Unit-1
Divide and Conquer

- Dr. K.RAGHAVA RAO
  - Professor in CSE
    - KL University
- krraocse@gmail.com
- http://mcadaa.blog.com

Divide-and-Conquer

- Divide the problem into a number of subproblems.

- Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, solve the subproblems in a straightforward manner.

Divide-and-Conquer

- Combine the solutions to the subproblems into the solution for the original problem.

Merge Sort Algorithm

Divide: Divide the n-element sequence into two subsequences of n/2 elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted sequences.

How to merge two sorted sequences

- We have two subarrays A[p..q] and A[q+1..r] in sorted order.
- Merge sort algorithm merges them to form a single sorted subarray that replaces the current subarray A[p..r]

To sort the entire sequence A[1 .. n], make the initial call to the procedure MERGE-SORT (*A*, 1, *n*).

```
MERGE-SORT (A, p, r){
1.    IF p < r                                  // Check for base case
2.       THEN q = FLOOR[(p + r)/2]              // Divide step
3.             MERGESORT (A, p, q)              // Conquer step.
4.             MERGE SORT(A, q + 1, r)          // Conquer step.
5.             MERGE (A, p, q, r)               // Conquer step.
}
```

The **pseudocode** of the MERGE procedure is as follow:

MERGE (A, p, q, r)

$n_1 \leftarrow q - p + 1$

$n_2 \leftarrow r - q$

Create arrays L[1 . . $n_1 + 1$] and R[1 . . $n_2 + 1$]

FOR $i \leftarrow 1$ **TO** n_1

    DO L[i] $\leftarrow$ A[$p + i - 1$]

FOR $j \leftarrow 1$ **TO** n_2

    DO R[j] $\leftarrow$ A[$q + j$]

L[$n_1 + 1$] $\leftarrow \infty$

R[$n_2 + 1$] $\leftarrow \infty$

$i \leftarrow 1$

$j \leftarrow 1$

FOR $k \leftarrow p$ **TO** r

    DO IF L[i] $\leq$ R[j]

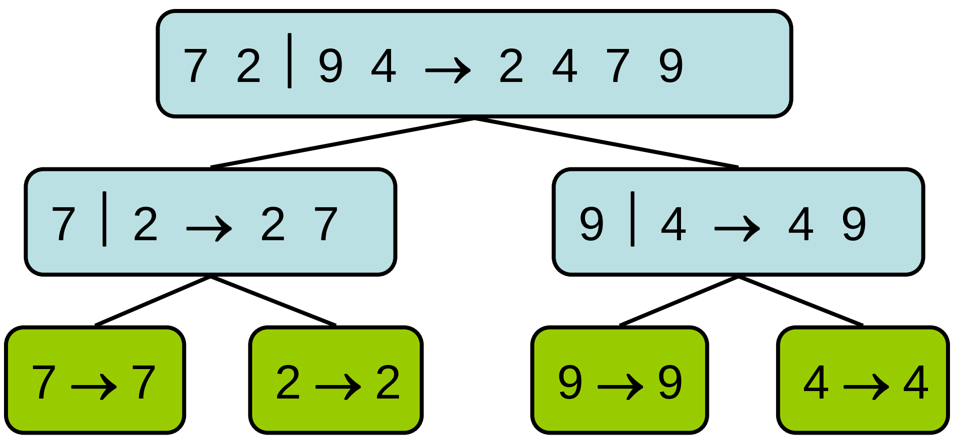
        THEN A[k] $\leftarrow$ L[i]

            $i \leftarrow i + 1$

        ELSE A[k] $\leftarrow$ R[j]

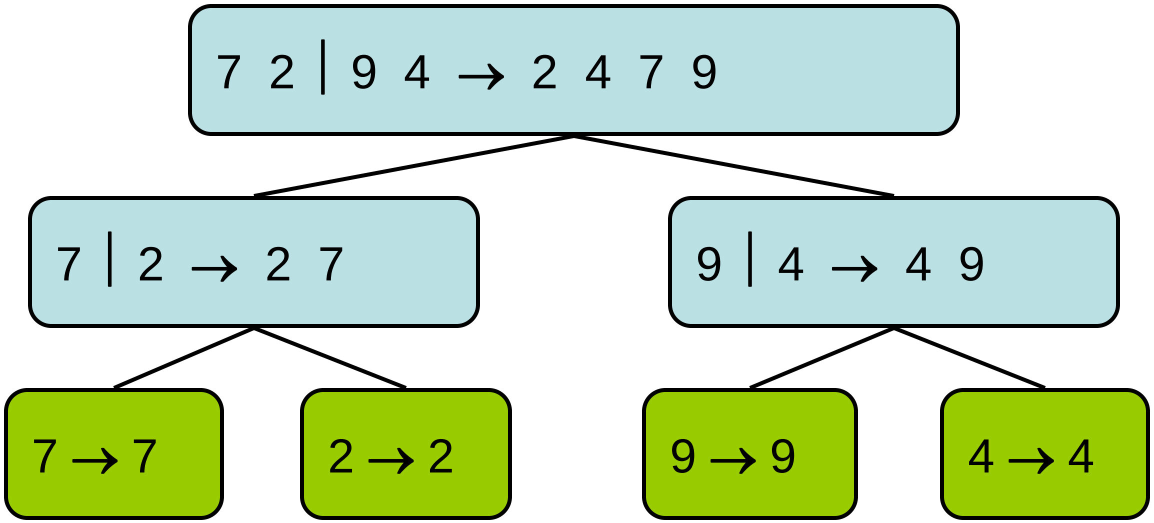
            $j \leftarrow j + 1$

Merge Sort



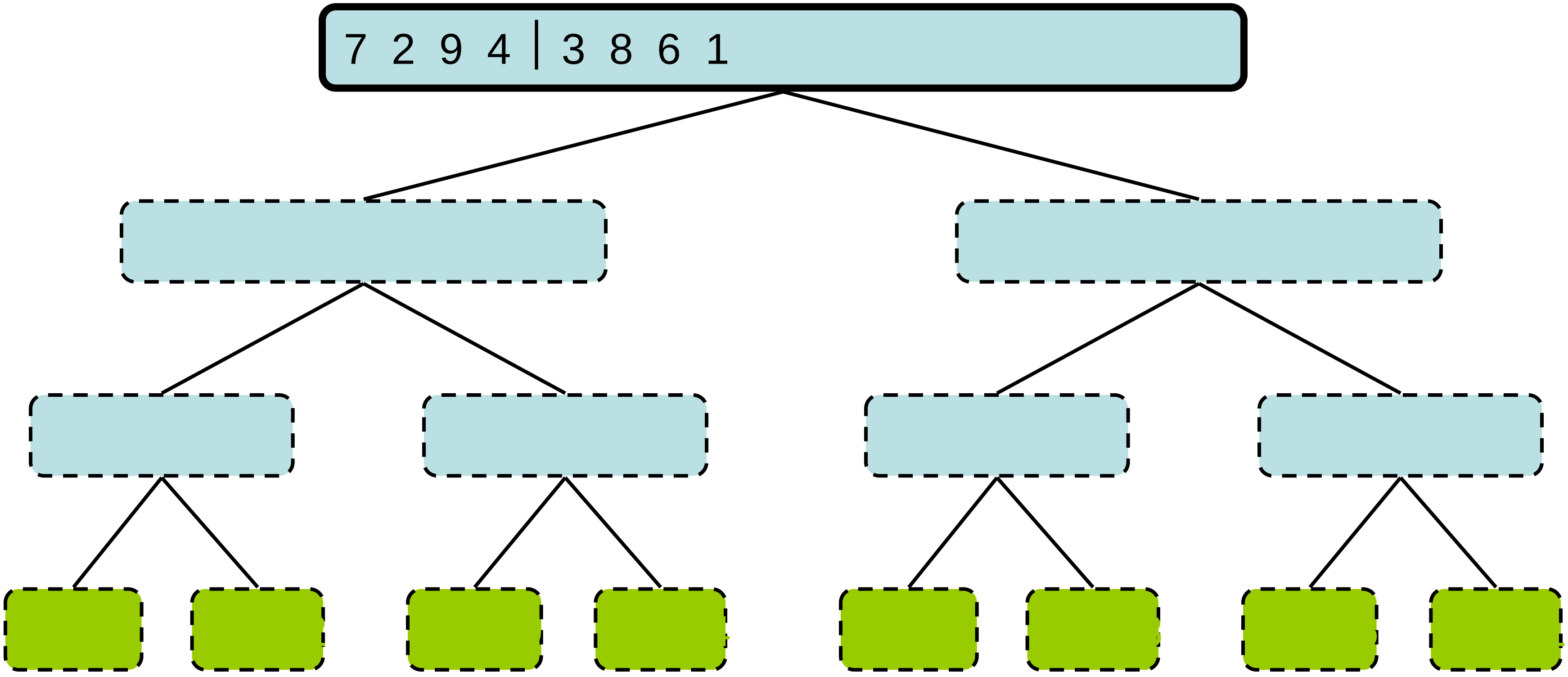
Merge-Sort Tree

- 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



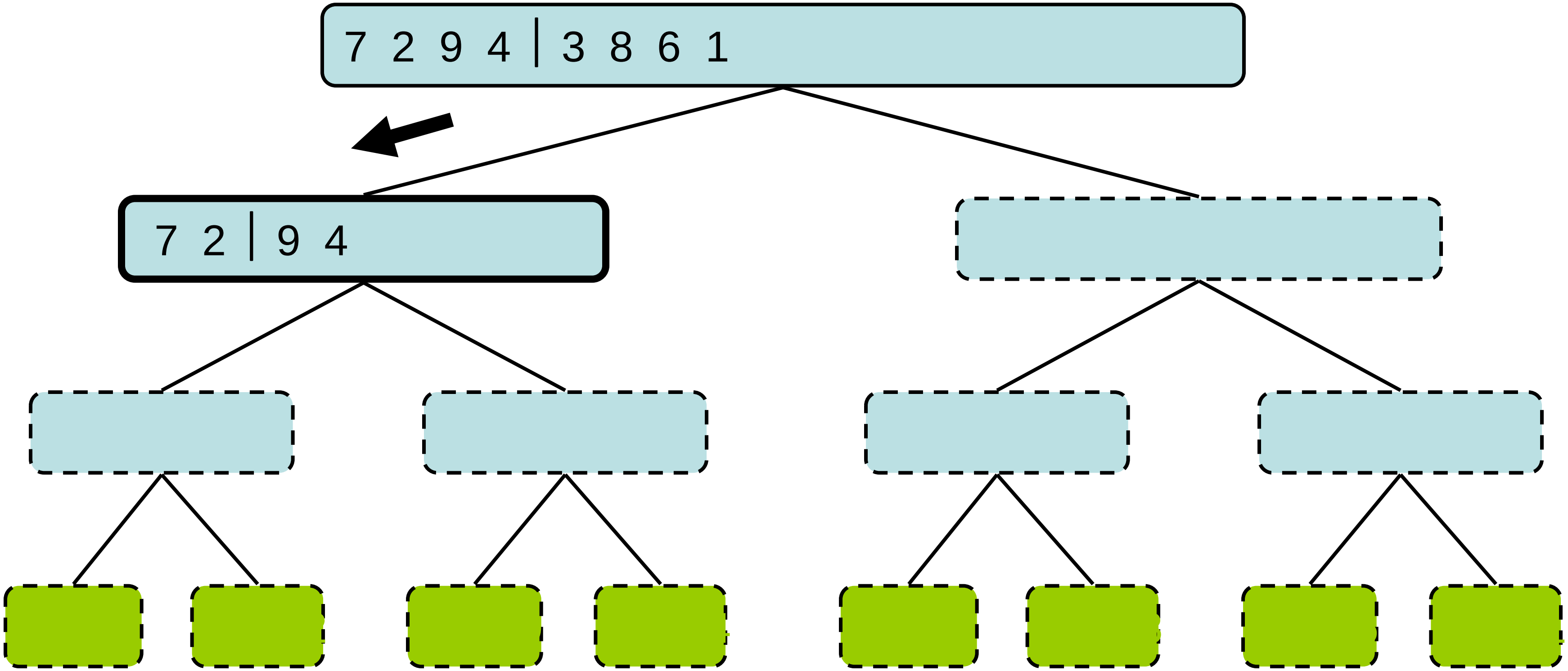
Execution Example

- Partition



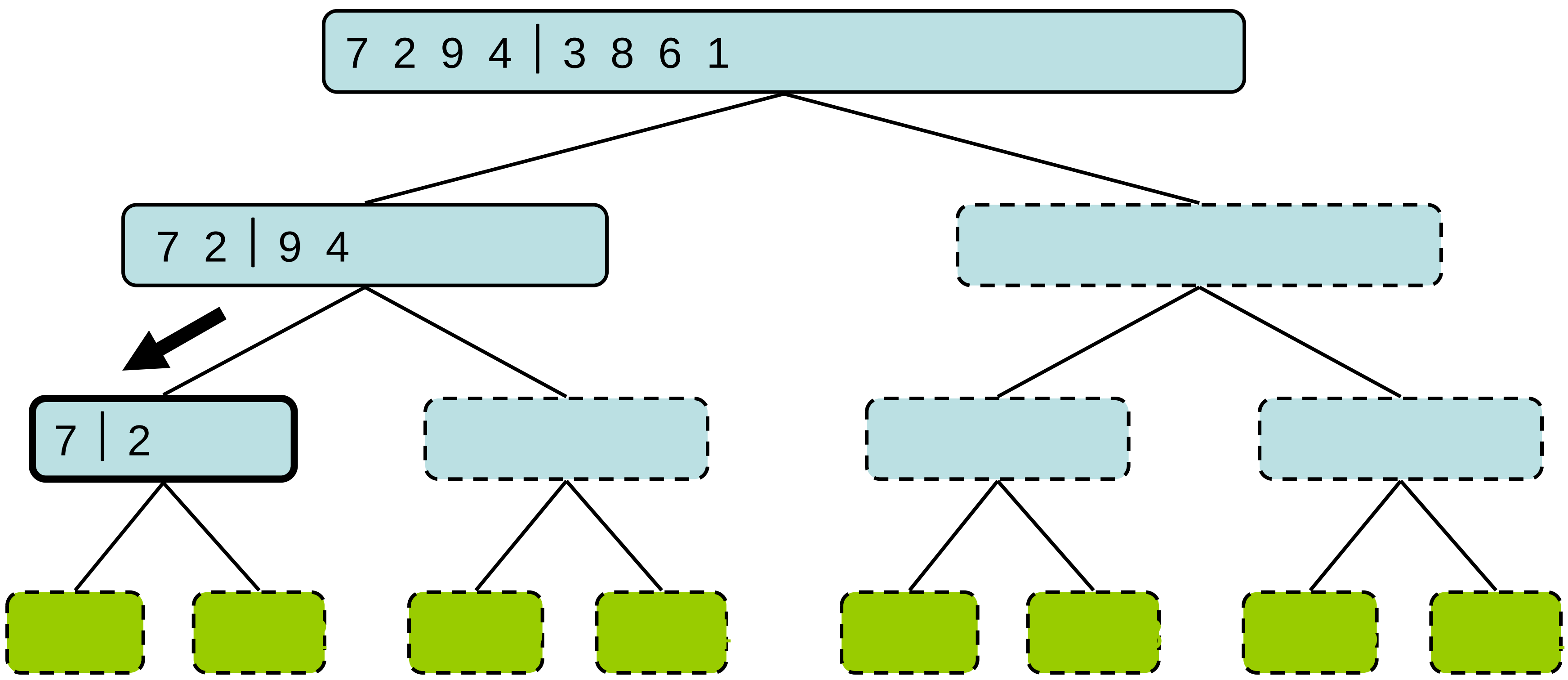
Execution Example (cont.)

- Recursive call, partition



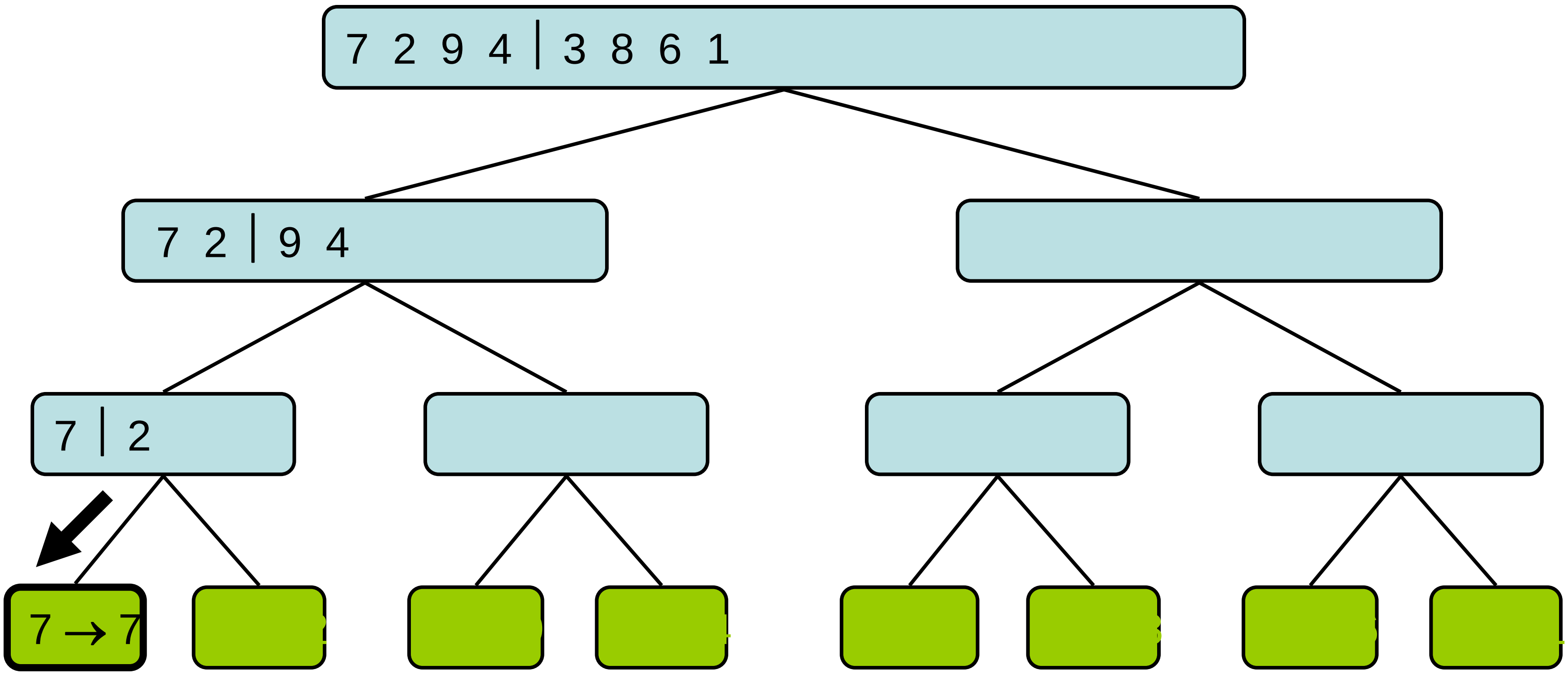
Execution Example (cont.)

- Recursive call, partition



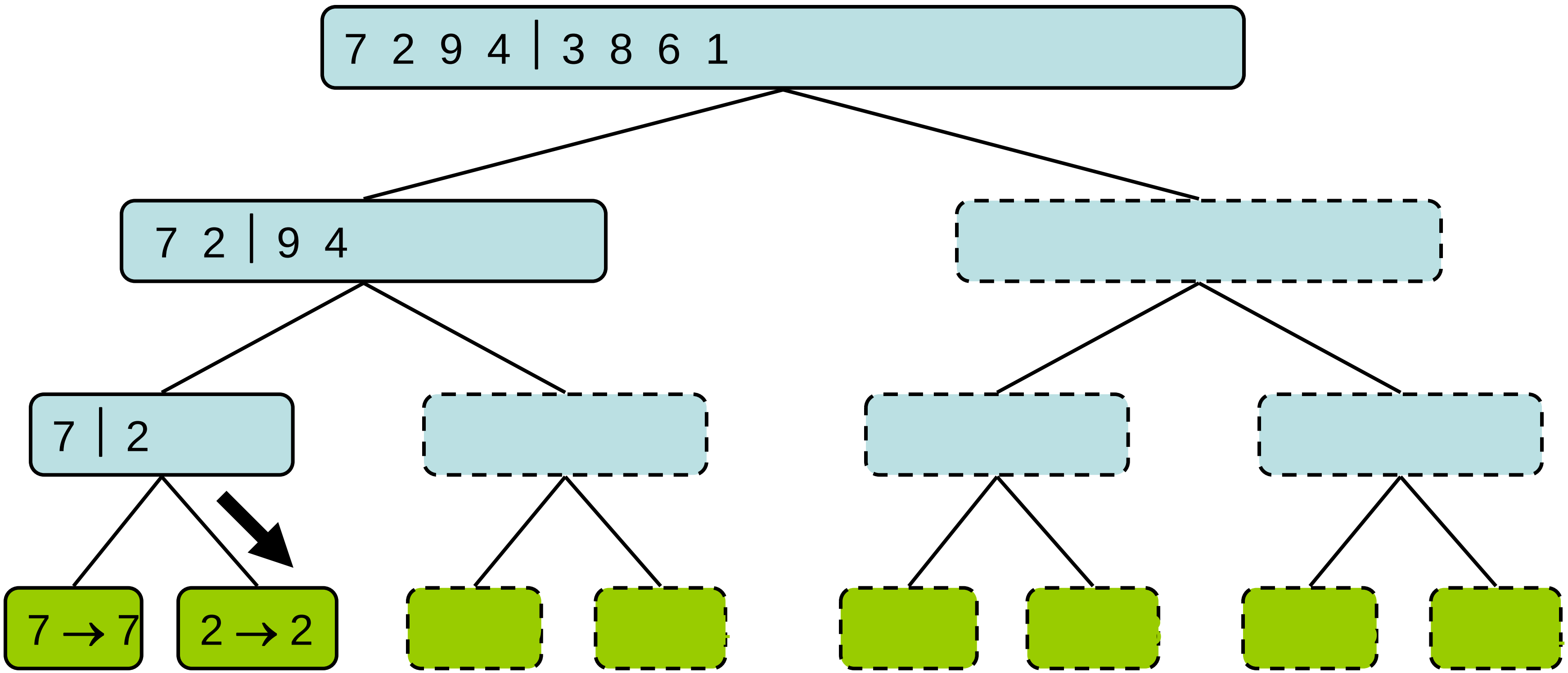
Execution Example (cont.)

- Recursive call, base case



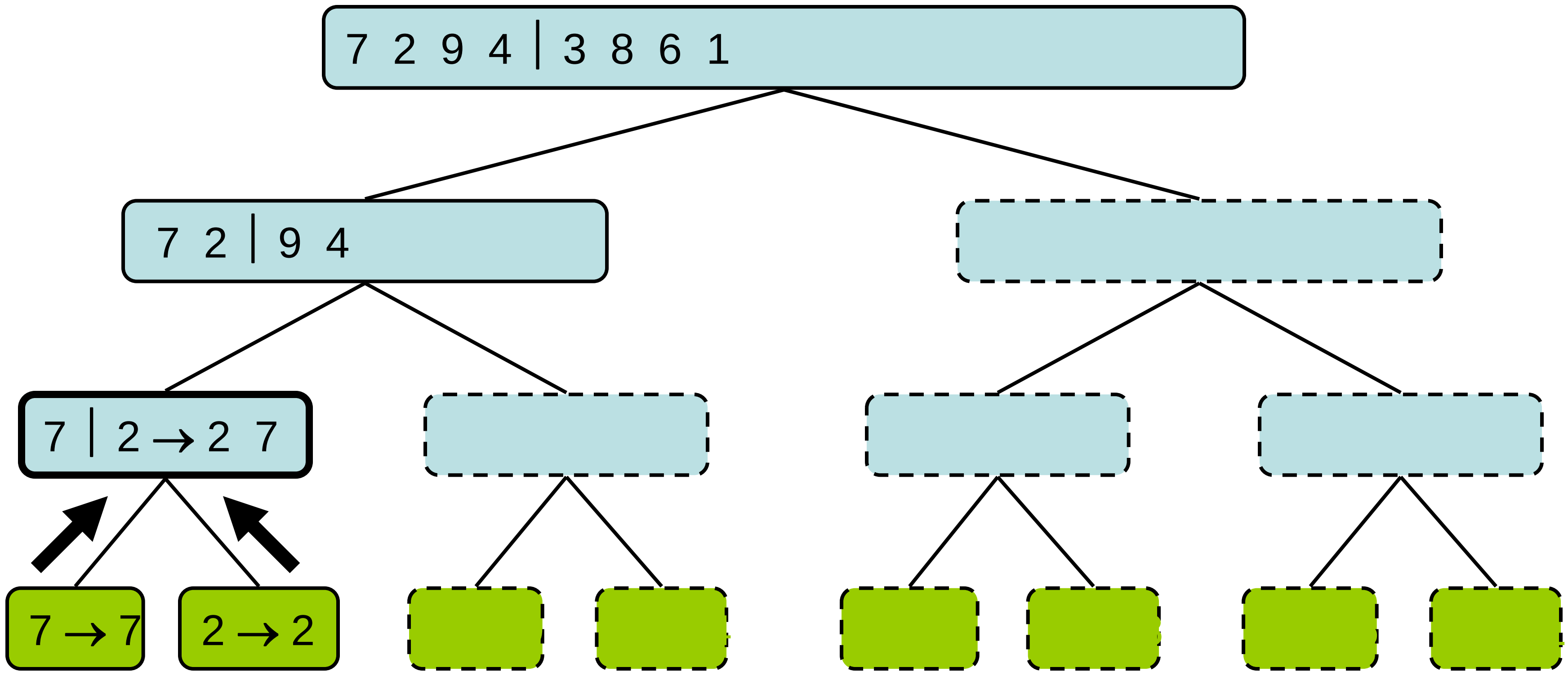
Execution Example (cont.)

- Recursive call, base case



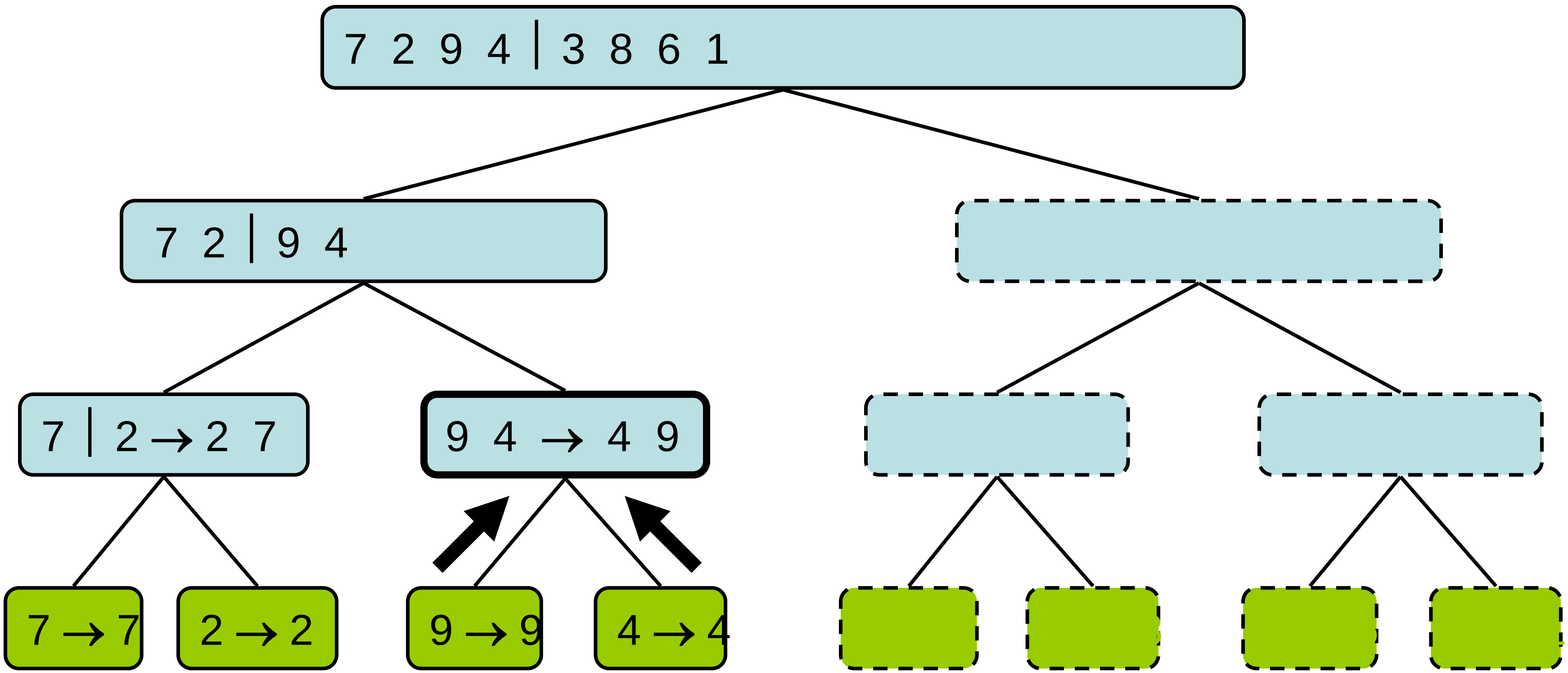
Execution Example (cont.)

- Merge



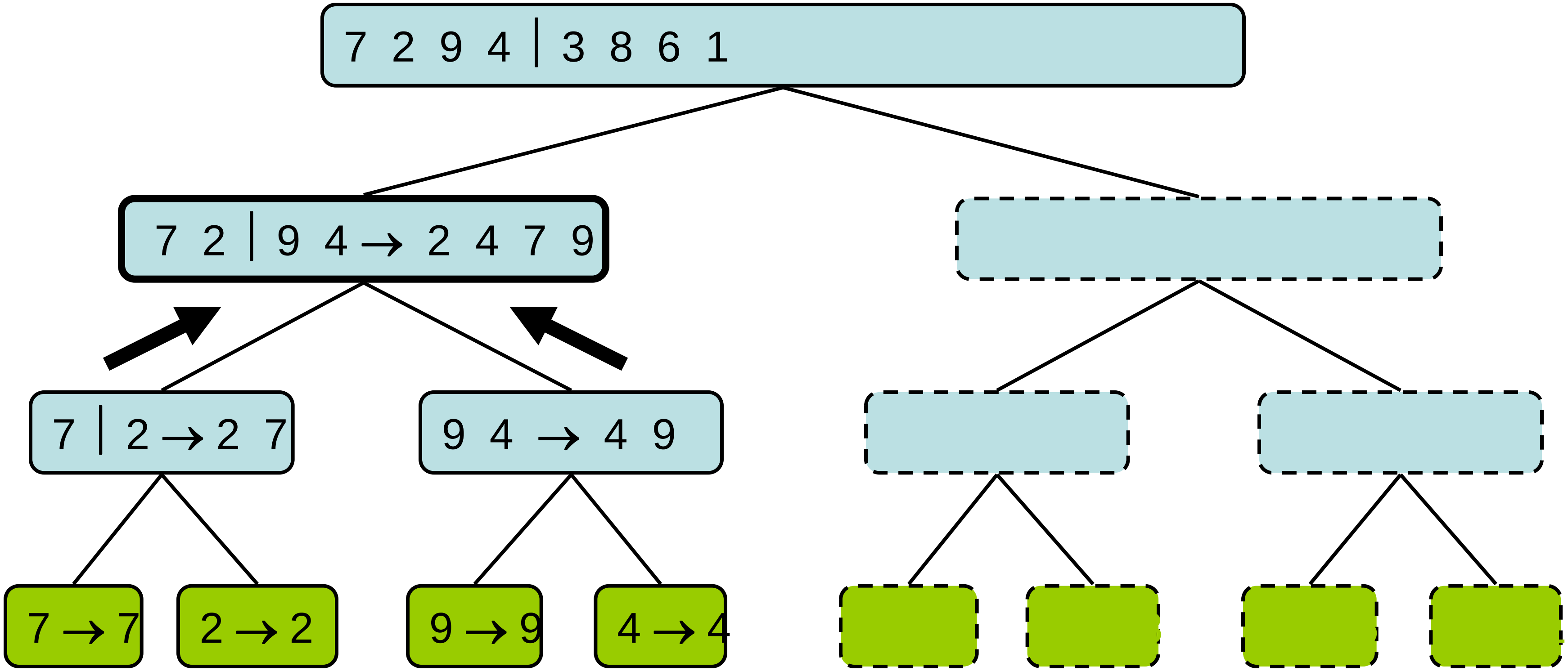
Execution Example (cont.)

- Recursive call, …, base case, merge



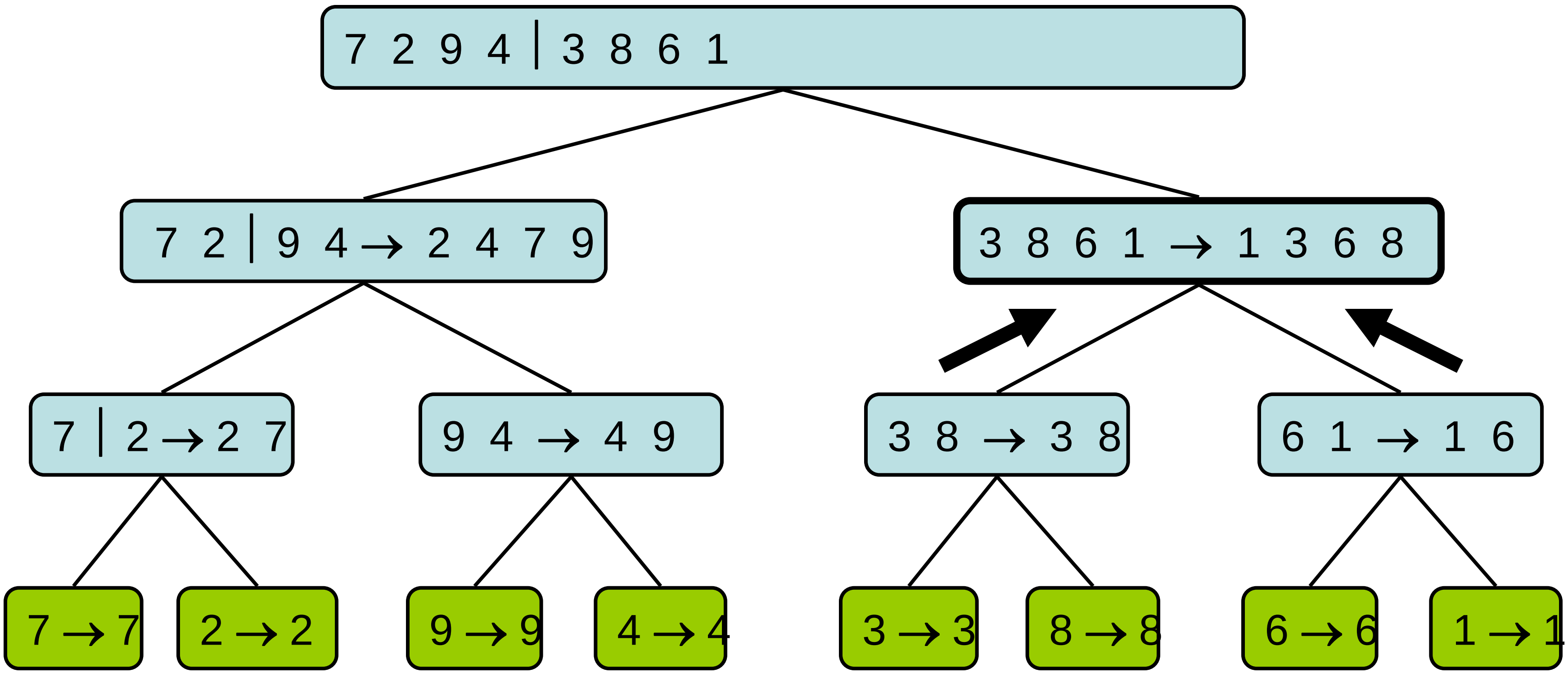
Execution Example (cont.)

- Merge



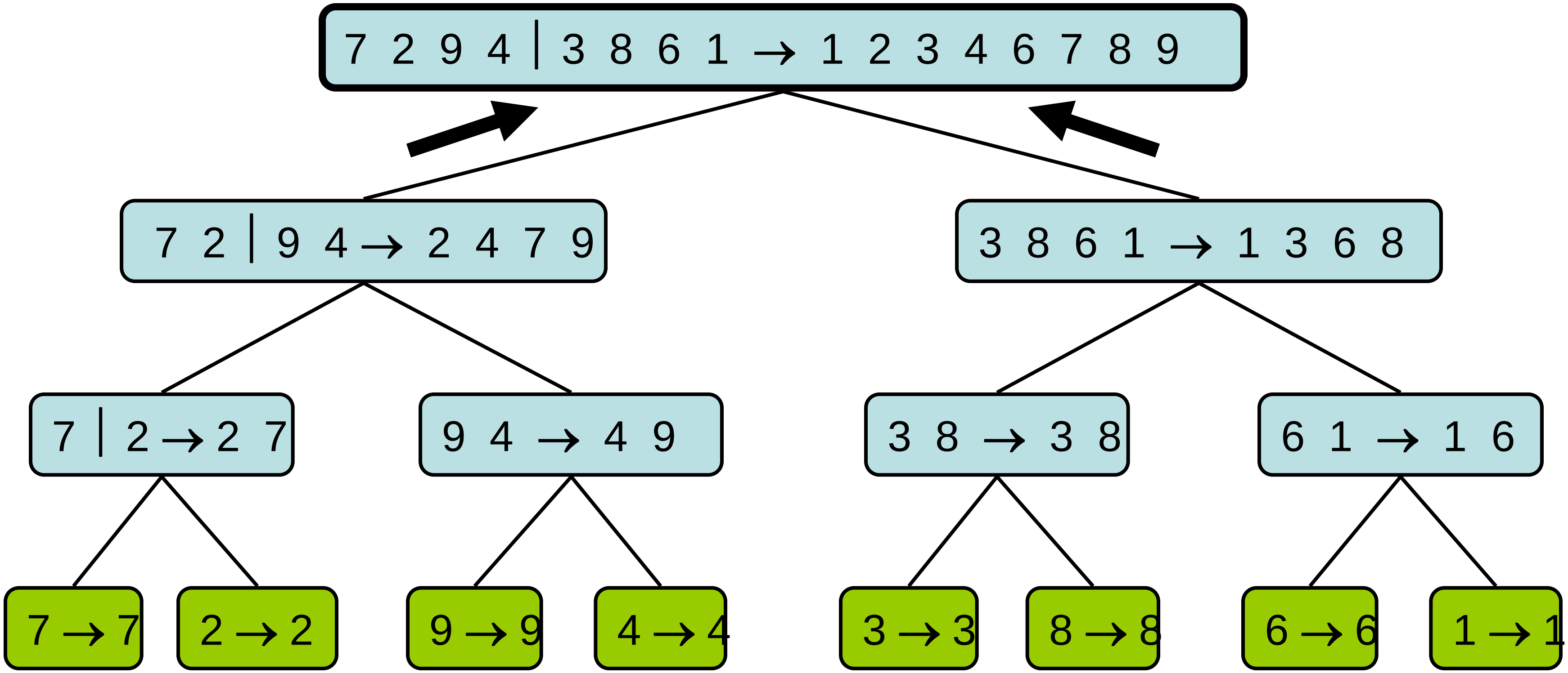
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

- Merge



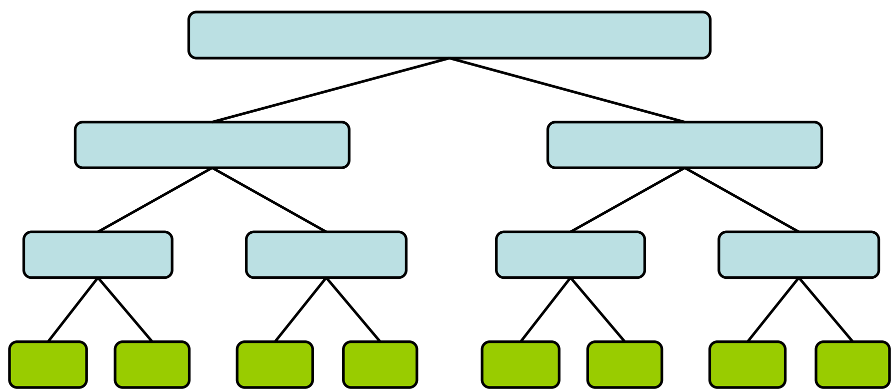
Analyzing Divide-and-Conquer Algorithm

When an algorithm contains a recursive call to itself, its running time can be described by a recurrence equation or recurrence which describes the running time

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
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Recurrence

If the problem size is small enough, say $n <= c$ for some constant c, the straightforward solution takes constant time, can be written as $\theta(1)$.

Recurrence

If we have *a* subproblems, each of which is *1/b* the size of the original. D(n) time to divide the problem and C(n) time to combine the solution.

Recurrence

The recurrence

$$T(n) = \begin{cases} \theta(1) & \text{if } n <= c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Recurrence

Divide: The divide step computes the

middle of the subarray which takes

constant time, $D(n)=\theta(1)$

Recurrence

Conquer: We recursively solve two subproblems, each of size n/2, which contributes 2T(n/2) to the running time.

Recurrence

Combine: Merge procedure takes θ(n) time

on an n-element subarray. C(n)=θ(n)

The recurrence

$$T(n) = \begin{cases} \theta(1) & \text{if } n=1 \\ 2T(n/2) + \theta(n) & \text{if } n>1 \end{cases}$$

Recurrence

Let us rewrite the recurrence

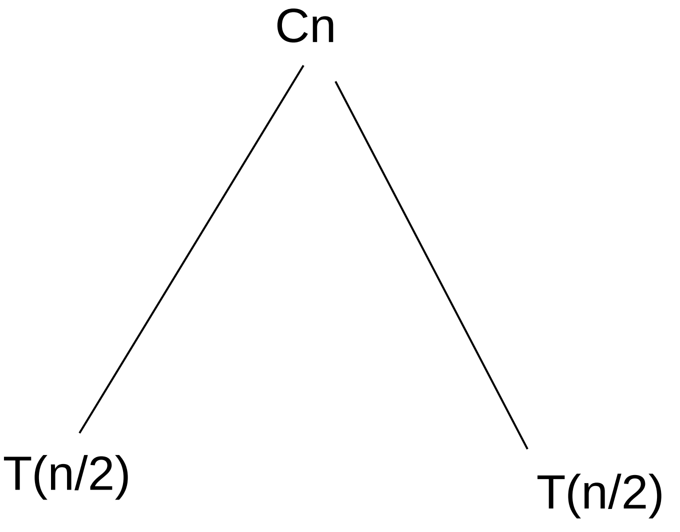
$$T(n) = \begin{cases} C & \text{if } n=1 \\ 2T(n/2) + cn & \text{if } n>1 \end{cases}$$

C represents the time required to solve

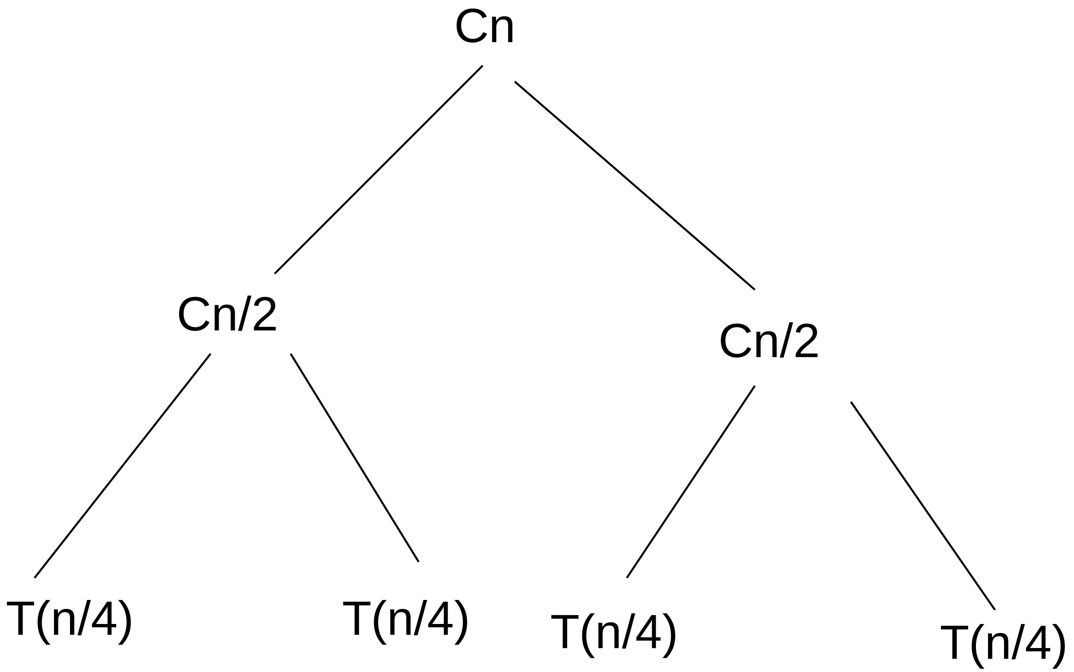
problems of size 1

A Recursion Tree for the Recurrence

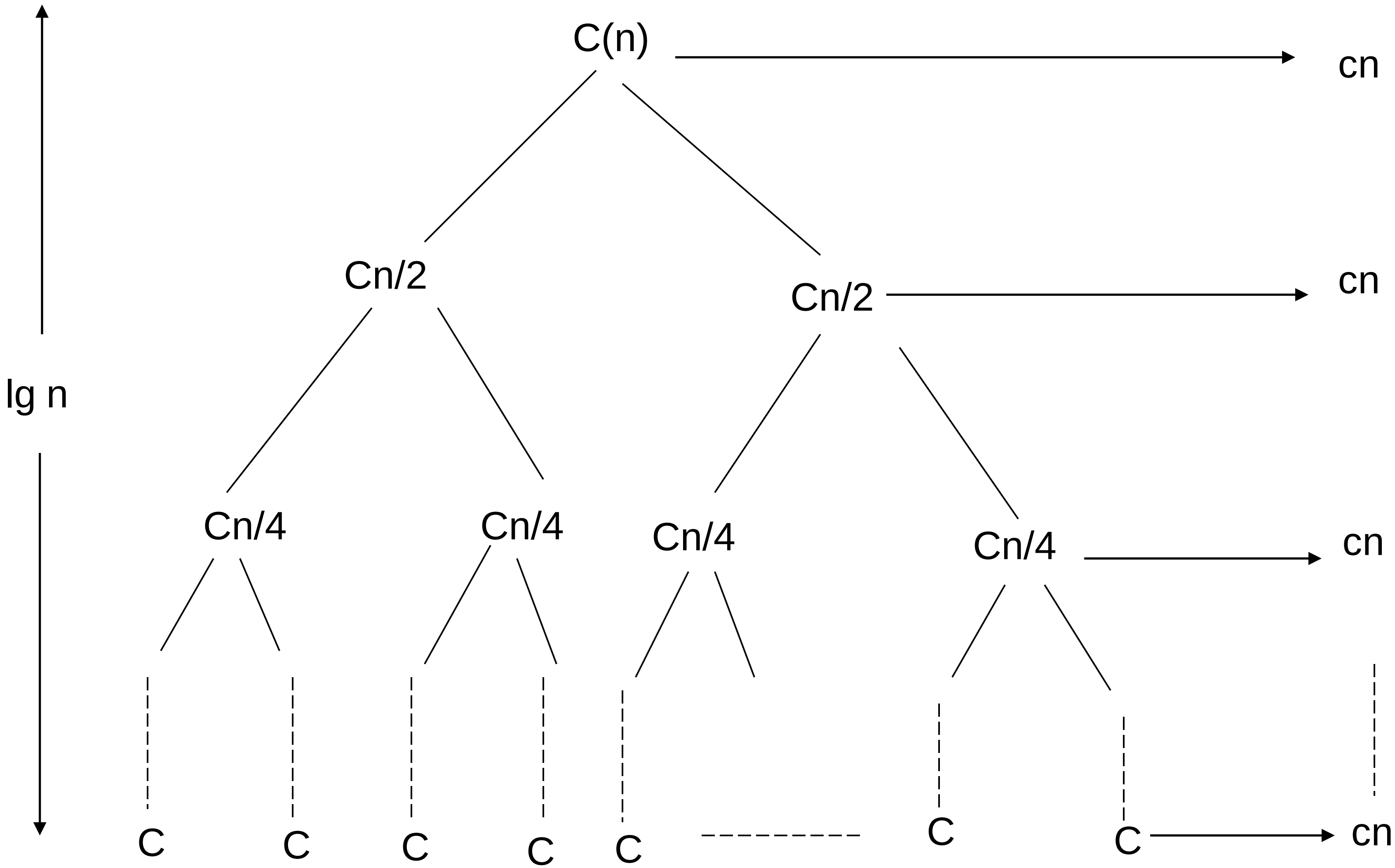
T(n)



A Recursion Tree for the Recurrence



A Recursion Tree for the Recurrence



A Recursion Tree for the Recurrence

- In the above recursion tree, each level has cost cn.
- The top level has cost cn.
- The next level down has 2 subproblems, each contributing cost $cn/2$.
- The next level has 4 subproblems, each contributing cost $cn/4$.
- Each time we go down one level, the number of subproblems doubles but the cost per subproblem halves. Therefore, cost per level stays the same.
- The height of this recursion tree is $\log n$ and there are $\log n + 1$ levels.

Total Running Time

- A tree for a problem size of 2^i has log 2^i + 1 = i +1 levels.
- The fully expanded tree recursion tree has log n+1 levels. When n=1 than 1 level log 1=0, so correct number of levels log n+1.

- Because we assume that the problem size is a power of 2, the next problem size up after 2^i is 2^i + 1. A tree for a problem size of 2^i + 1 has one more level than the size-2^i tree implying i + 2 levels.

- Since log 2^i + 1 = i + 2, we are done with the inductive argument.

- Total cost is sum of costs at each level of the tree. Since we have log n +1 levels, each costing cn, the total cost is cn log n + cn.

- Ignore low-order term of cn and constant coeffcient c, and we have, Θ(n log n)

Total Running Time

The fully expanded tree has lg n +1 levels and

each level contributes a total cost of cn.

Therefore $T(n)= cn \log n + cn = \theta(n\log n)$

Growth of Functions

We look at input sizes large enough to

make only the order of growth of the

running time relevant.