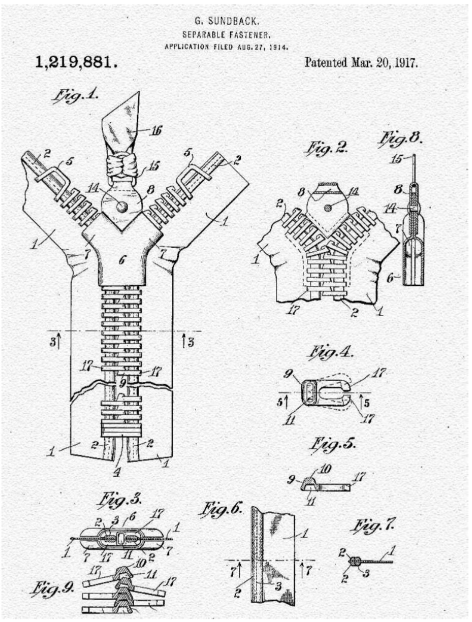
Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

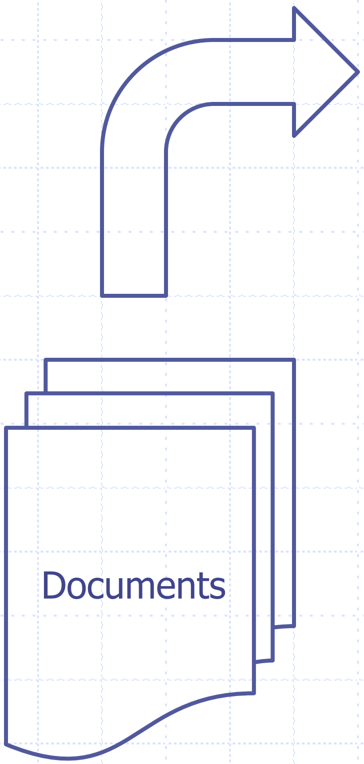
Merge Sort



Separable Fastener, U.S. Patent 1,219,881, 1917.
Public domain image.

Application: Internet Search Engines

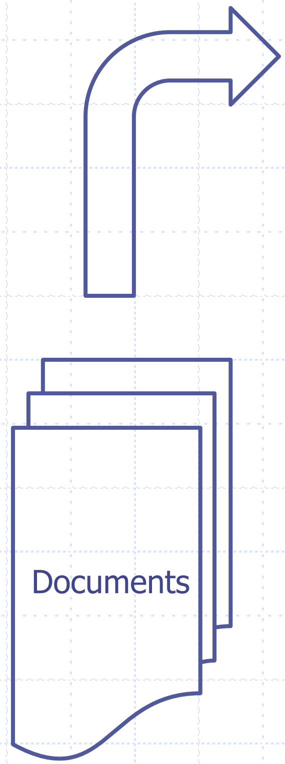
- Sorting has a lot of applications, including uses in Internet search engines.
- Sorting arises in the steps needed to build a data structure, known as the **inverted file** or **inverted index**, that allows a search engine to quickly return a list of the documents that contain a given keyword.



Word	Document Number & word location
banana	1:3, 2:45
butterfly	2:15, 3:12
camel	4:40
dog	1:60, 1:70, 2:22, 3:20, 4:11
horse	4:21
pig	2:55
pizza	1:56, 3:33

Application: How Sorting Builds an Internet Search Engine

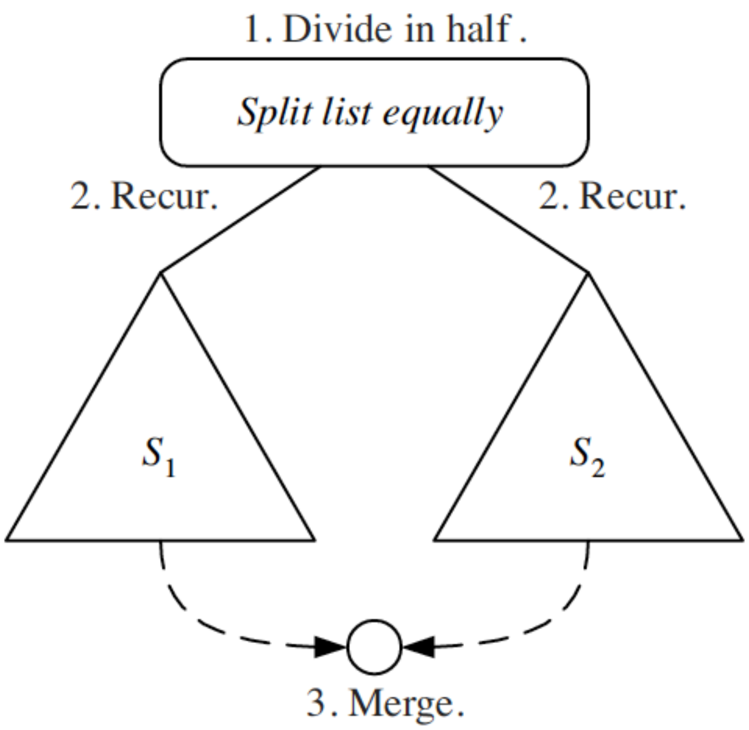
- To build an inverted file we need to identify, for each keyword, k, the documents containing k.
- Bringing all such documents together can be done simply by sorting the set of keyword-document pairs by keywords.
- This places all the (k, d) pairs with the same keyword, k, right next to one another.
- From this sorted list, it is then a simple computation to scan the list and build a lookup table of documents for each keyword that appears in this sorted list.



Word	Document Number & word location
banana	1:3, 2:45
butterfly	2:15, 3:12
camel	4:40
dog	1:60, 1:70, 2:22, 3:20, 4:11
horse	4:21
pig	2:55
pizza	1:56, 3:33

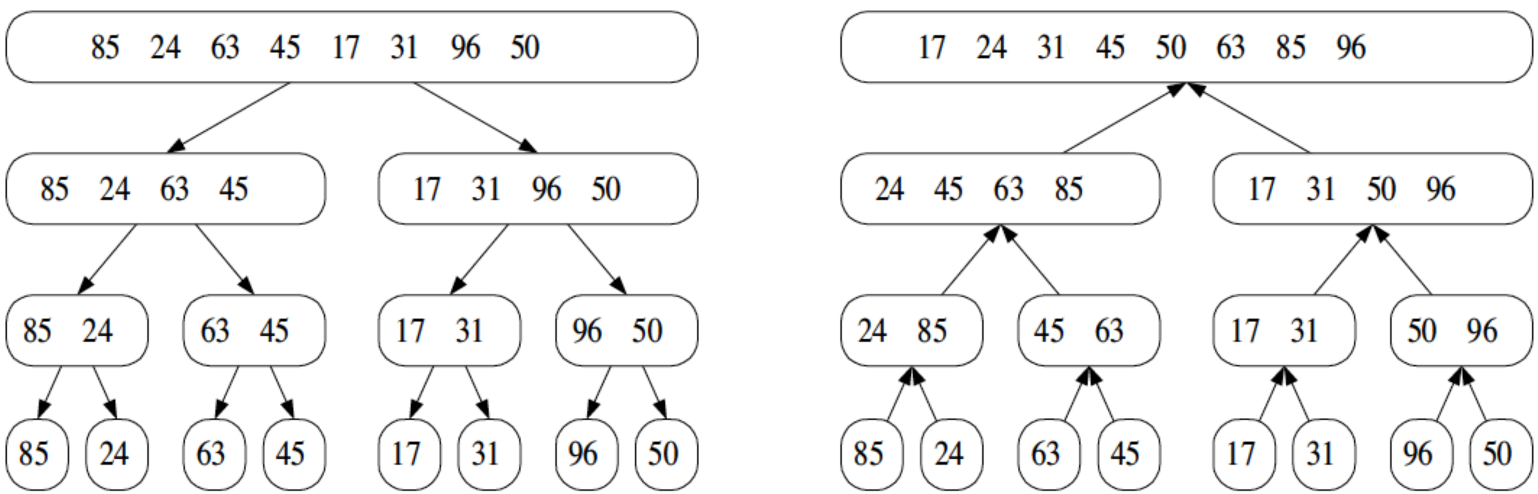
Divide-and-Conquer

- 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

- 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)



The Merge-Sort Algorithm

- Merge-sort on an input sequence 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 $\textit{mergeSort}(S)$
    Input sequence S with n elements
    Output sequence S sorted according to C
    if $S.\textit{size}() > 1$
        $(S_1, S_2) \leftarrow \textit{partition}(S, n/2)$
        $\textit{mergeSort}(S_1)$
        $\textit{mergeSort}(S_2)$
        $S \leftarrow \textit{merge}(S_1, S_2)$

Merging Two Sorted Sequences

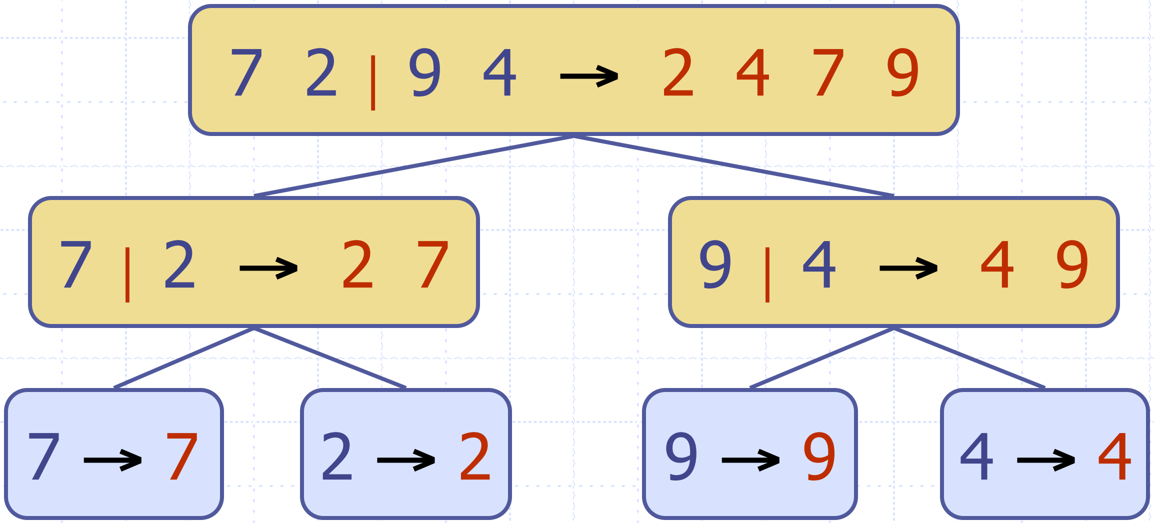
- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

Algorithm merge(S_1, S_2, S):
    Input: Two arrays, S_1 and S_2, of size n_1 and n_2, respectively, sorted in non-decreasing order, and an empty array, S, of size at least $n_1 + n_2$
    Output: S, containing the elements from S_1 and S_2 in sorted order

```
i ← 1
j ← 1
while i ≤ n and j ≤ n do
    if S1[i] ≤ S2[j] then
        S[i + j − 1] ← S1[i]
        i ← i + 1
    else
        S[i + j − 1] ← S2[j]
        j ← j + 1
while i ≤ n do
    S[i + j − 1] ← S1[i]
    i ← i + 1
while j ≤ n do
    S[i + j − 1] ← S2[j]
    j ← j + 1
```

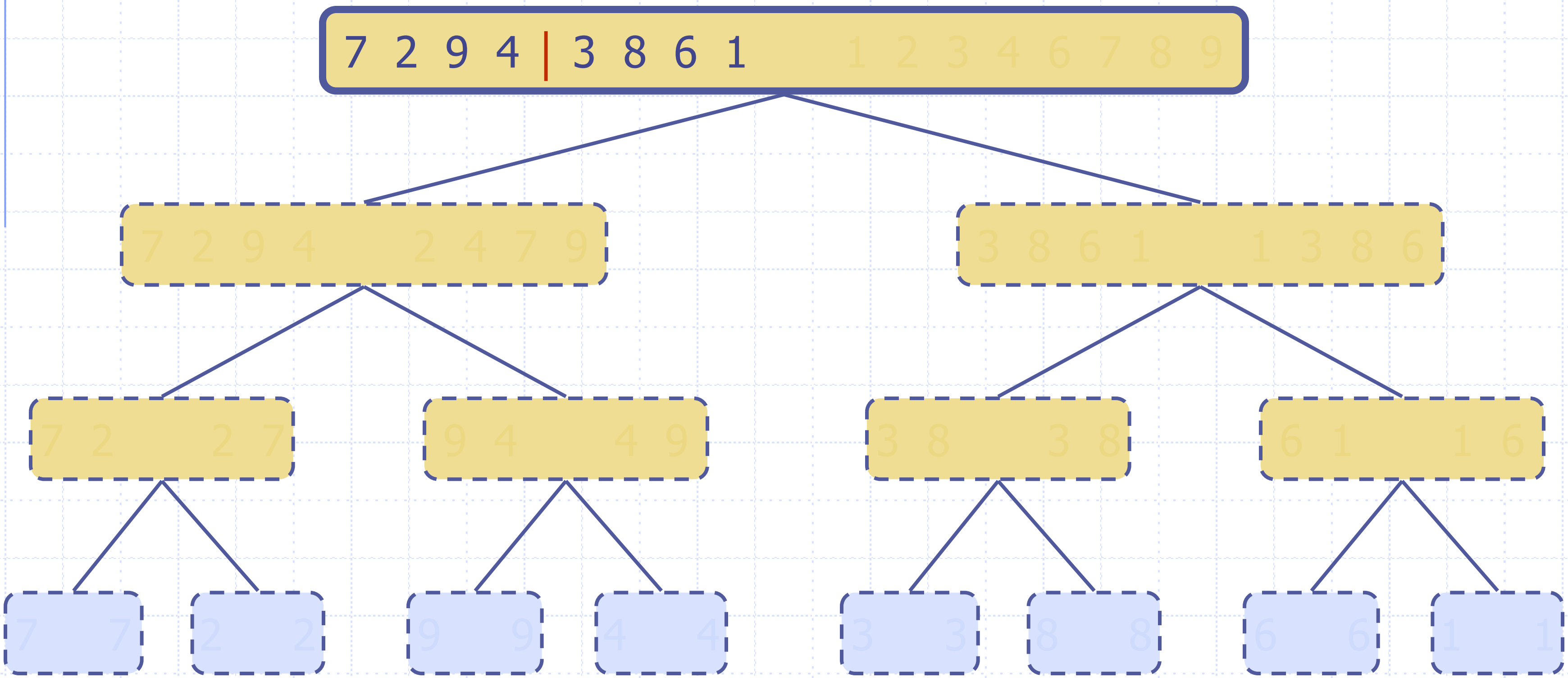
Merge-Sort Tree

- 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



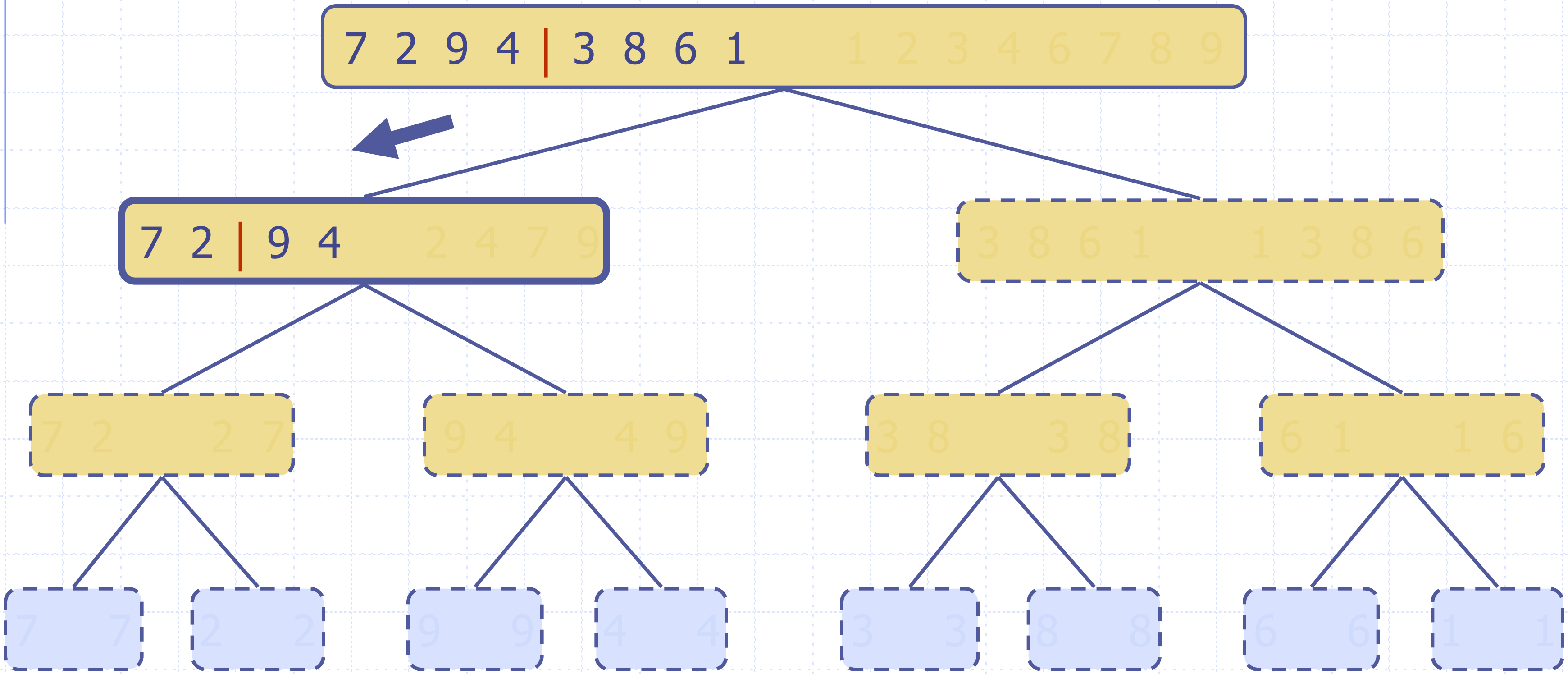
Execution Example

- Partition



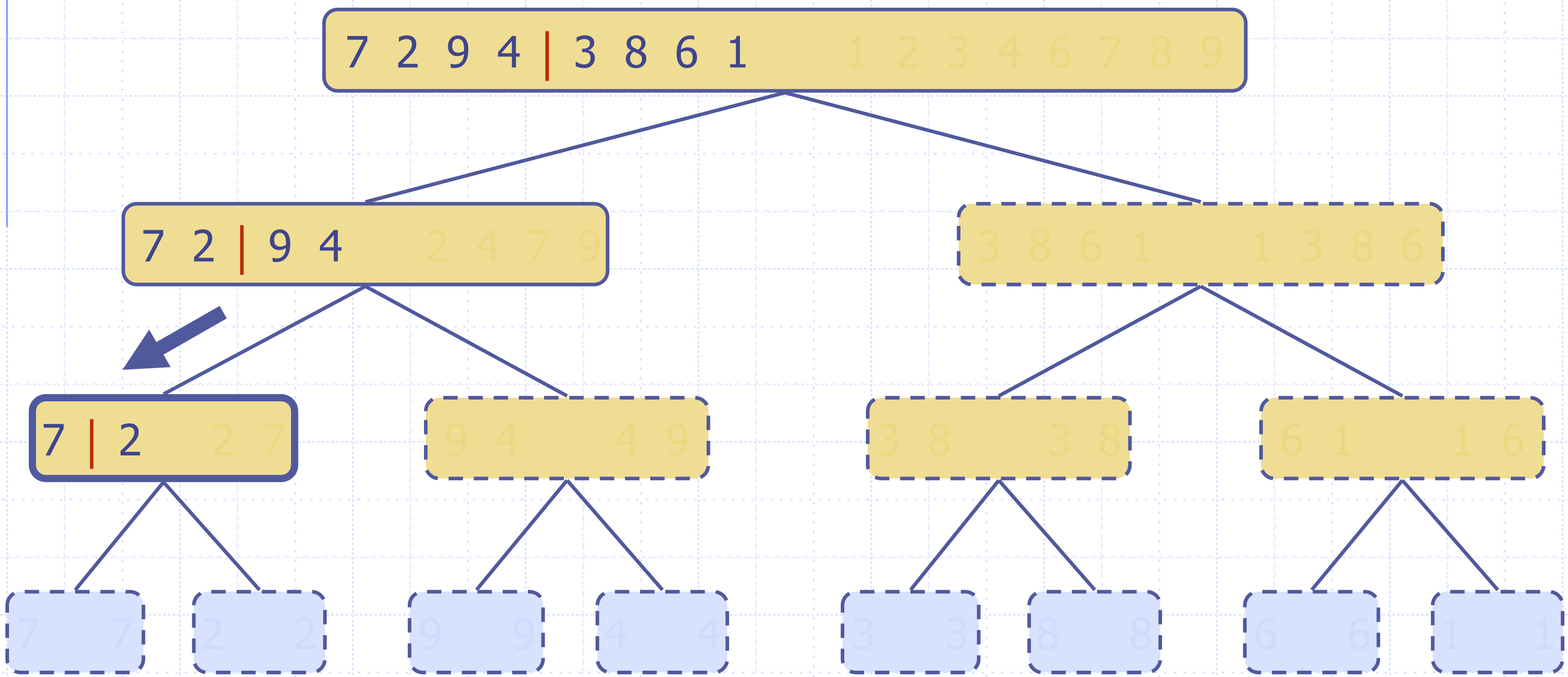
Execution Example (cont.)

- Recursive call, partition



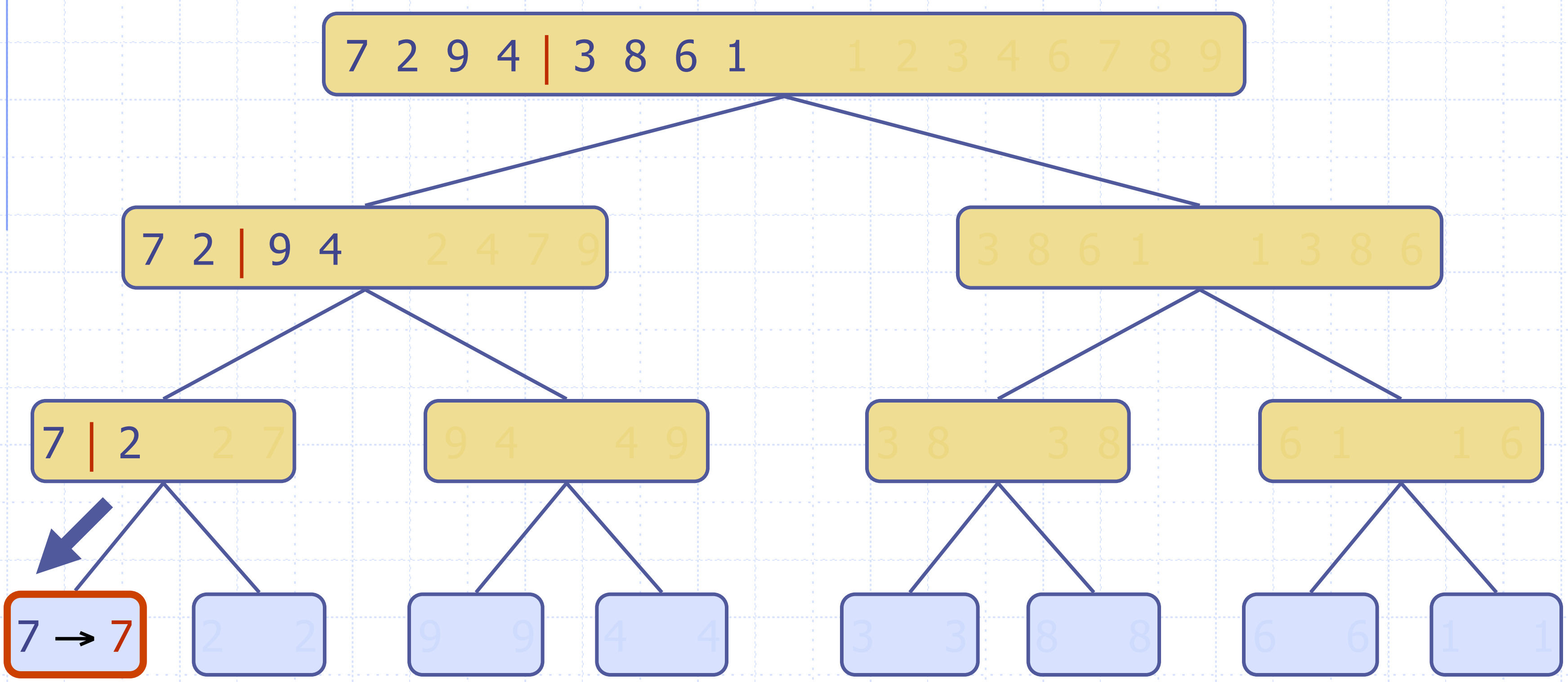
Execution Example (cont.)

- Recursive call, partition



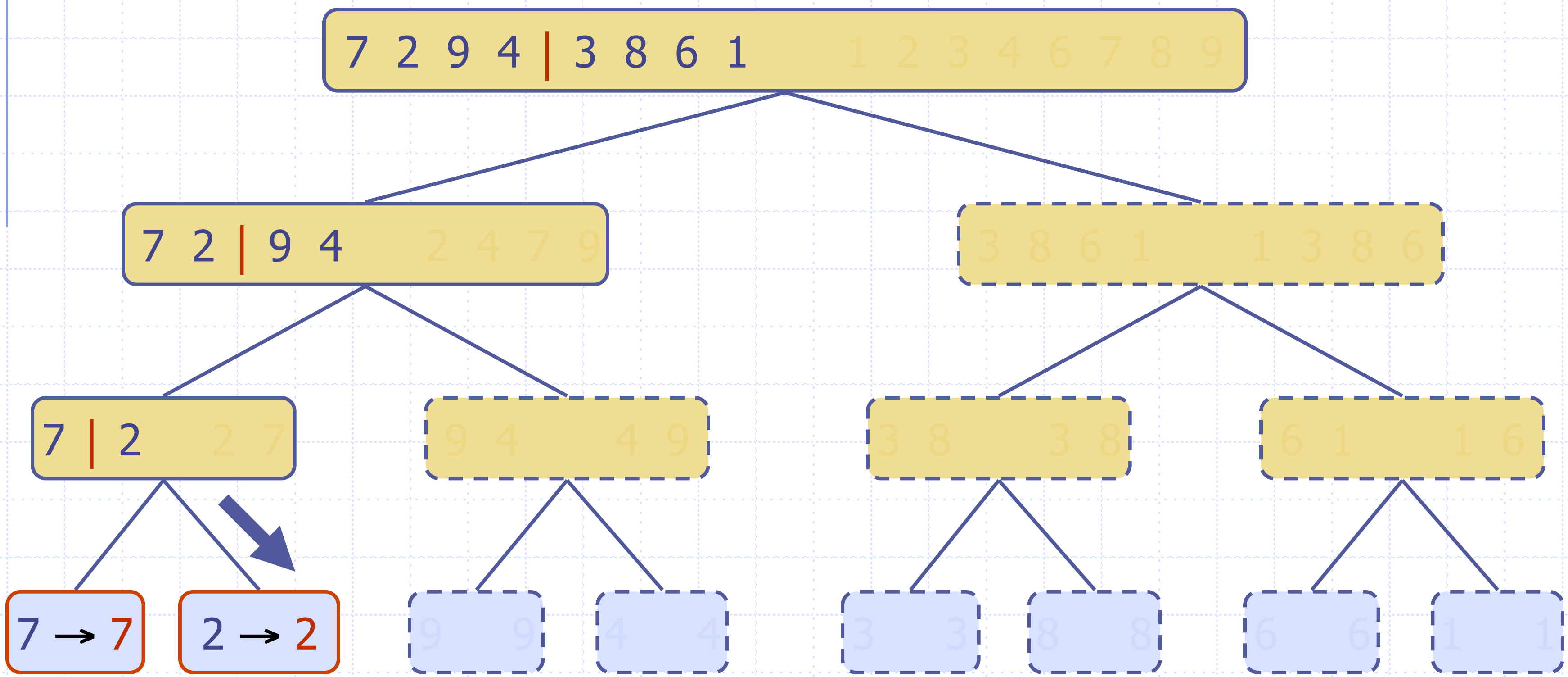
Execution Example (cont.)

- Recursive call, base case



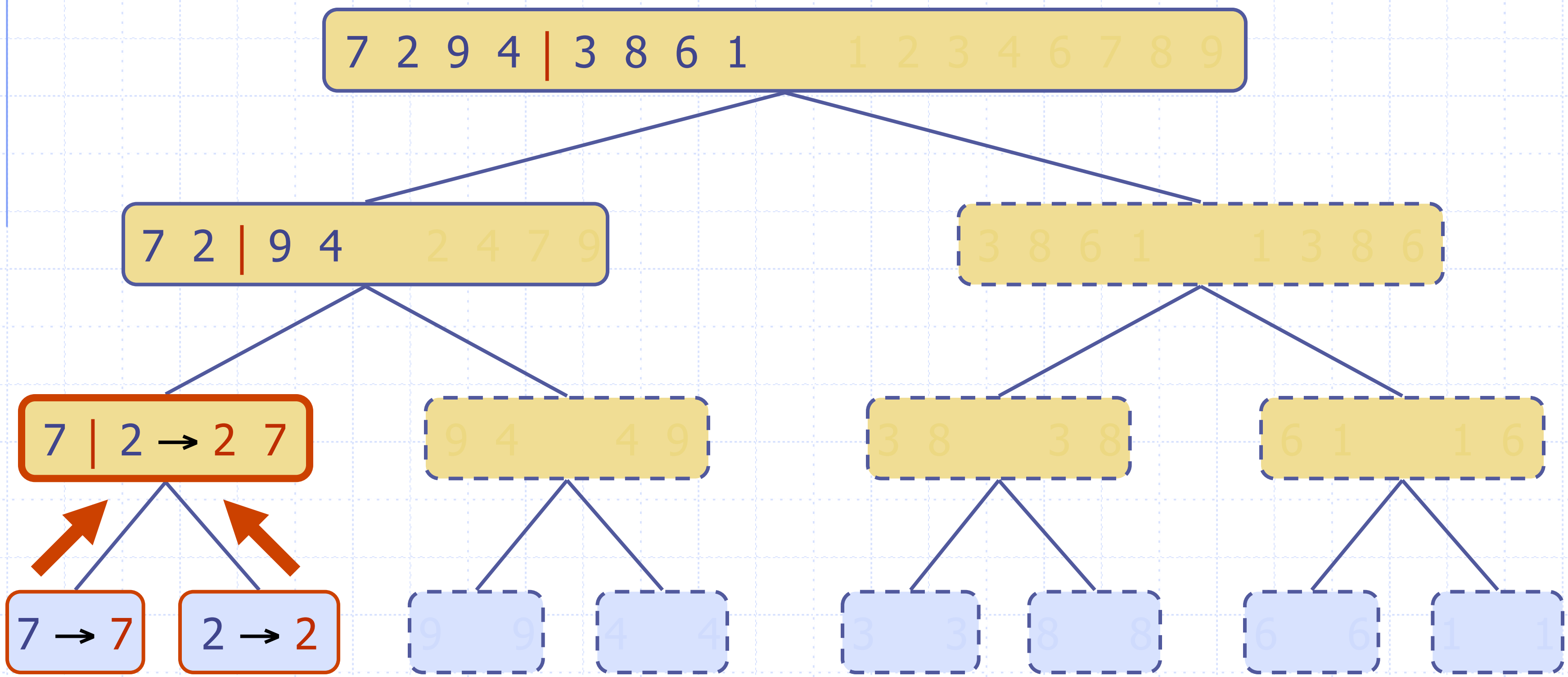
Execution Example (cont.)

- Recursive call, base case



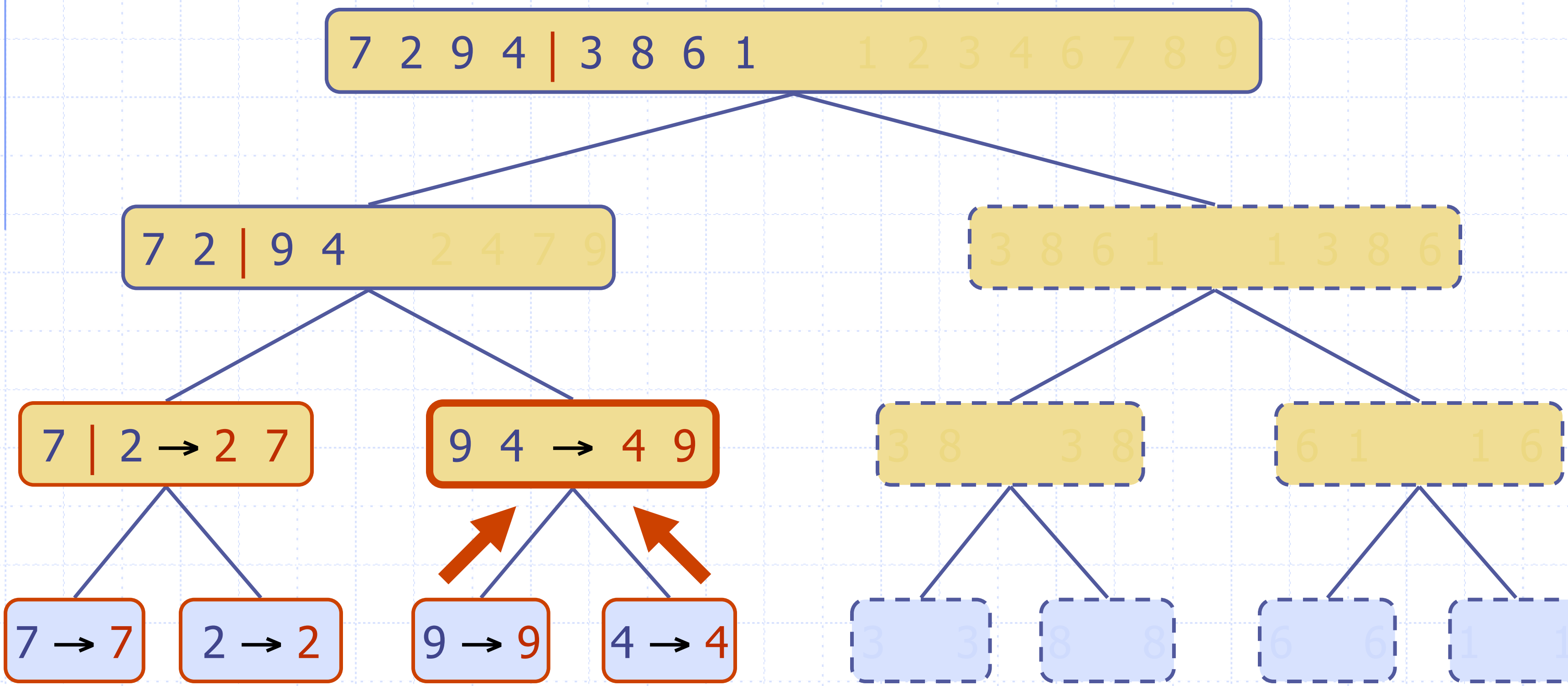
Execution Example (cont.)

- Merge



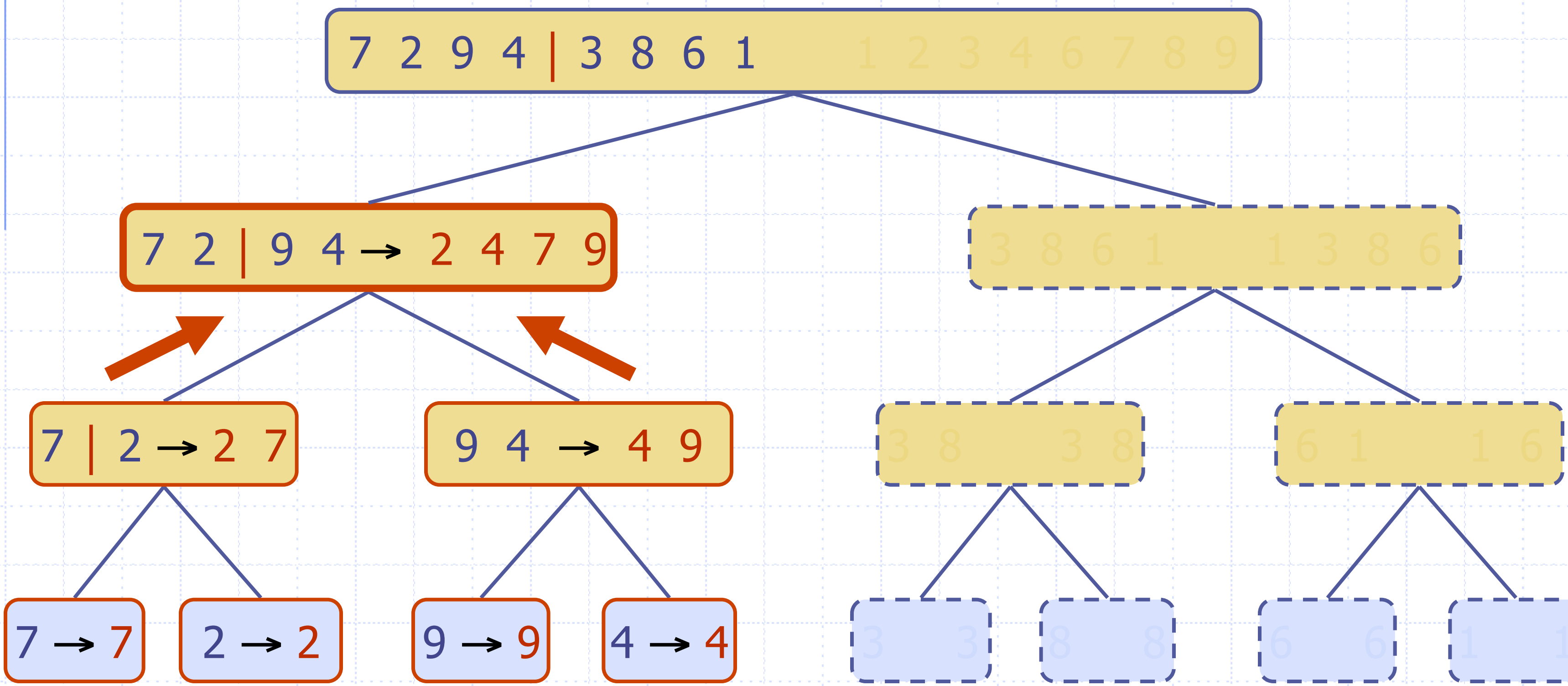
Execution Example (cont.)

- Recursive call, …, base case, merge



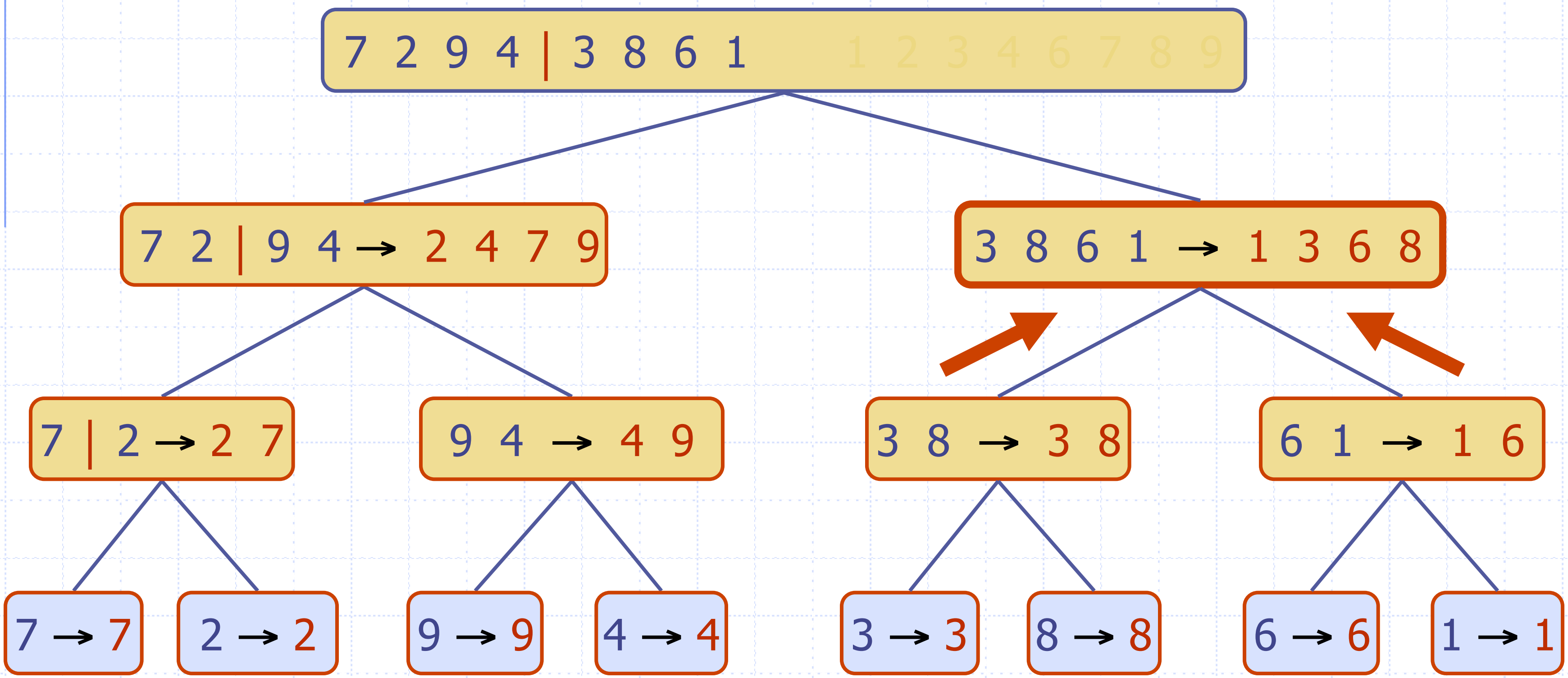
Execution Example (cont.)

- Merge



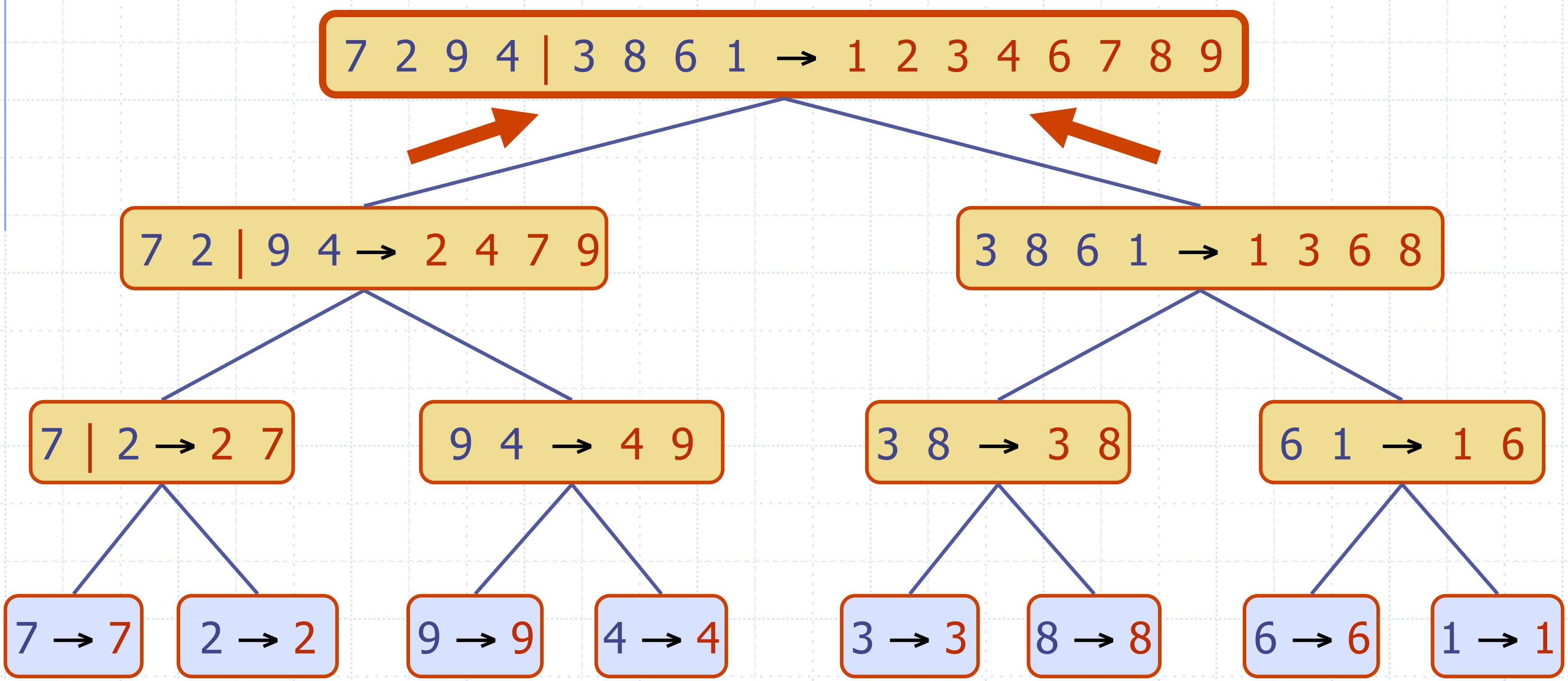
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

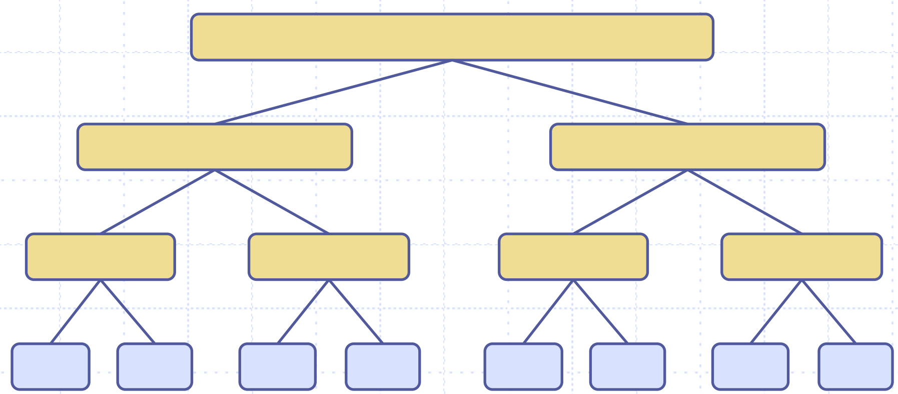
- Merge



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 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	$n/2^i$
…	…	…



Summary of Sorting Algorithms

Algorithm	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 ▪ sequential data access ▪ for huge data sets (> 1M)