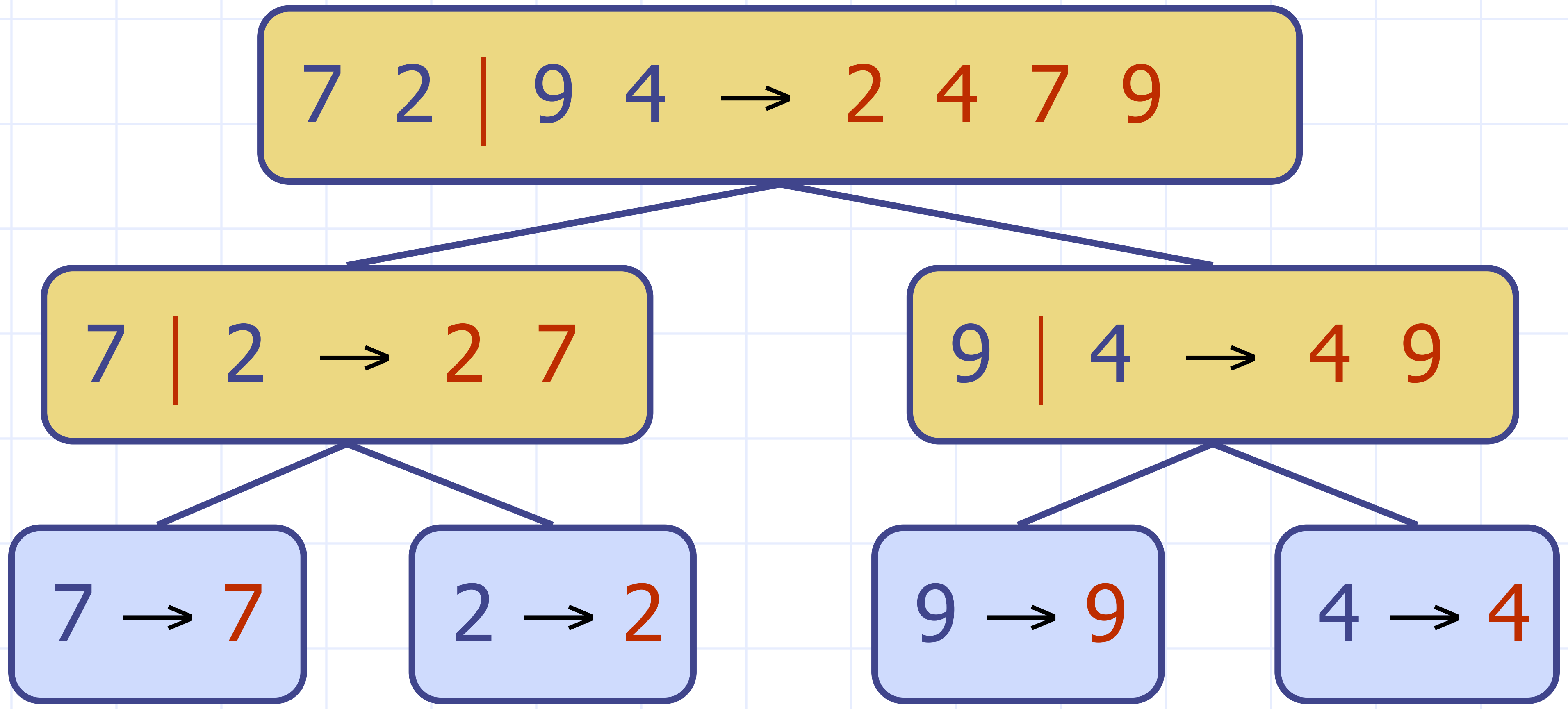
Merge Sort & Quick Sort



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
  - Recursive step: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

Algorithm *mergeSort*(S)

if $S.size() \leq 1$
  return;
else
  $(S_1, S_2) \leftarrow partition(S, S.size()/2)$
  $mergeSort(S_1)$
  $mergeSort(S_2)$
  $S \leftarrow merge(S_1, S_2)$

- This is known as a divide and conquer algorithm.

Merging Two Sorted Sequences

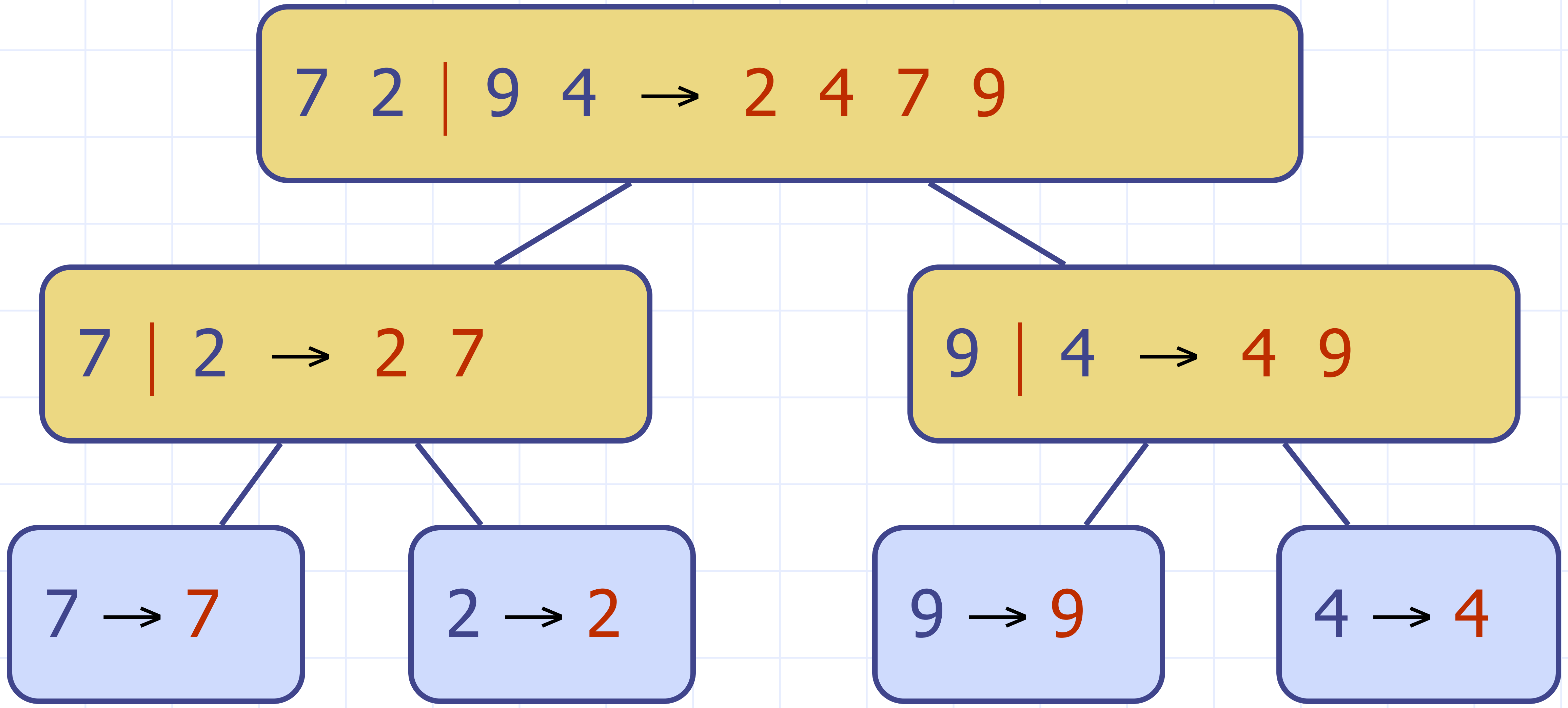
- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

```
Algorithm merge(A, B)

  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
```

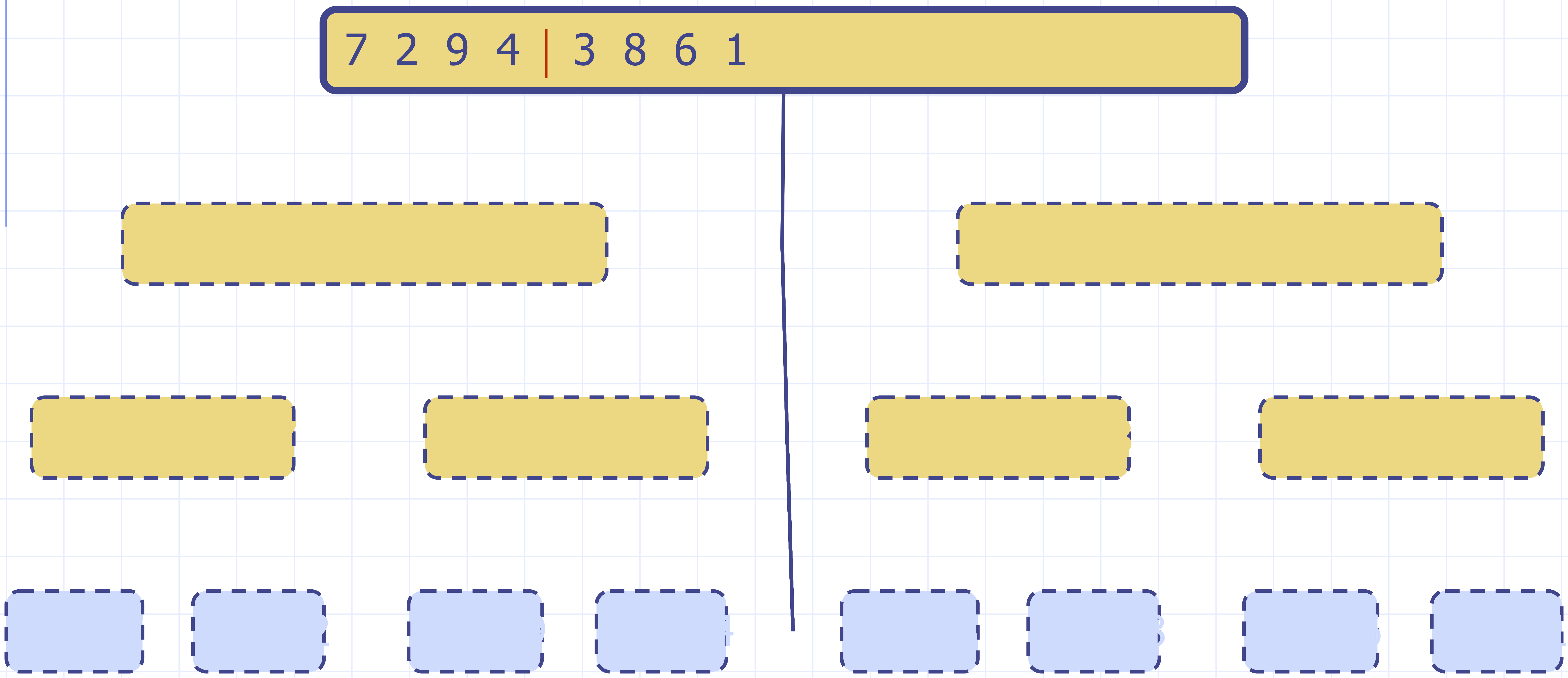
Merge-Sort Tree

- 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



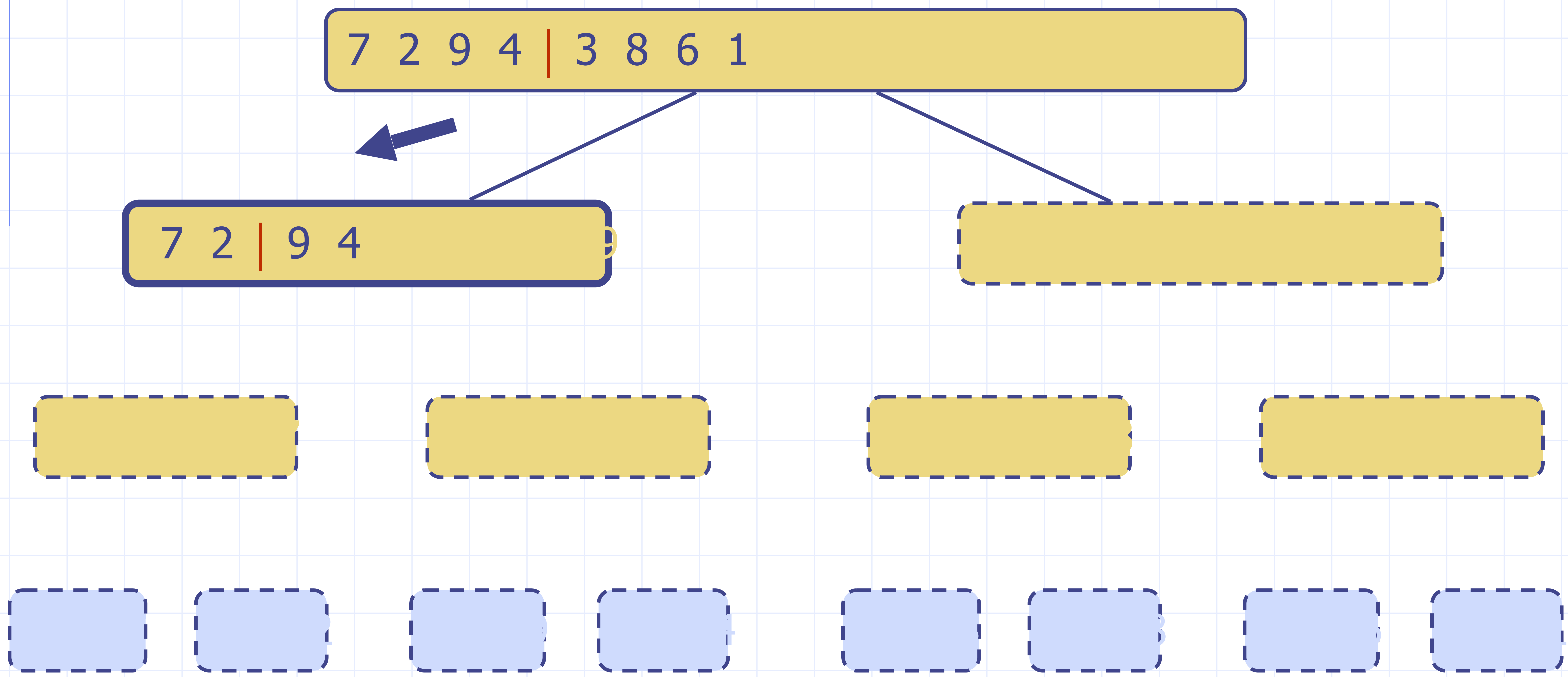
Execution Example

- Partition - recursively



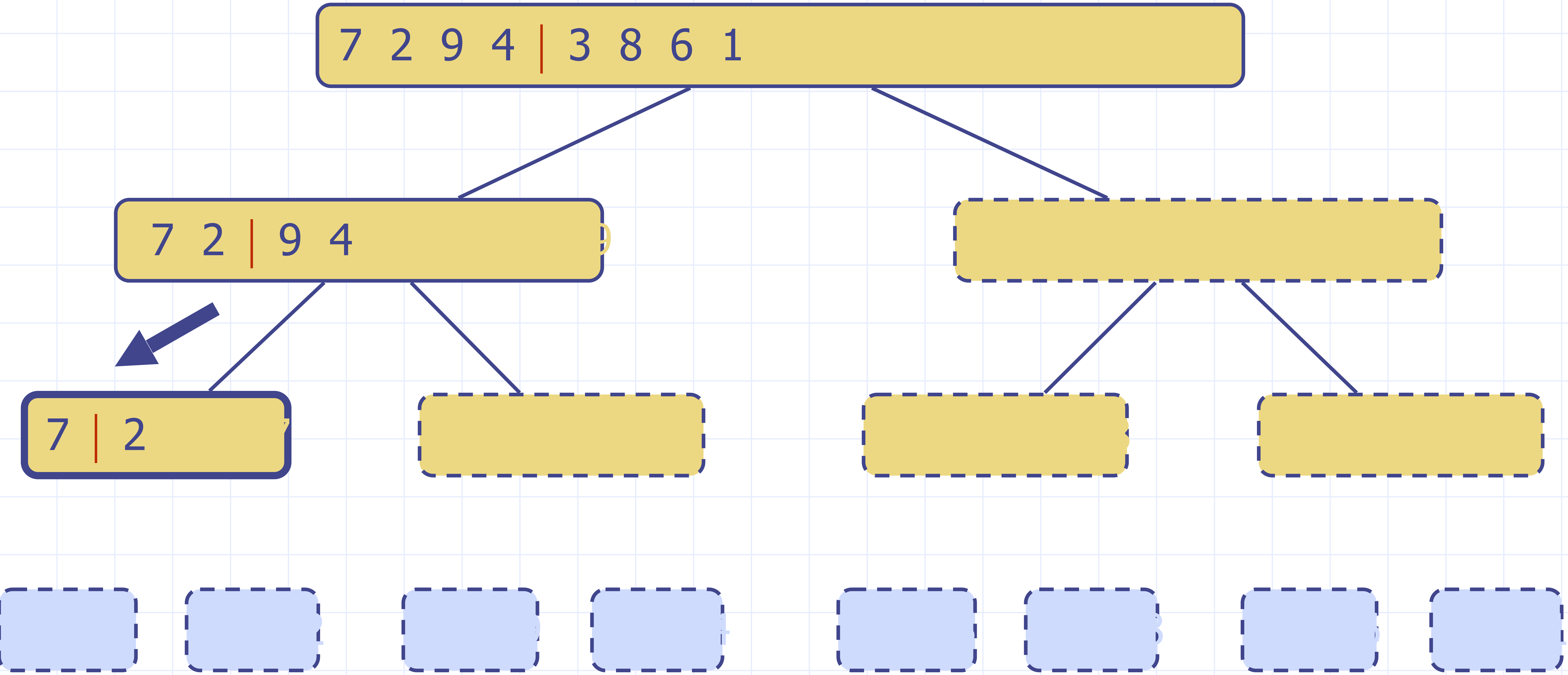
Execution Example (cont.)

- Recursive call, partition



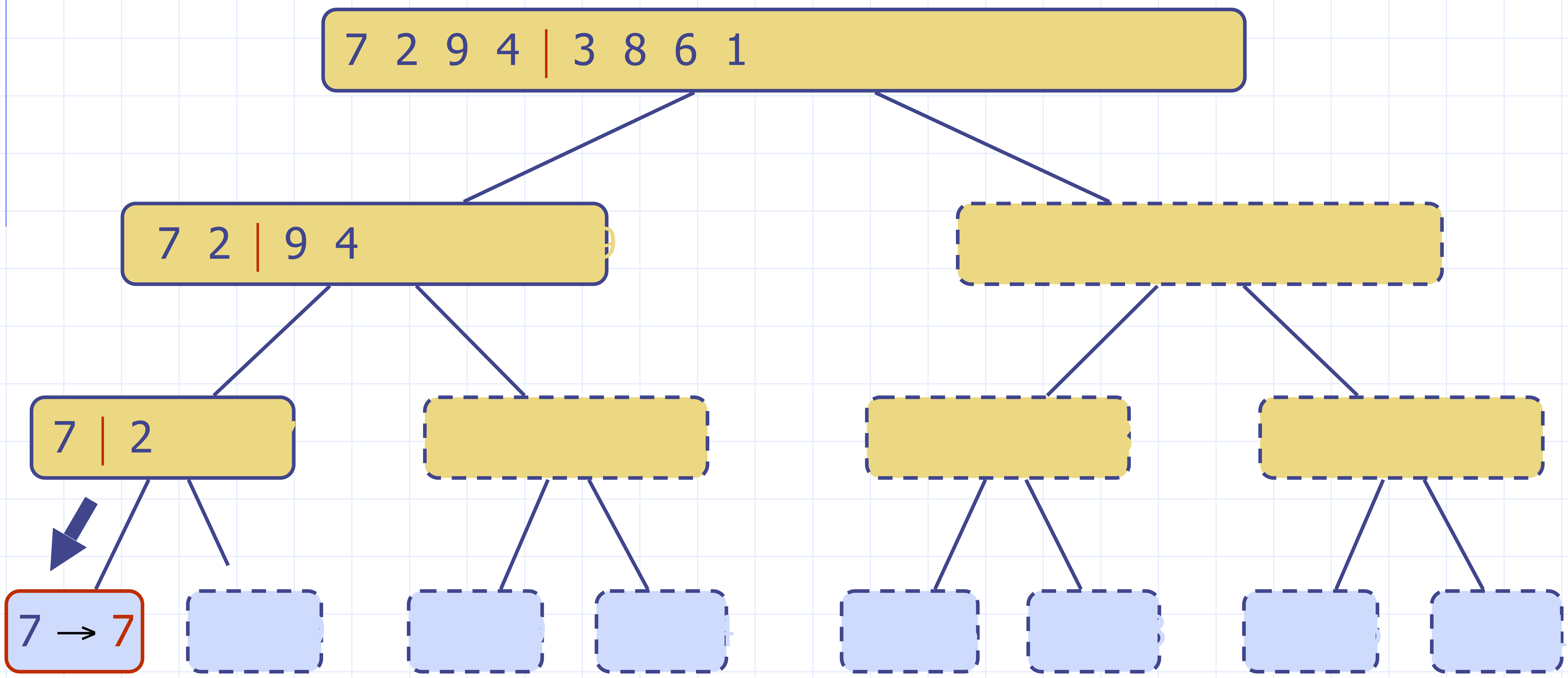
Execution Example (cont.)

- Recursive call, partition



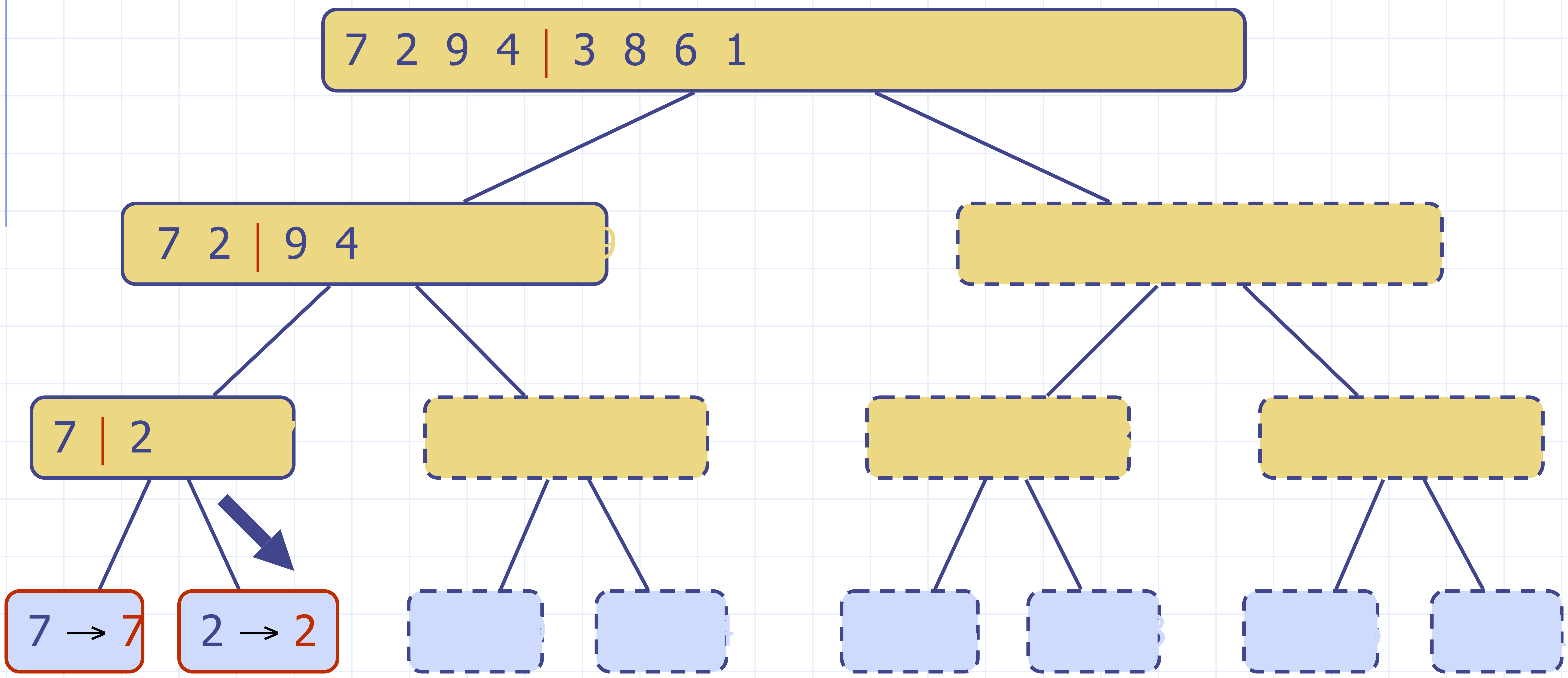
Execution Example (cont.)

- Recursive call, base case



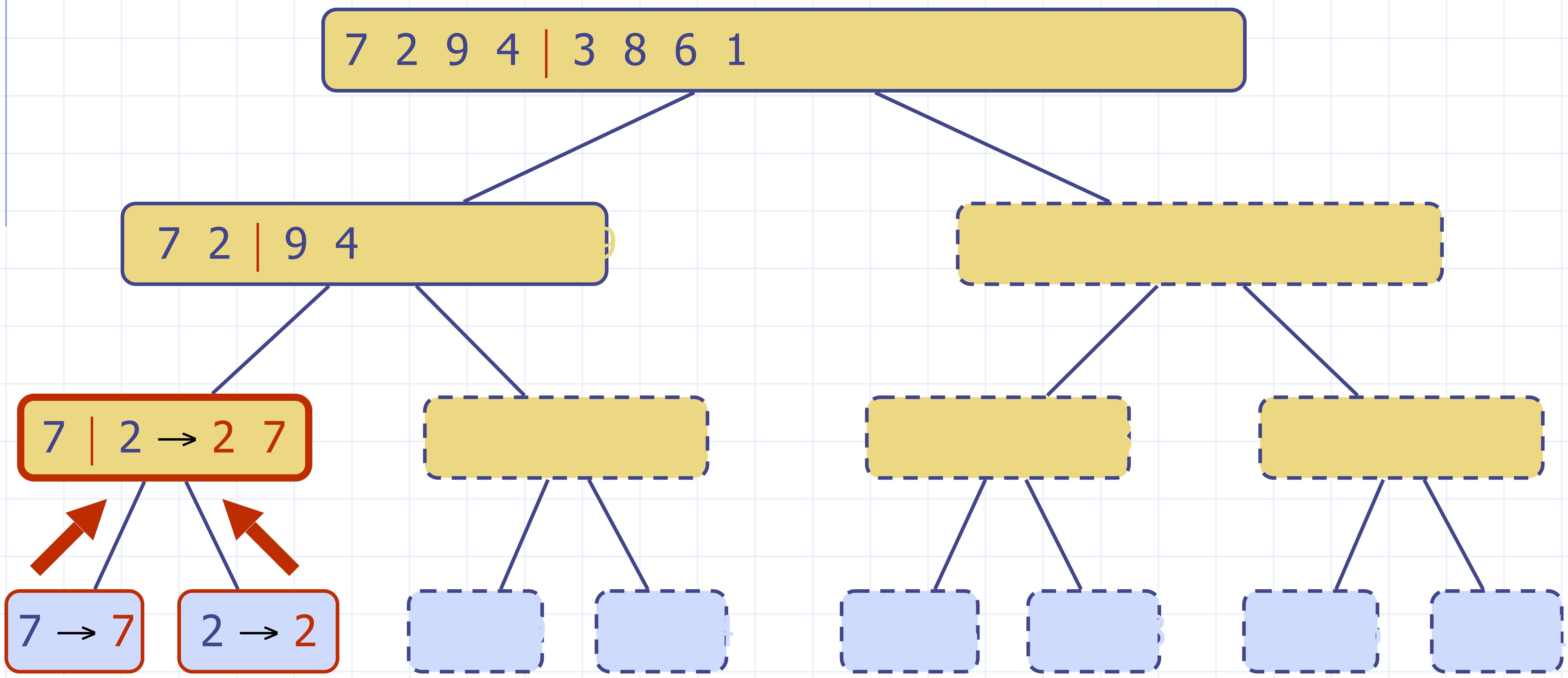
Execution Example (cont.)

- Recursive call, base case



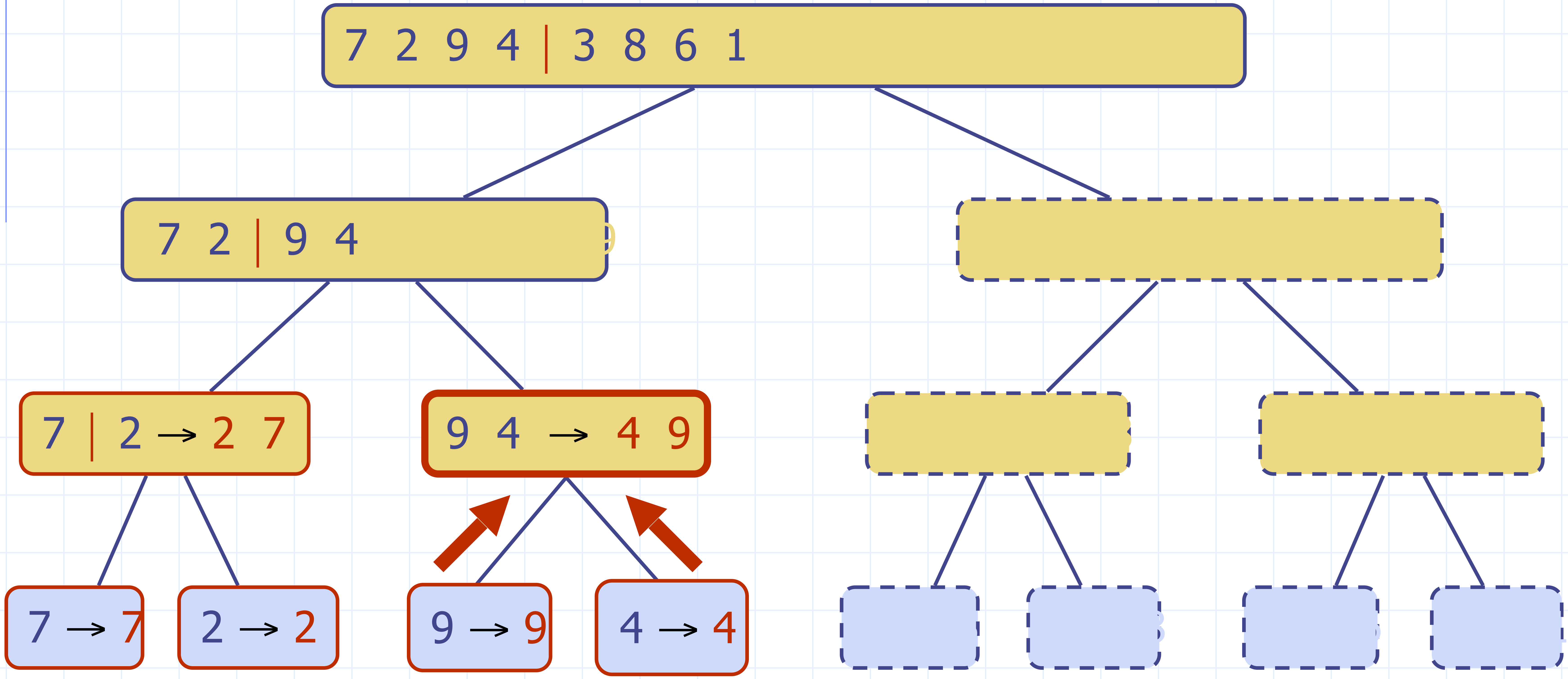
Execution Example (cont.)

- Merge



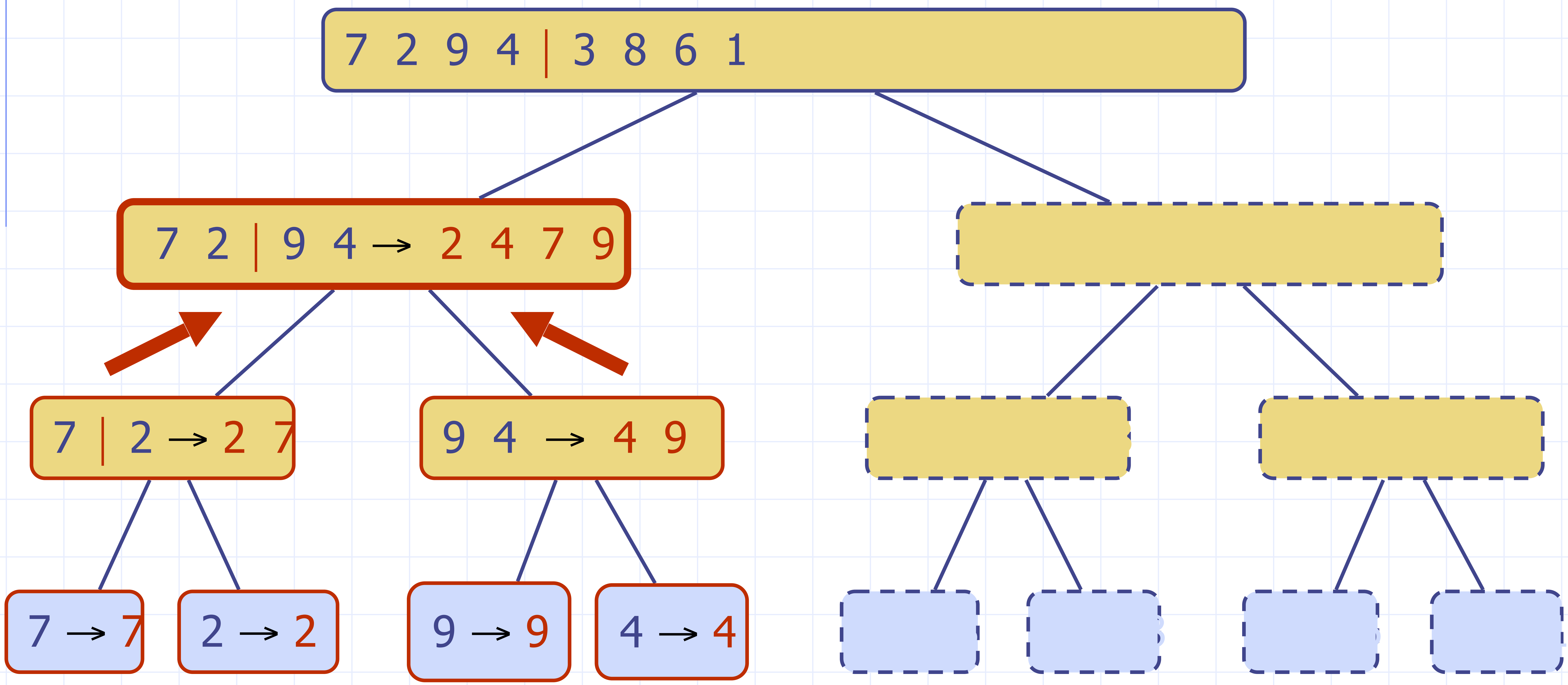
Execution Example (cont.)

- Recursive call, ..., base case, merge



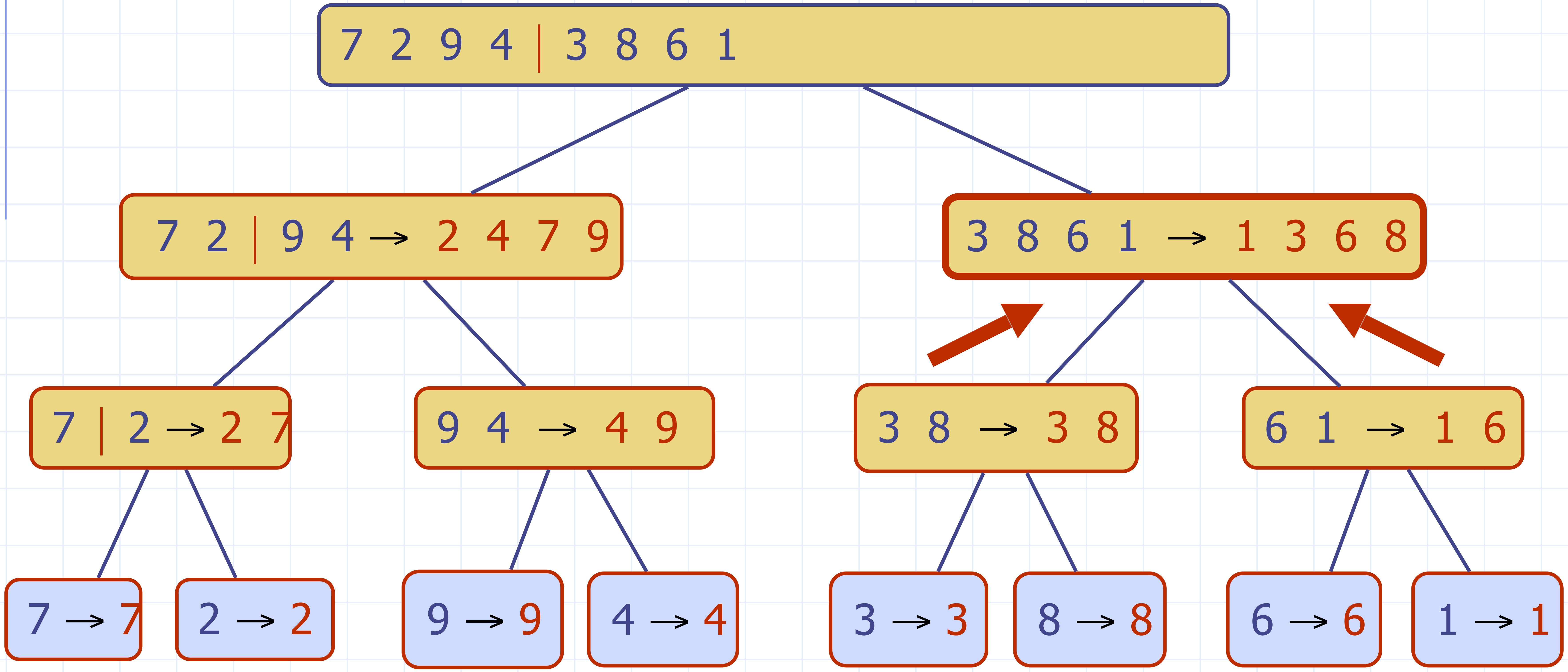
Execution Example (cont.)

- Merge



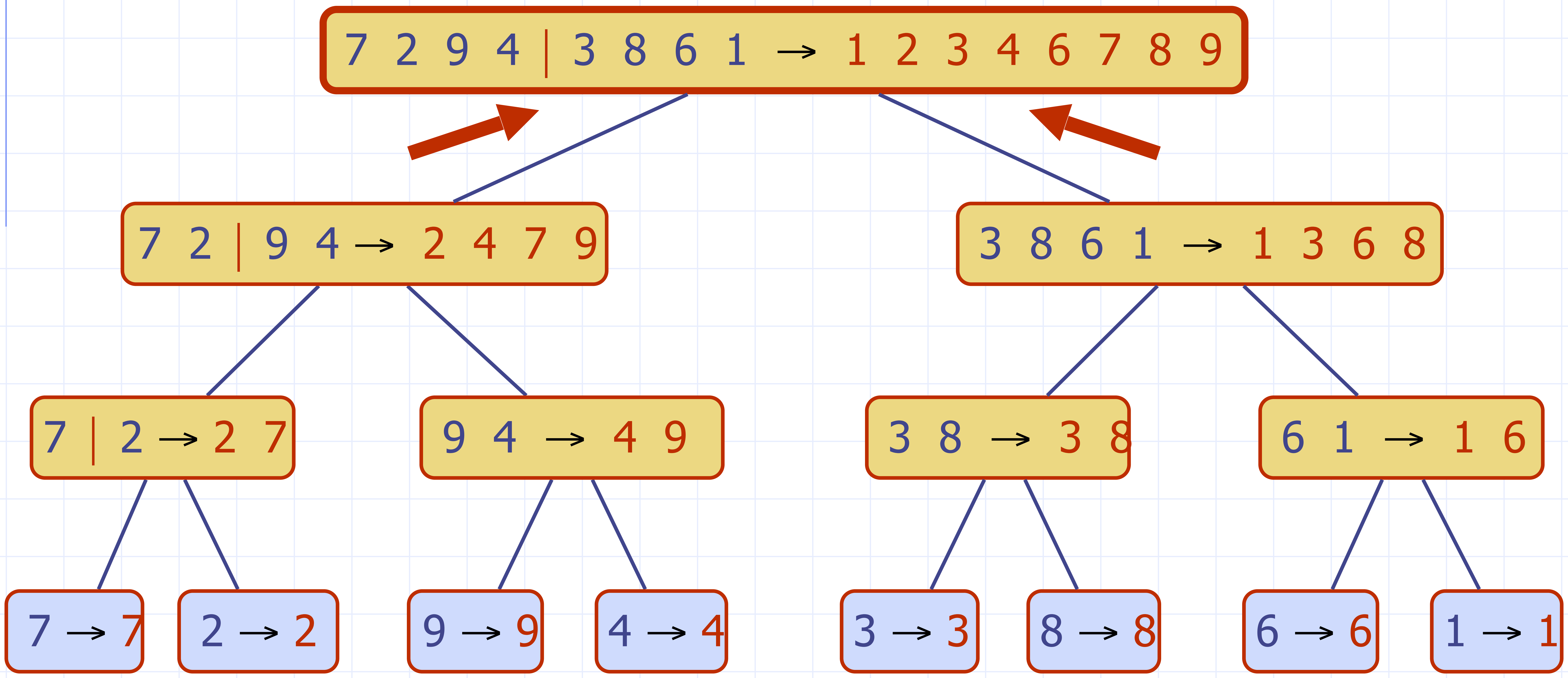
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

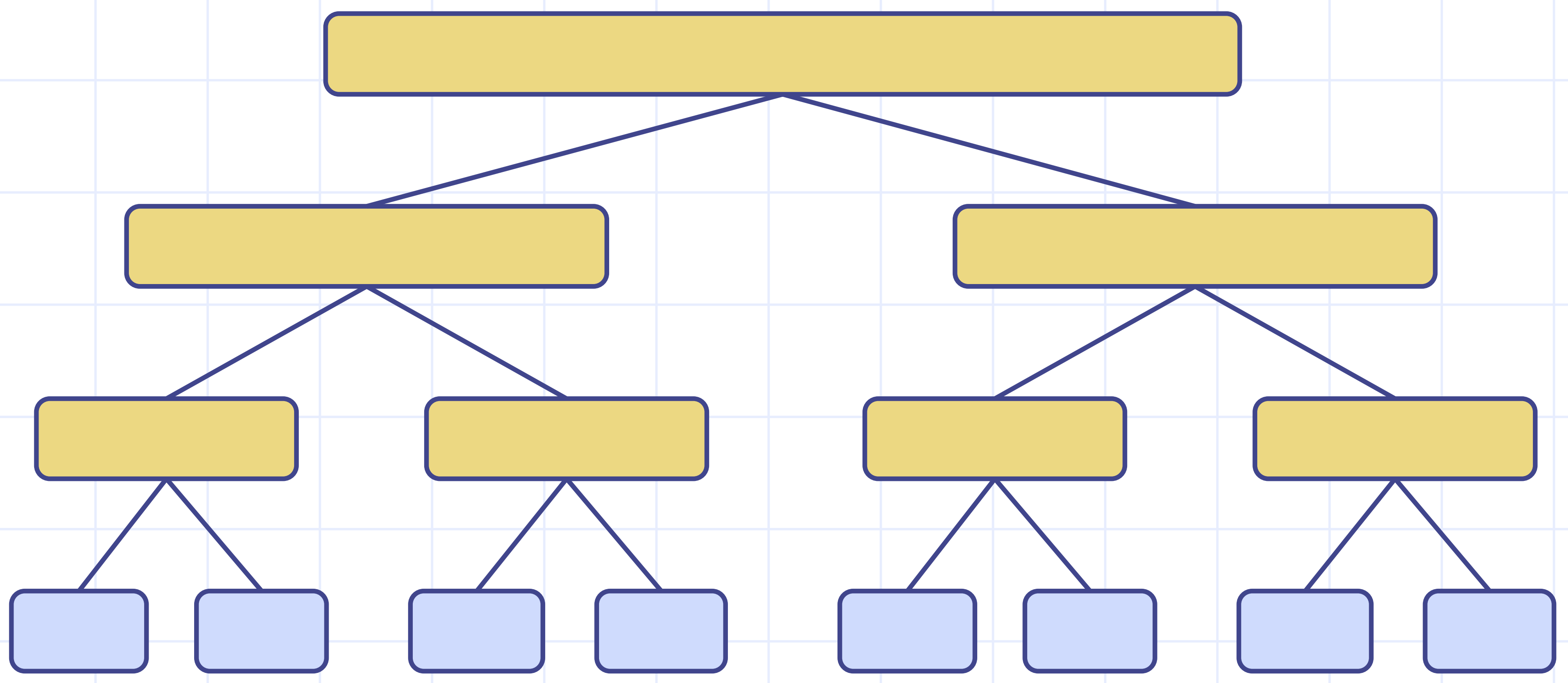
- Merge



Analysis of Merge-Sort

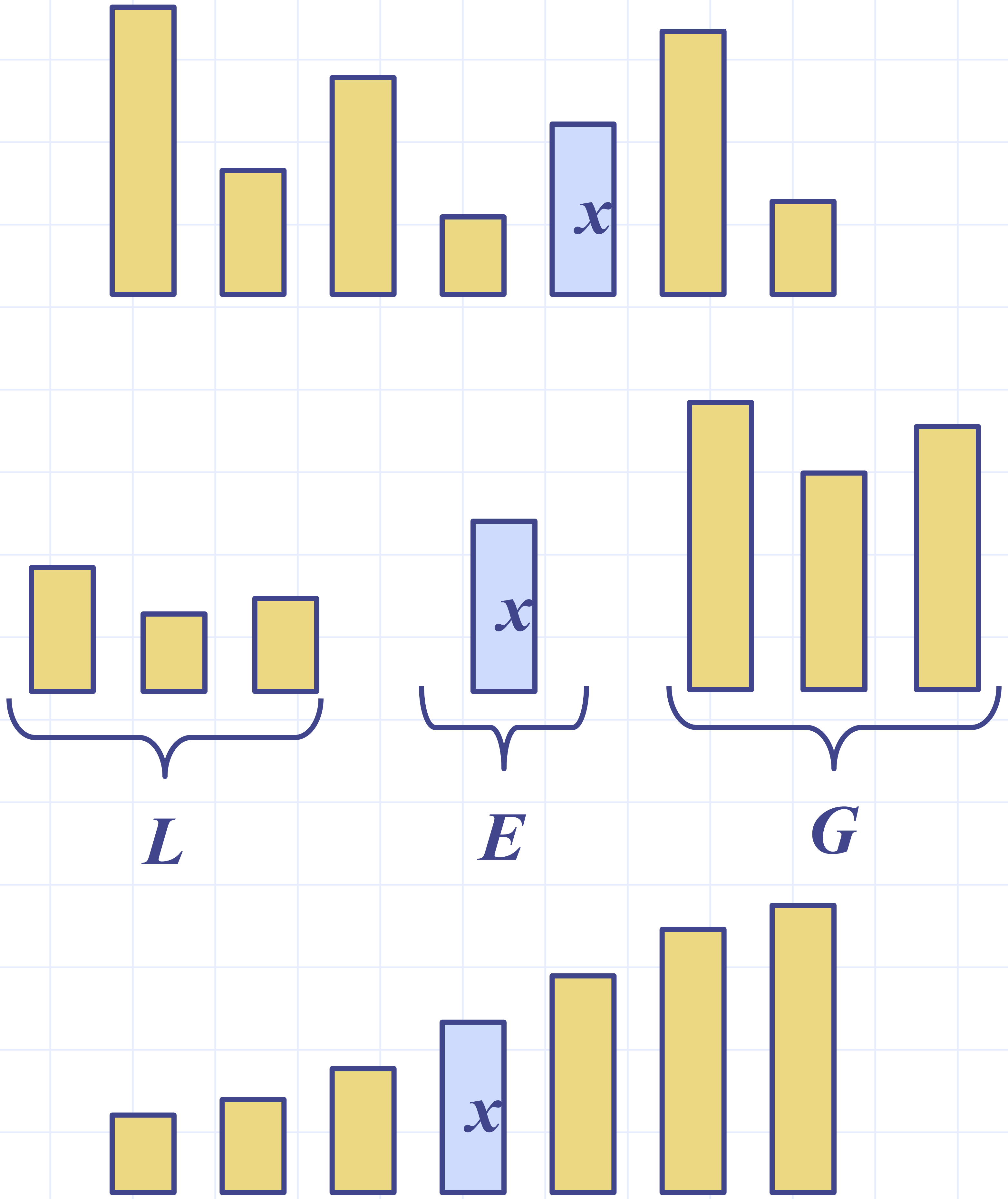
- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

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

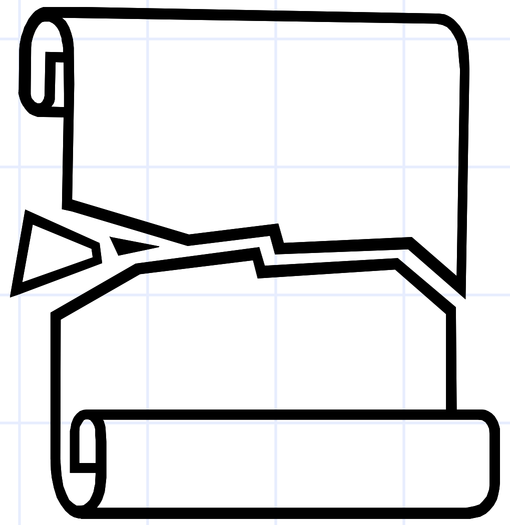


Quick-Sort

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element x (called pivot) and partition S into
    - L : elements less than x
    - E : elements equal x
    - G : elements greater than x
  - Recurse: sort L and G
  - Conquer: join L, E and G
    - ***this turns out to be trivial!***



Partition

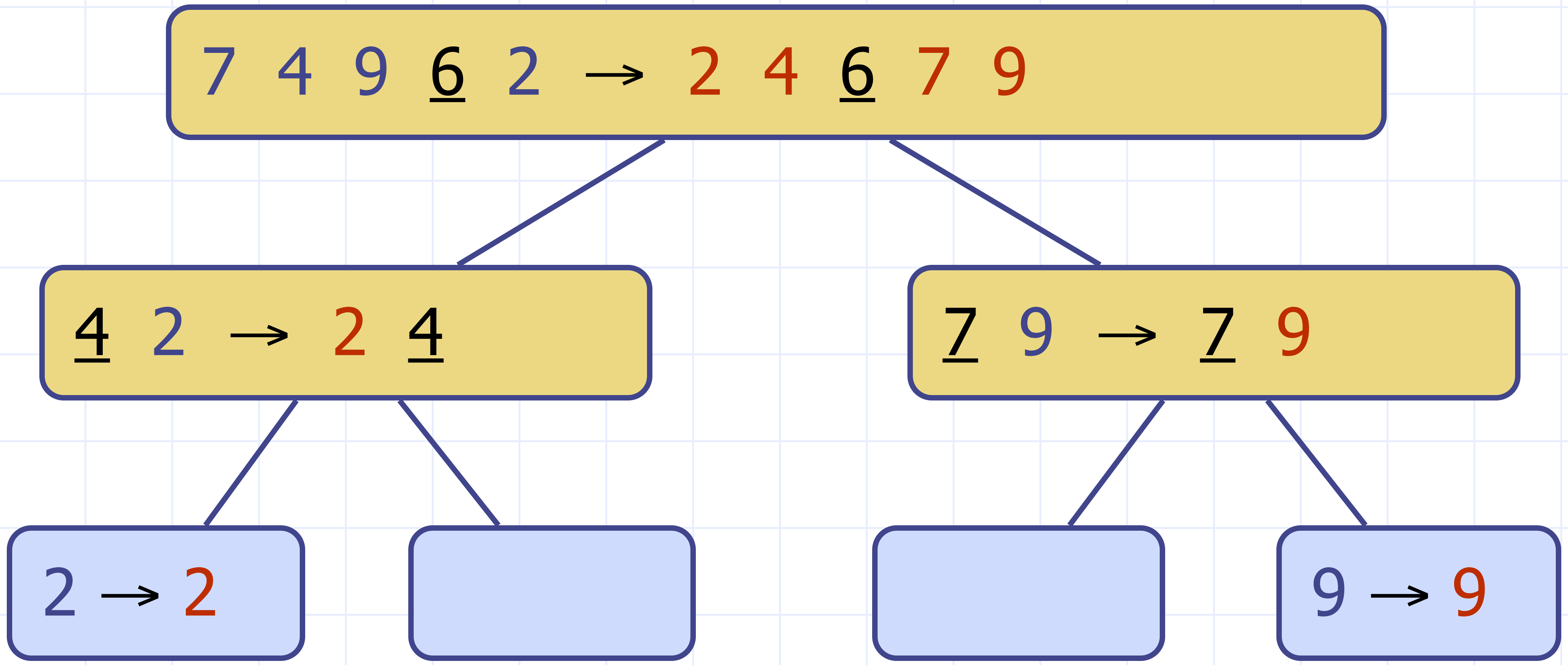


- Partition the input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot p
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

Algorithm *partition*(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the elements of S less than, equal to, or greater than the pivot, resp.

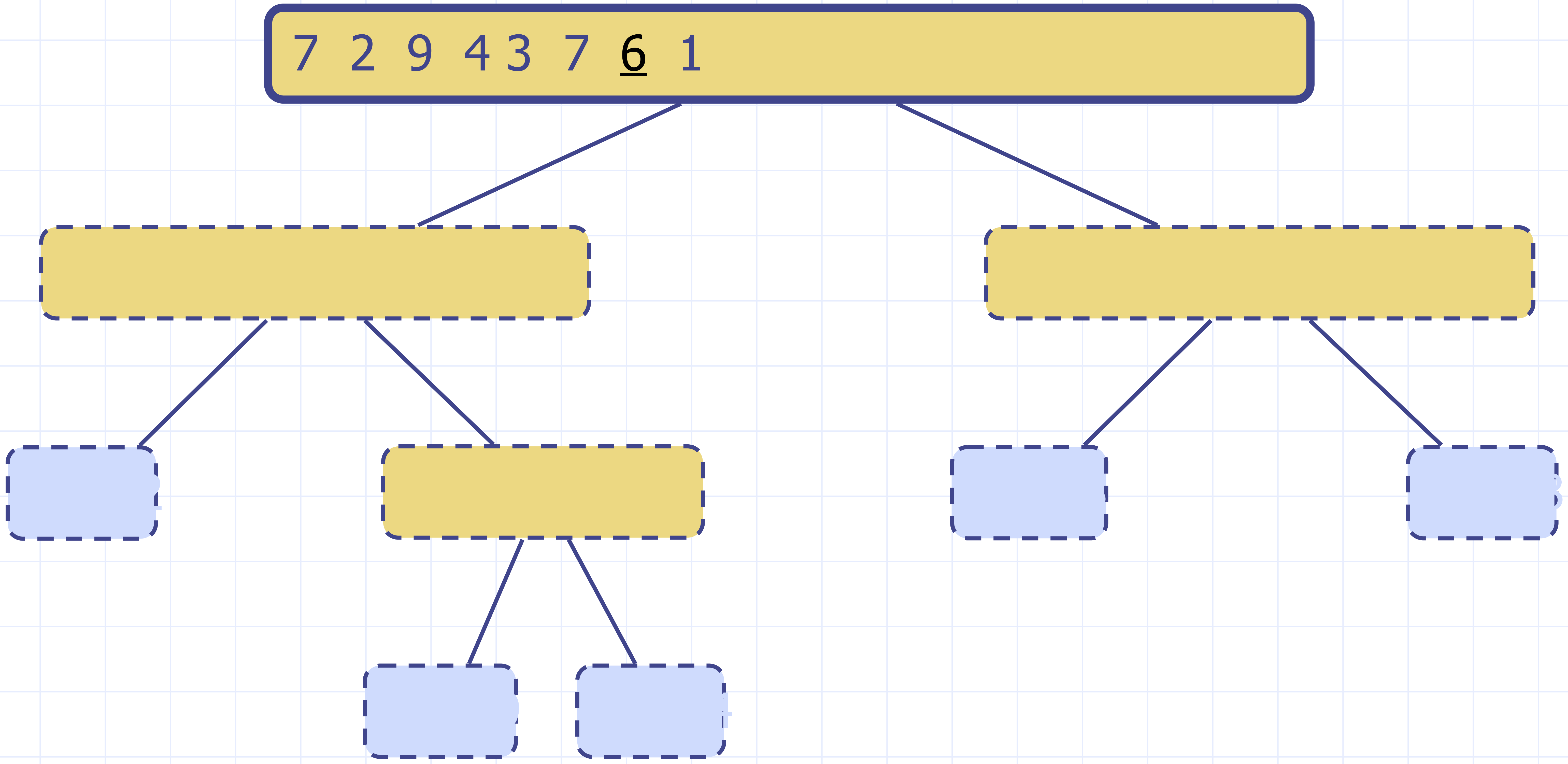
Quick-Sort Tree

- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - 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



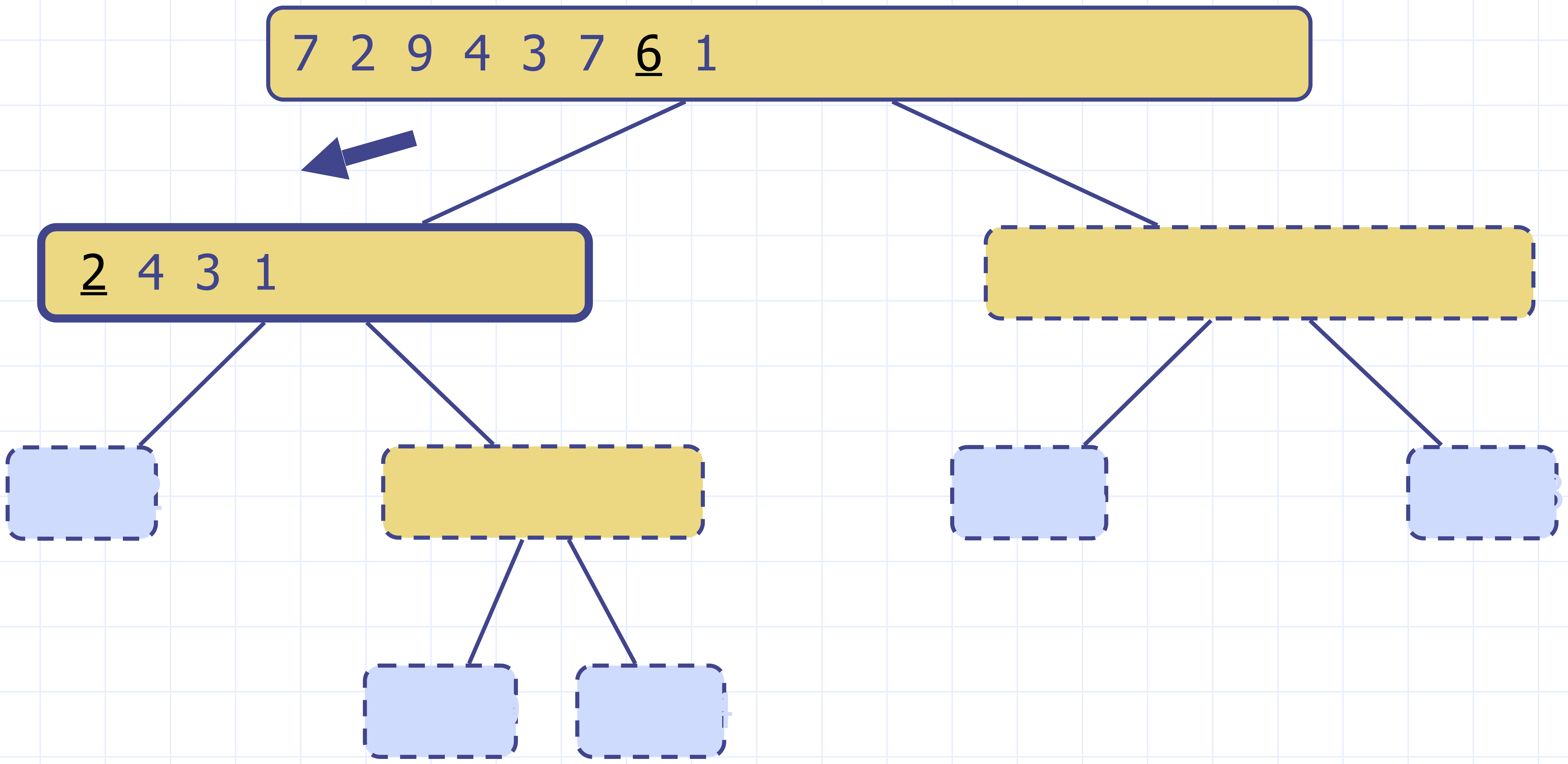
Execution Example

- Pivot selection



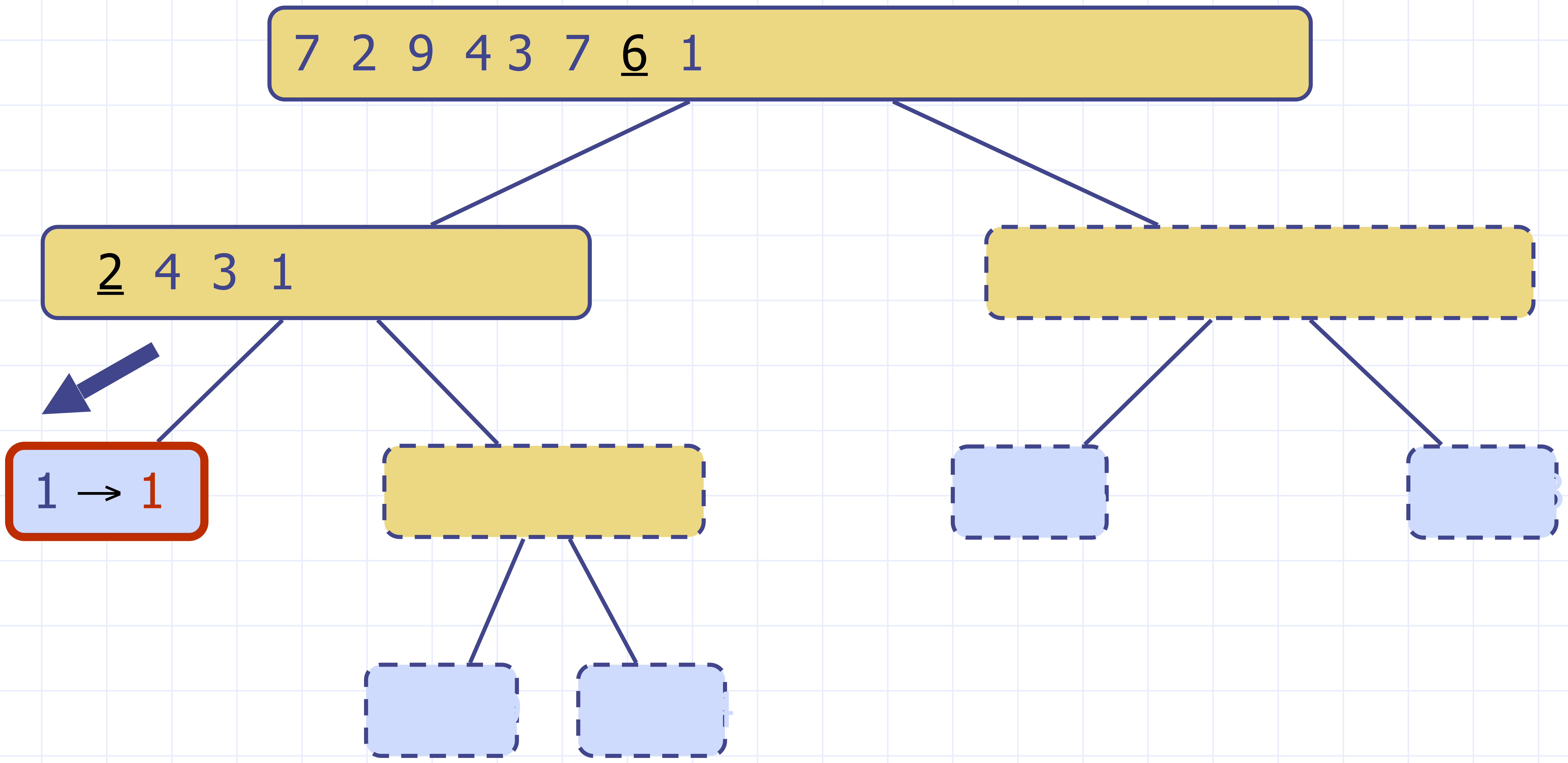
Execution Example (cont.)

- Partition, recursive call, pivot selection



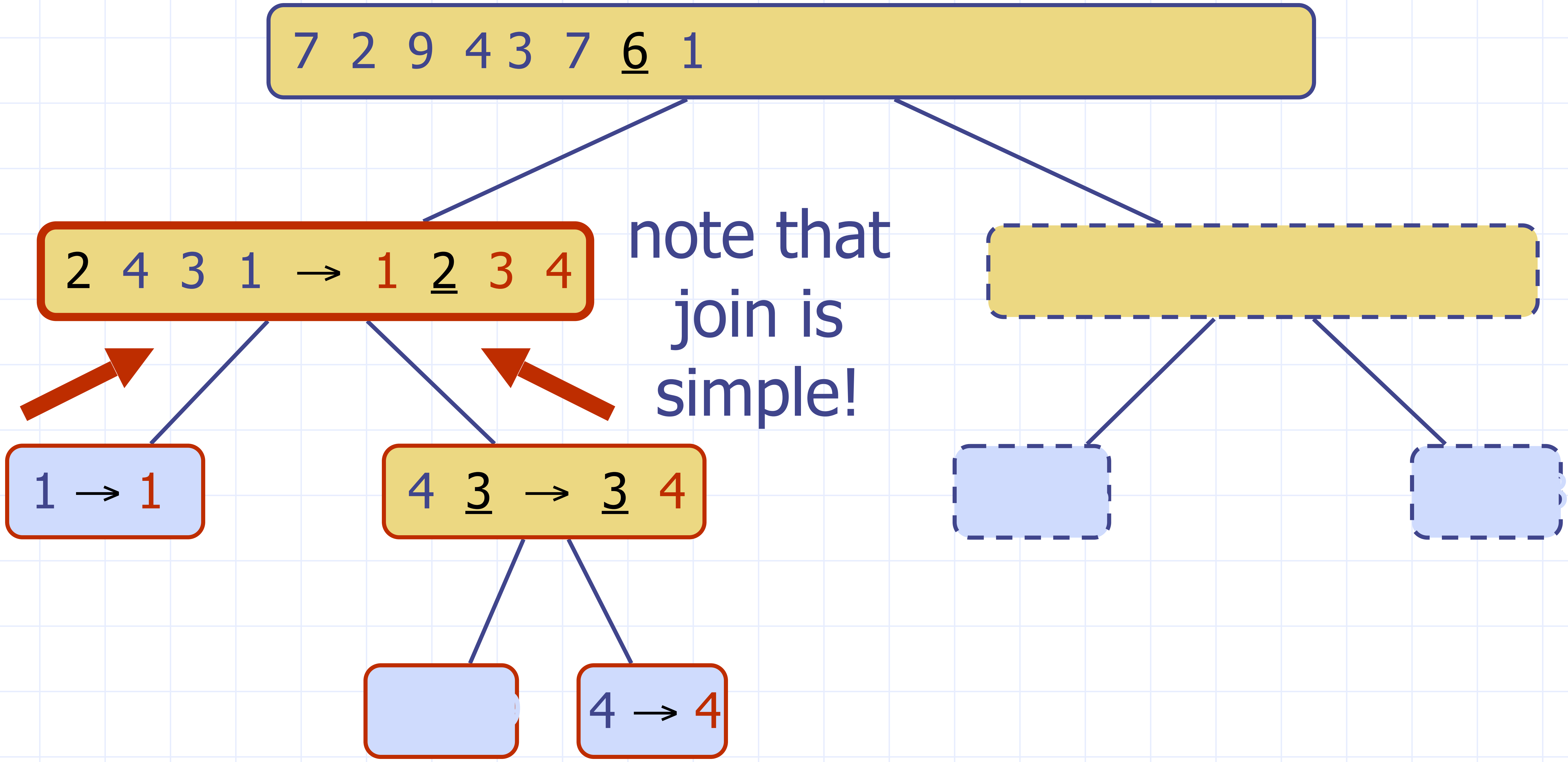
Execution Example (cont.)

- Partition, recursive call, base case



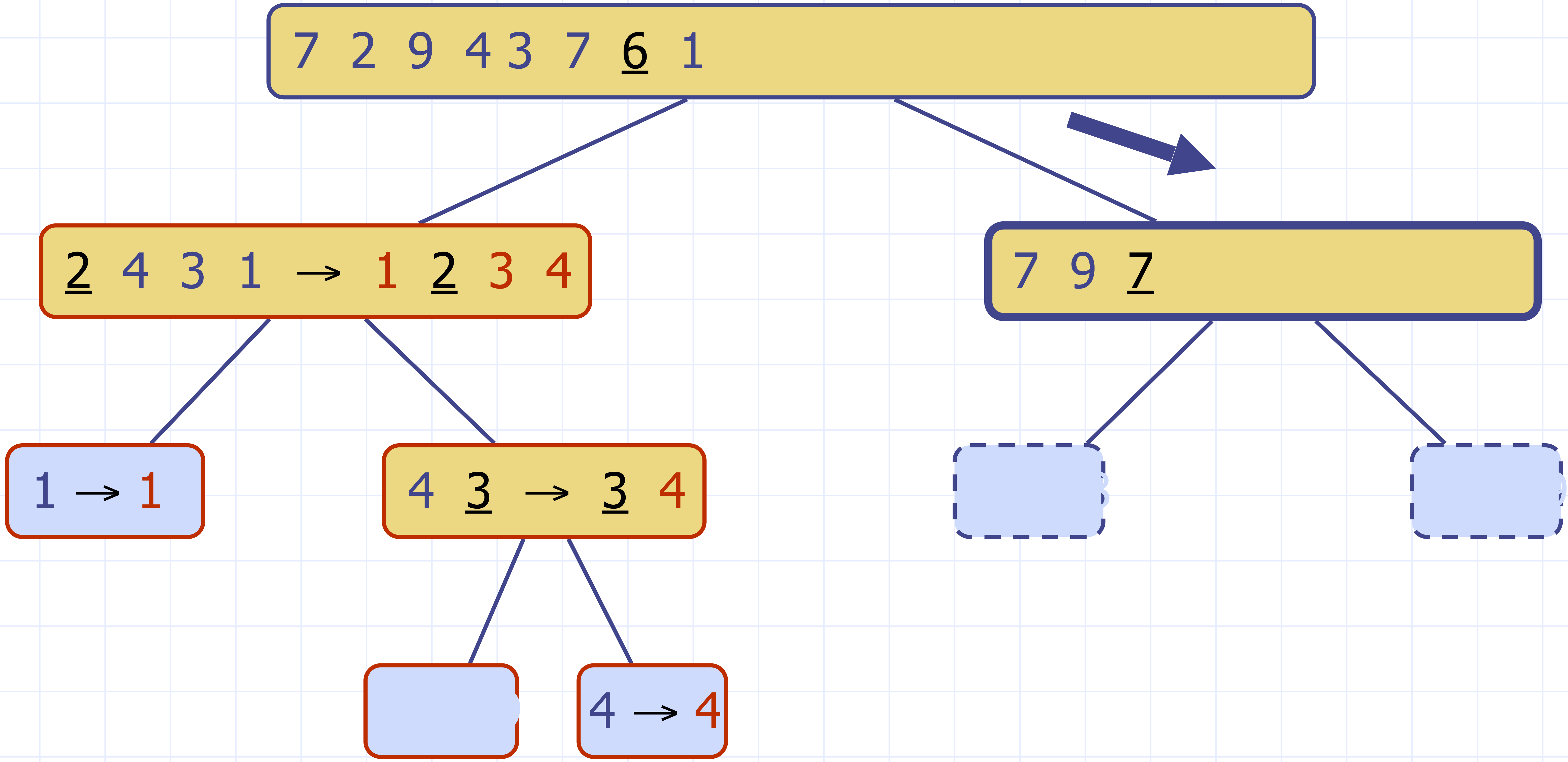
Execution Example (cont.)

- Recursive call, …, base case, join



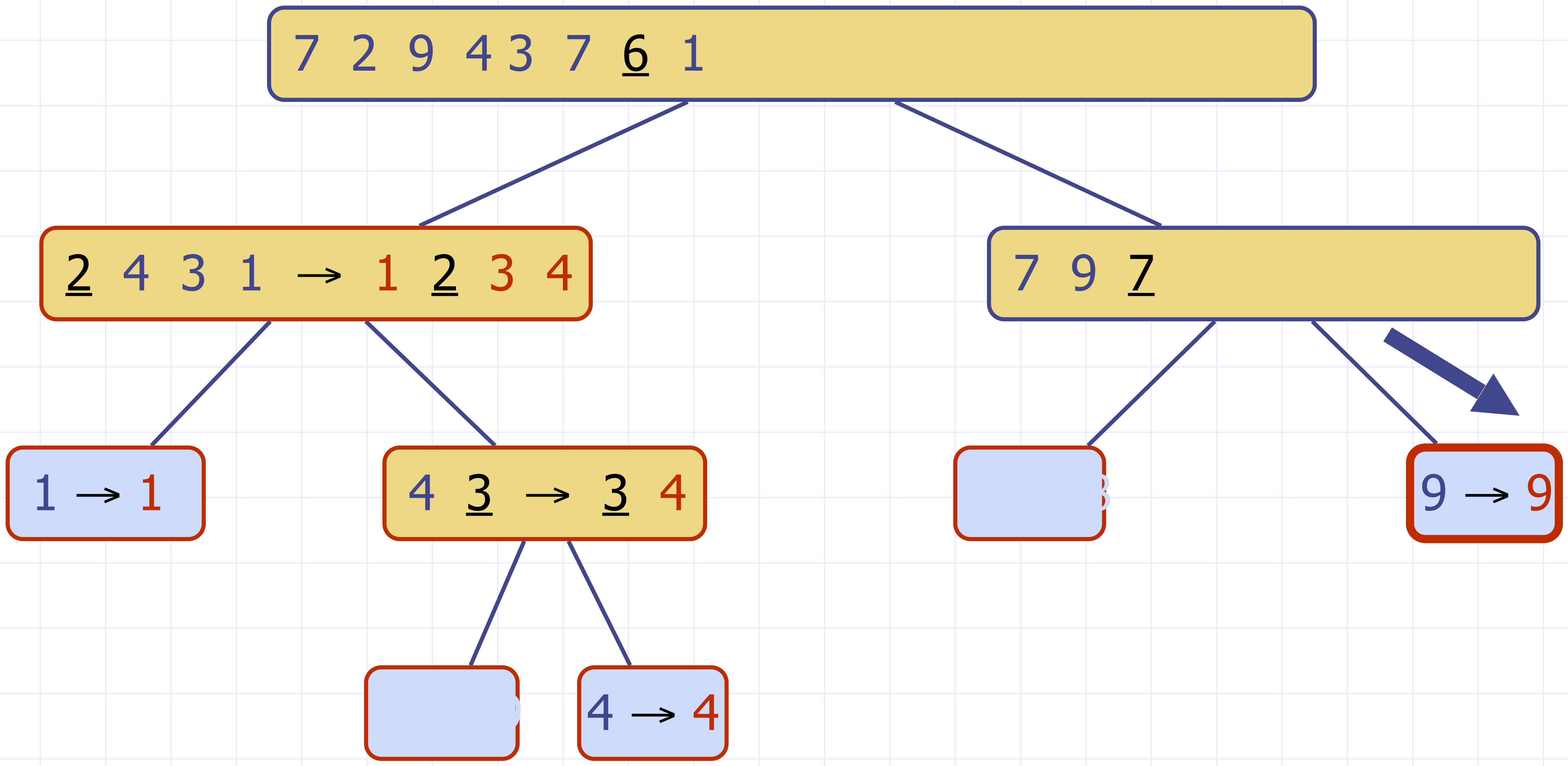
Execution Example (cont.)

- Recursive call, pivot selection



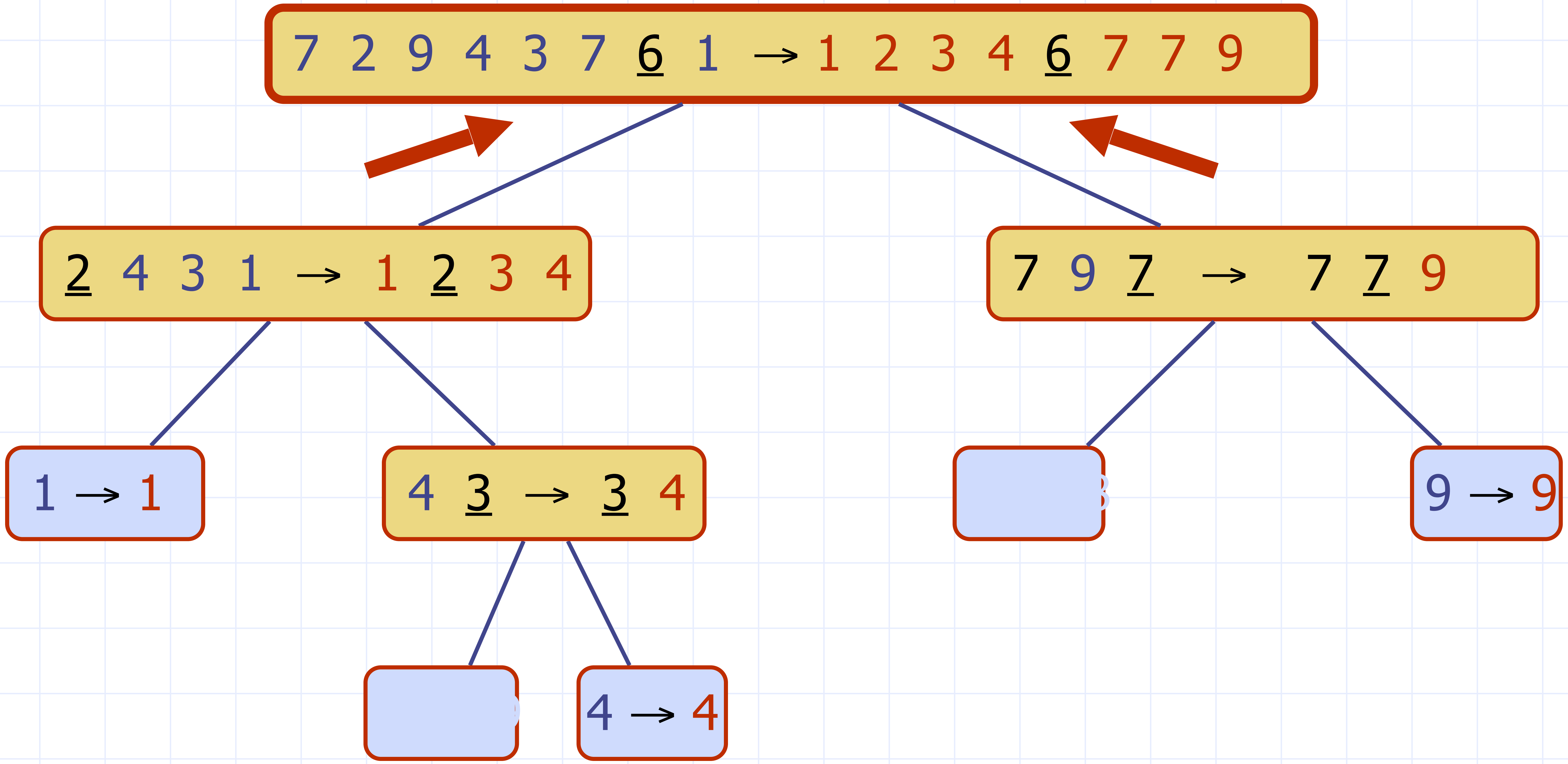
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join … again, these are easy here!



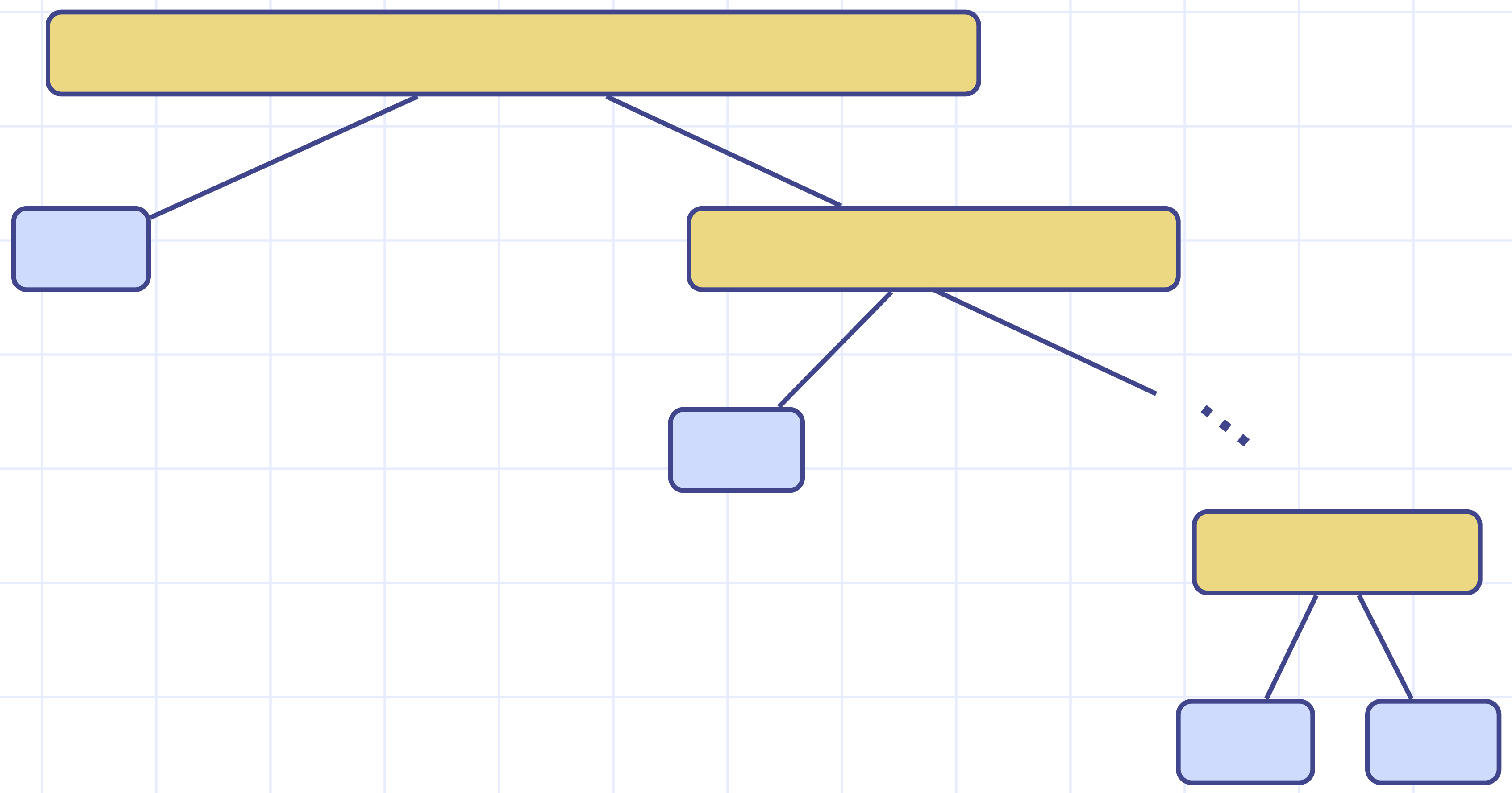
Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of L and G has size $n - 1$ and the other has size 0
- The running time is proportional to the sum

$$n + (n - 1) + \ldots + 2 + 1$$

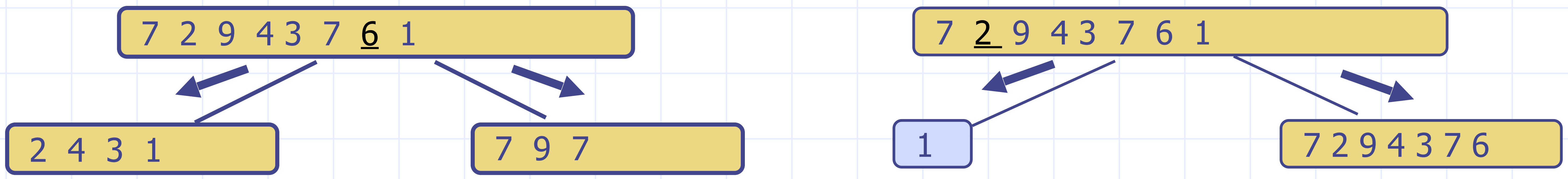
- Thus, the worst-case running time of quick-sort is $O(n^2)$

depth	time
0	n
1	$n - 1$
…	…
$n - 1$	1



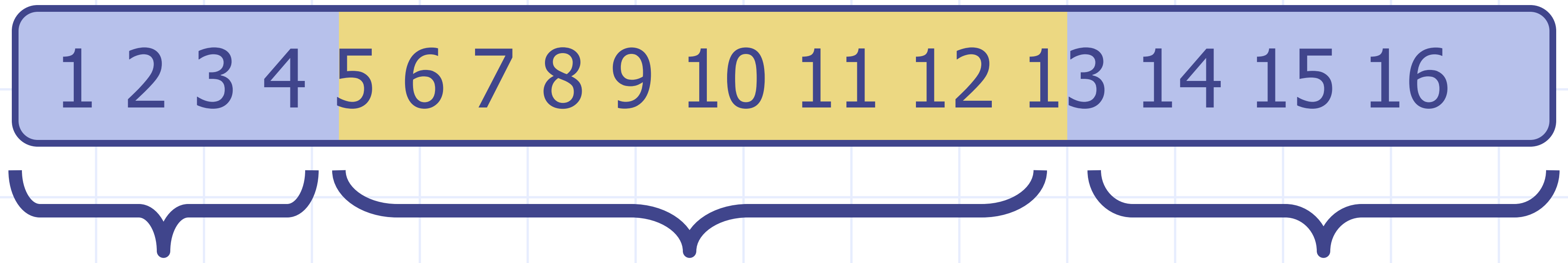
Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size s. Define:
  - **Good call:** the sizes of L and G are each less than $3s/4$
  - **Bad call:** one of L and G has size greater than $3s/4$



Good call **Bad call**

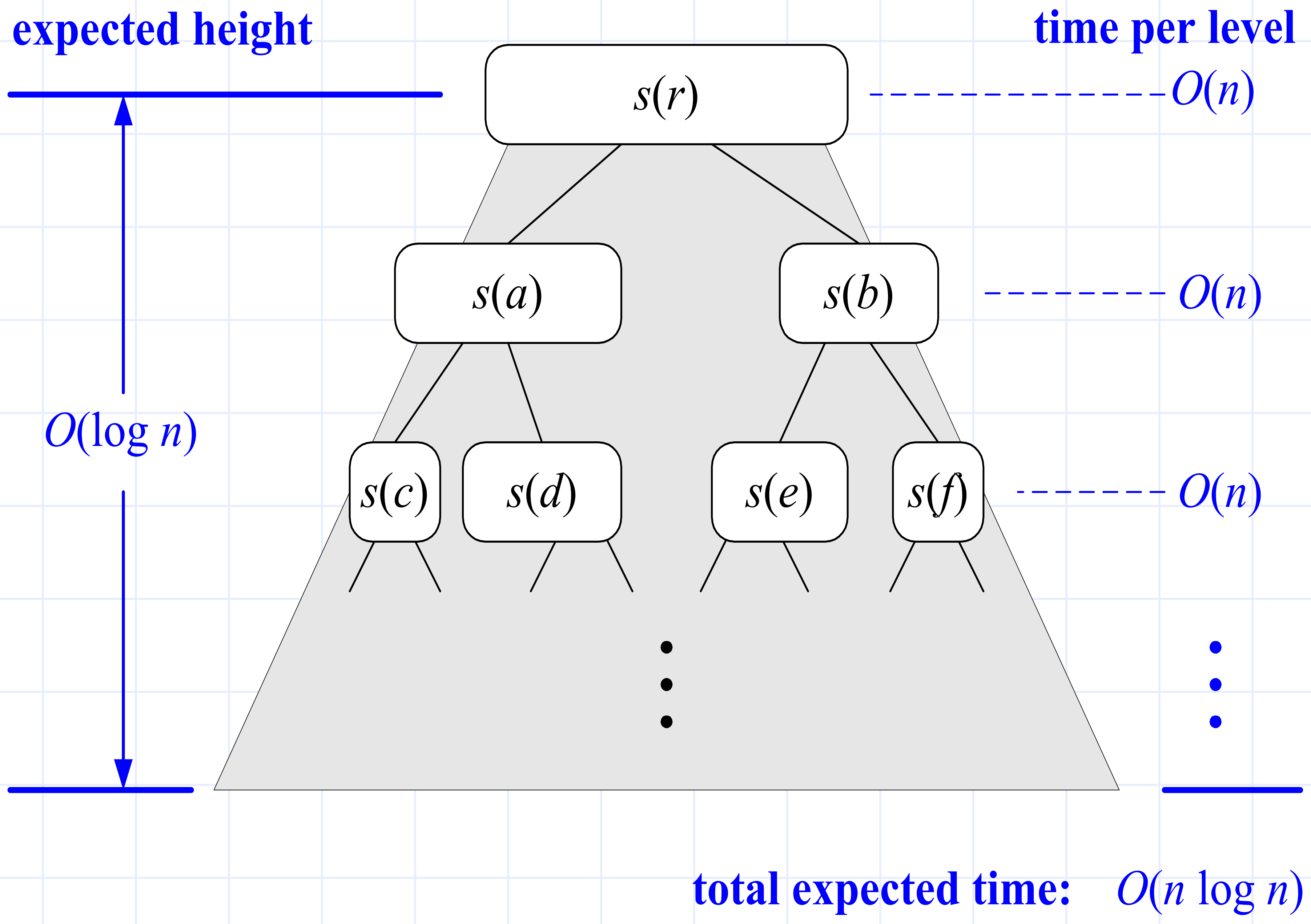
- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:



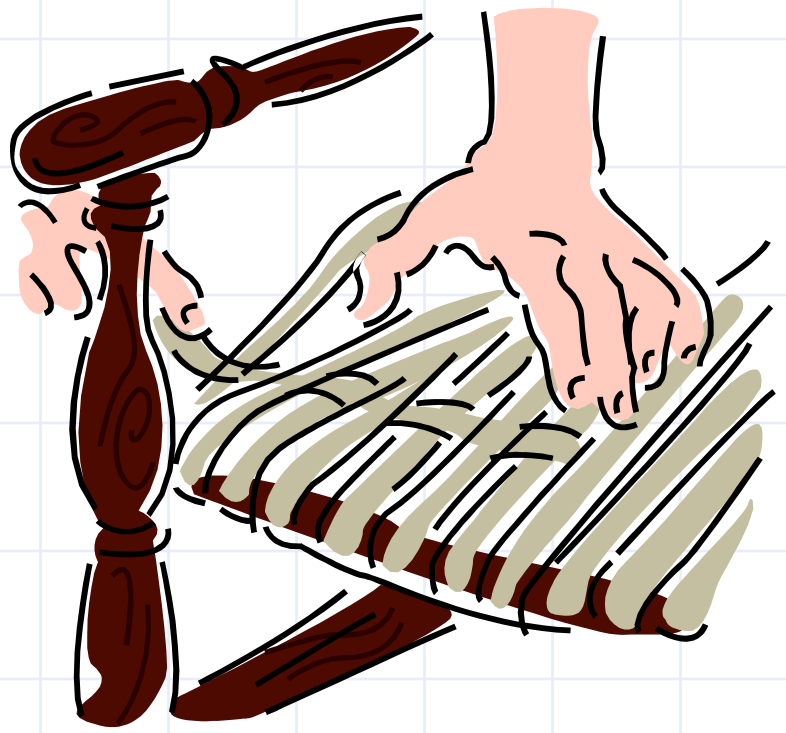
Bad pivots **Good pivots** **Bad pivots**

Expected Running Time, Part 2

- For a node of depth i, we expect
  - $i/2$ ancestors are good calls
  - The size of the input sequence for the current call is at most $(3/4)^{i/2}n$
- Therefore:
  - For a node of depth $2\log_{4/3}n$, the expected input size is one
  - The expected height of the quick-sort tree is $O(\log n)$
- The amount or work done at the nodes of the same depth is $O(n)$
- Thus, the expected running time of quick-sort is $O(n \log n)$



In-Place Quick-Sort



- Quick-sort can be implemented to run “in-place”
- In the partition step, we use replace operations to rearrange the elements of the input sequence such that:
  - the elements less than the pivot have index less than h
  - the elements equal to the pivot have index between h and k
  - the elements greater than the pivot have index greater than k
- The recursive calls consider
  - elements with index less than h
  - elements with index greater than k

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
bubble-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
quick-sort	$O(n \log n)$ ***expected***	▪ in-place, randomized ▪ <u>usually</u> fast (good for large inputs)
merge-sort	$O(n \log n)$	▪ needs extra storage ▪ fast (good for huge inputs)
heap-sort (next lecture)	$O(n \log n)$	▪ in-place ▪ fast (good for large inputs)