Merge Sort

Algorithms & Data Structures
ITCS 6114/8114

Dr. Dewan Tanvir Ahmed
Department of Computer Science
University of North Carolina at Charlotte

Outline and Reading

- Divide-and-conquer paradigm (§4.1.1)
- Merge-sort (§4.1.1)
  - Algorithm
  - Merging two sorted sequences
  - Merge-sort tree
  - Execution example
  - Analysis
- Summary of sorting algorithms (§4.2.1)

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
- 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
- **Like heap-sort**
  - It has $O(n \log n)$ running time
- **Unlike heap-sort**
  - It does not use an auxiliary priority queue
  - It accesses data in a sequential manner (suitable to sort data on a disk)

Merge-Sort

```
Algorithm mergeSort(S, C)
   Input sequence S with n elements, comparator C
   Output sequence S sorted according to C
   if S.size() > 1
      (S1, S2) ← partition(S,n/2)
      mergeSort(S1, C)
      mergeSort(S2, C)
      S ← merge(S1, S2)
```

Merging Two Sorted Sequences

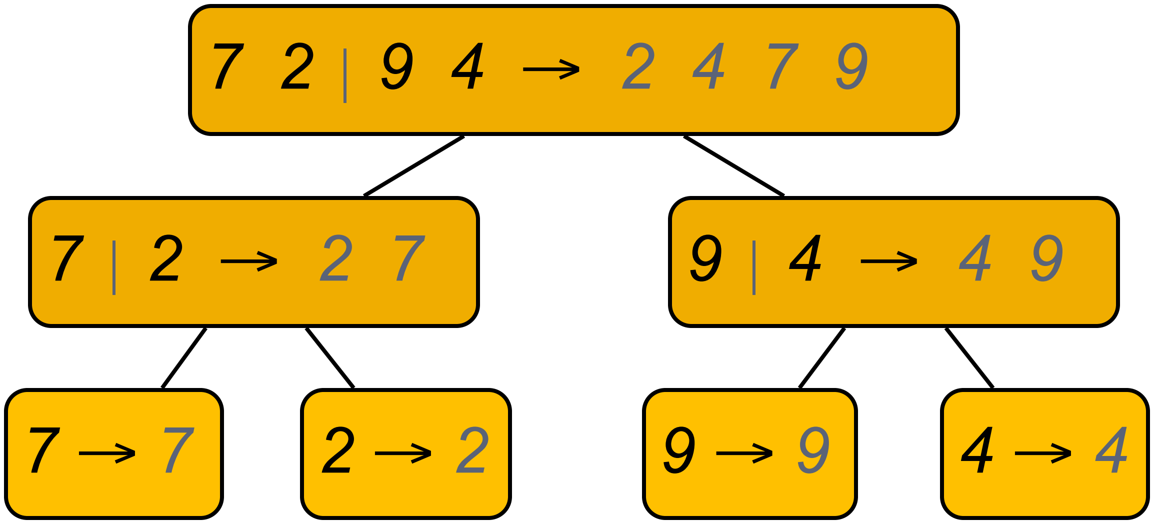
```
Algorithm merge(A, B)
   Input sequences A and B with n/2 elements each
   Output sorted sequence of A ∪ B

   S ← empty sequence
   while ¬A.isEmpty()  ∧ ¬B.isEmpty()
      if A.first().element() < B.first().element()
         S.insertLast(A.remove(A.first()))
      else
         S.insertLast(B.remove(B.first()))

   while ¬A.isEmpty()
      S.insertLast(A.remove(A.first()))

   while ¬B.isEmpty()
      S.insertLast(B.remove(B.first()))

   return S
```

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

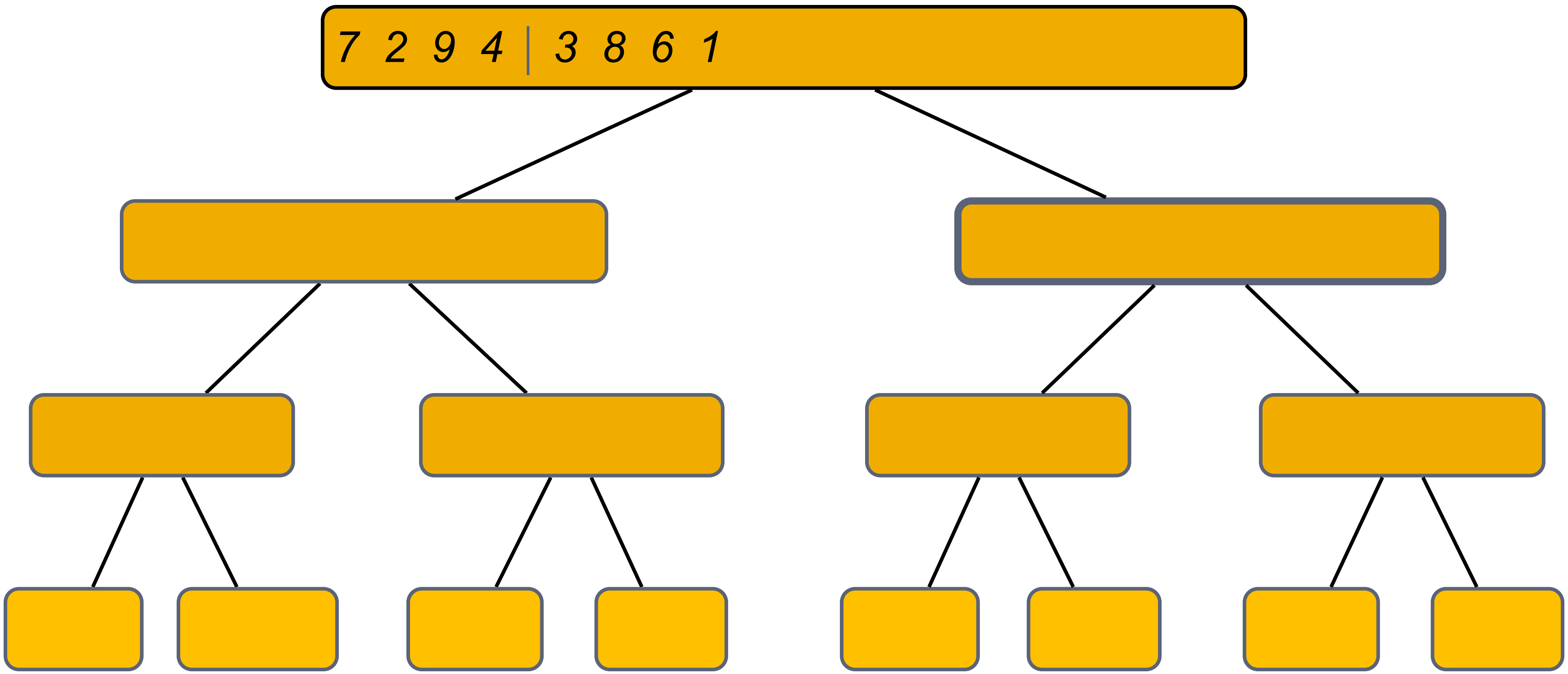
Merge-Sort Tree

- 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



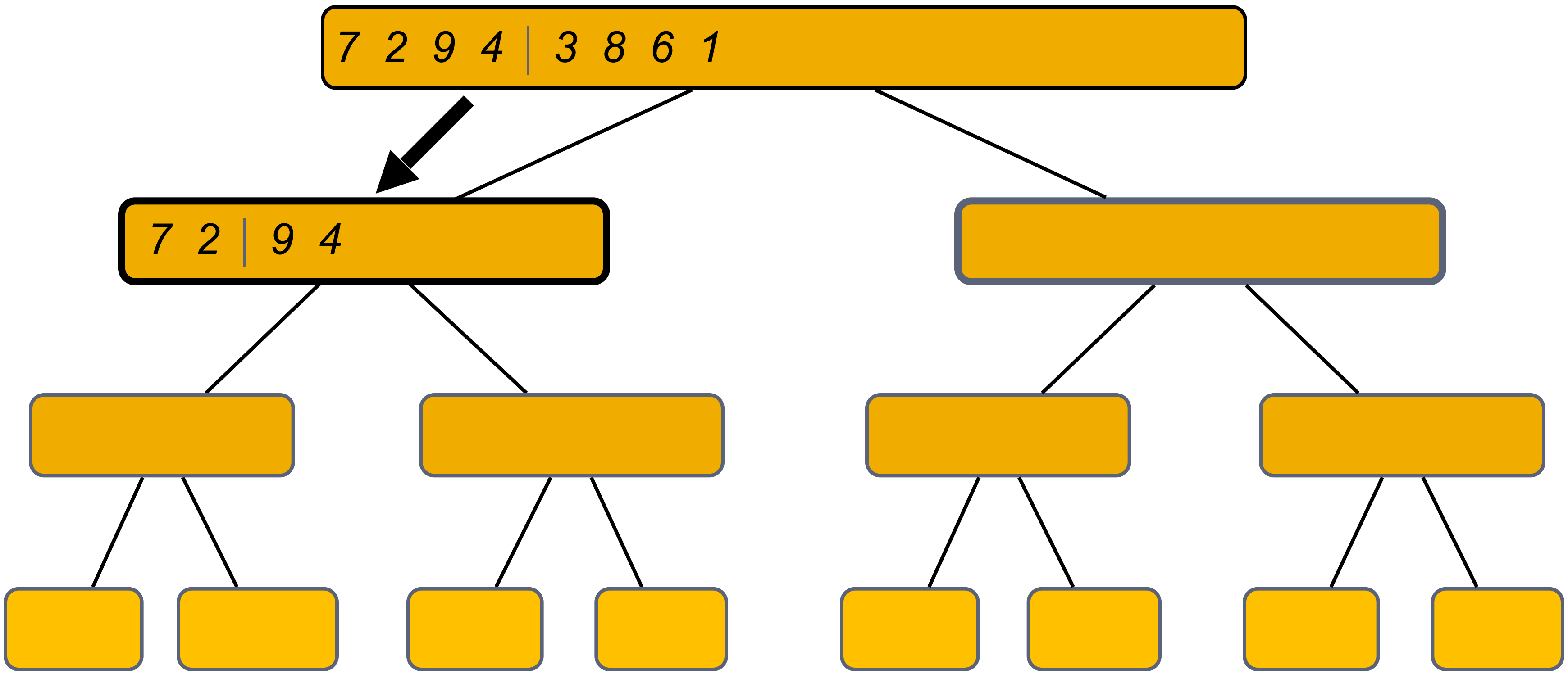
Execution Example

- Partition



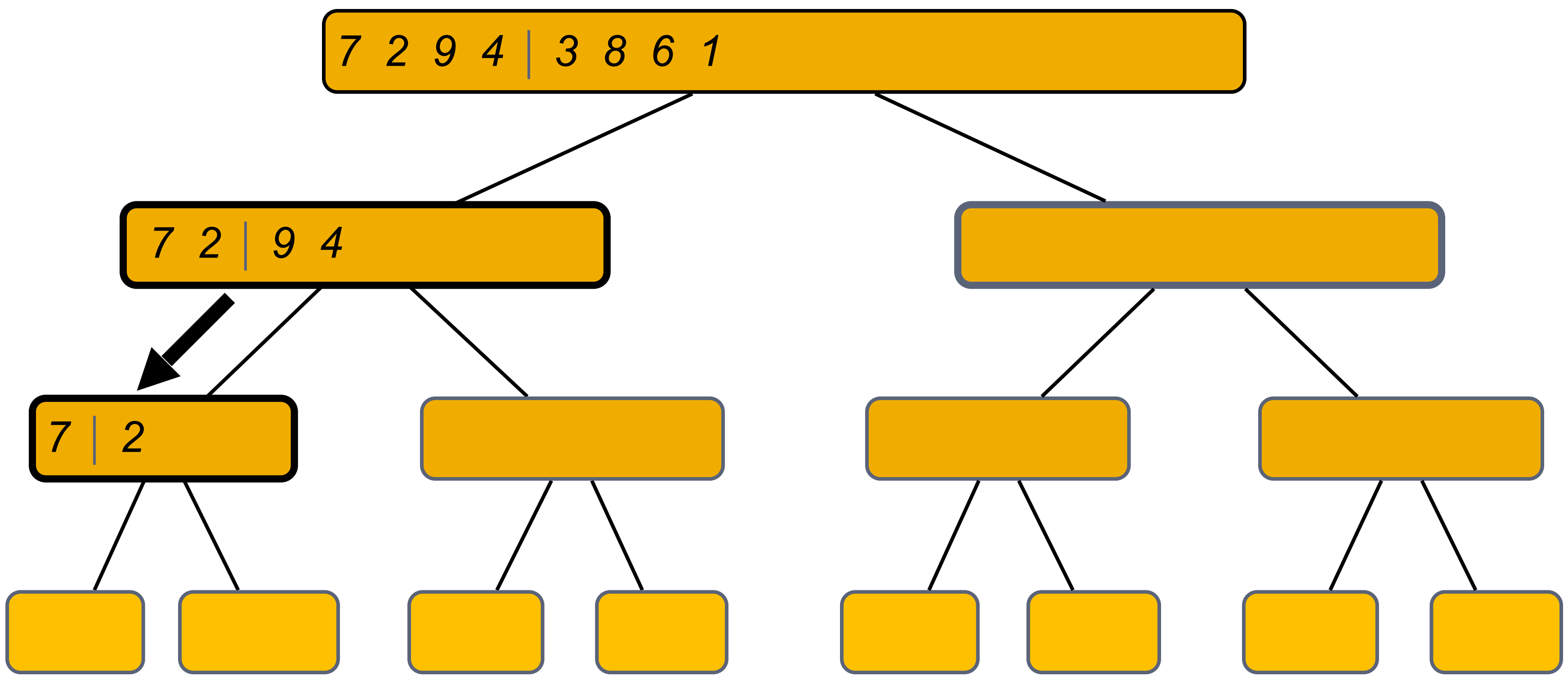
Execution Example (cont.)

- Recursive call, partition



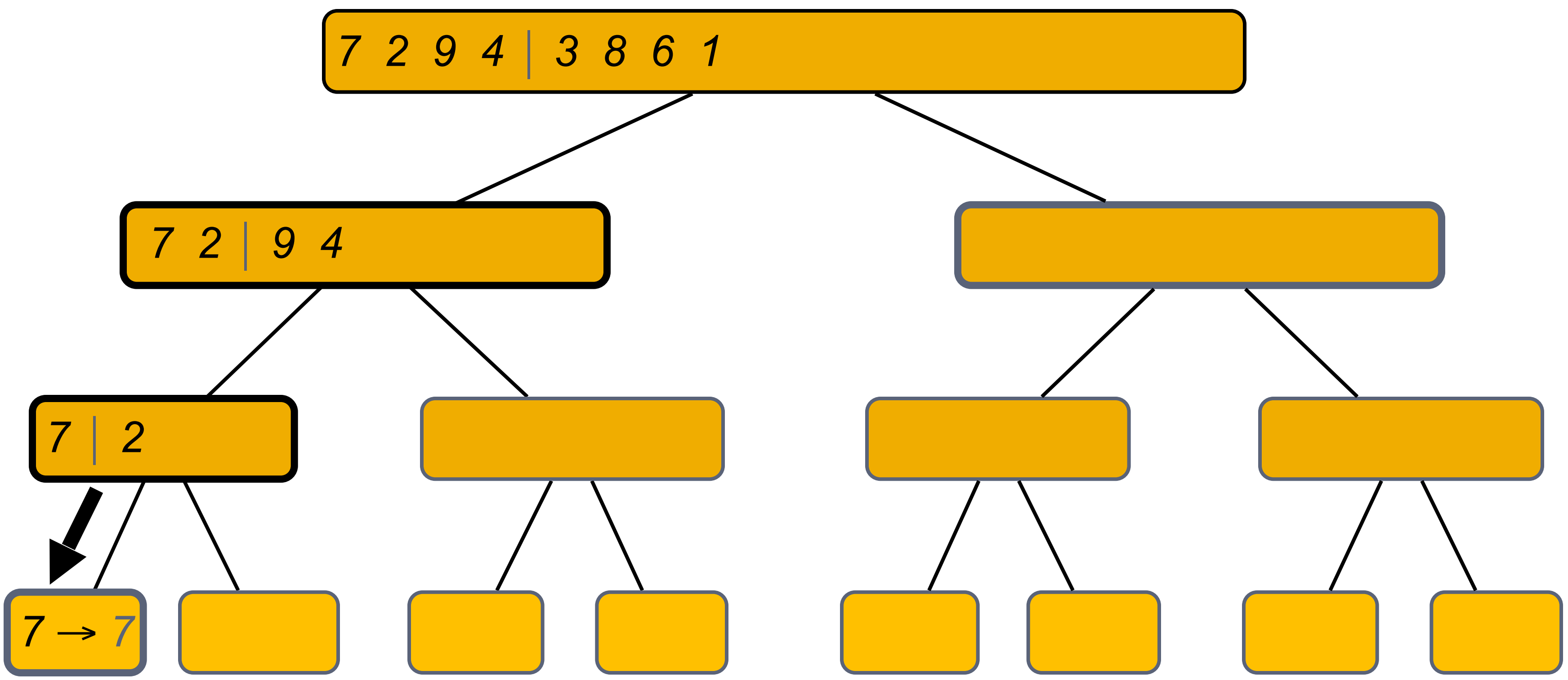
Execution Example (cont.)

- Recursive call, partition



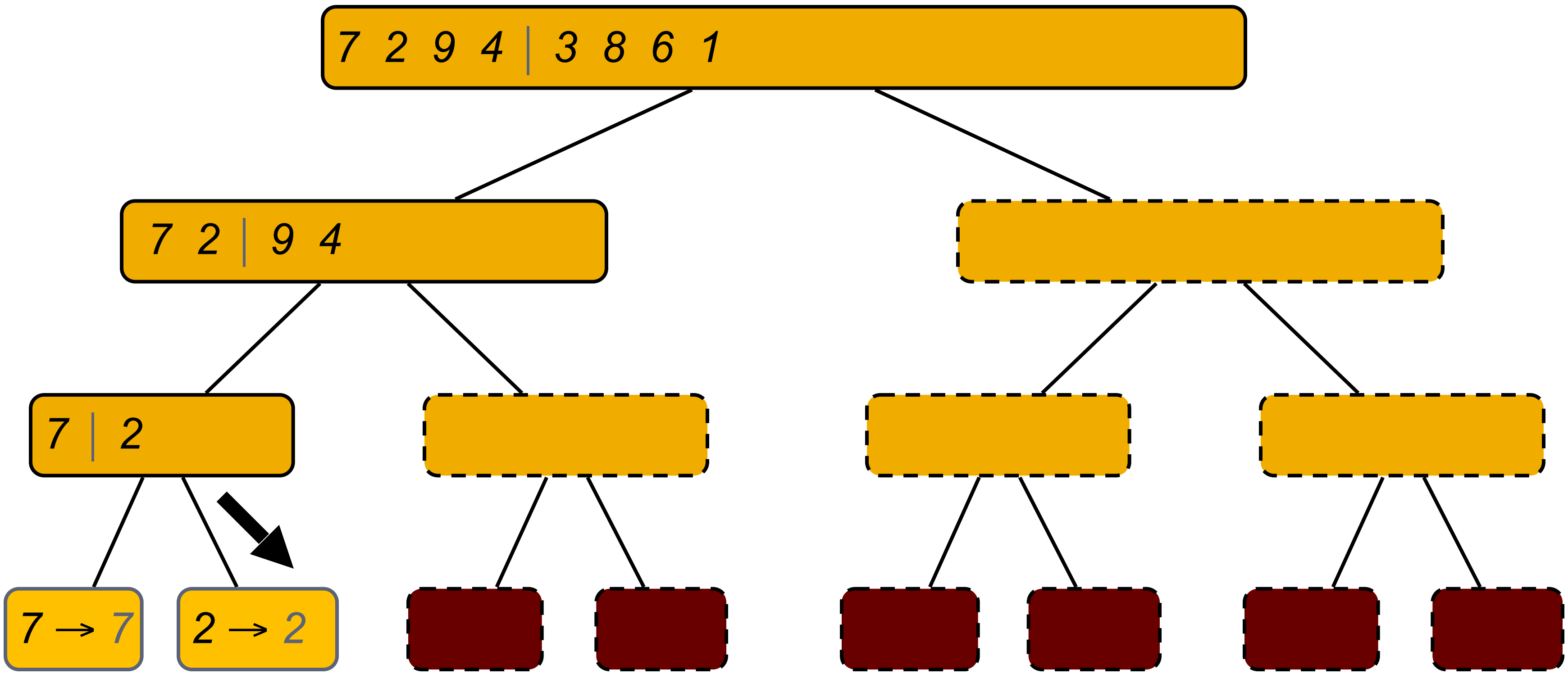
Execution Example (cont.)

- Recursive call, base case



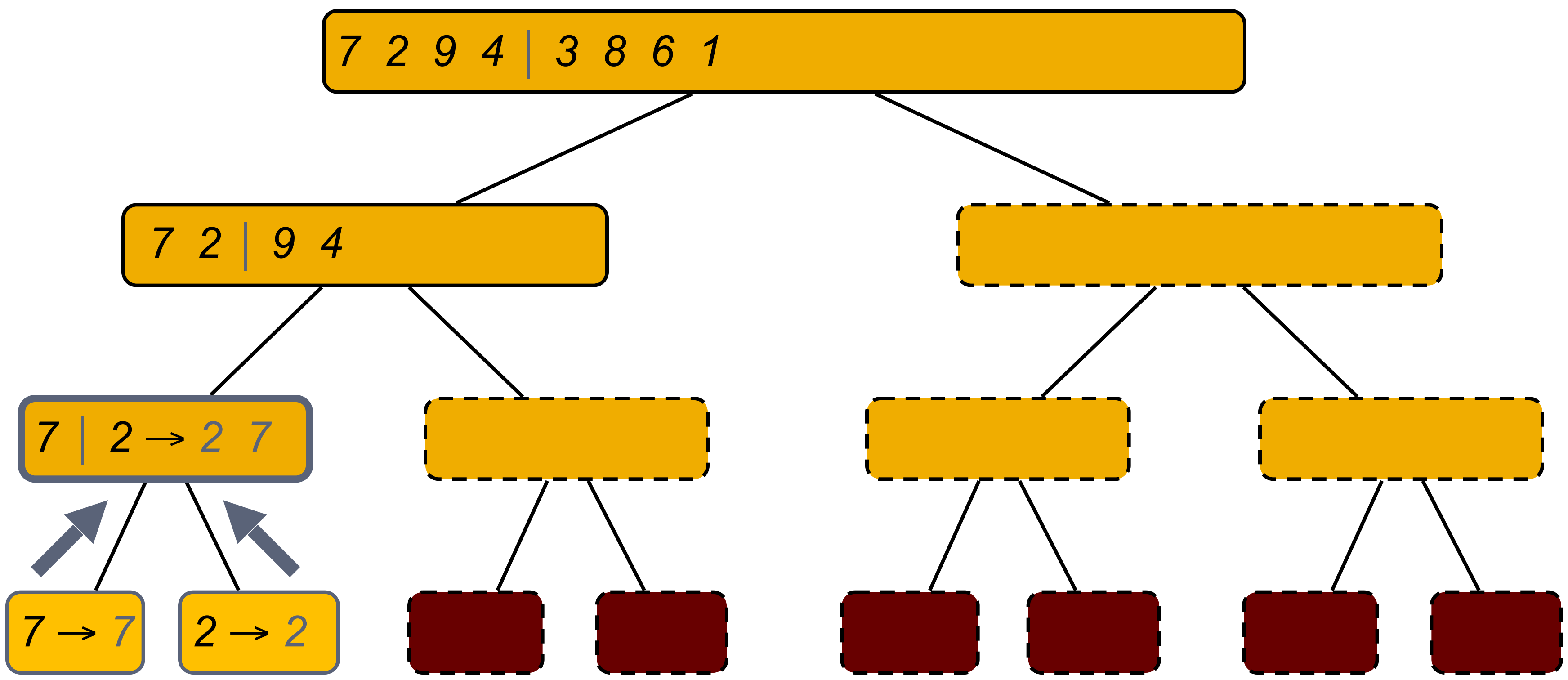
Execution Example (cont.)

- Recursive call, base case



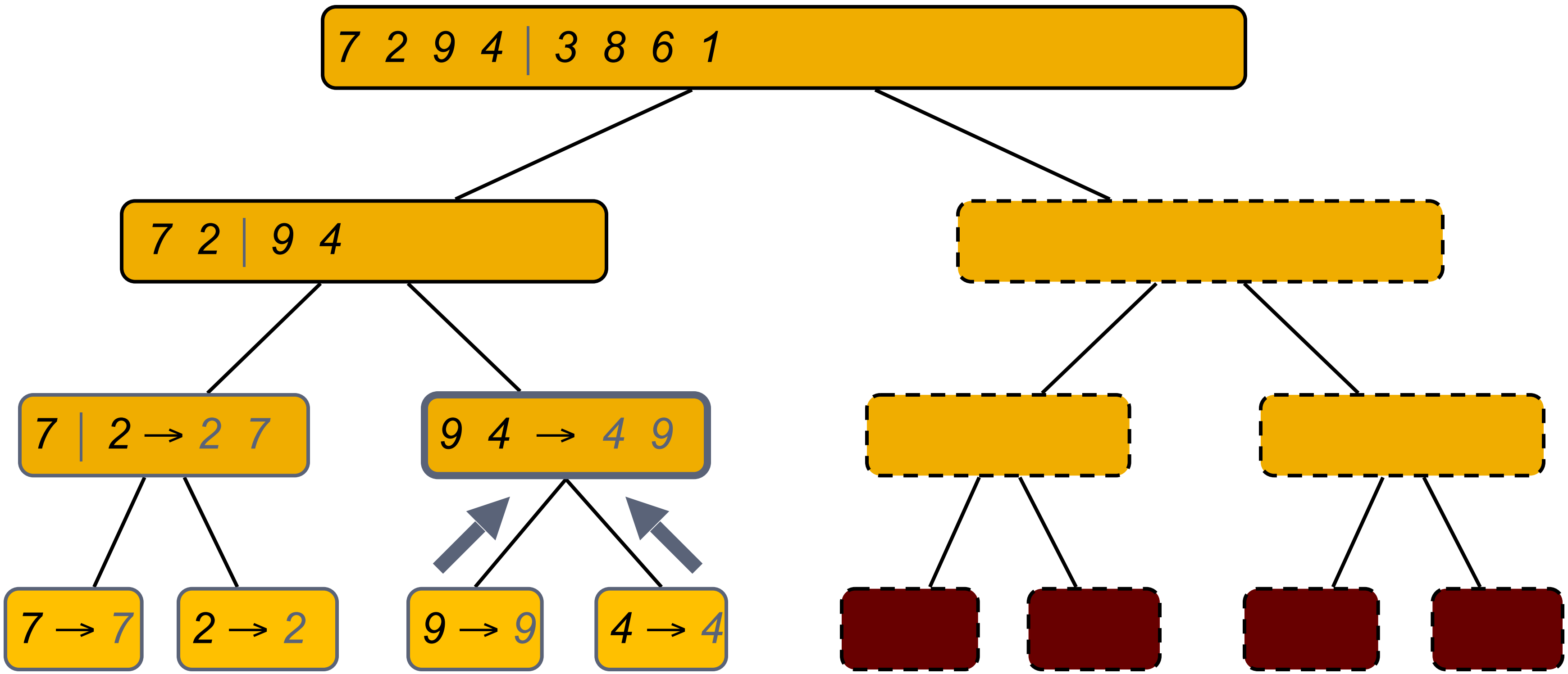
Execution Example (cont.)

- Merge



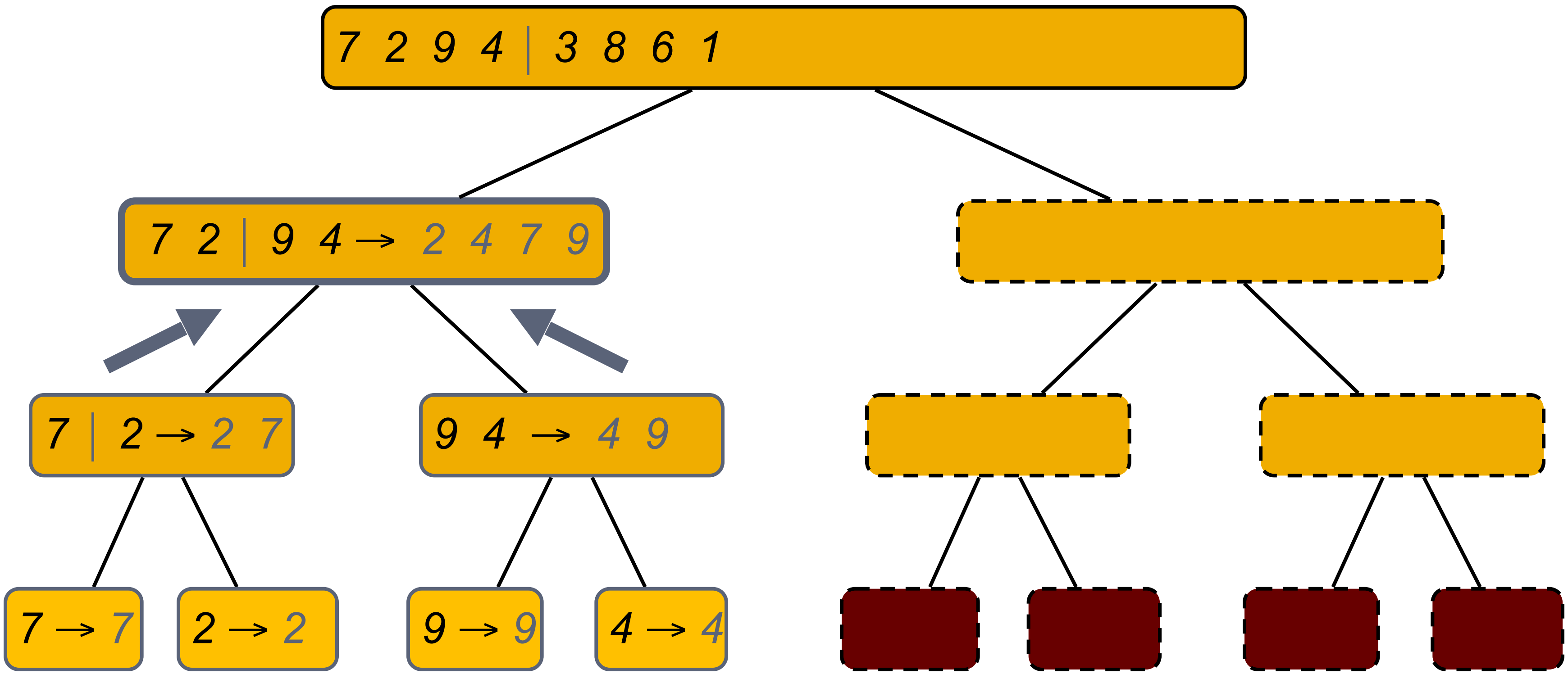
Execution Example (cont.)

- Recursive call, …, base case, merge



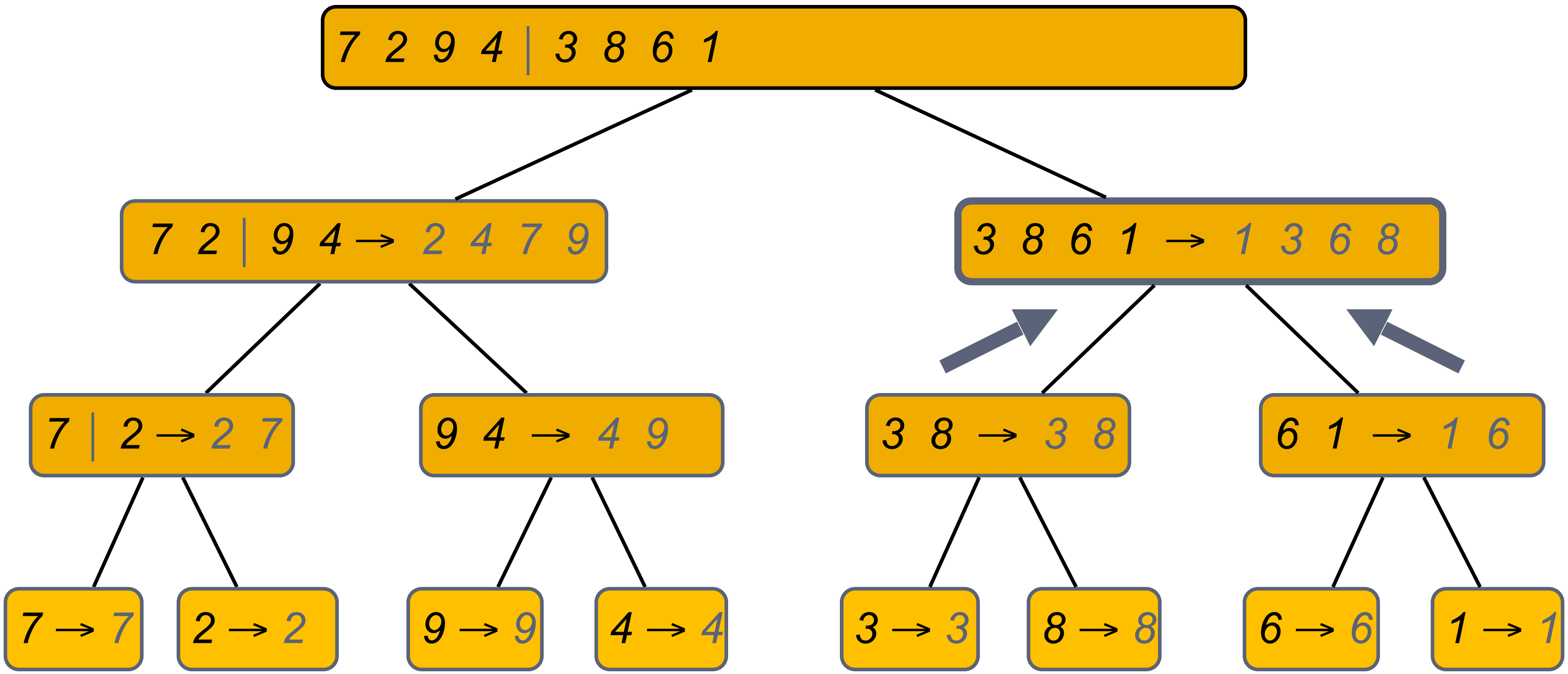
Execution Example (cont.)

- Merge



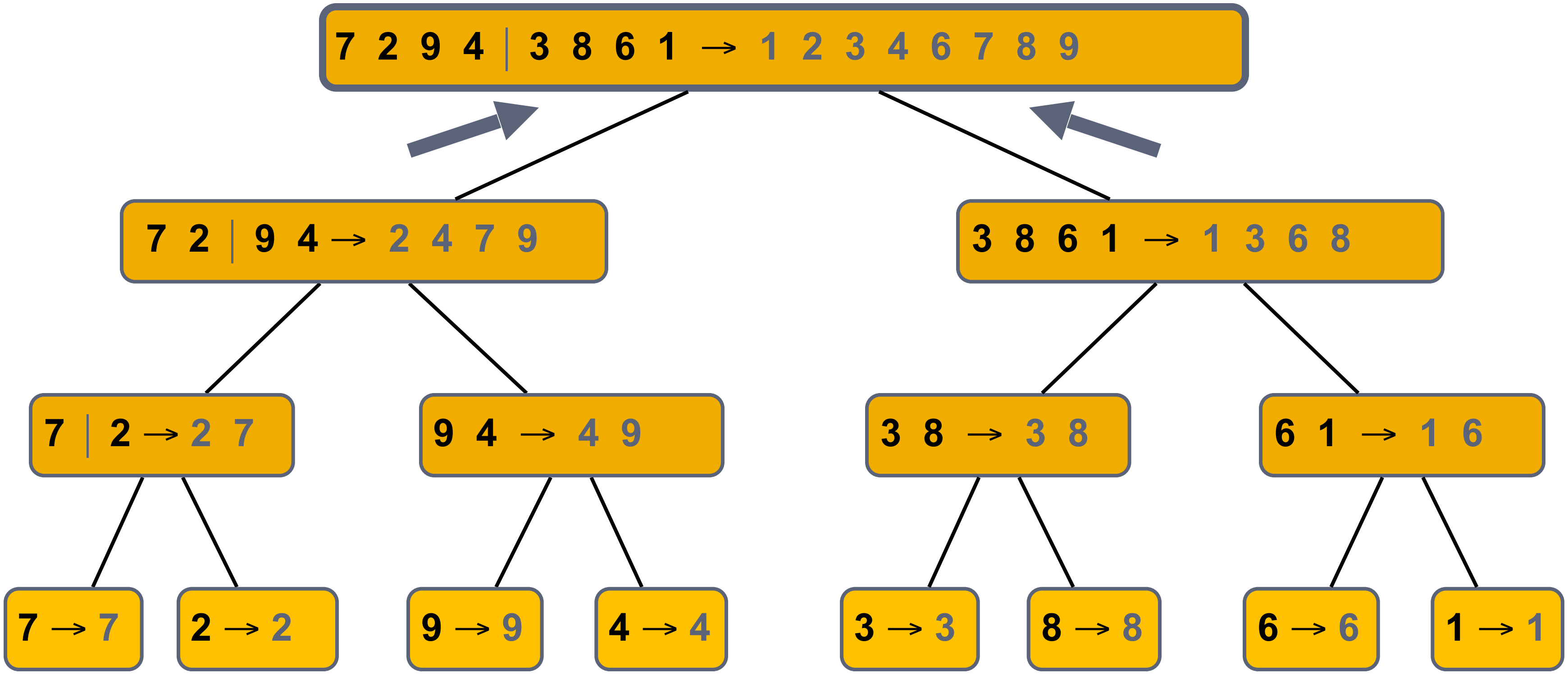
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

- Merge



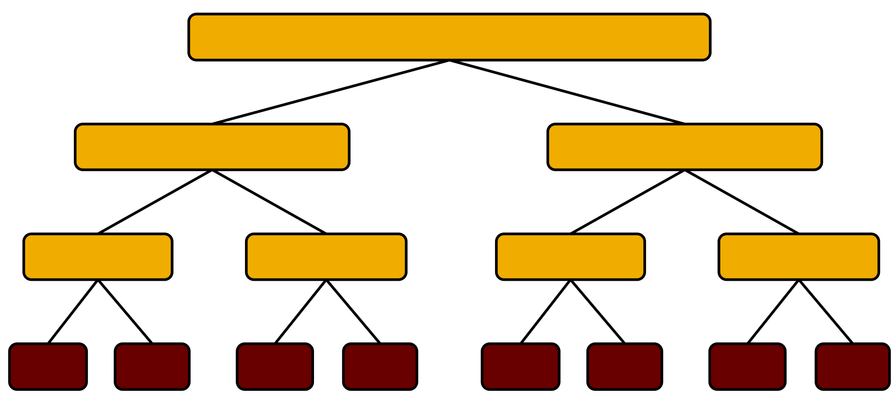
Analysis of Merge-Sort

- 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

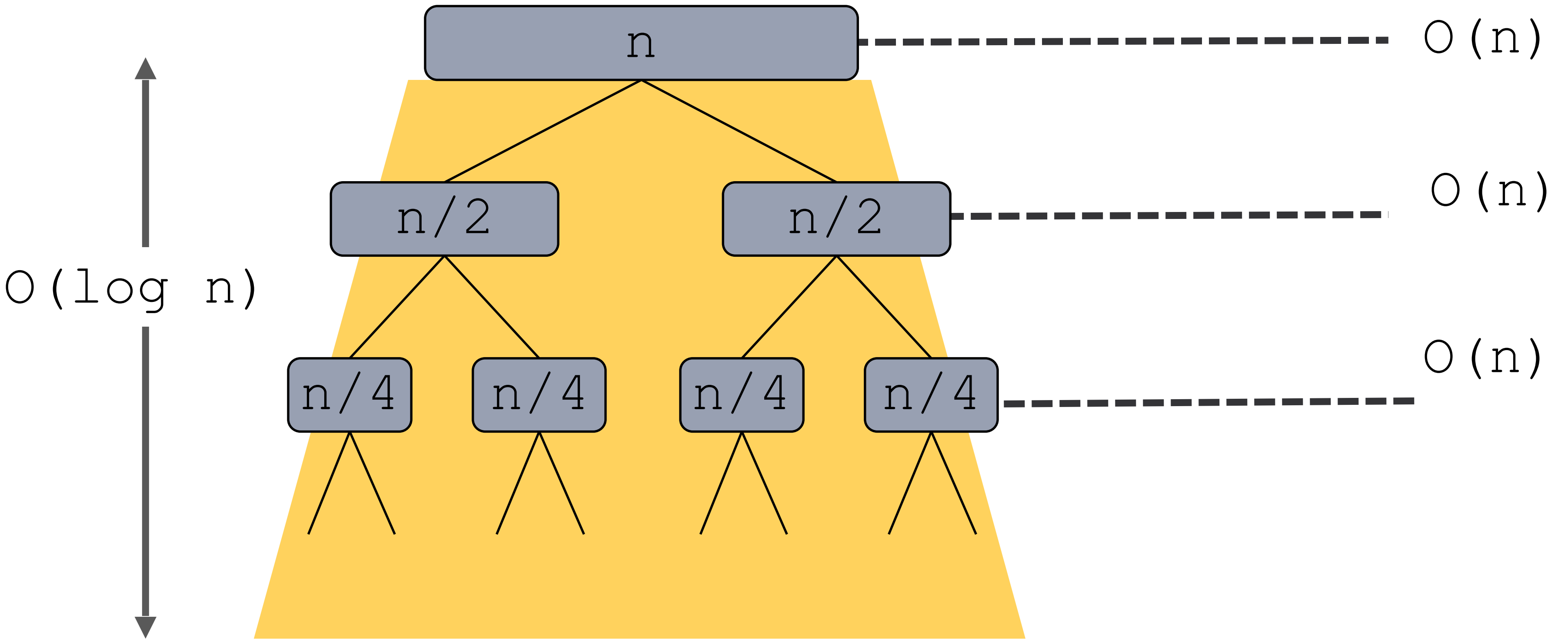
Thus, the total running time of merge-sort is $O(n \log n)$

T has exactly 2^i nodes at each depth i. This implies that the overall time spent at all the nodes at depth i is O(2^i. n/ 2^i), which is O(n)

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



Analysis of Merge-Sort



Merge-sort and Recurrence Equation

- Let, $T(n)$: the worst-case running time of input size n.
- Since merge-sort is recursive, we can characterize the function $T(n)$ by recursive equation
- $T(n) = \begin{cases} b & \textit{if } n = 1 \\ T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn & \textit{otherwise} \end{cases}$

  Where $b > 0$ and $c > 0$

Find its **closed-form** characterization (does not involve T(n) itself).

We restrict n is a power of 2 .

Merge-sort and Recurrence Equation

- $$T(n) = \begin{cases} b & if\ n = 1 \\ T(n/2) + T(n/2) + cn & otherwise \end{cases}$$

$$T(n) = \begin{cases} b & if\ n = 1 \\ 2T(n/2) + cn & otherwise \end{cases}$$

$$T(n) = \begin{cases} b & if\ n = 1 \\ 2(2T(n/2^2) + cn/2) + cn & otherwise \end{cases}$$

$$T(n) = \begin{cases} b & if\ n = 1 \\ 2^2T(n/2^2) + 2cn & otherwise \end{cases}$$

$$T(n) = \begin{cases} b & if\ n = 1 \\ 2^3T(n/2^3) + 3cn & otherwise \end{cases}$$

Merge-sort and Recurrence Equation

- After applying this eq. i times

$$T(n) = \begin{cases} b & \text{if } n = 1 \\ 2^i T(n/2^i) + icn & \text{otherwise} \end{cases}$$

To stop, T(n) = b when n =1

$$\begin{aligned} T(n) &= 2^{\log n} T(n/2^{\log n}) + (\log n)cn \\ &= nT\left(\frac{n}{n}\right) + cn\log n \\ &= nT(1) + cn\log n \\ &= nb + cn\log n \end{aligned}$$

T(n) is O($n\log n$)

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$\boldsymbol{O}(\boldsymbol{n}^2)$	◆ slow ◆ in-place ◆ for small data sets (< 1K)
insertion-sort	$\boldsymbol{O}(\boldsymbol{n}^2)$	◆ slow ◆ in-place ◆ for small data sets (< 1K)
heap-sort	$\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$	◆ fast ◆ in-place ◆ for large data sets (1K — 1M)
merge-sort	$\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$	◆ fast ◆ sequential data access ◆ for huge data sets (> 1M)

Reference

- Algorithm Design: Foundations, Analysis, and Internet Examples. Michael T. Goodrich and Roberto Tamassia. John Wiley & Sons.
- Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.