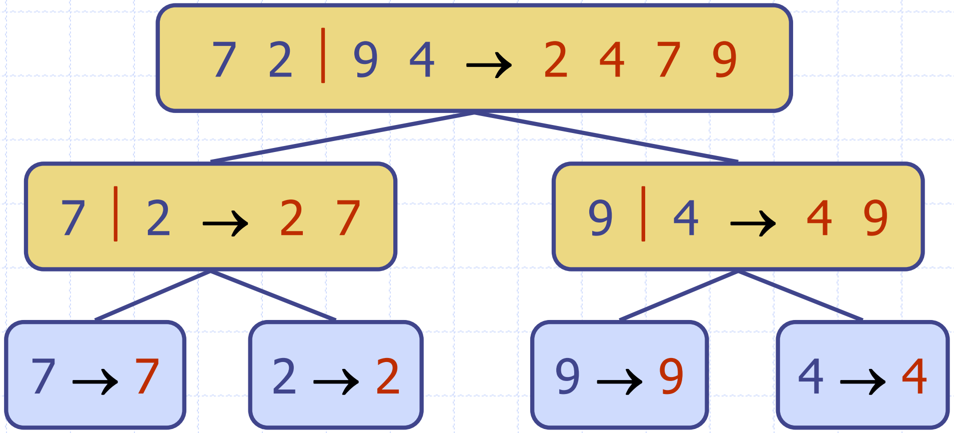
Merge Sort



Divide-and-Conquer (§ 10.1.1)

- Divide-and-conquer (分治演算法) is a general algorithm design paradigm:
  - Divide: divide the input data S in two disjoint subsets S_1 and S_2
  - Conquer: solve the subproblems associated with S_1 and S_2
  - Combine: combine the solutions for S_1 and S_2 into a solution for S

化整為零
各個擊破

- Merge-sort: A sorting algorithm based on divide and conquer
- Like heap-sort
  - It uses a comparator
  - 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/local manner (suitable to sort data on a disk)

Some don't!

Good for external sorting

Merge Sort

- Merge sort
  - A divide-and-conquer algorithm
  - Invented by John von Neumann in 1945

約翰·馮·紐曼（**John von Neumann**，**1903**年**12**月**28**日－**1957**年**2**月**8**日），出生於匈牙利的美國籍猶太人數學家，現代電腦創始人之一。他在電腦科學、經濟、物理學中的量子力學及幾乎所有數學領域都作過重大貢獻，被譽為「電腦之父」。**(**圖及說明摘自維基百科**)**

Merge-Sort (§ 10.1)

- Three steps of merge-sort on an input sequence S with n elements:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Conquer: recursively sort S_1 and S_2
  - Combine: merge S_1 and S_2 into a sorted sequence

Key step!

Algorithm *mergeSort*(S, C)
    Input sequence S with n elements, comparator C
    Output sequence S sorted according to C
    if $S.size() > 1$
        $(S_1, S_2) \leftarrow partition(S, n/2)$
        $mergeSort(S_1, C)$
        $mergeSort(S_2, C)$
        $S \leftarrow merge(S_1, S_2)$

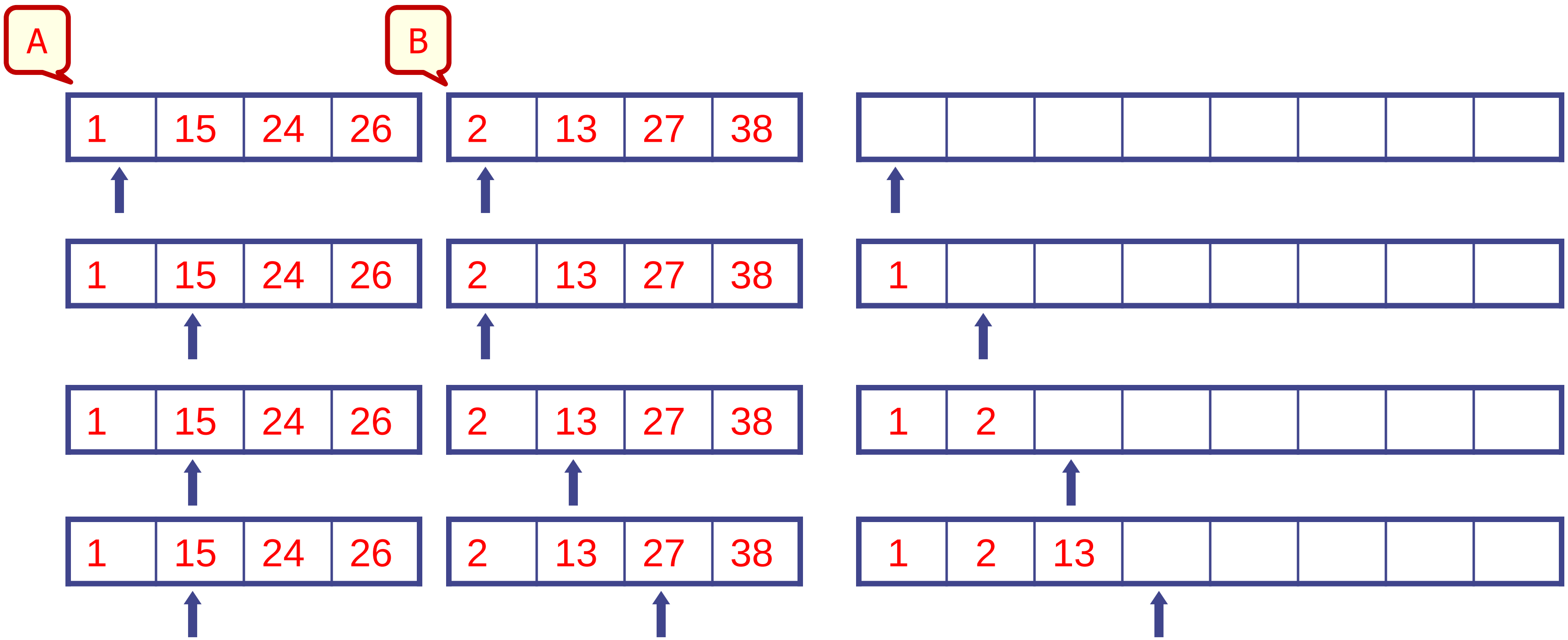
Merging Two Sorted Sequences

- Merging two sorted sequences (implemented as linked lists) with $n/2$ elements each, takes $O(n)$ time.

Algorithm *merge*(*A*, *B*)
  Input sequences A and B with
    $n/2$ elements each
  Output sorted sequence of $A \cup B$

  $S \leftarrow$ empty sequence
  while $\neg A.empty() \wedge \neg B.empty()$
    if $A.front() < B.front()$
      $S.addBack(A.front()); A.eraseFront();$
    else
      $S.addBack(B.front()); B.eraseFront();$
  while $\neg A.empty()$
    $S.addBack(A.front()); A.eraseFront();$
  while $\neg B.empty()$
    $S.addBack(B.front()); B.eraseFront();$
  return S

To Merge 2 Sorted Sequences

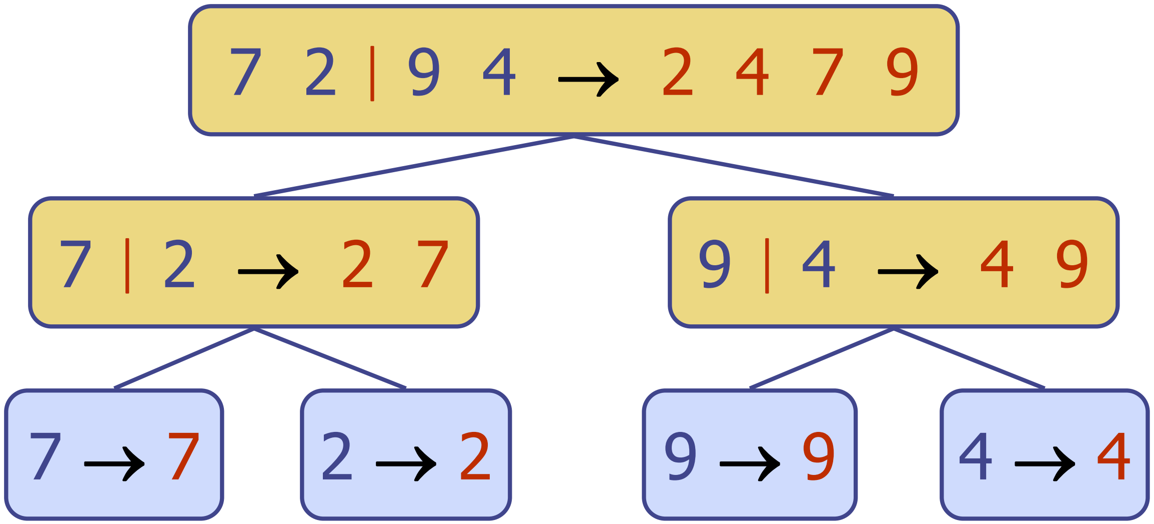


- Properties
  - Need extra space to store the sorted results → Not an in-place sort
  - Total time = $O(|A|+|B|) = O(m+n)$

Also good for singly linked lists

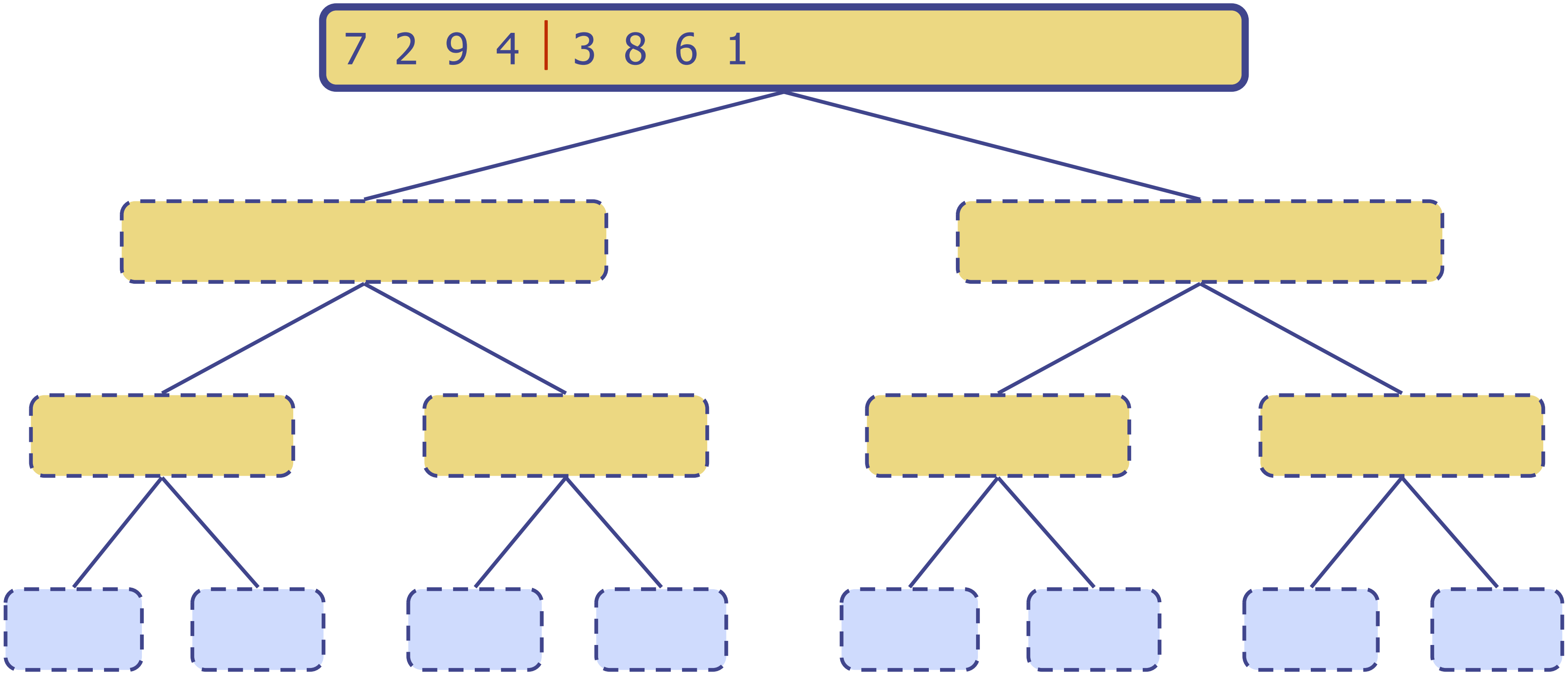
Merge-Sort Tree

- 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



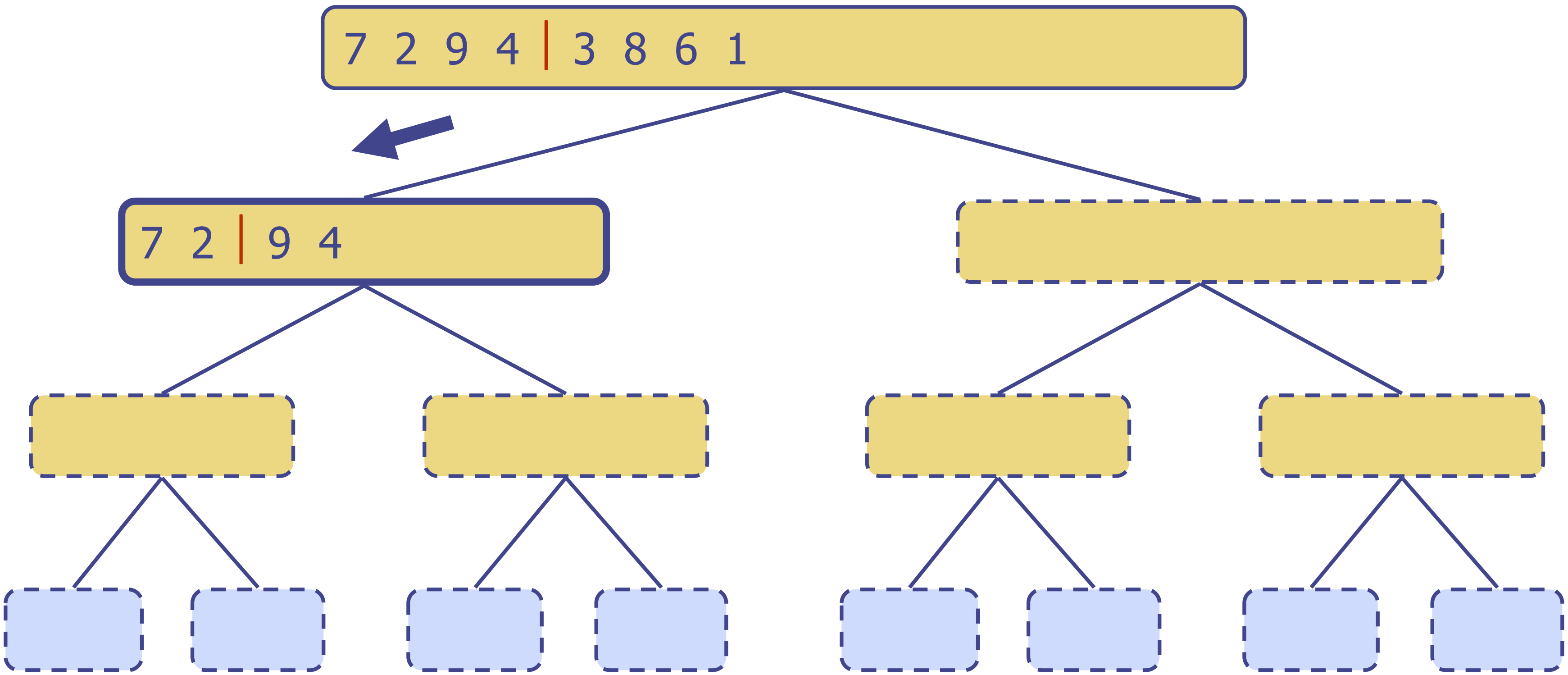
Execution Example

- Partition



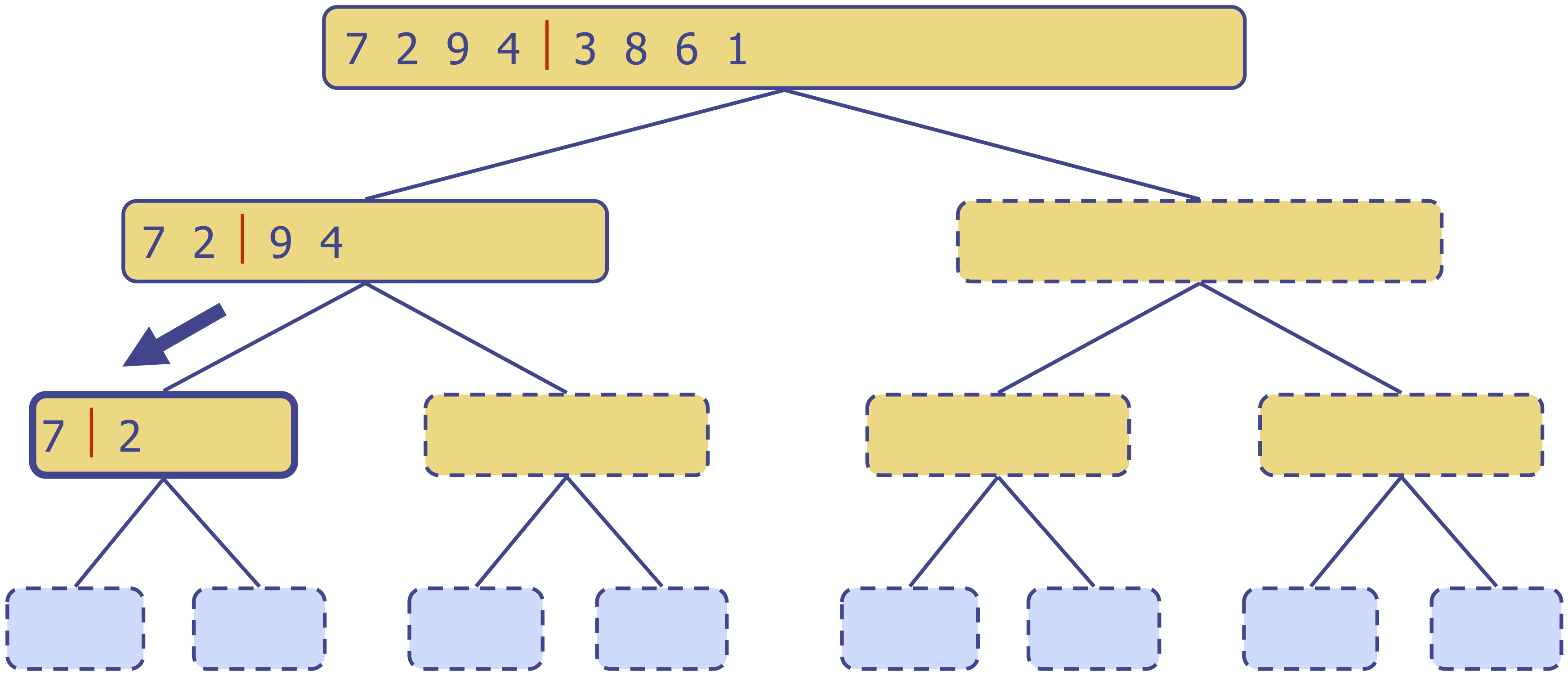
Execution Example (cont.)

- Recursive call, partition



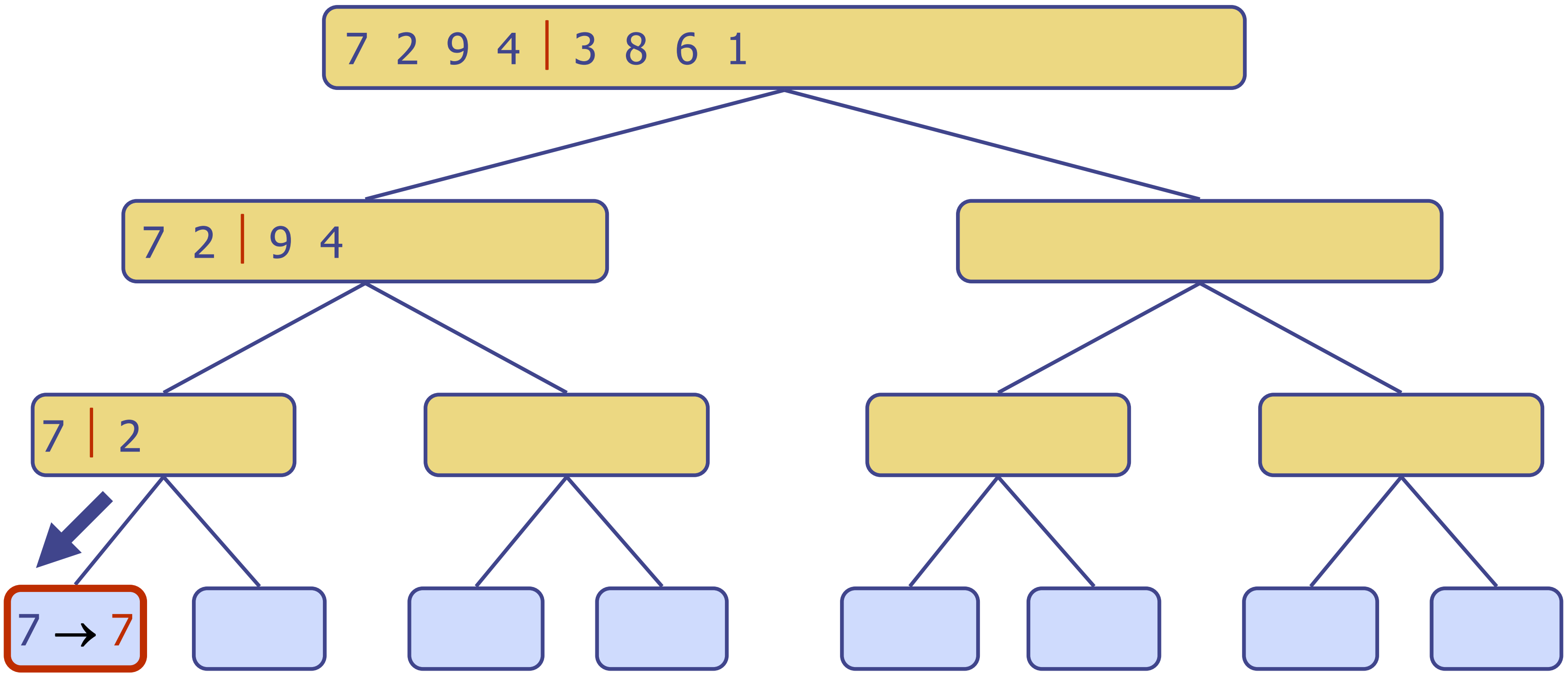
Execution Example (cont.)

- Recursive call, partition



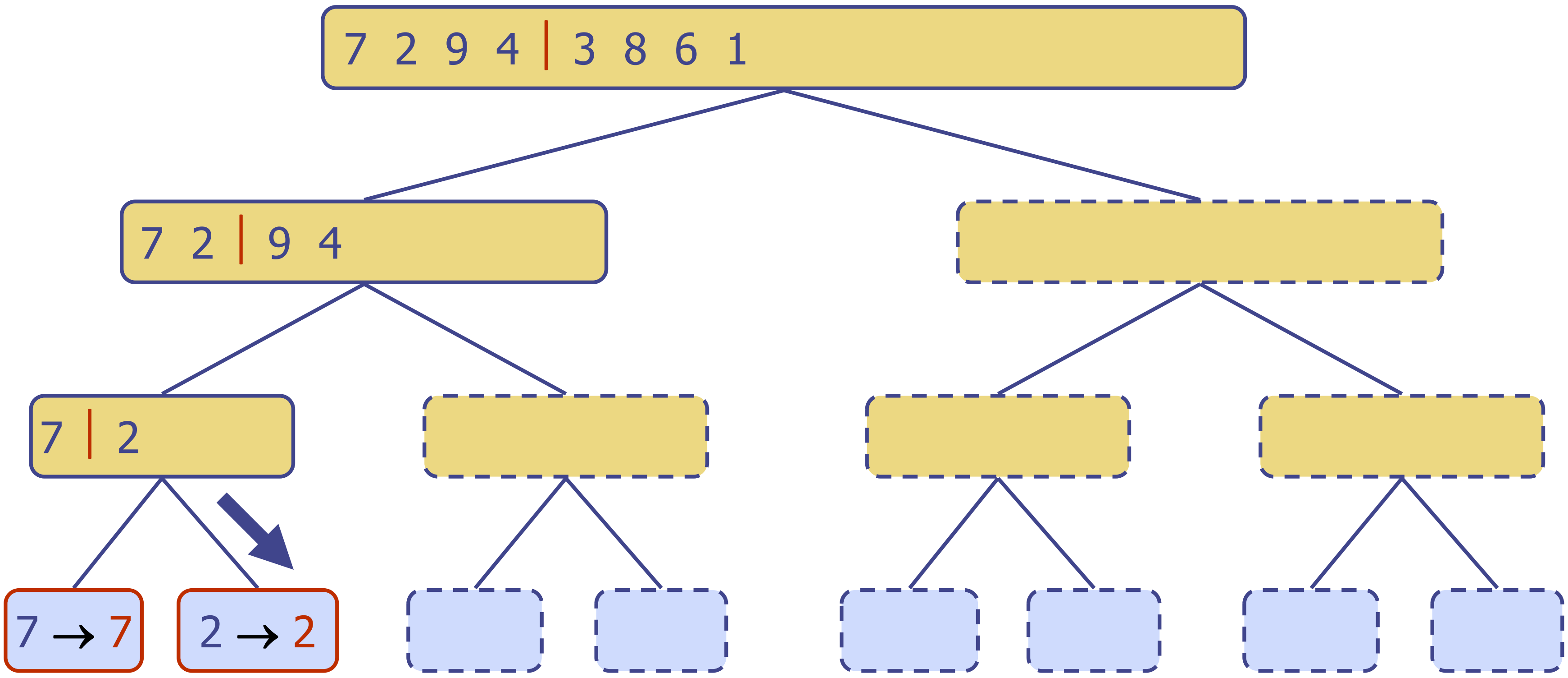
Execution Example (cont.)

- Recursive call, base case



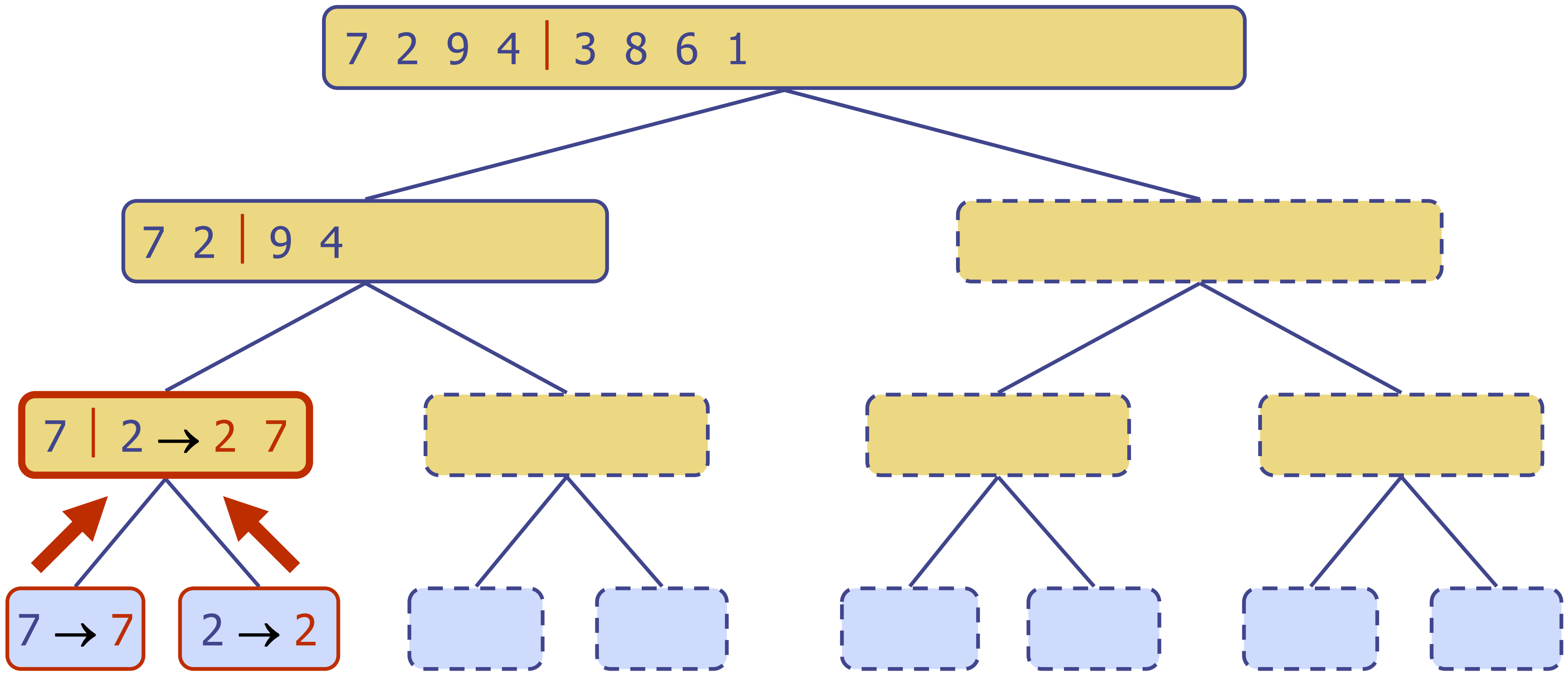
Execution Example (cont.)

- Recursive call, base case



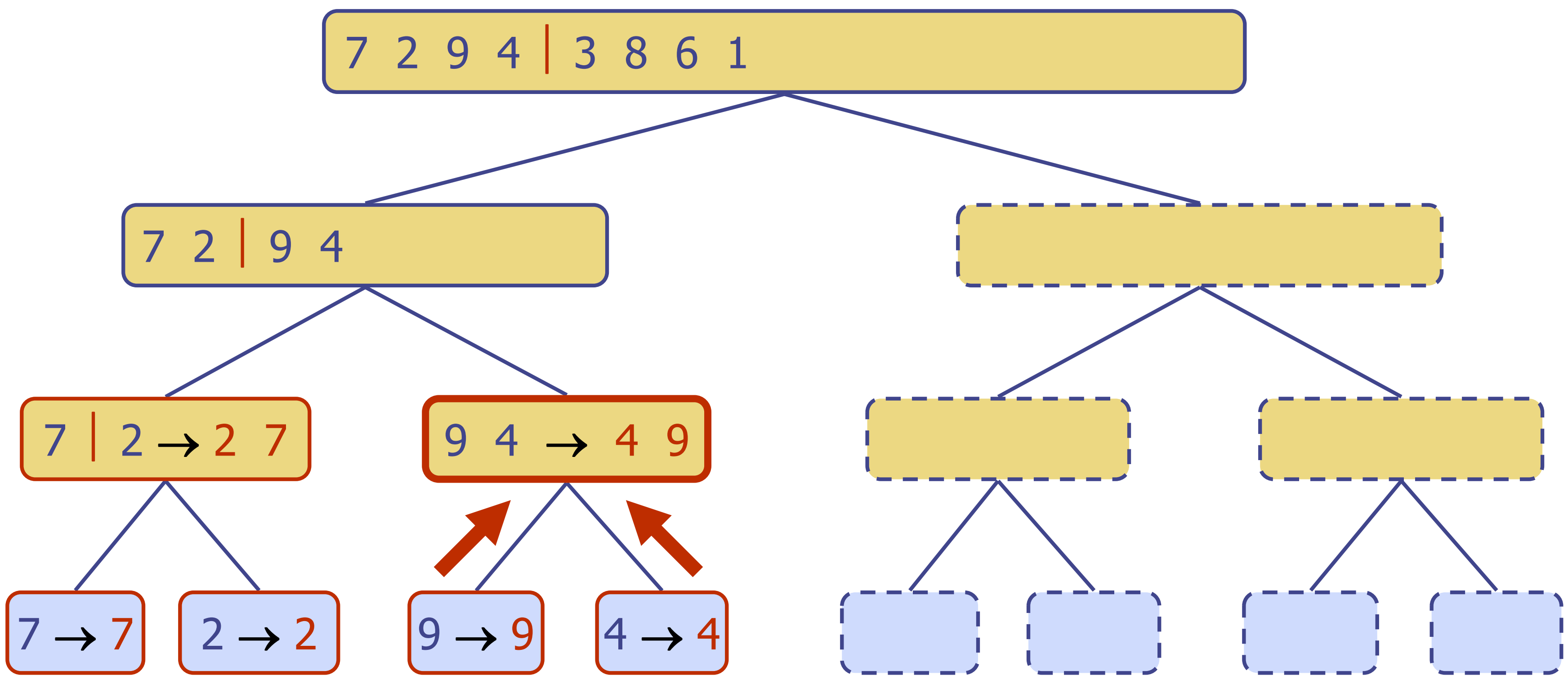
Execution Example (cont.)

- Merge



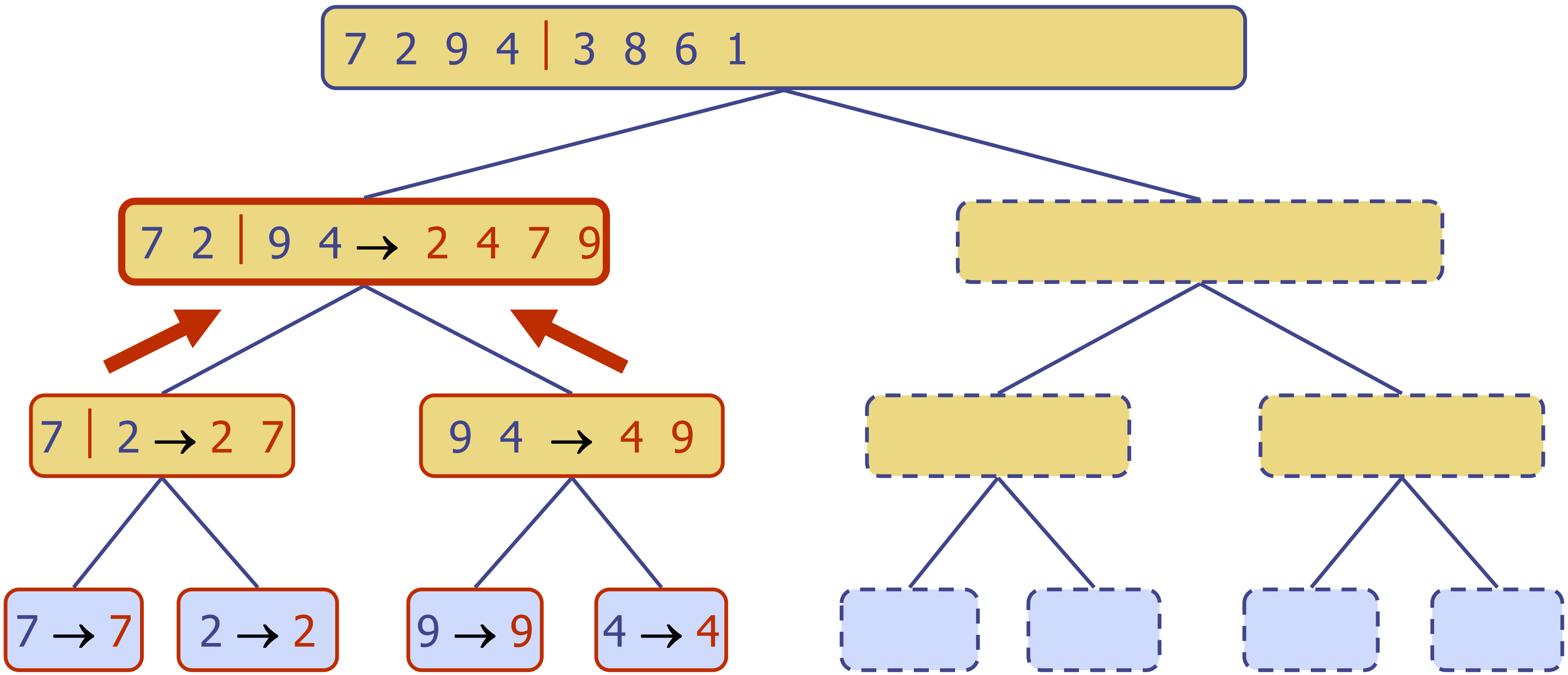
Execution Example (cont.)

- Recursive call, …, base case, merge



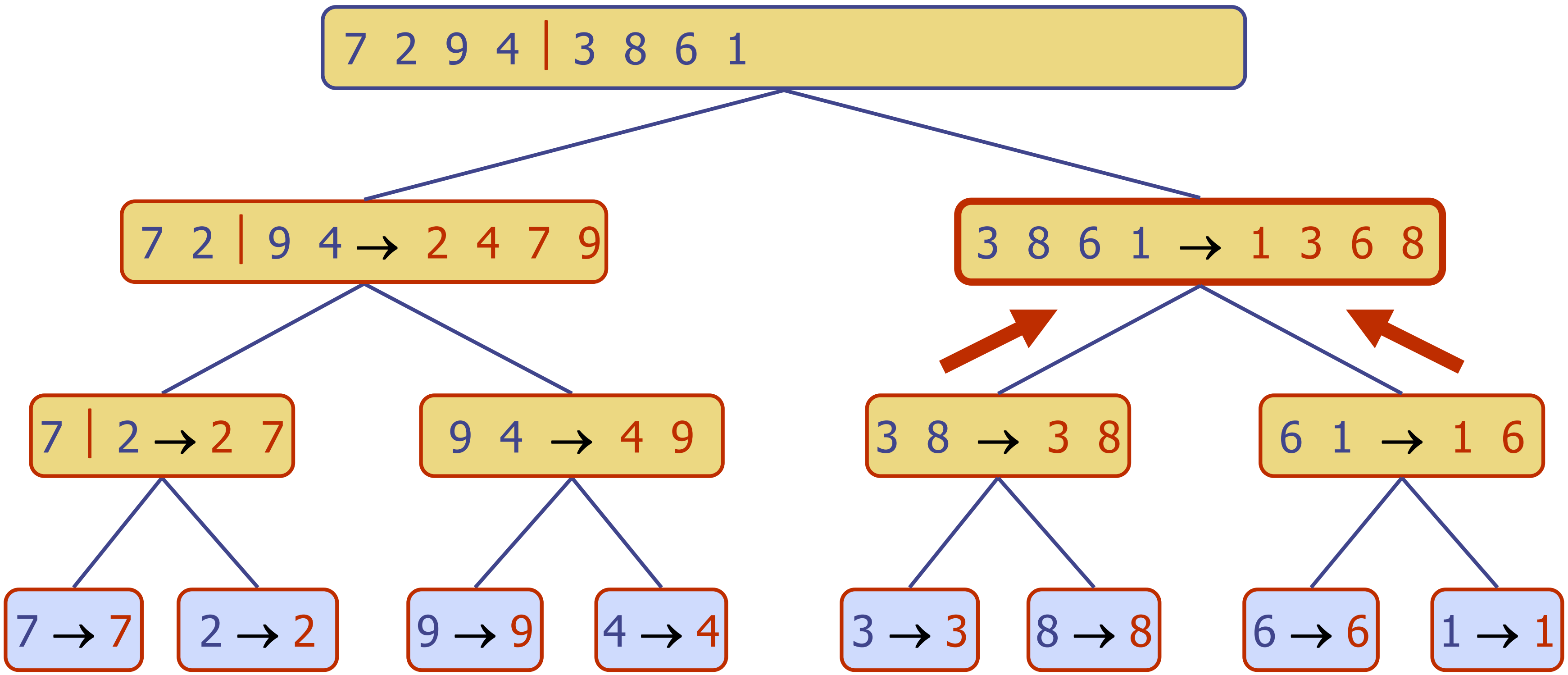
Execution Example (cont.)

- Merge



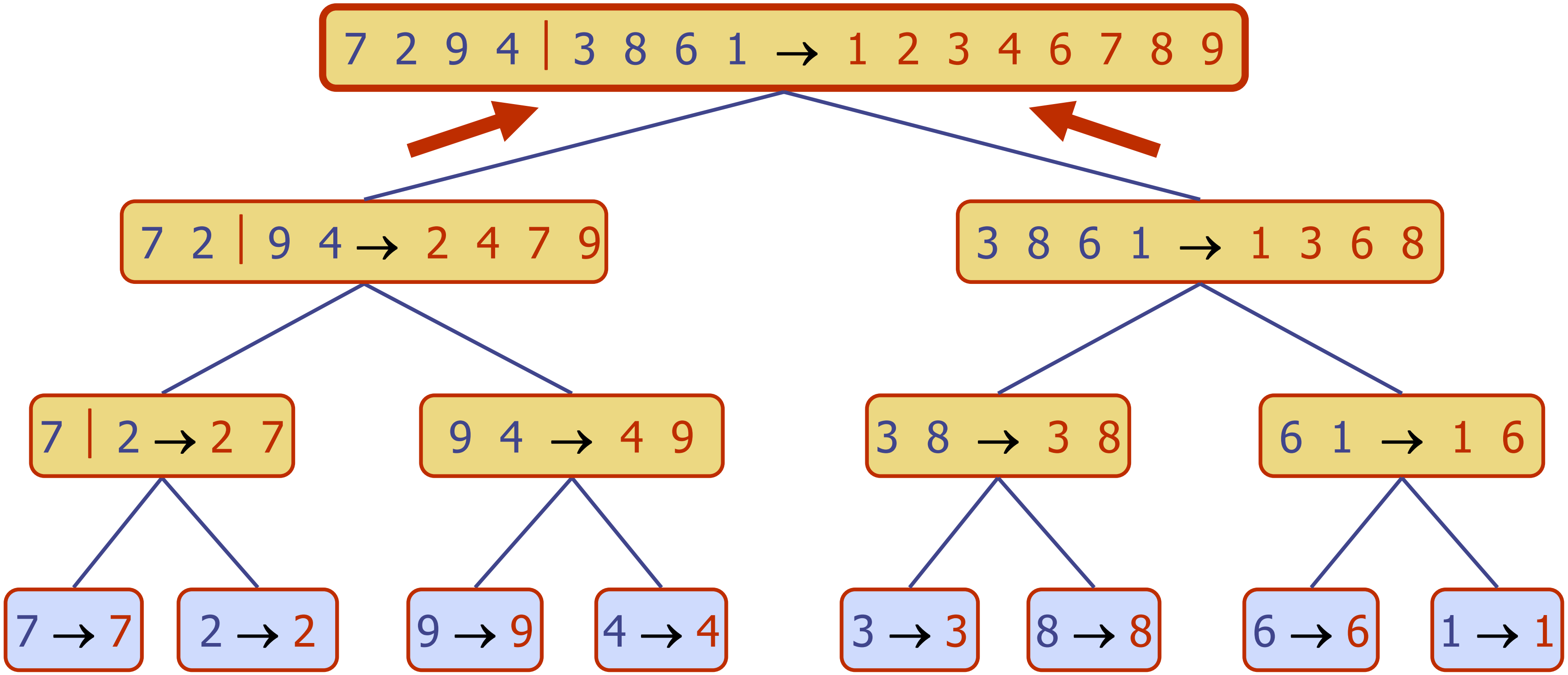
Execution Example (cont.)

- Recursive call, …, merge, merge

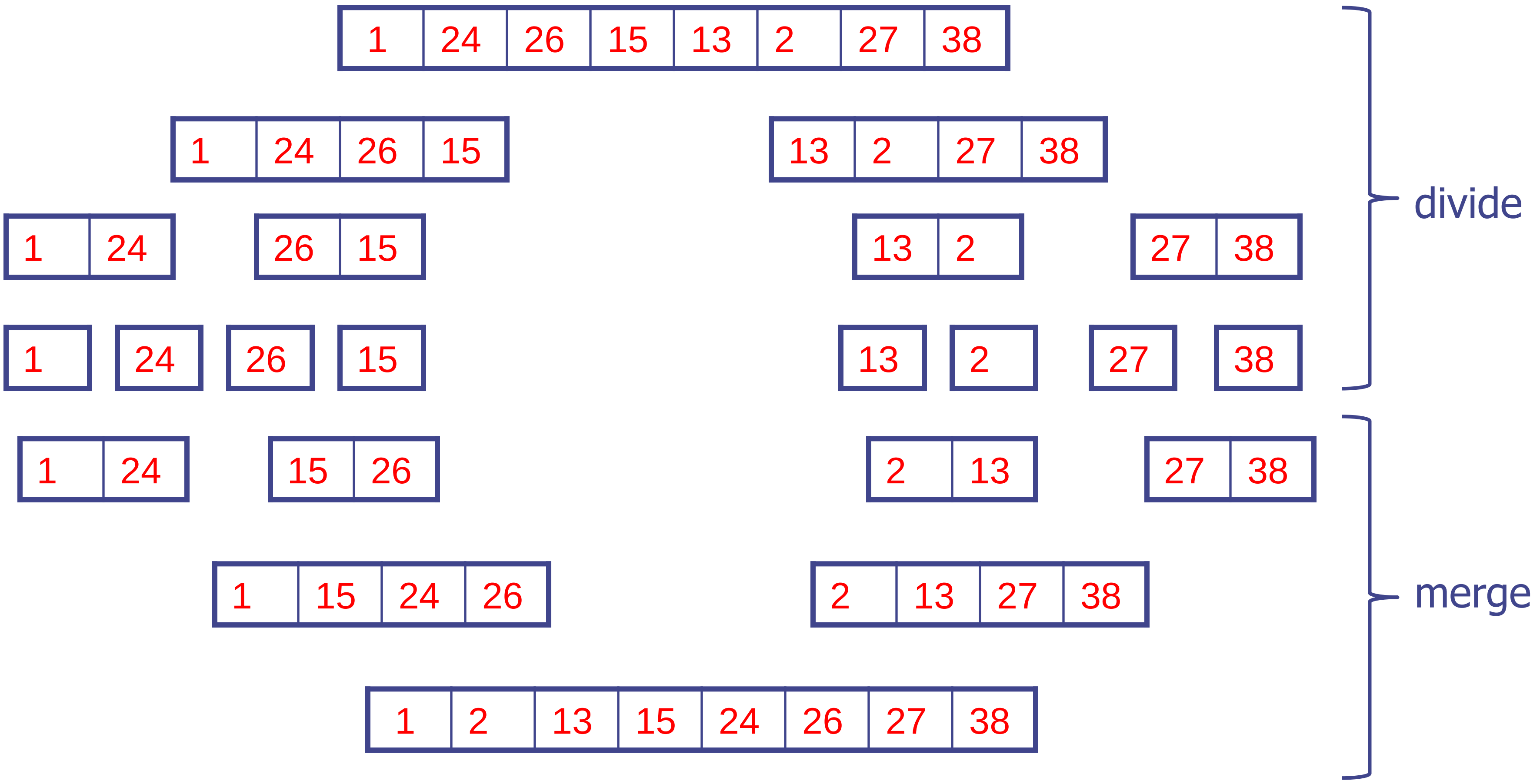


Execution Example (cont.)

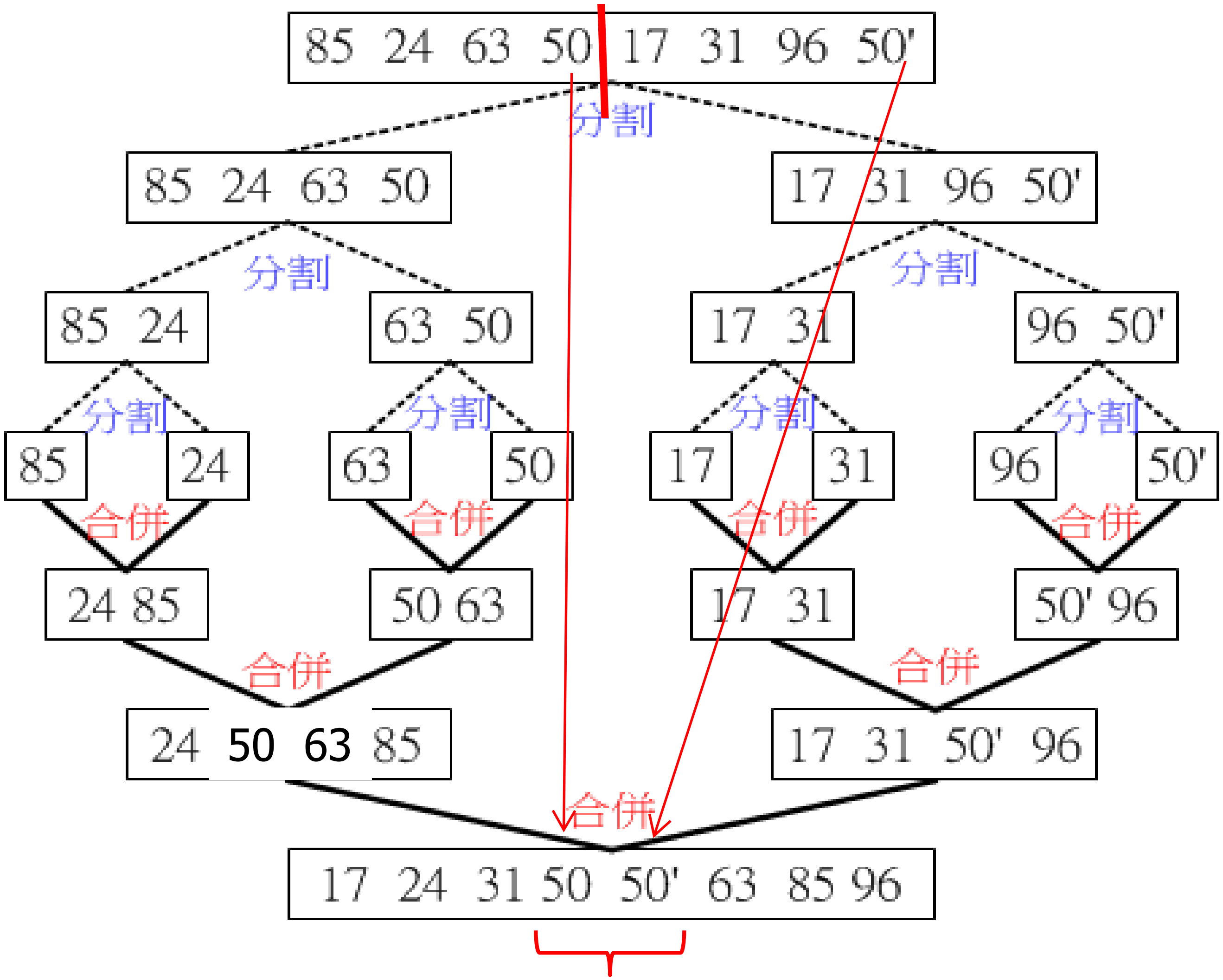
- Merge



Merge Sort: Example



Why Merge Sort Is Stable?



Same relative positions after merging!

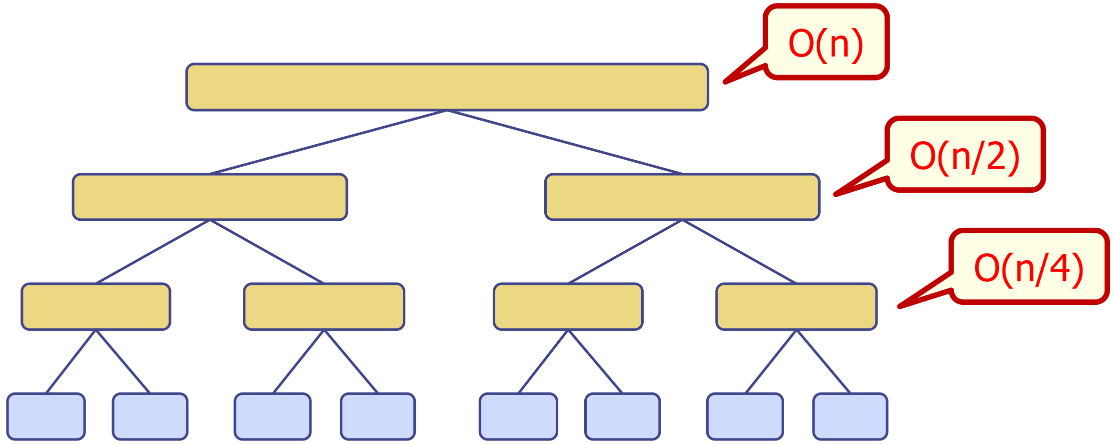
Resources on Merge Sort

- Numerous resources on merge sort
  - Wiki
    - Animation by sorting a vector
    - Animation by dots
  - Youtube
    - Detailed explanation with pseudo code

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	$n/2$
2	4	$n/4$
…	…	…



Summary of Sorting Algorithms

Algorithms	Time	Notes
selection-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
insertion-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
heap-sort	$O(n \log n)$	▪ fast ▪ in-place ▪ for large data sets (1K — 1M)
merge-sort	$O(n \log n)$	▪ fast ▪ Not in-place ▪ sequential data access ▪ for huge data sets (> 1M)