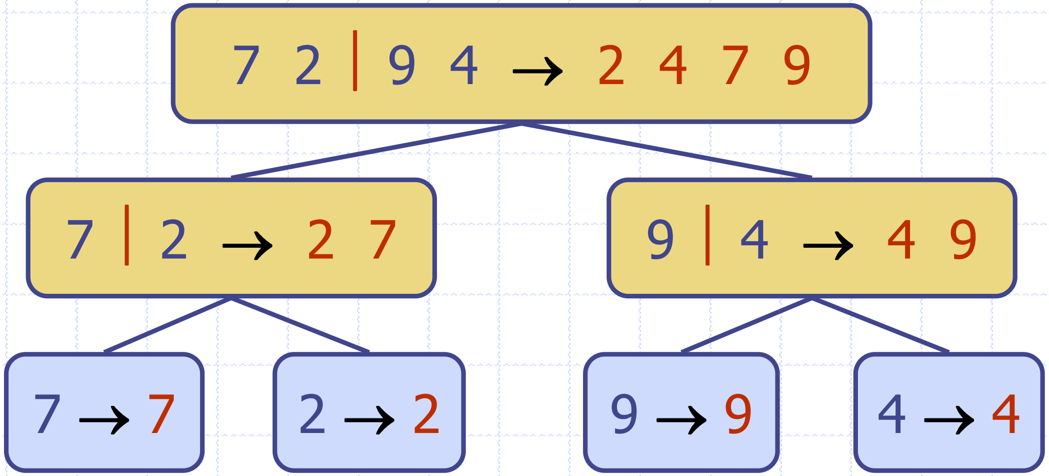
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014 With notes by R. Solis-Oba

Merge Sort



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
- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm

Merge-Sort

- Merge-sort on an input array 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*(*A,first,last*)
Input array A[first, …, last]
Output Sorted array A

```
    if first < last then {
        mid ← (first + last) / 2
        mergeSort(A,first,mid)
        mergeSort(A,mid+1,last)
        merge(A,first,mid,last)
    }
```

Algorithm ***merge(A, first, mid, last)***

Input: *Array A and* ***indices first, mid, last. The first half of the array A[first, …, mid] is sorted, and the second half A[mid+1, …,last] is also sorted***

Output: Sorted array *A*

```
B ← empty array of size n
i ← first
j ← mid + 1
iB ← first
while (i ≤ mid) and (j ≤ last) do {
    if A[i] < A[j] then {
        B[iB] ← A[i]
        i ←  i + 1
    }
    else {
        B[iB] ← A[j]
        j ← j + 1
    }
    iB ← iB + 1
}
```

```
if  i ≤ mid then   // There are values remaining in the first half of the array
    while (i ≤ mid) do {
        B[i_B] ← A[i]
        i ←  i + 1
        i_B ← i_B + 1
      }
else     // There are values remaining in the second half of the array
    while (j ≤ last) do {
        B[i_B] ← A[j]
        j ←  j + 1
        i_B ← i_B + 1
      }

for i ← first to last do   // Copy back all values to A
   A[i] ← B[i]

return A
```

Time Complexity

Algorithm **merge** has several loops. Each iteration of each loop performs a constant number of operations. To determine the total number of iterations performed by all the loops, note that the **while** loops copy each value from A to B, so the total number of iterations that the 3 loops perform is n.

The **for** loop copies all values back from B to A and so it also performs n iterations. Therefore, the total number of operations performed by algorithm *merge* is c_2n for some constant c_2, so the time complexity is O(n).

Let f(n) be the time complexity of *mergesort* when the input has size n. The following recurrence equation characterizes the time complexity of the algorithm:

$f(1) = c$, where c is a constant

$f(n) = c_1 + c_2 n + 2f(n/2)$, if $n > 1$, where c_1, c_2 are constants

Time Complexity

We solve the above recurrence equation using the method of repeated substitution. For simplicity, so we do not have to round numbers up or down, we assume that n is a power of 2, i.e. $n = 2^k$, for some integer k. Hence the recurrence equation can be written as

$f(2^0) = c$
$f(2^k) = c_1 + c_2 2^k + 2^1 f(2^{k-1})$, if $n > 1$. We need to compute $2^1 f(2^{k-1})$:
$2^1 f(2^{k-1}) = 2^1 c_1 + 2^1 c_2 2^{k-1} + 2^2 f(2^{k-2})$, now we need to compute $2^2 f(2^{k-2})$
$2^2 f(2^{k-2}) = 2^2 c_1 + c_2 2^2 2^{k-2} + 2^3 f(2^{k-3})$, and so on.
.
.
.
$2^{k-1} f(2^1) = 2^{k-1} c_1 + c_2 2^{k-1} 2^1 + 2^k f(2^0)$.

Substituting the value of $2^1 f(2^{k-1})$ in the formula for $f(2^k)$, then substituting the value of $2^2 f(2^{k-2})$ in this formula and so on, we get

$f(2^k) = c_1 + c_2 2^k + 2^1 c_1 + 2^1 c_2 2^{k-1} + 2^2 c_1 + c_2 2^2 2^{k-2} + \ldots + 2^{k-1} c_1 + c_2 2^{k-1} 2^1 + 2^k f(0)$

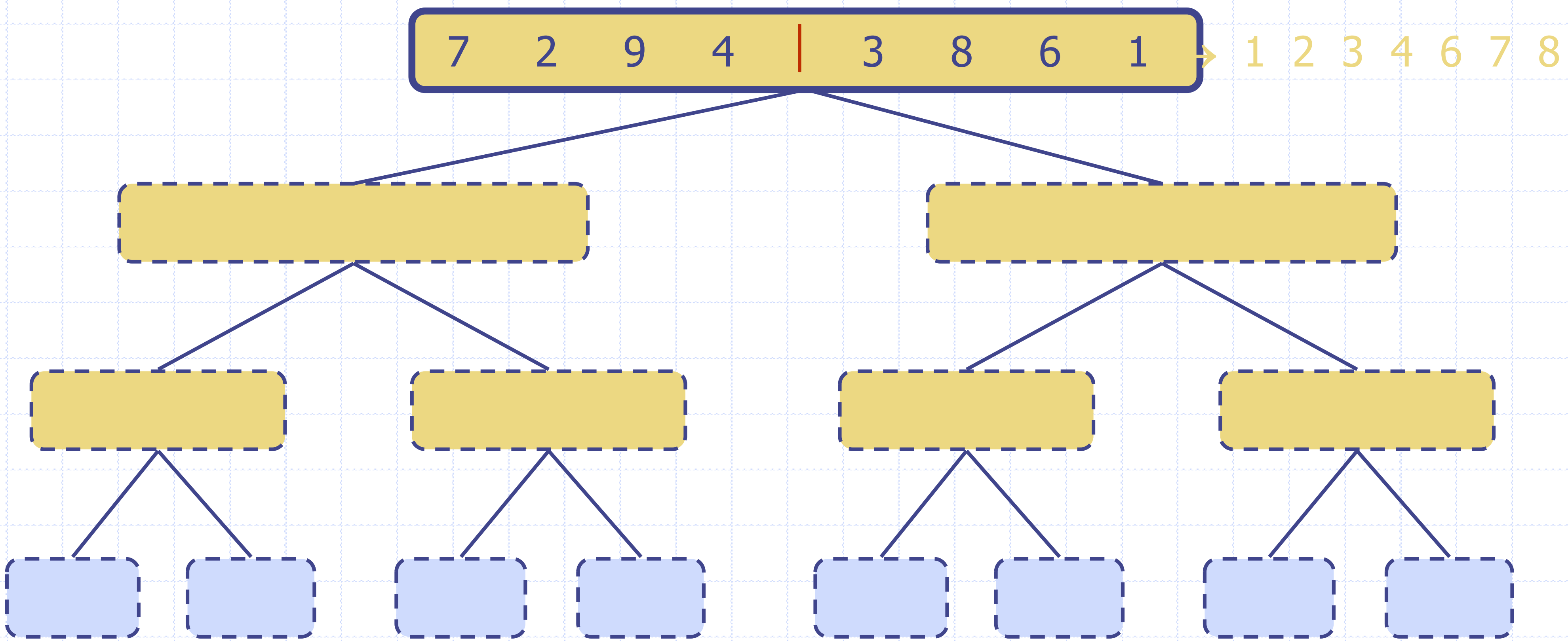
Time Complexity

Then,

$$
\begin{aligned}
f(n) &= f(2^k) = c_1 + c_2 2^k + 2^1 c_1 + c_2 2^k + 2^2 c_1 + c_2 2^k + \ldots + 2^{k-1} c_1 + c_2 2^k + 2^k f(0) \\
&= c_1 (2^0 + 2^1 + 2^2 + \ldots + 2^{k-1}) + c_2 2^k k + 2^k c \\
&= c_1 (2^k - 1) + c_2 2^k k + 2^k c = 2^k (c_1 + c) + c_2 2^k k - c_1 \\
&= (c_1 + c)\, n + c_2\, n \log n - c_1 \text{ is } O(n \log n)
\end{aligned}
$$

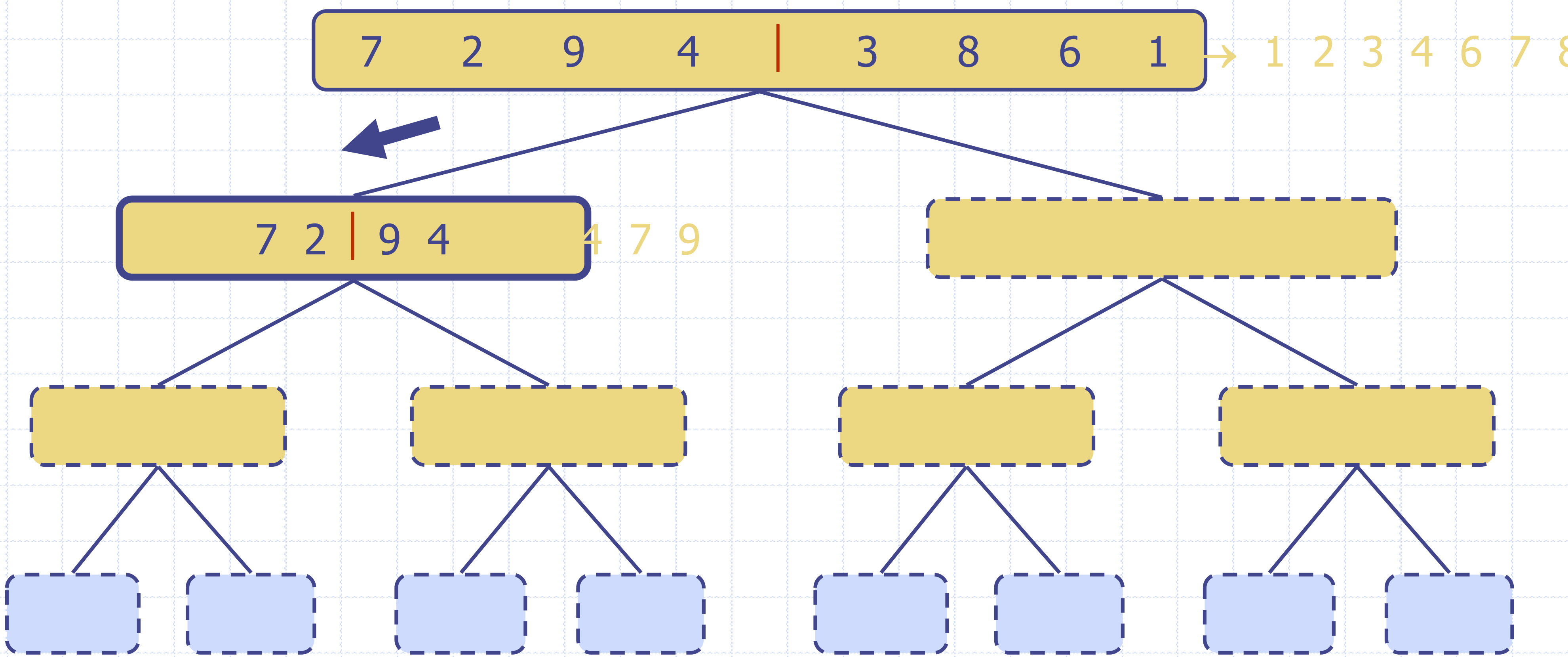
Execution Example. Execution tree

- Partition



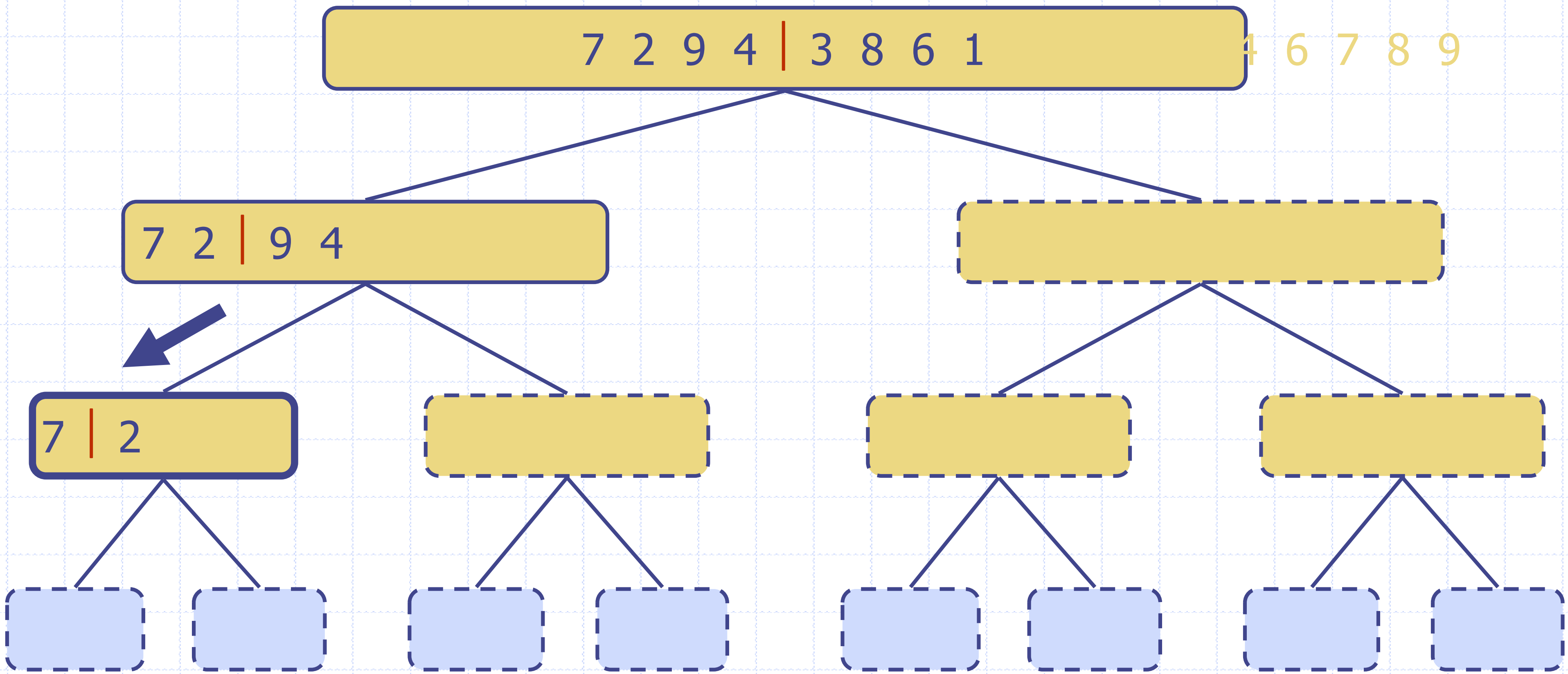
Execution Example (cont.)

- Recursive call, partition



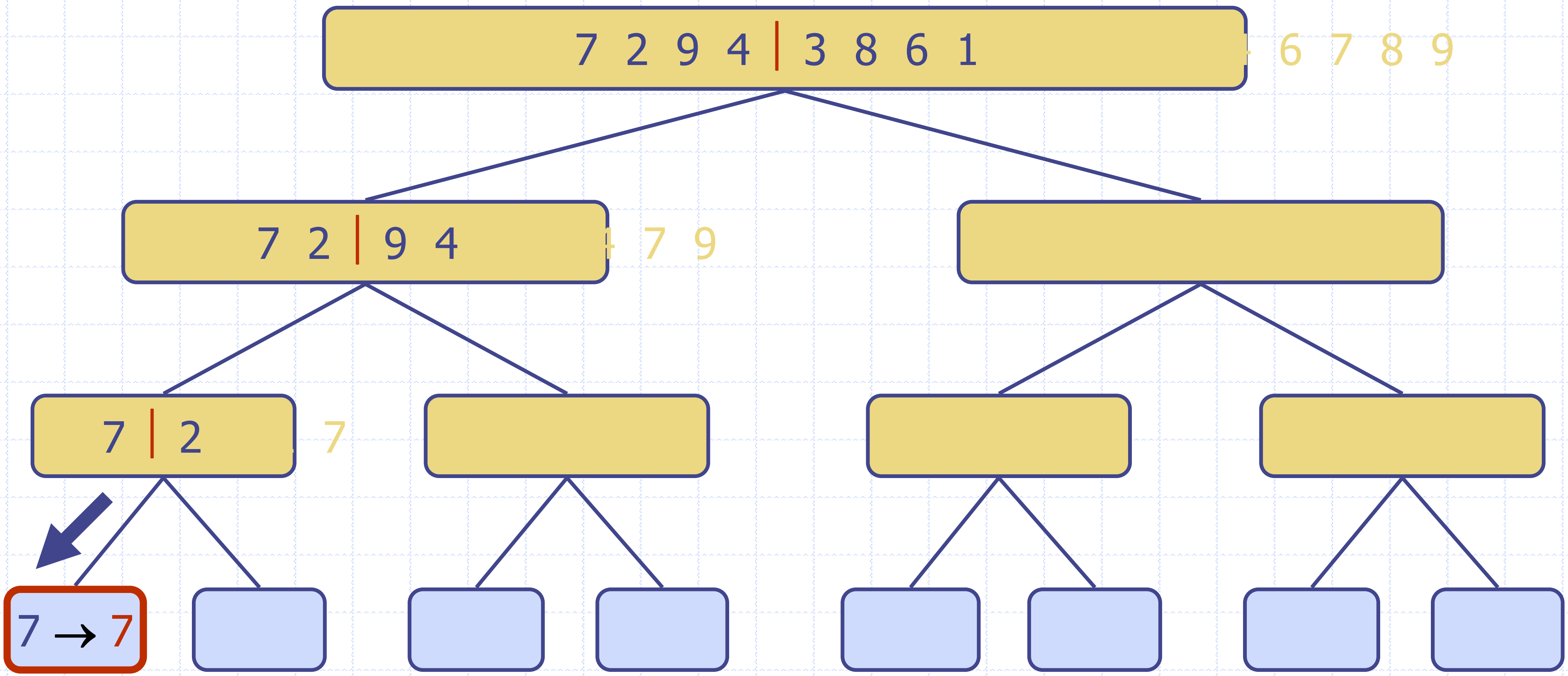
Execution Example (cont.)

- Recursive call, partition



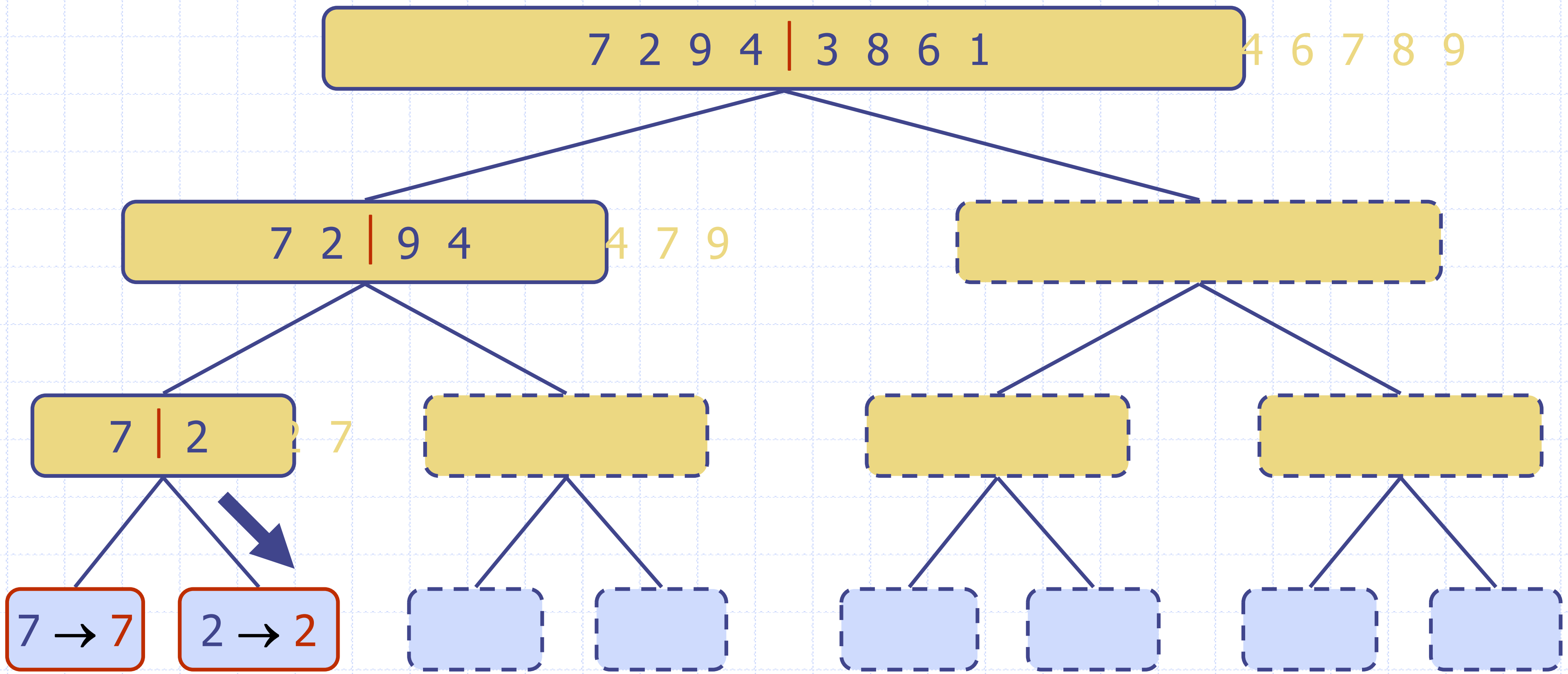
Execution Example (cont.)

- Recursive call, base case



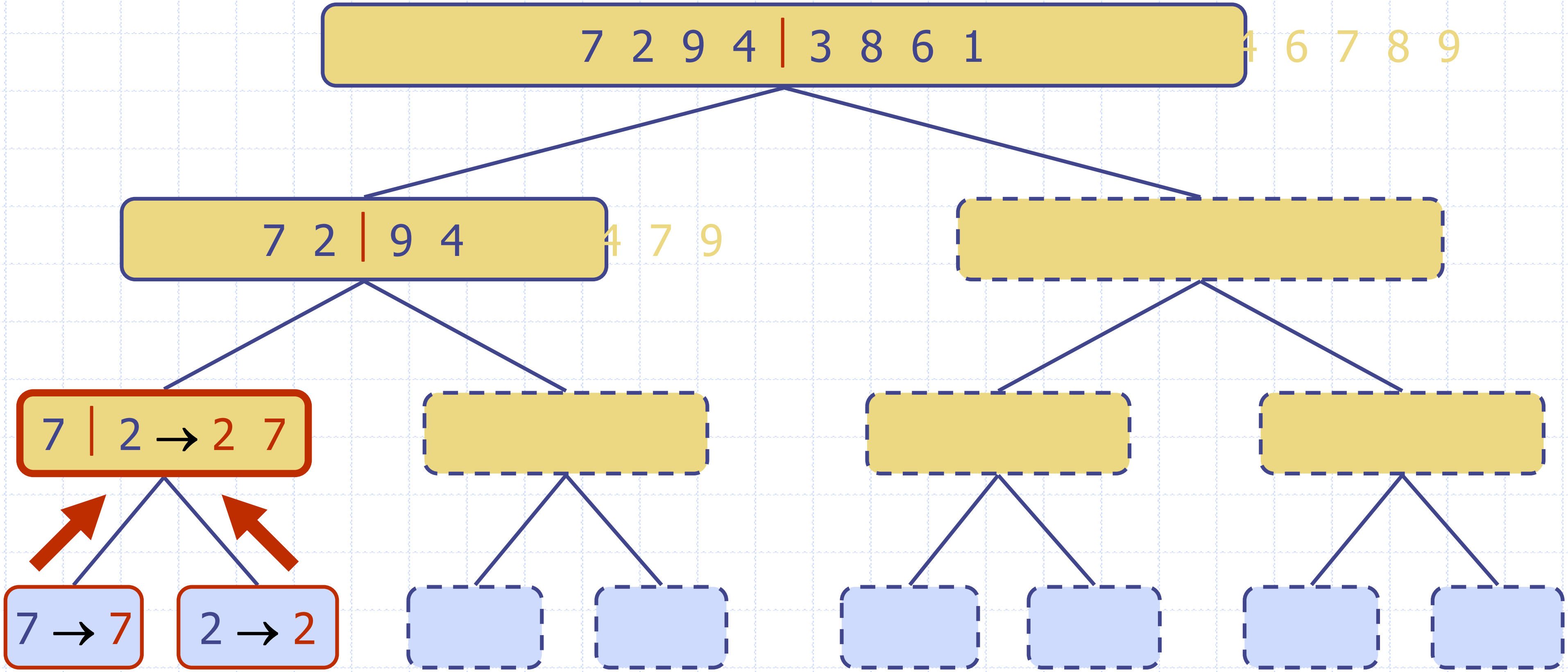
Execution Example (cont.)

- Recursive call, base case



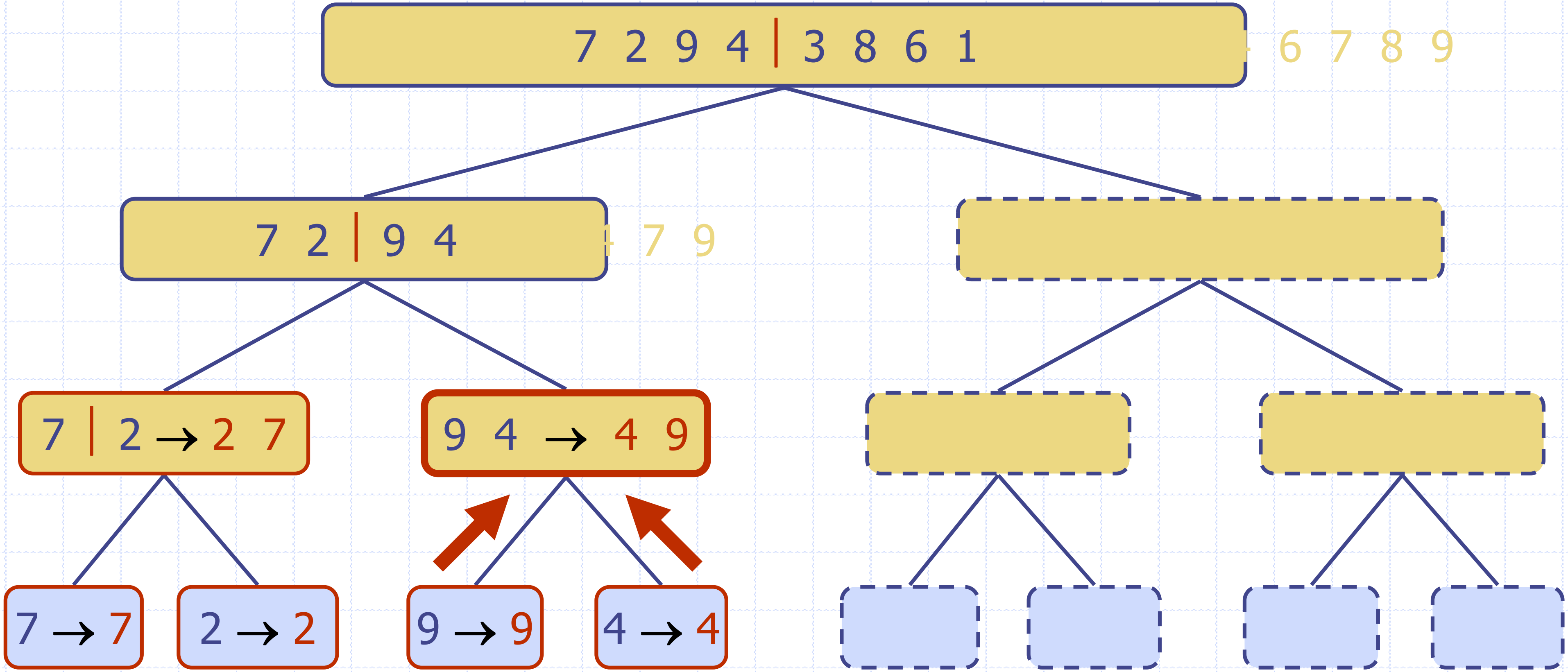
Execution Example (cont.)

- Merge



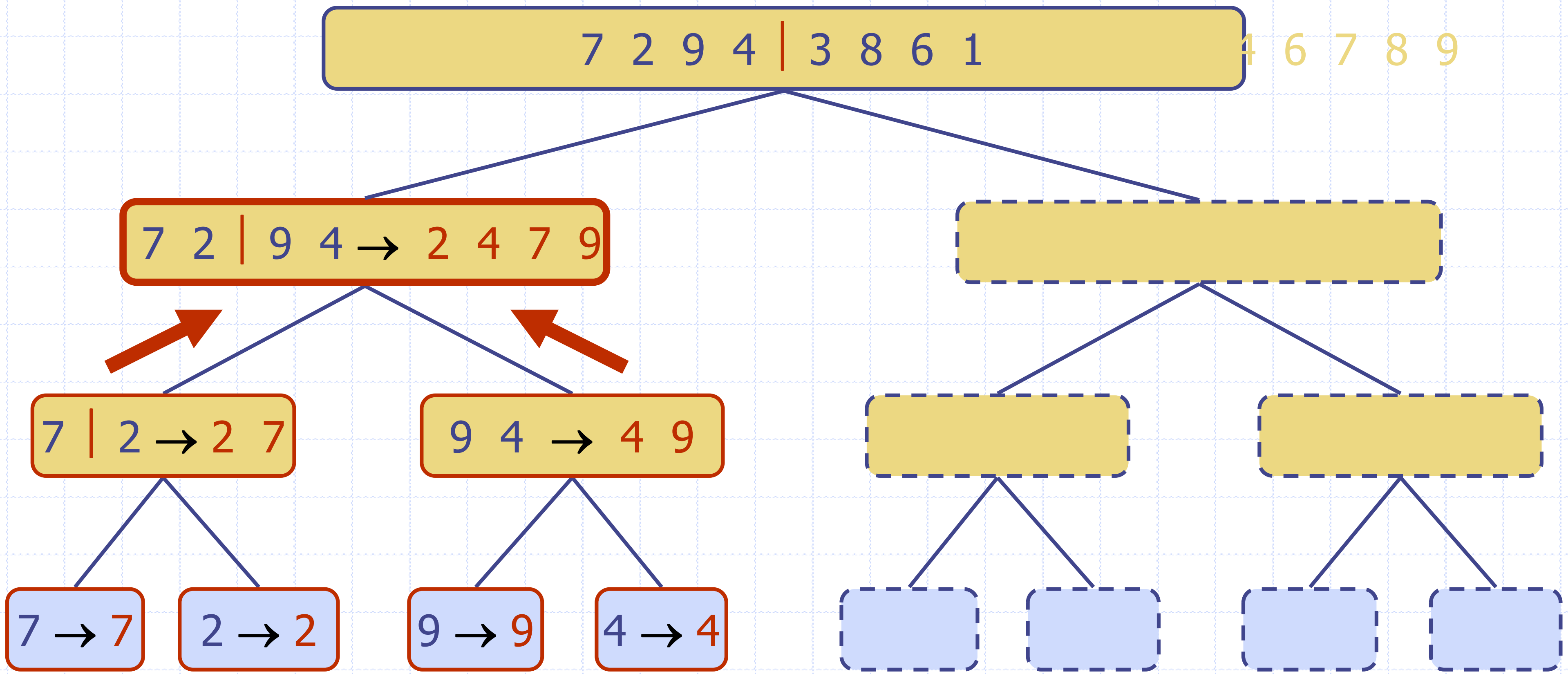
Execution Example (cont.)

- Recursive call, …, base case, merge



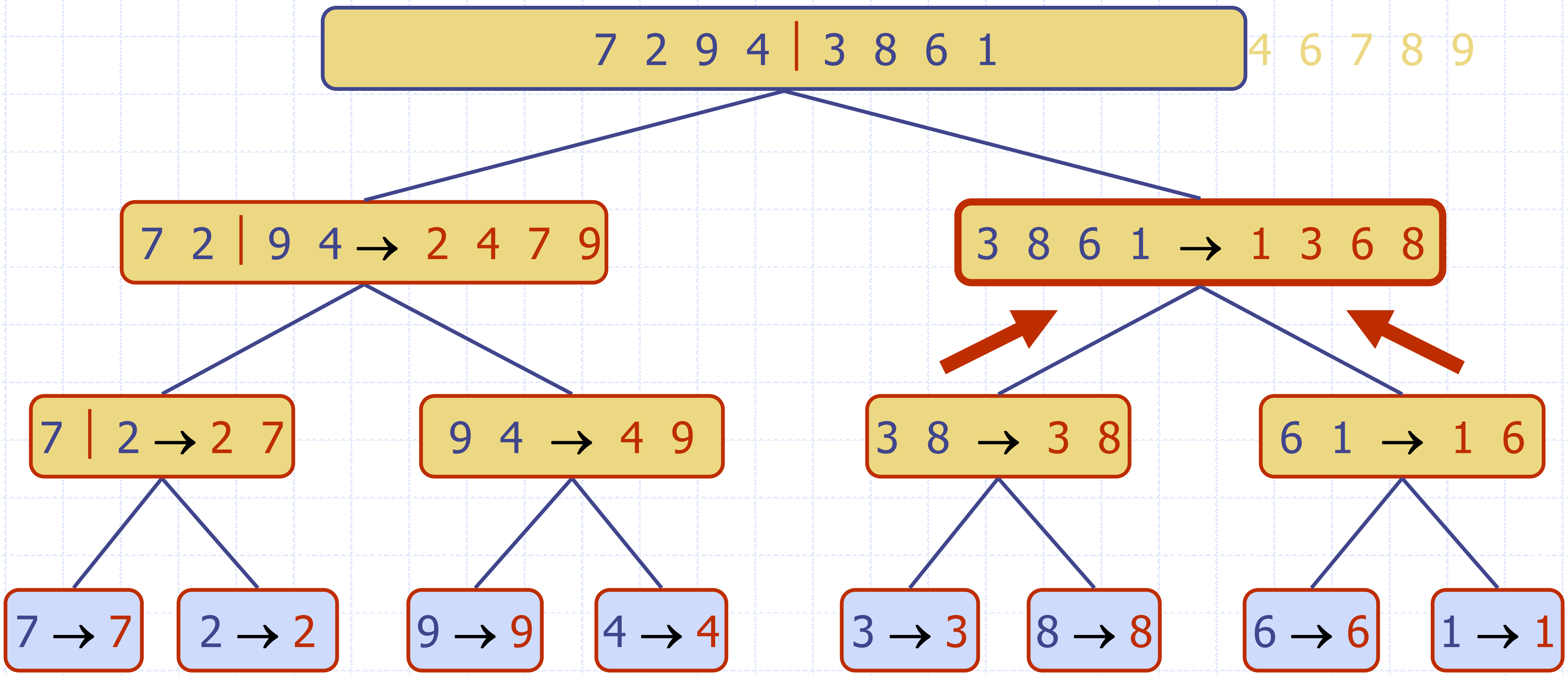
Execution Example (cont.)

- Merge



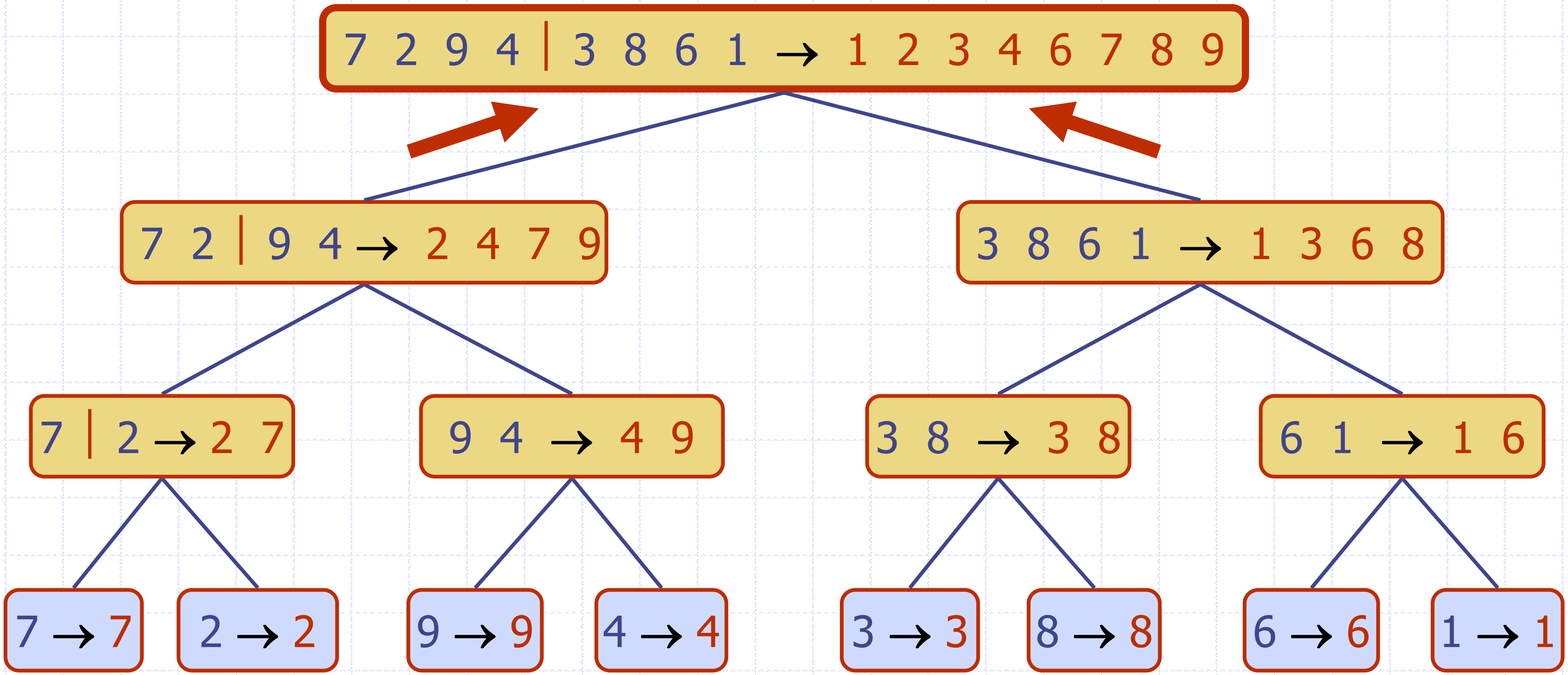
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

- Merge



© 2014 Goodrich, Tamassia, Goldwasser