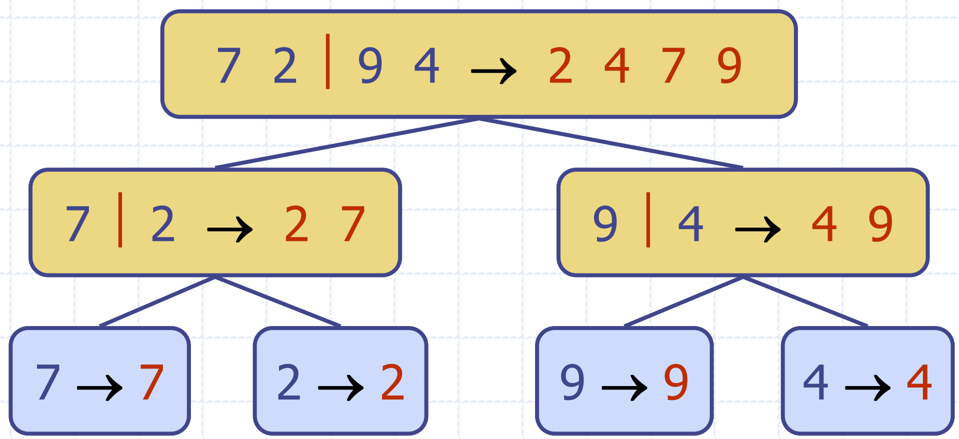
Merge Sort and Sorting Lower Bound

Sections 11.1 and 11.3.1



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 often subproblems of size 1 or 2
- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- 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 manner (suitable to sort data on a disk)

Merge-Sort

- 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 *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

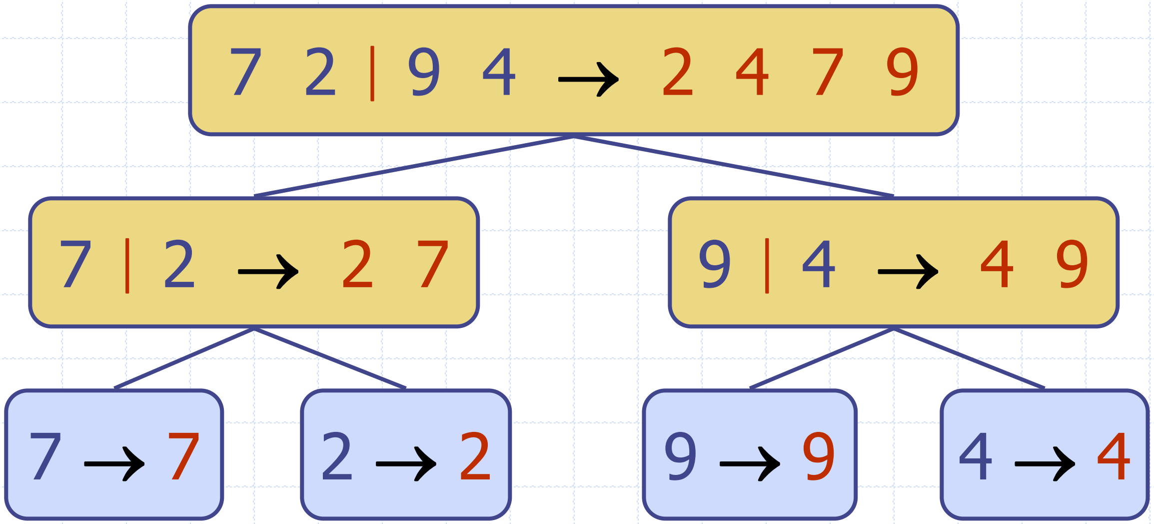
- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

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

    S ← empty sequence
    while ¬A.empty() ∧ ¬B.empty()
        if A.front() < B.front()
            S.addBack(A.front()); A.eraseFront();
        else
            S.addBack(B.front()); B.eraseFront();
    while ¬A.empty()
        S.addBack(A.front()); A.eraseFront();
    while ¬B.empty()
        S.addBack(B.front()); B.eraseFront();
    return S
```

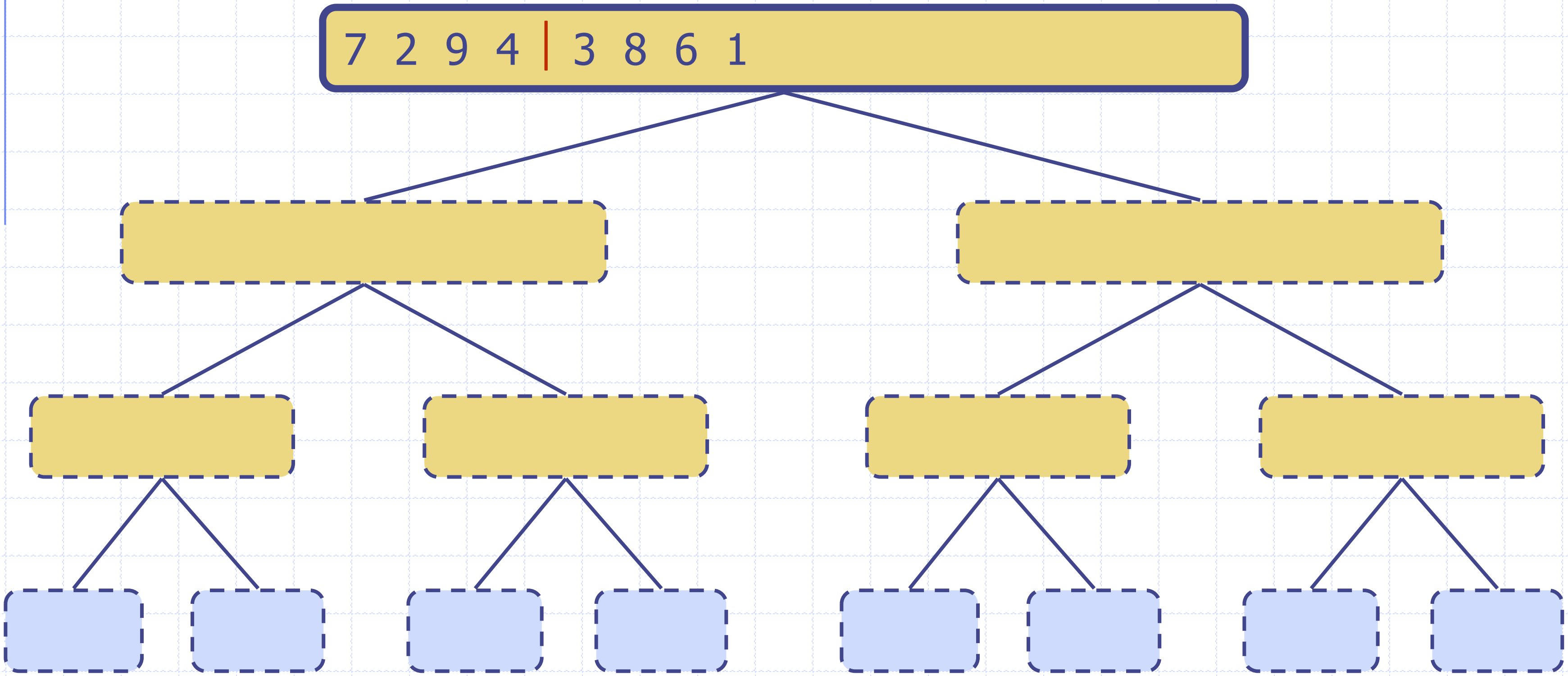
Merge-Sort Tree

- 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 1



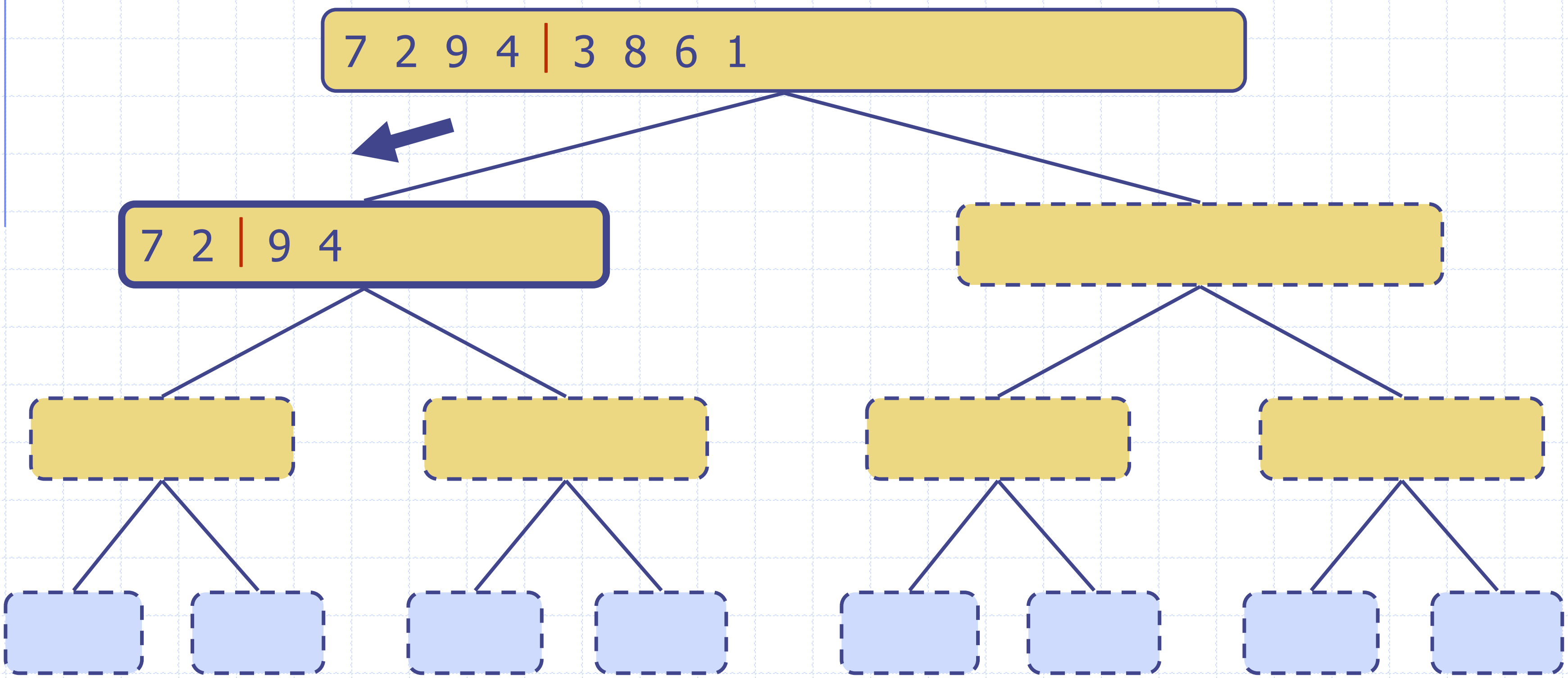
Execution Example

- Partition



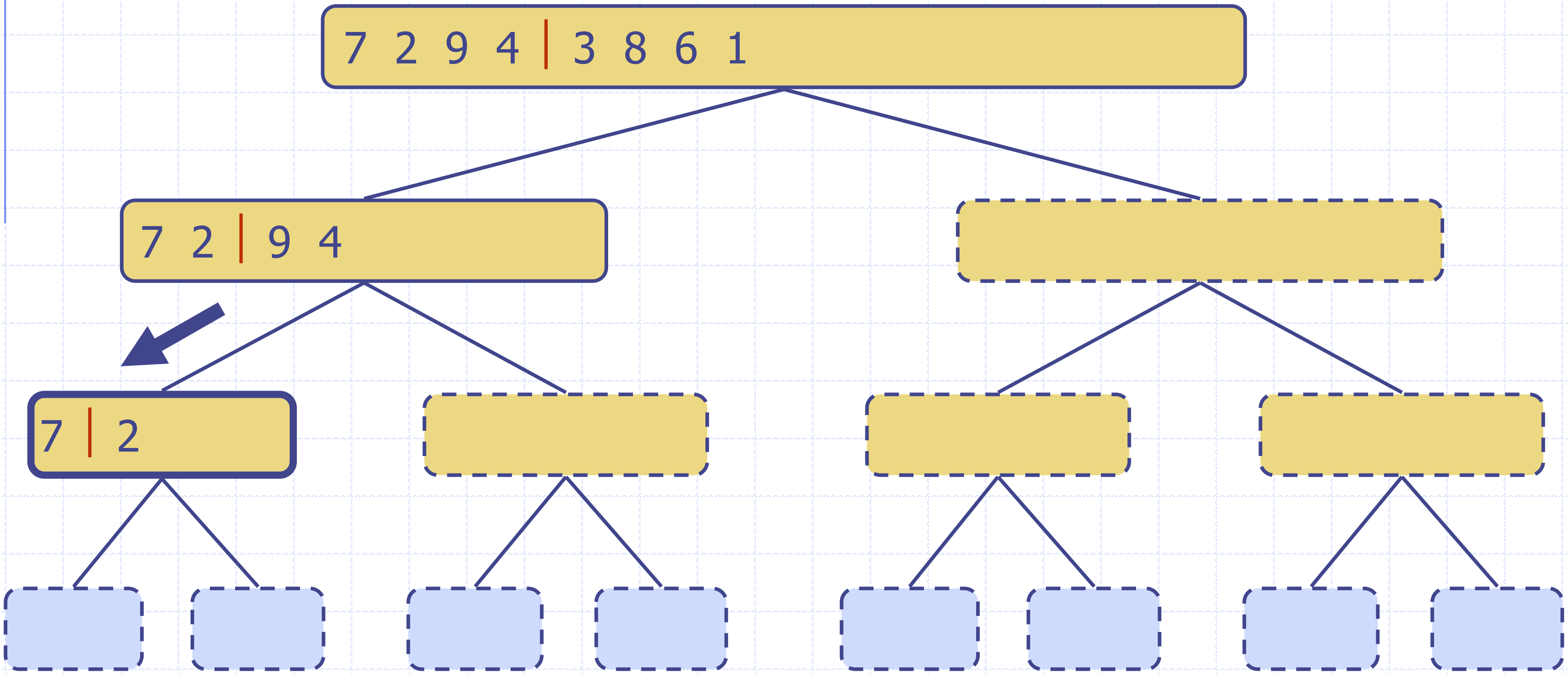
Execution Example (cont.)

- Recursive call, partition



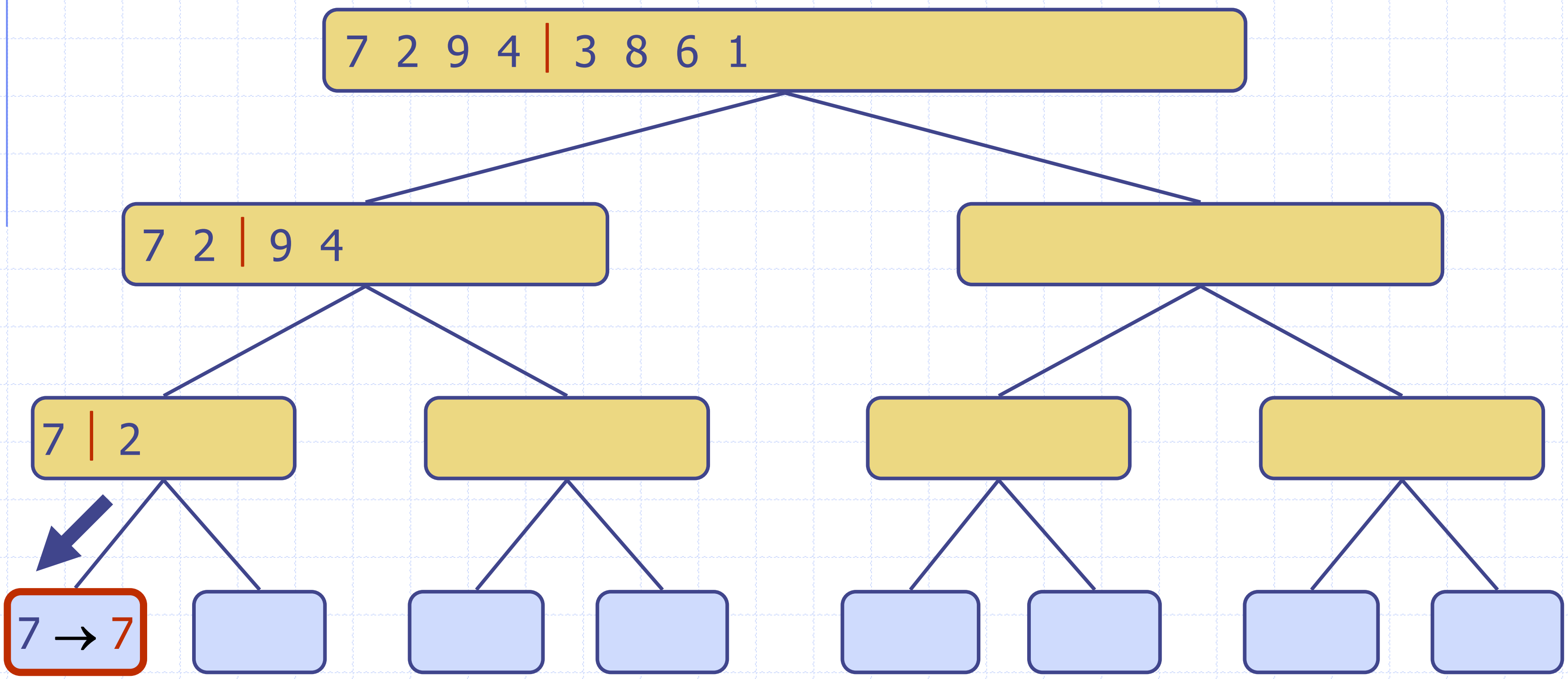
Execution Example (cont.)

- Recursive call, partition



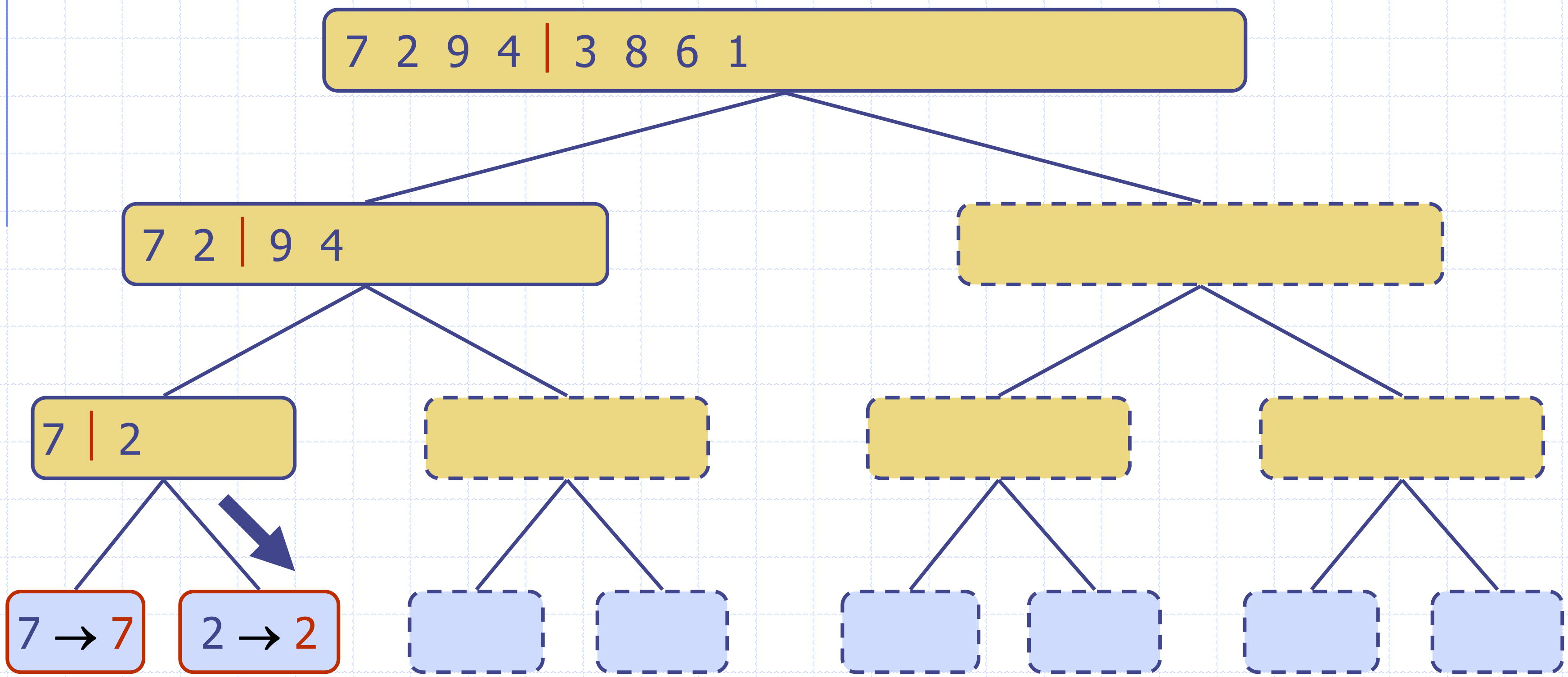
Execution Example (cont.)

- Recursive call, base case



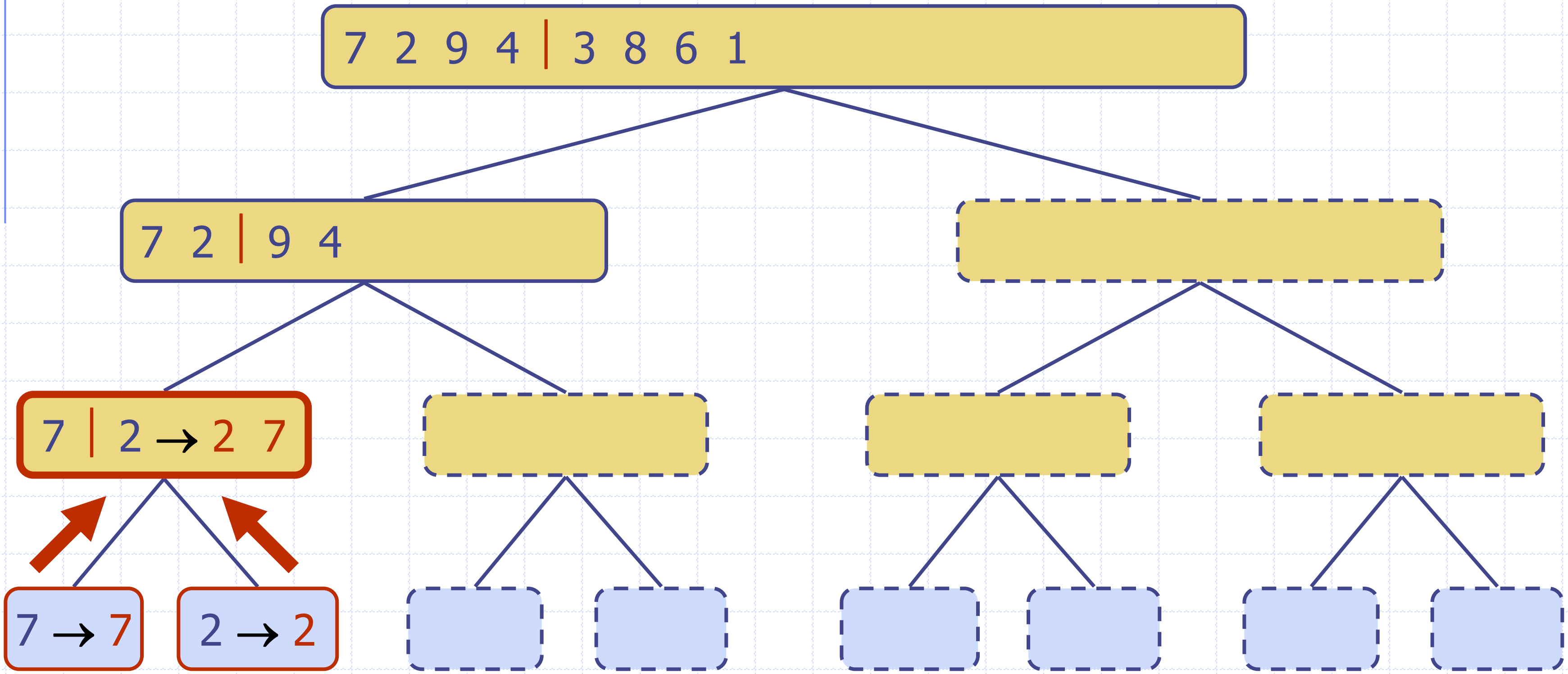
Execution Example (cont.)

- Recursive call, base case



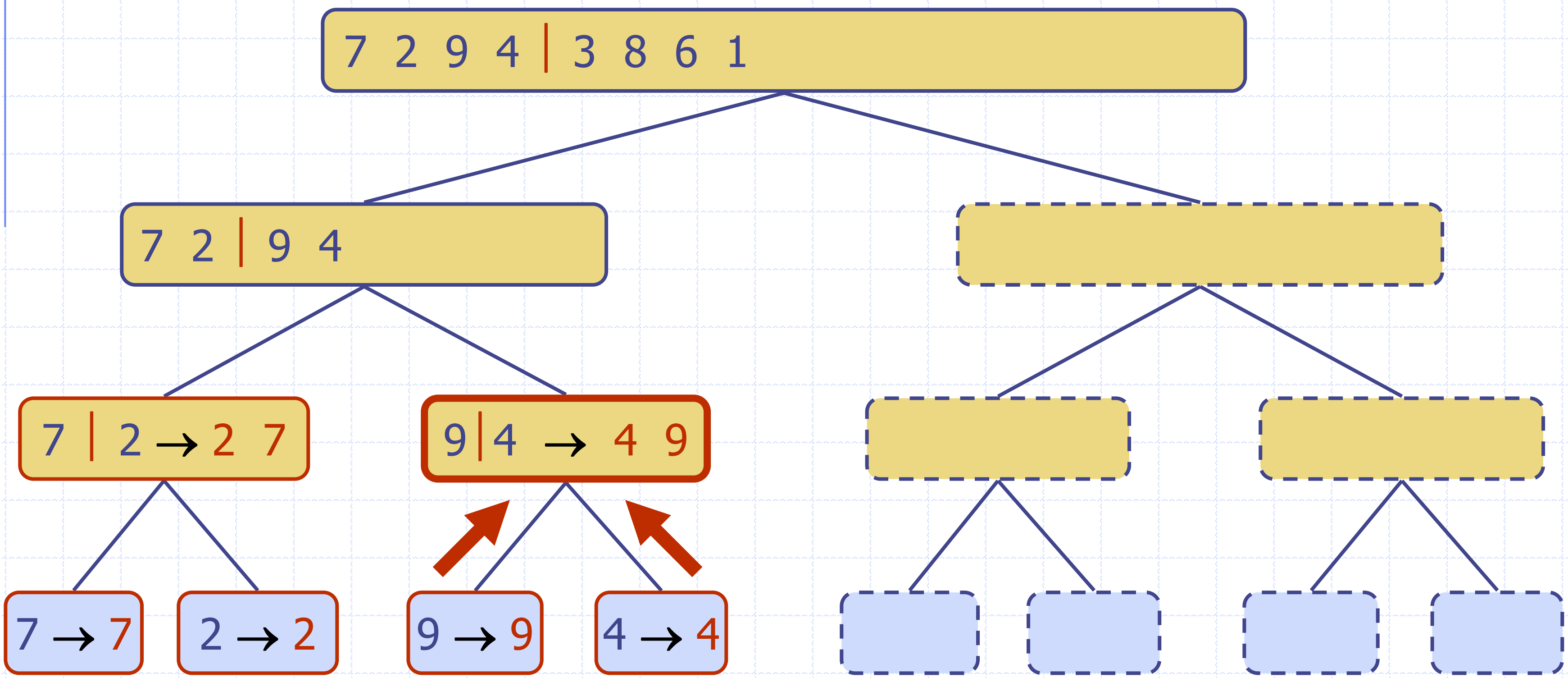
Execution Example (cont.)

- Merge



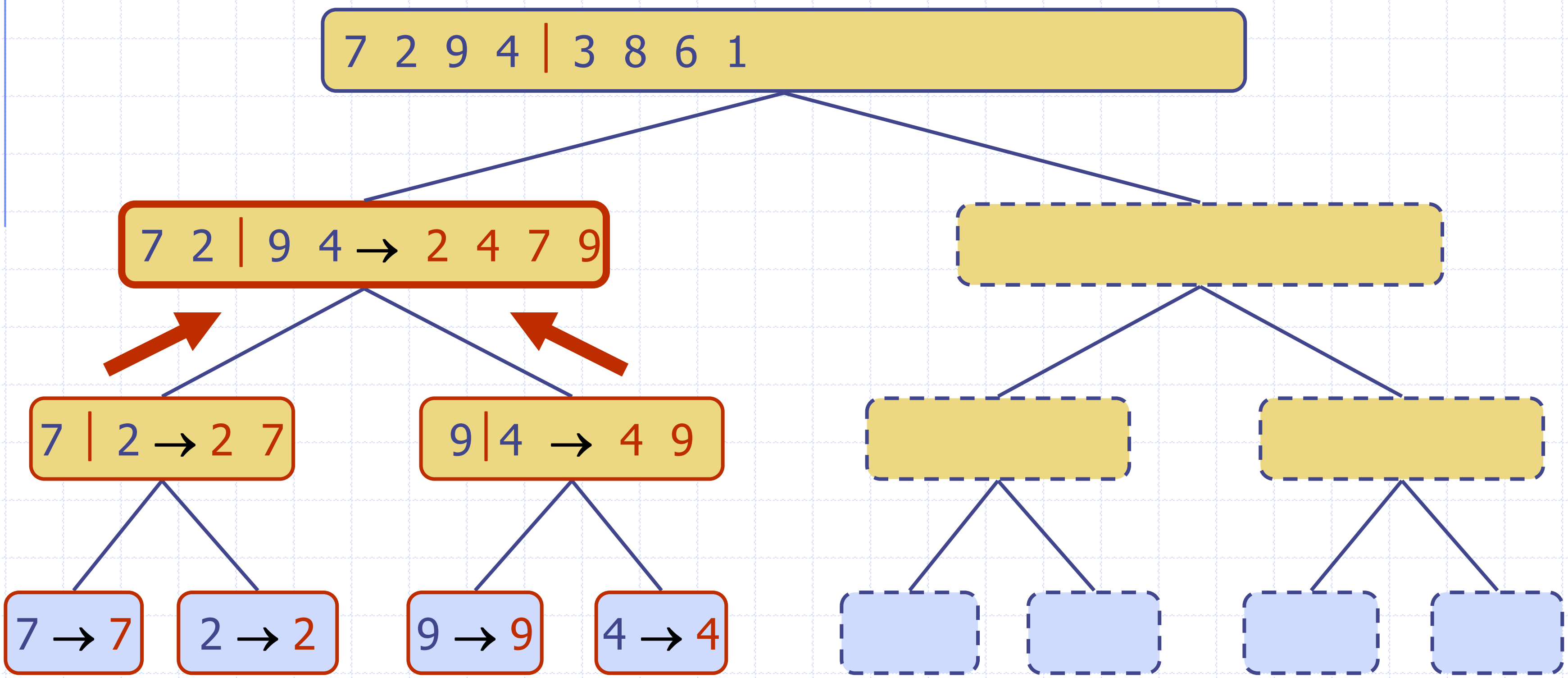
Execution Example (cont.)

- Recursive call, …, base case, merge



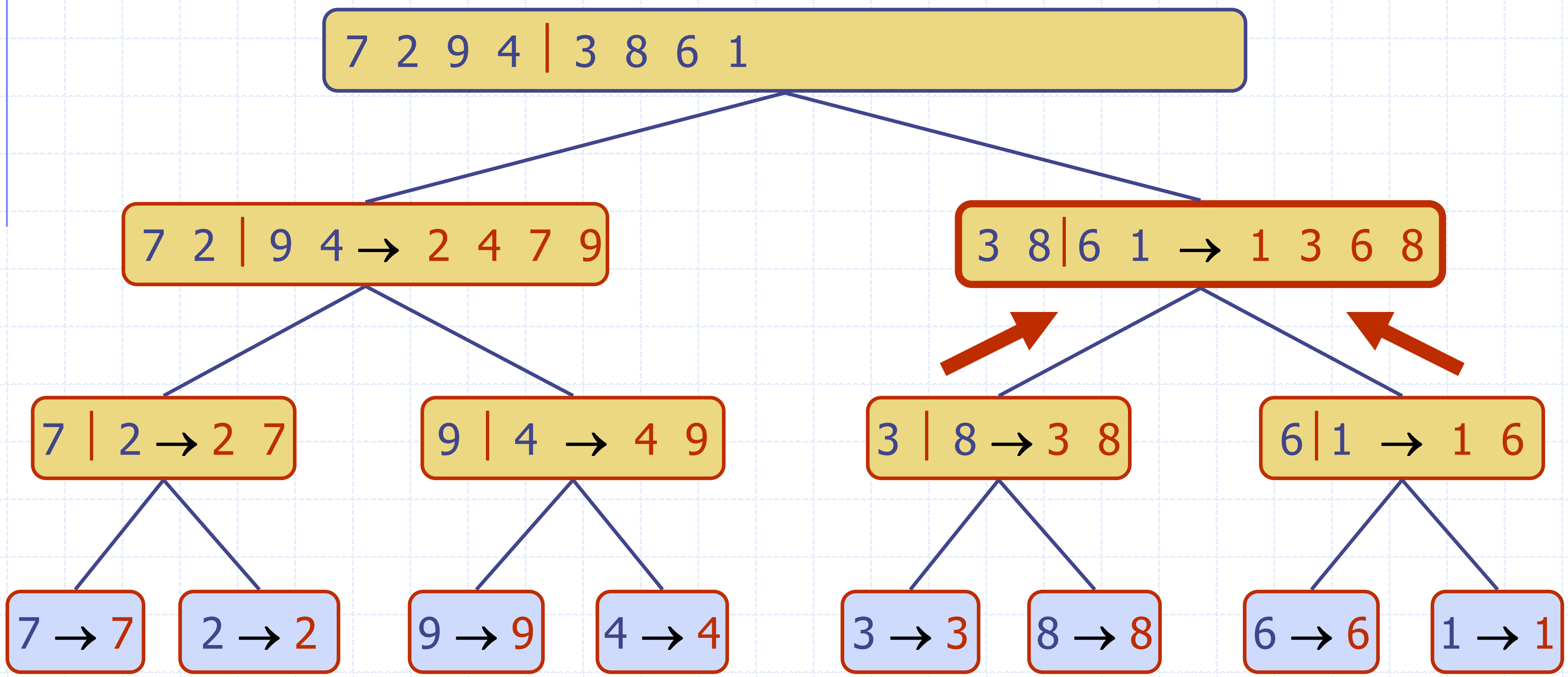
Execution Example (cont.)

- Merge



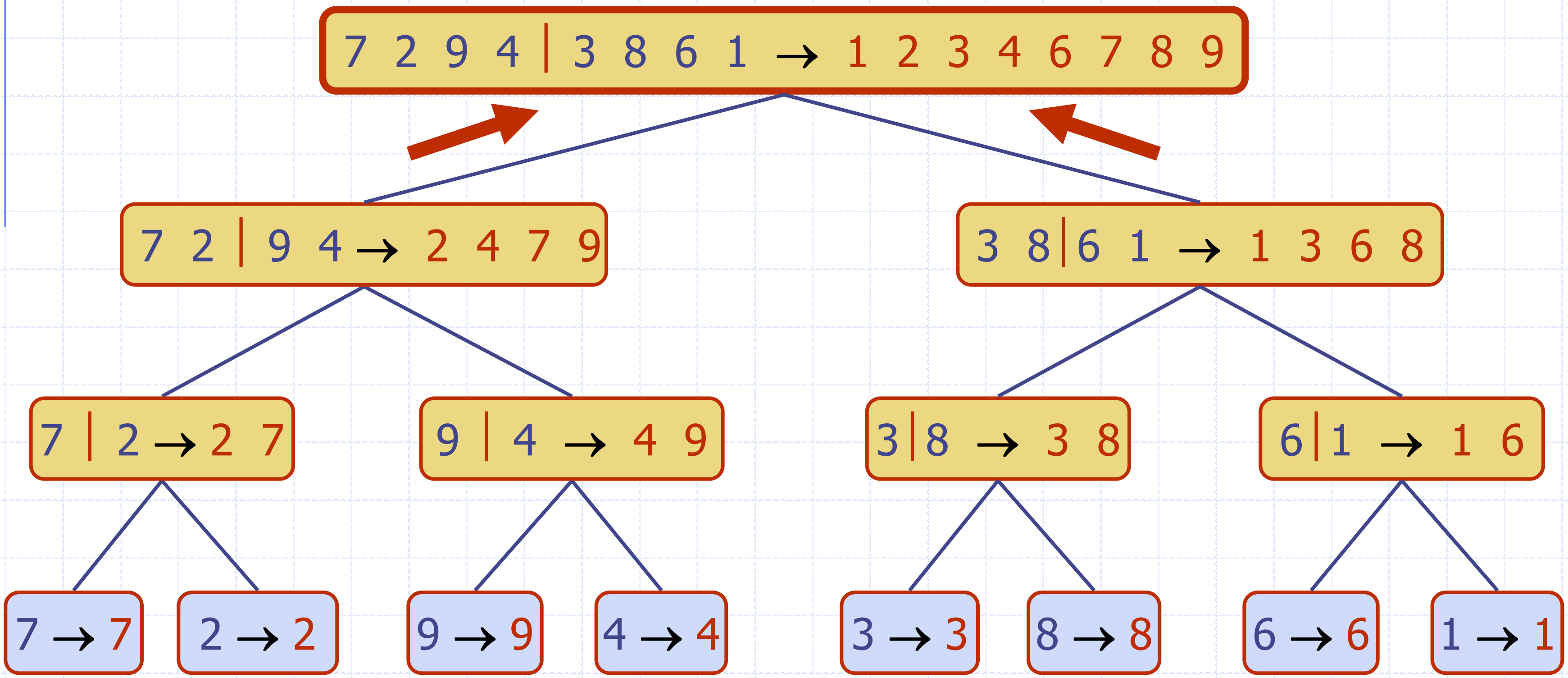
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

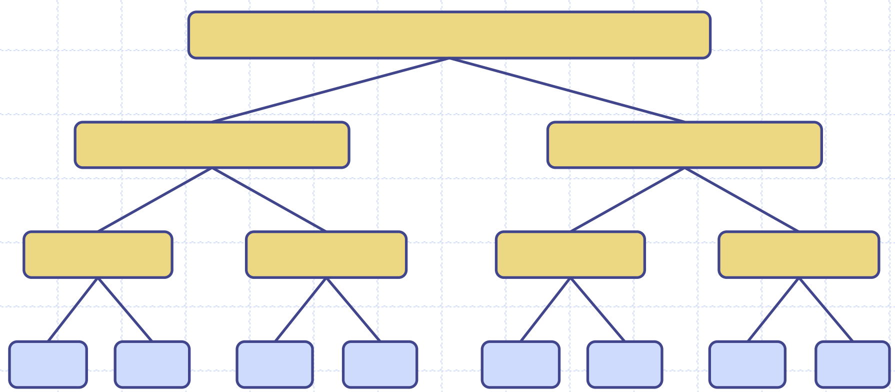
- Merge



Analysis of Merge-Sort

- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide the sequence in half,
- 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 $O(n \log n)$

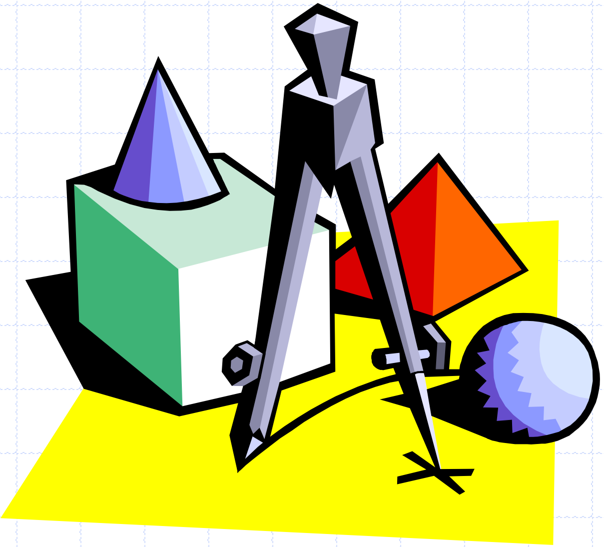
depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



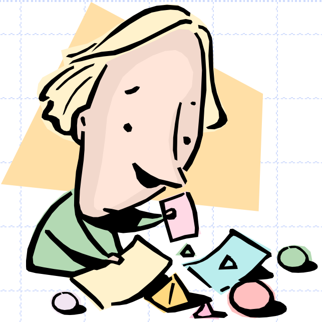
Summary of Sorting Algorithms

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)

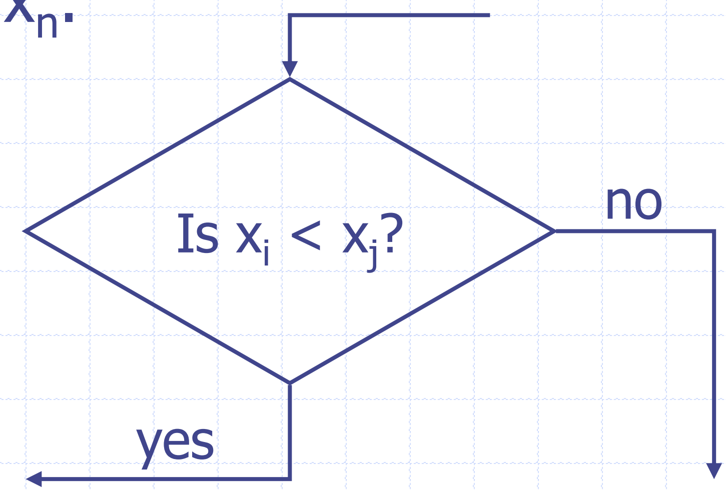
Sorting Lower Bound



Comparison-Based Sorting

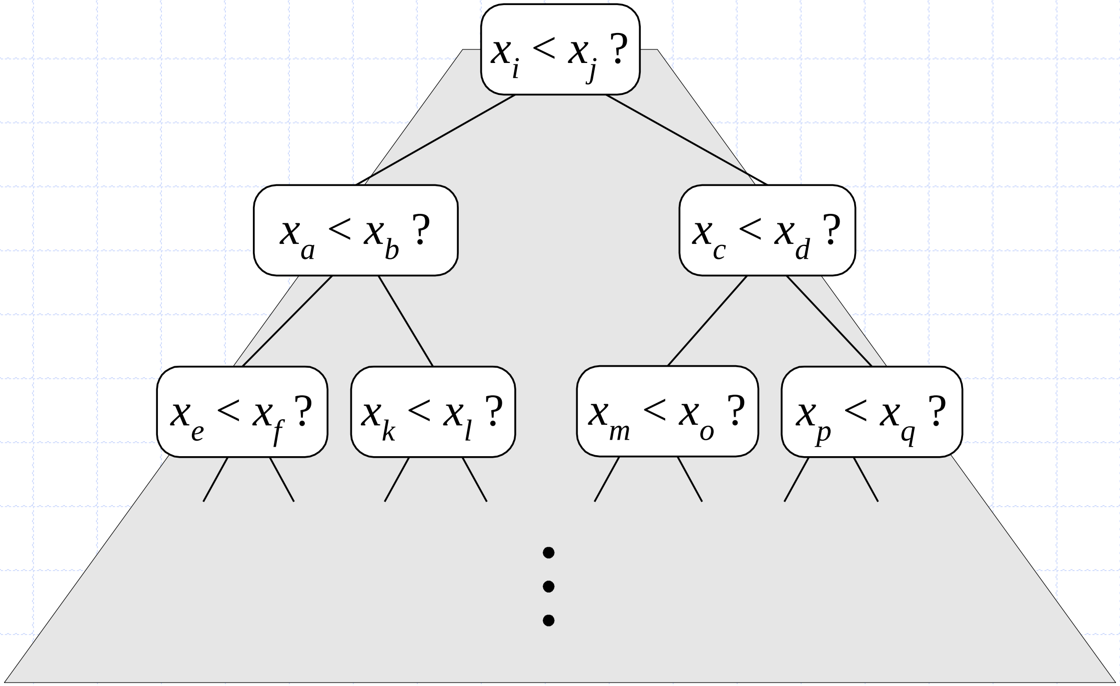


- Many sorting algorithms are comparison based.
  - They sort by making comparisons between pairs of objects
  - Examples: bubble-sort, selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, ...
- Let us therefore derive a lower bound on the running time of any algorithm that uses comparisons to sort n elements, $x_1, x_2, \ldots, x_n$.



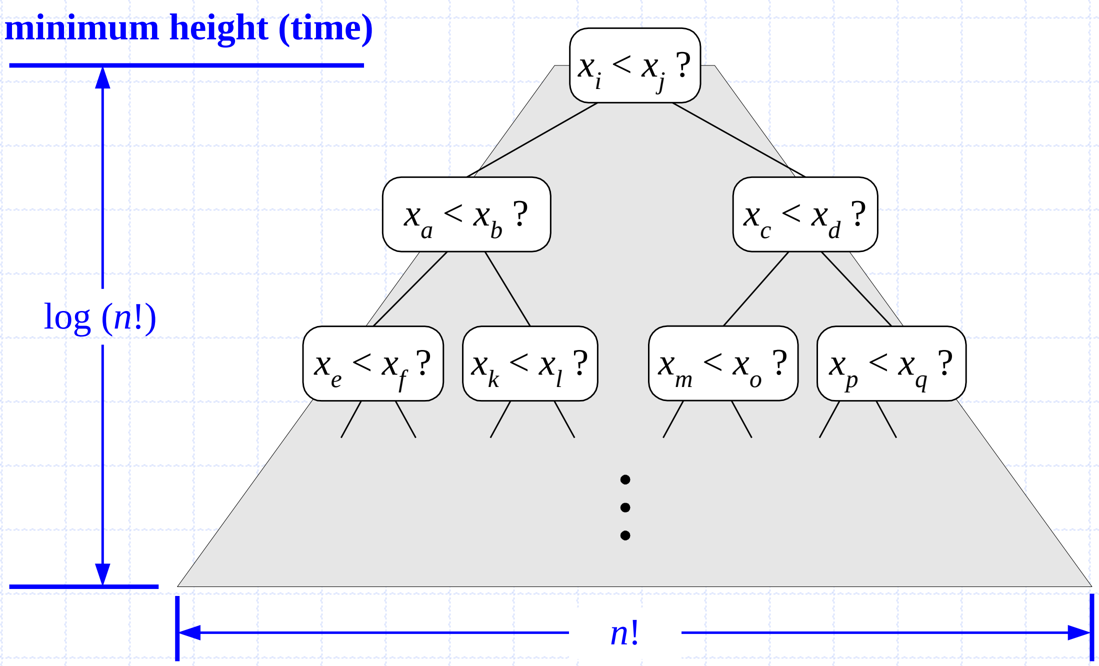
Counting Comparisons

- Let us just count comparisons then.
- Each possible run of the algorithm corresponds to a root-to-leaf path in a decision tree

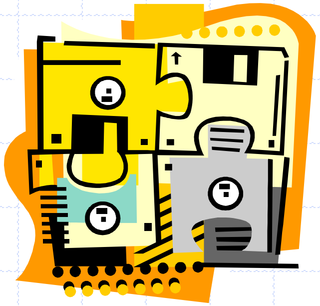


Decision Tree Height

- The height of the decision tree is a lower bound on the running time
- Each leaf specifies how to "unscramble" an input permutation.
- Every input permutation must lead to a separate leaf.
- There are n!=1·2 · … ·n leaves.
- So the height is at least log (n!)



The Lower Bound



- Any comparison-based sorting algorithms takes at least log (n!) time
- Therefore, any such algorithm takes time at least

$$\log(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = (n/2)\log(n/2).$$

- That is, any comparison-based sorting algorithm must run in Ω(n log n) time.

The Lower Bound

- The preceding argument uses the fact that

$$n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

which we can easily see by writing out what n! means:

$$\begin{aligned} n! &= n \cdot (n-1) \cdot \cdots \cdot \left(\frac{n}{2}+1\right) \cdot \frac{n}{2} \cdot \left(\frac{n}{2}-1\right) \cdot \ \cdots \ \cdot 2 \cdot 1 \\ &\geq n \cdot (n-1) \cdot \cdots \cdot \left(\frac{n}{2}+1\right) \\ &\geq \underbrace{\frac{n}{2} \cdot \frac{n}{2} \cdot \qquad \cdots \ \cdot \frac{n}{2}}_{\frac{n}{2} \text{ factors}} = \left(\frac{n}{2}\right)^{\frac{n}{2}} \end{aligned}$$