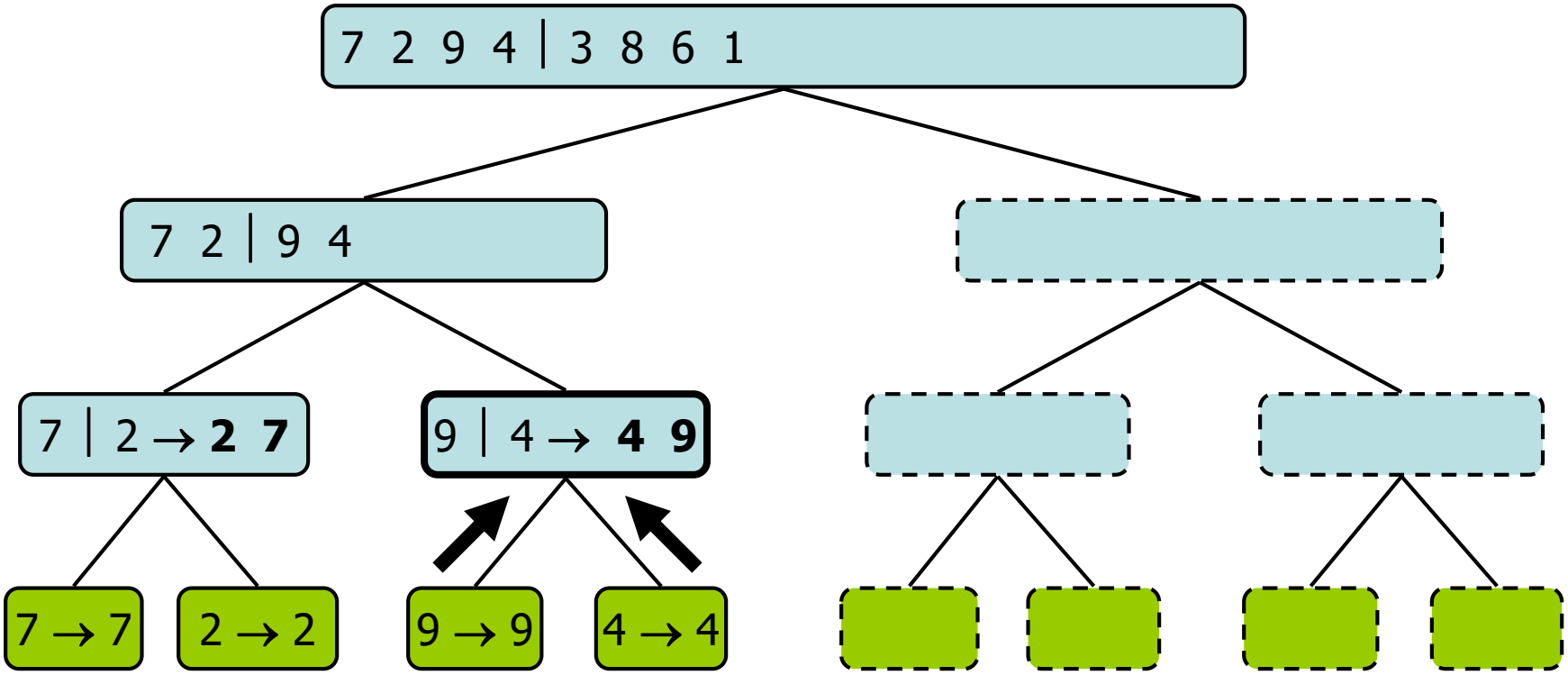
Execution Example (cont.)

- Recursive call, …, base case, merge



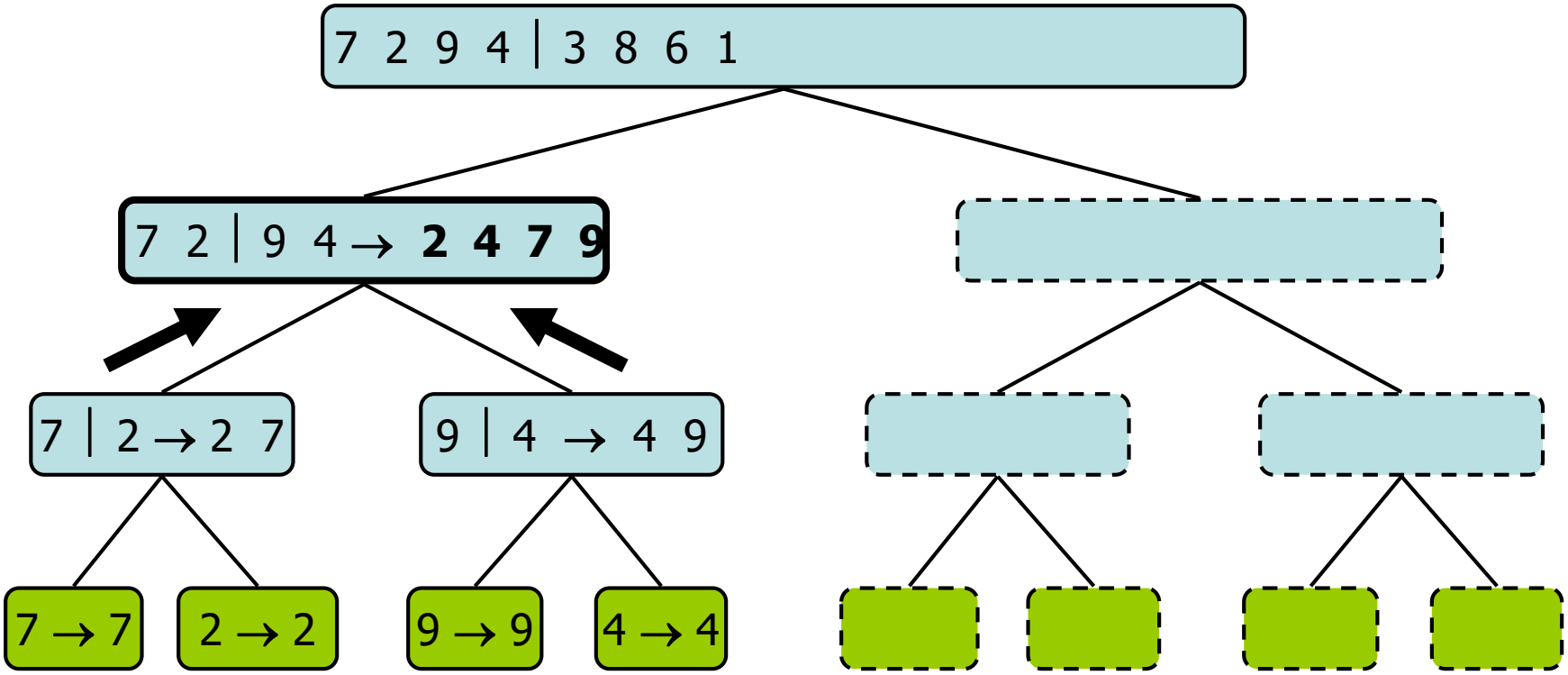
Execution Example (cont.)

- Recursive call, …, base case, merge



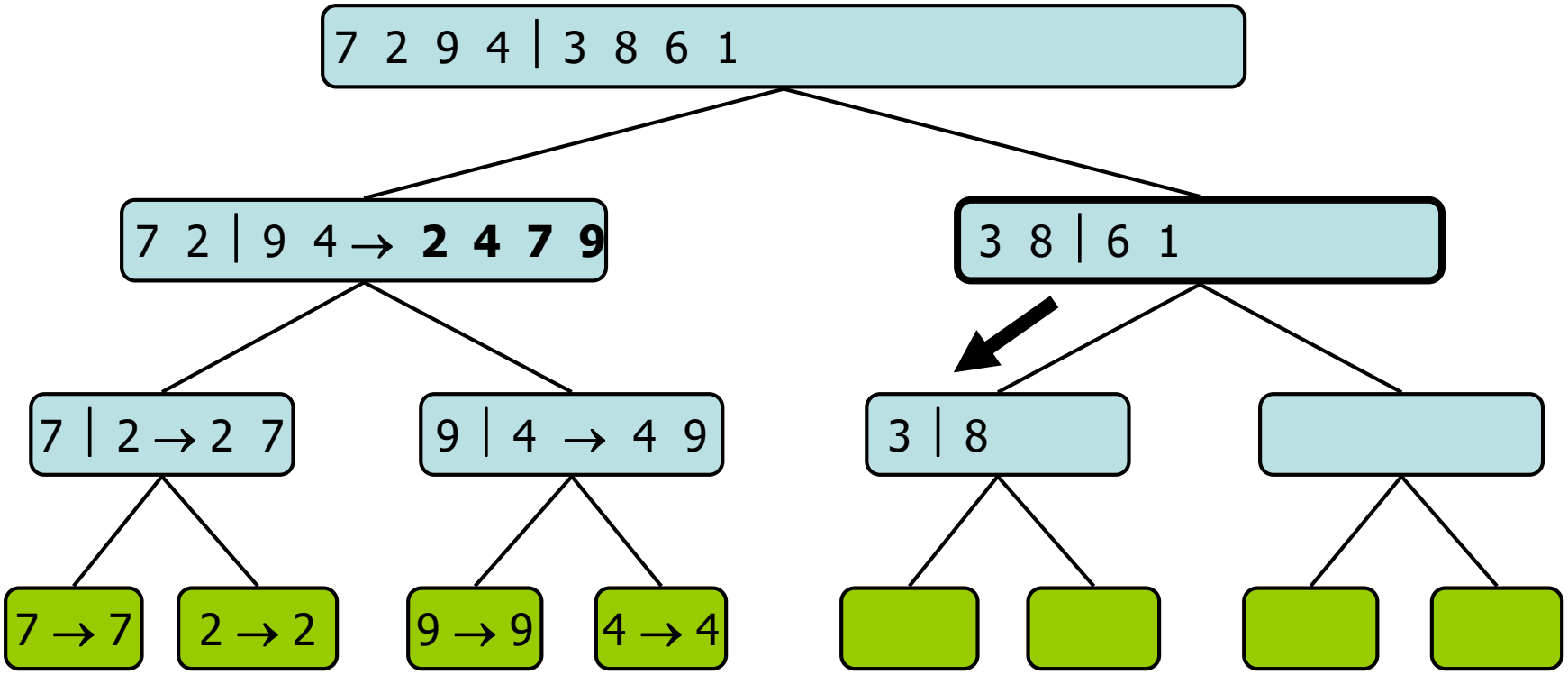
Execution Example (cont.)

- Merge



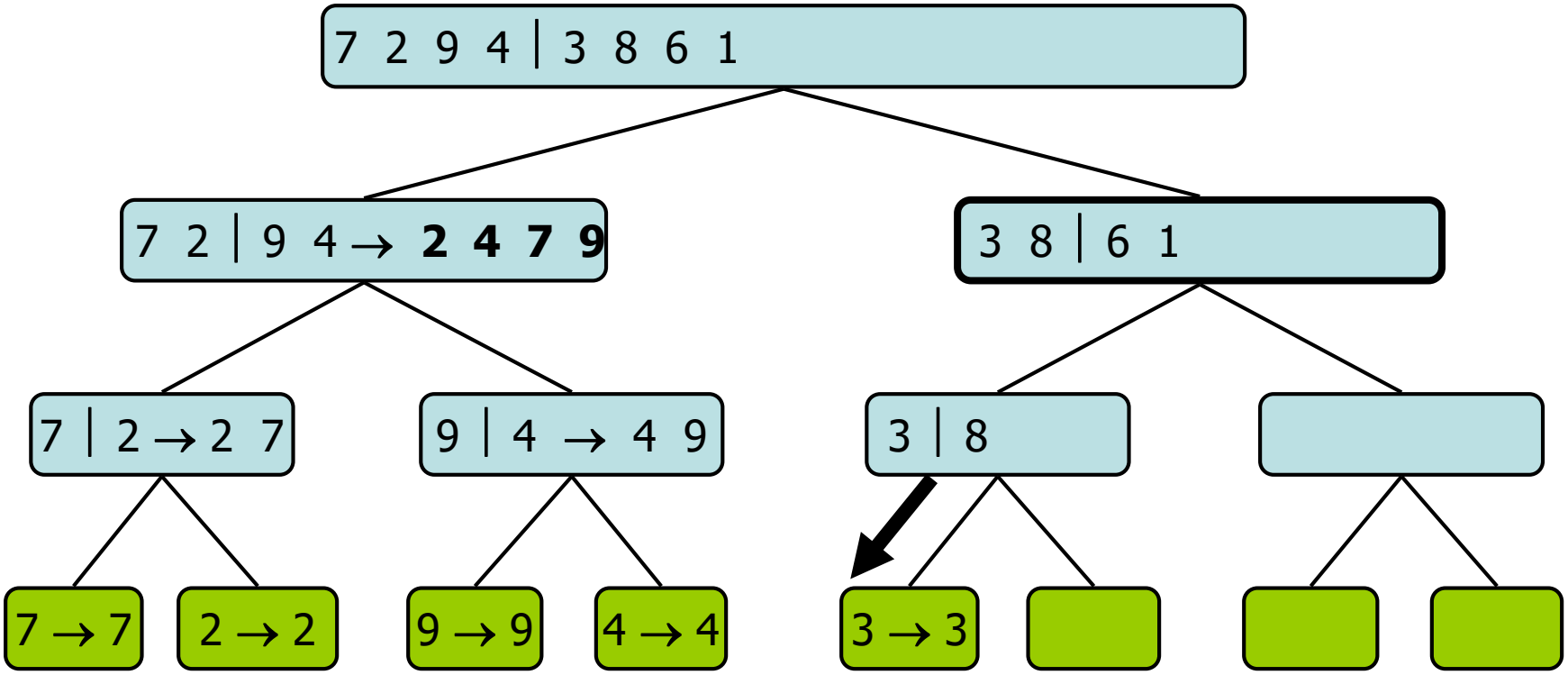
Execution Example (cont.)

- Recursive call, …., merge, merge



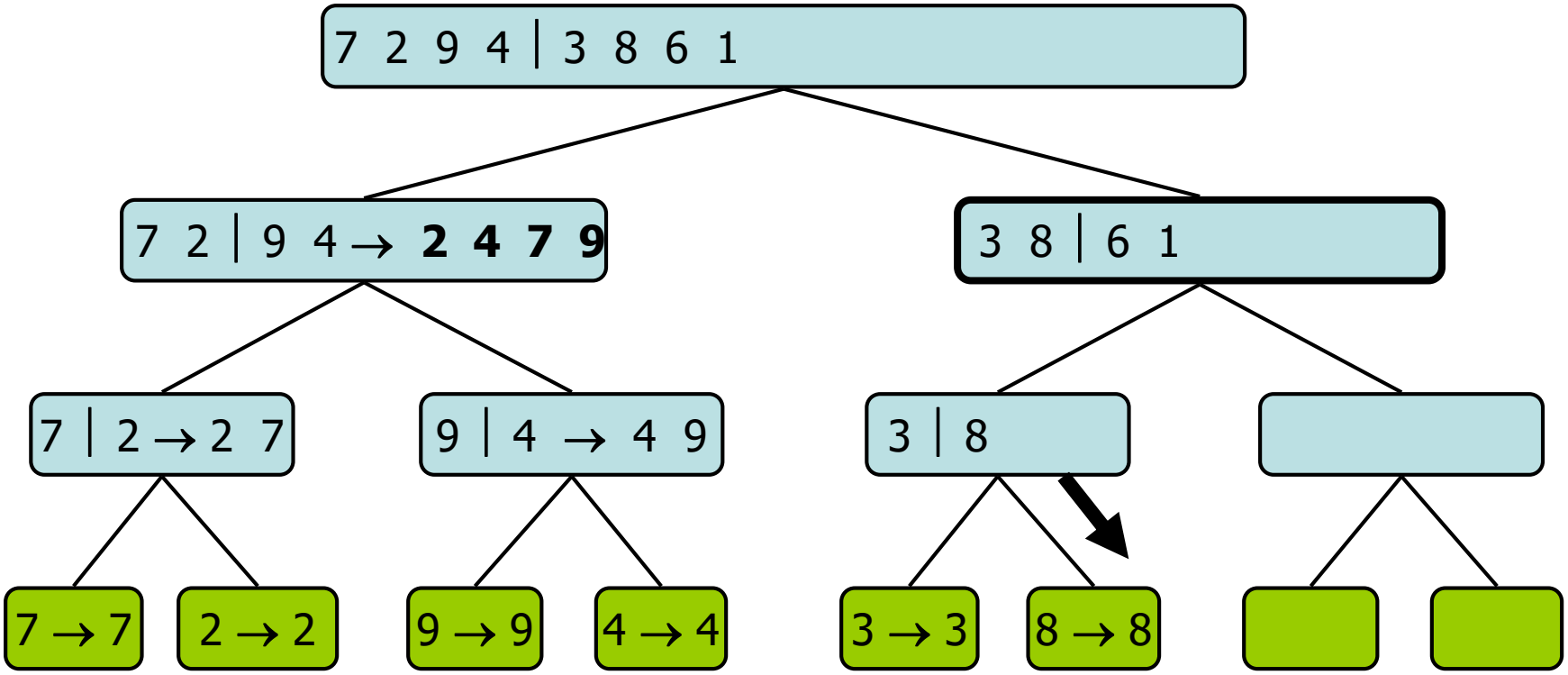
Execution Example (cont.)

- Recursive call, …., merge, merge



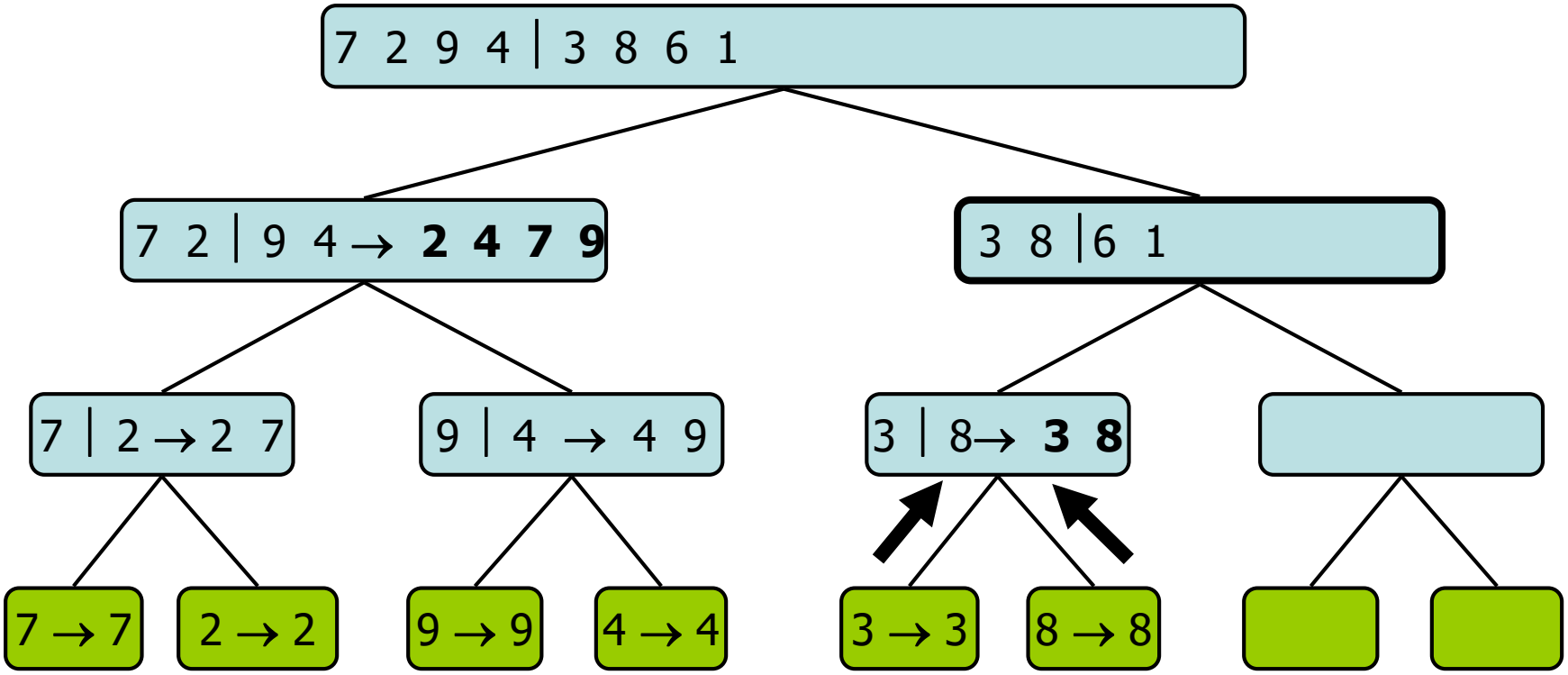
Execution Example (cont.)

- Recursive call, …., merge, merge



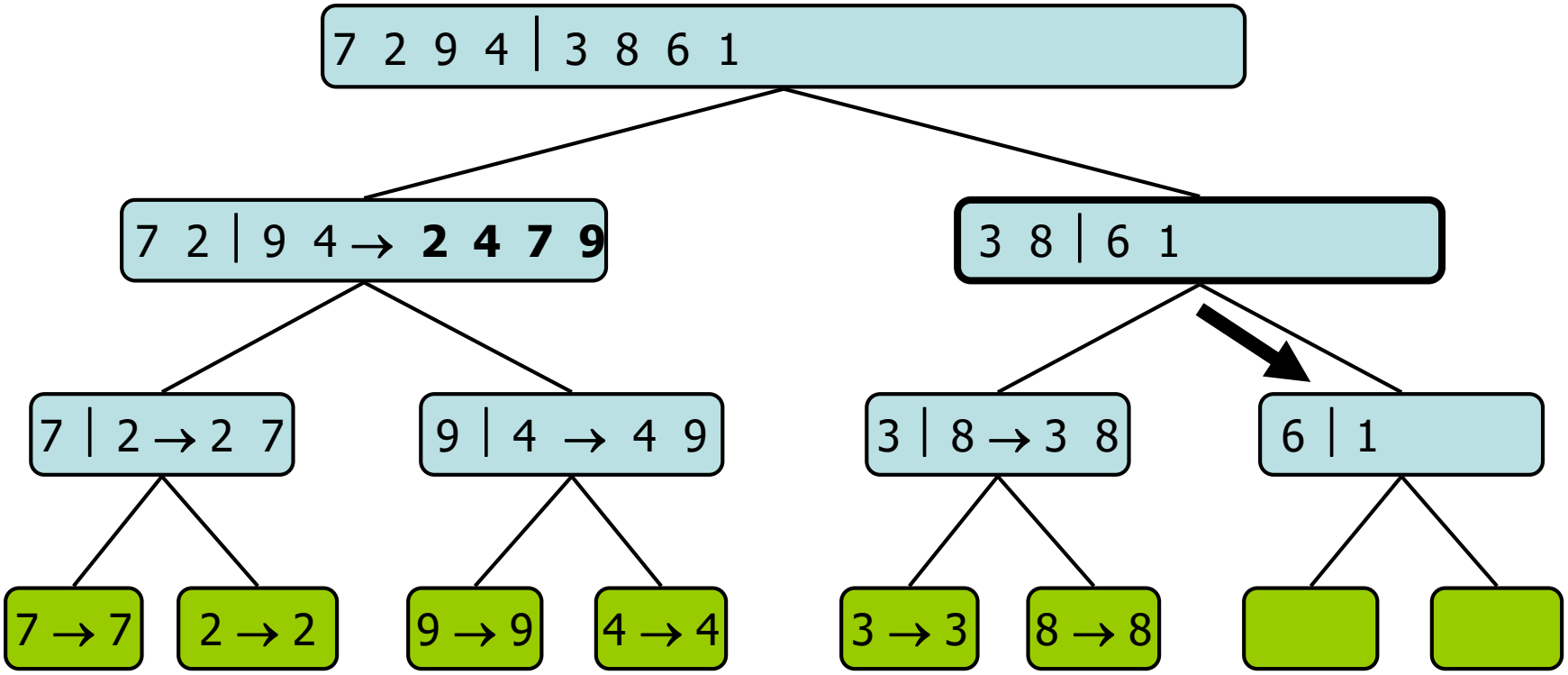
Execution Example (cont.)

- Recursive call, …., merge, merge



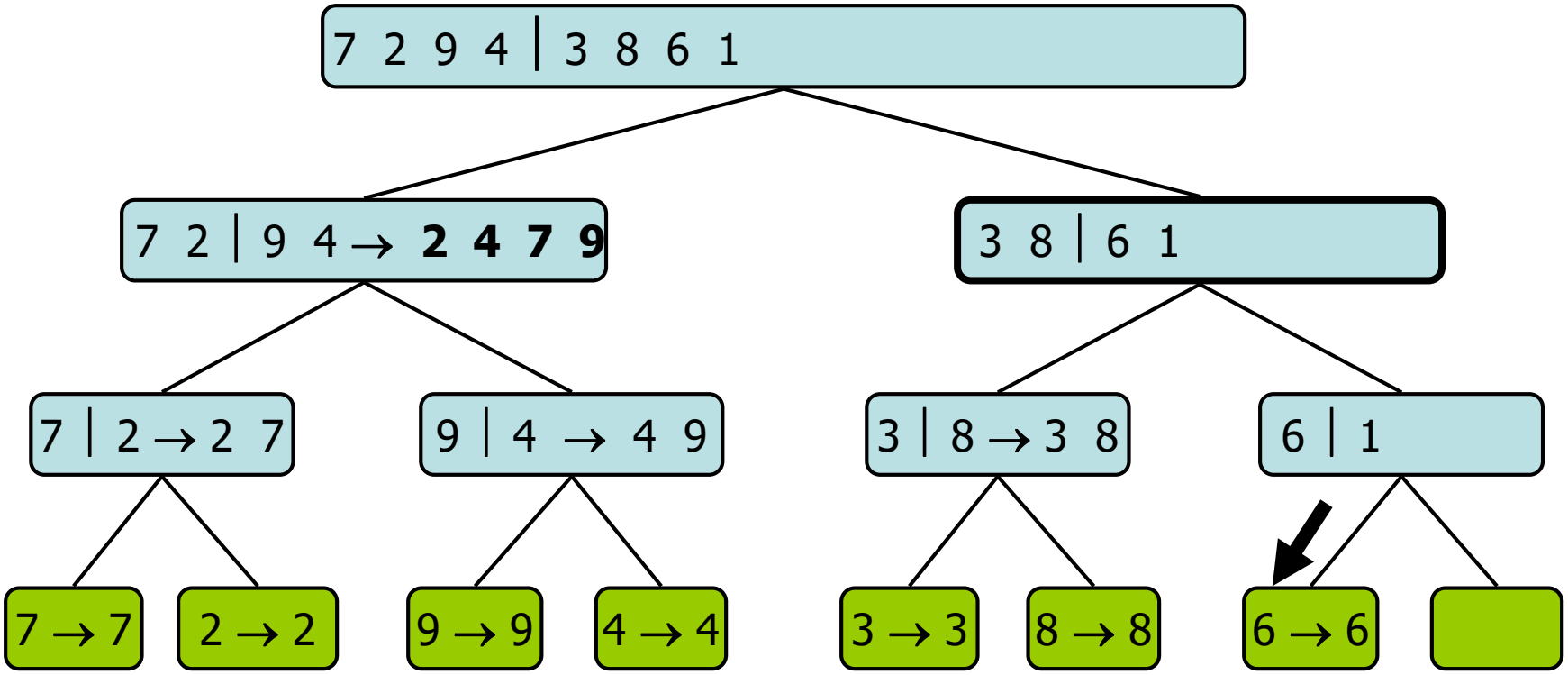
Execution Example (cont.)

- Recursive call, …, merge, merge



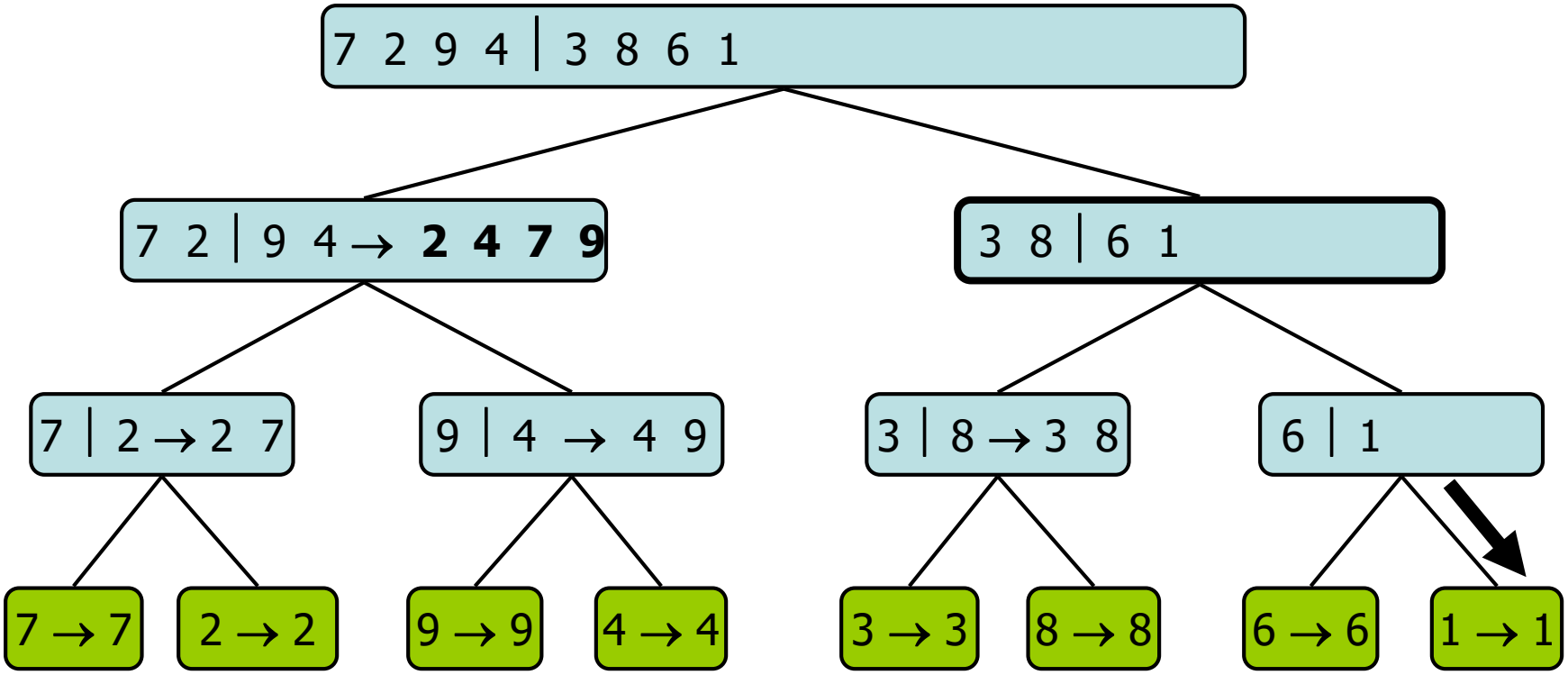
Execution Example (cont.)

- Recursive call, …., merge, merge



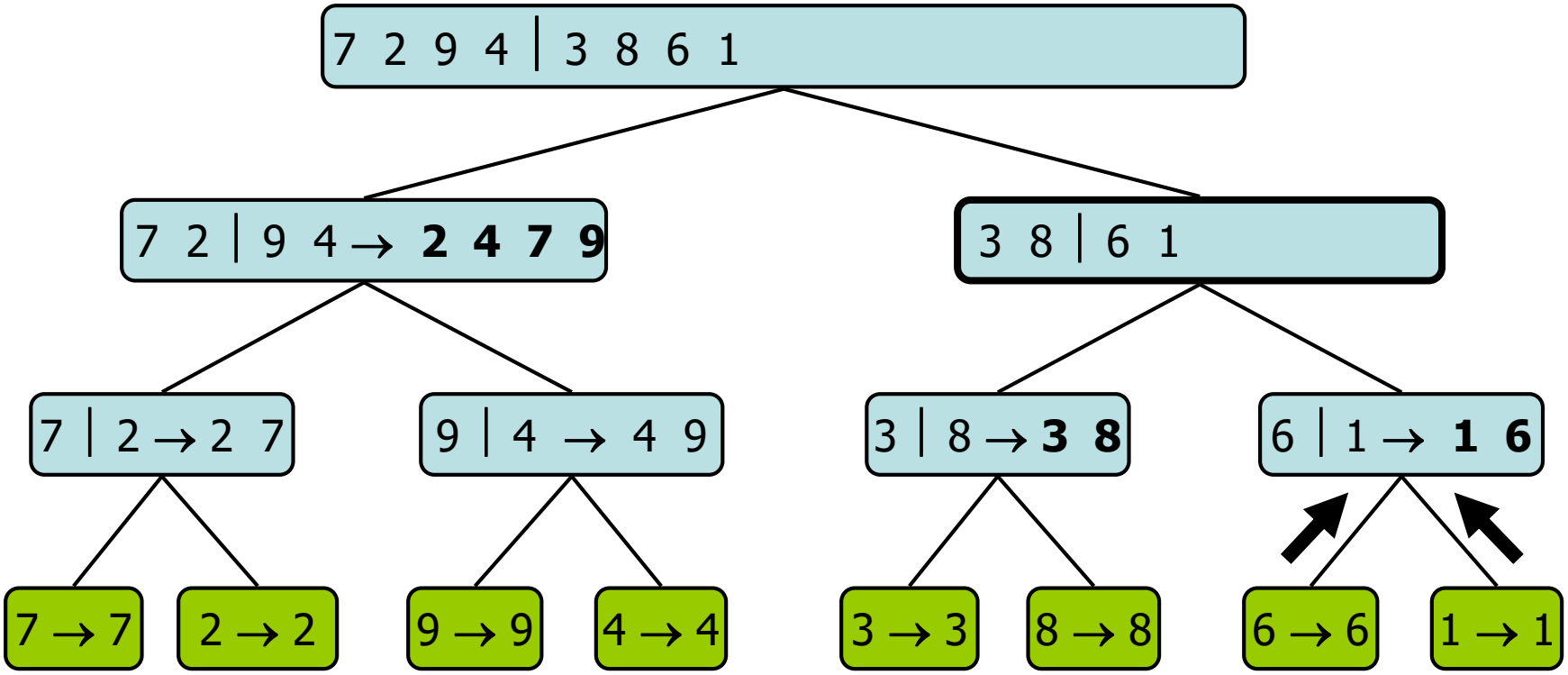
Execution Example (cont.)

- Recursive call, …., merge, merge



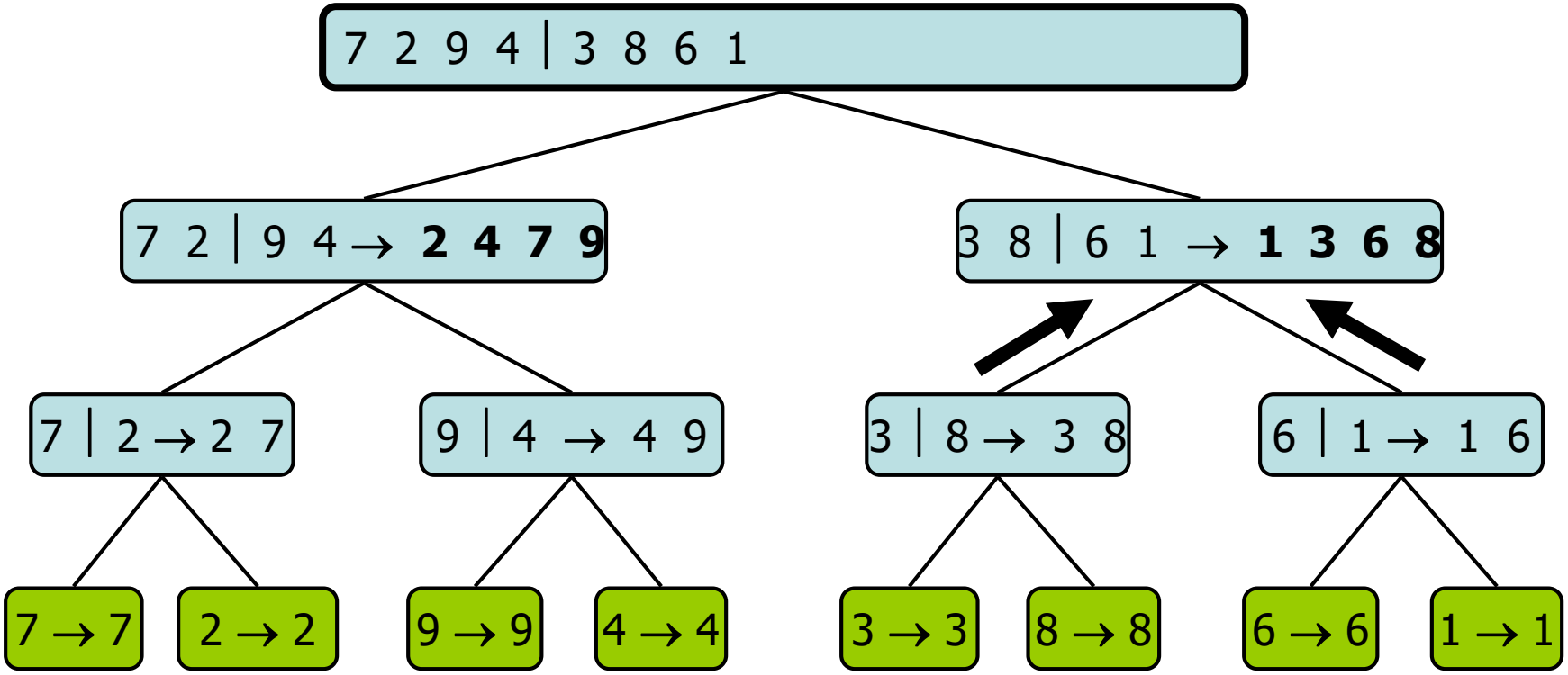
Execution Example (cont.)

- Recursive call, …., merge, merge



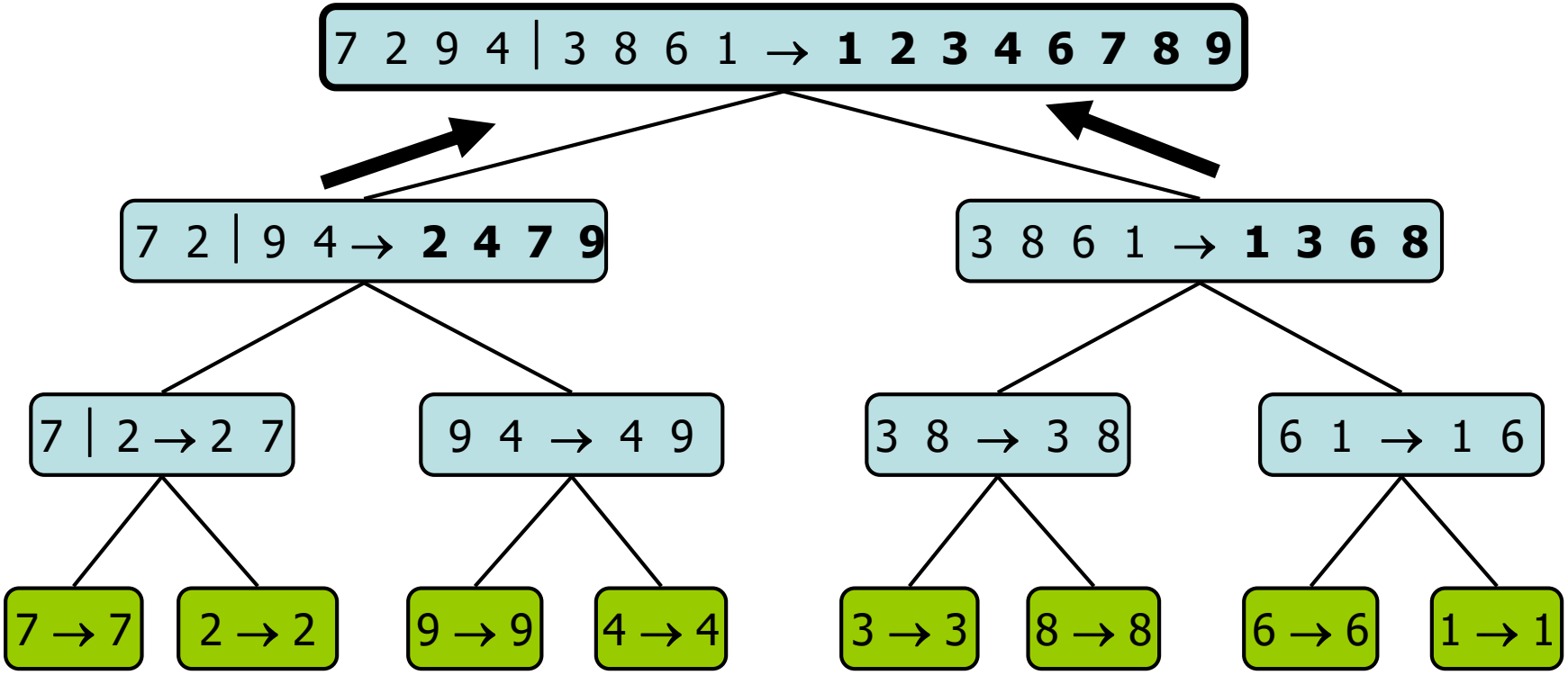
Execution Example (cont.)

- Merge



Execution Example (cont.)

- Merge



Functions for Merge Sort

```
// l is for left index and r is
// right index of the sub-array
// of arr to be sorted
void mergeSort(int arr[], int l, int r)
{
  if (l < r) {
    // Same as (l+r)/2, but avoids
    // overflow for large l and r
    int m = (l + r) / 2;

    // Sort first and second halves
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r - 1);

    merge(arr, l, m, r);
  }
}
```

```
// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
  int i, j, k;
  int n1 = m - l + 1;
  int n2 = r - m;

  // Create temp arrays
  int L[n1], R[n2];

  // Copy data to temp arrays
  // L[] and R[]
  for (i = 0; i < n1; i++)
    L[i] = arr[l + i];
  for (j = 0; j < n2; j++)
    R[j] = arr[m + 1 + j];
```

```
// Merge the temp arrays back
 // into arr[l..r]
 // Initial index of first subarray
  i = 0;
// Initial index of second subarray
  j = 0;
// Initial index of merged subarray
  k = l;
  while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
      arr[k] = L[i];
      i++;
    }
    else {
      arr[k] = R[j];
      j++;
    }
    k++;
  }
```

```
// Copy the remaining elements
  // of L[], if there are any
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }
 // Copy the remaining elements of
// R[], if there are any
  while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
  }
}
```

Improved version of merge sort by 23BCS100

```
#include<stdio.h>
/* Merges two subarrays of arr[].First subarray
is arr[l..m] Second subarray is arr[m+1..r] */
void merge(int arr[], int l, int m, int r)
{int i, j, k;
int n1 = m - l + 1;
int n2 = r - m;
// Create temp arrays
int L[n1], R[n2];
// Copy data to temp arrays L[] and R[]
for (i = 0; i < n1; i++)
            L[i] = arr[l + i];
for (j = 0; j < n2; j++)
            R[j] = arr[m + 1 + j];
// Merge the temp arrays back into arr[l..r]
//Initial index of first subarray
i = 0;
// Initial index of second subarray
j = 0;
// Initial index of merged subarray
k = l;
while (i < n1 && j < n2)
{if (L[i] <= R[j])
            {arr[k] = L[i];i++;}
else { arr[k] = R[j];
       j++;}
k++;}
// Copy the remaining elements of L[],
// if there are any
while (i < n1)
{ arr[k] = L[i];
i++;
k++;}
// Copy the remaining elements of R[],
//if there are any
while (j < n2)
{arr[k] = R[j];
j++;
k++;}
for(i=0;i<k;i++)
            printf("%d  ",arr[i]);
printf("\n\n");}
```

```
// l is for left index and r is right index of the
sub-array of arr to be sorted
void mergeSort(int arr[], int l, int r)
{
if (l < r) {
// Same as (l+r)/2, but avoids overflow for large
l and r
int m = (l + r) / 2;
// Sort first and second halves
mergeSort(arr, l, m);
mergeSort(arr, m + 1, r);
//instead of r-1 , take r
merge(arr, l, m, r);
}
}
main()
{int a[]={3,5,1,6,7,8,2,4}, l=0, r=7;
 //instead of taking length as r , take r=length-1
mergeSort(a,l,r);
for(int i = 0 ; i <= r; i++)
 //instead of i<r , keep i<=r
printf("%d  ",a[i]);
}
```

Program Explanation

1. The MergeSort function takes the value of the minimum and maximum index of the array

2. It calculates the mid by the formula (low+high)/2 and recursively calls itself for low to mid and mid+1 to high and then calls the merge function to merge the sorted array.

3. The Merge function merges the two arrays by comparing the value in each and taking the minimum value in the newly created auxiliary array.

4. At last, whichever array is non-empty its value is just copied in the auxiliary array.

5. In the Driver Program user just declare the array and takes the input and calls the Merge Sort and merge function as shown in the code and prints the sorted result using for loop.

Time Complexity: O(nlogn)

- Time Complexity of Merge Sort is O(nlogn) for best, average, and worst case.

Space Complexity: O(n)

- Space Complexity of Merge Sort using an array is O(n), because Merge Sort requires an Auxiliary/temporary array for copying the elements.