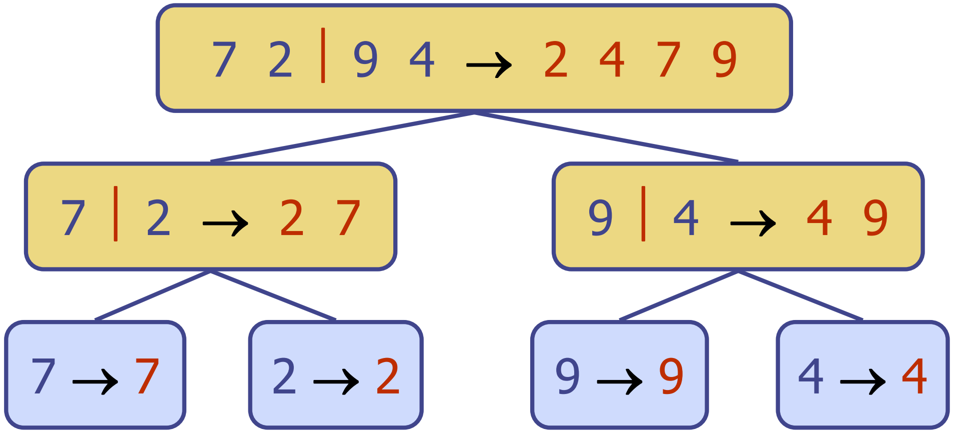
G52ACE 2017-18

Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

Merge Sort & Quicksort



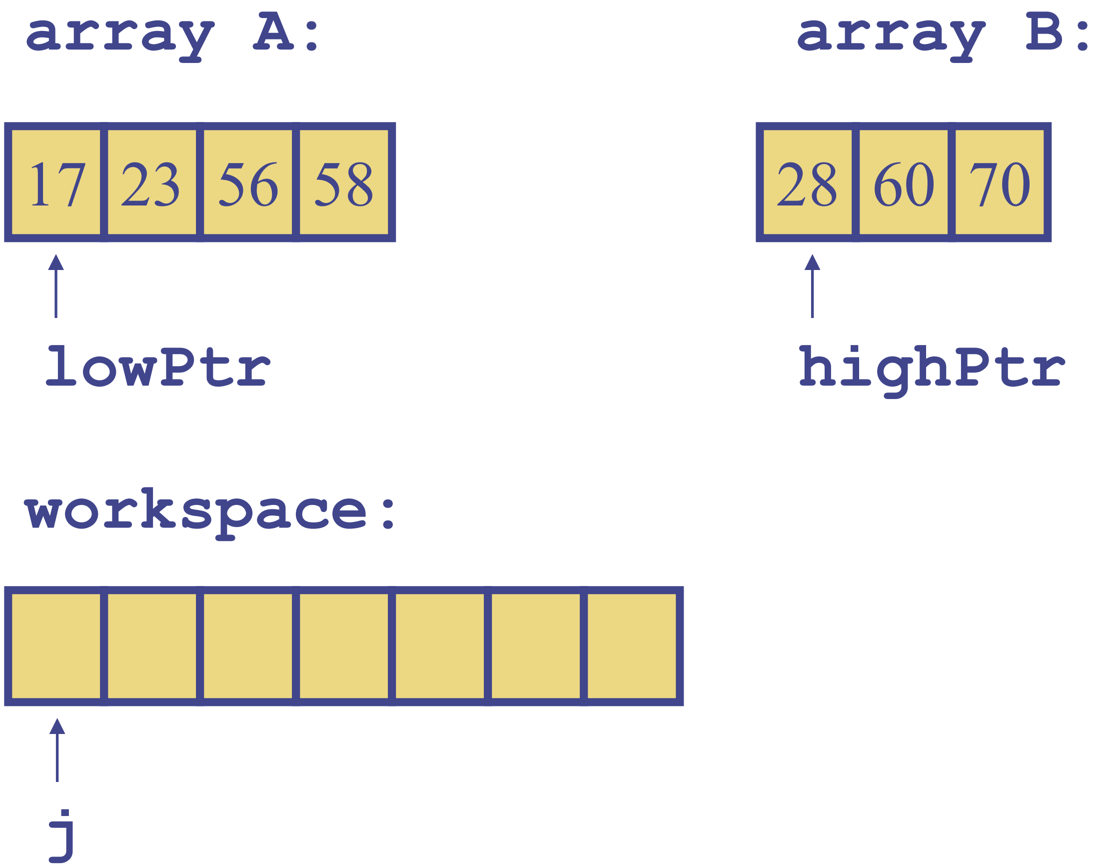
Divide-and-Conquer

- 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 (or "small enough to be done directly")
- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm

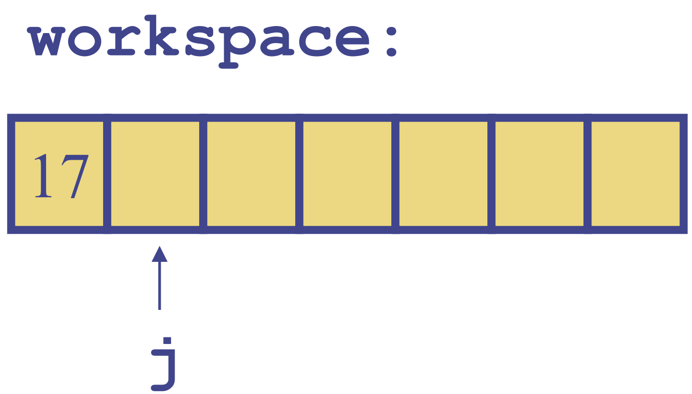
Merge-Sort

- Merge-sort on an input sequence $\boldsymbol{S}$ with $\boldsymbol{n}$ elements consists of three steps:
  - Divide: partition $\boldsymbol{S}$ into two sequences $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ of about $\boldsymbol{n}/2$ elements each
  - Recur: recursively sort $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$
  - Conquer: merge $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ into a unique sorted sequence
- First questions:
  Is the merge easy?
  What is the big-Oh of the merge?

Merging sorted arrays

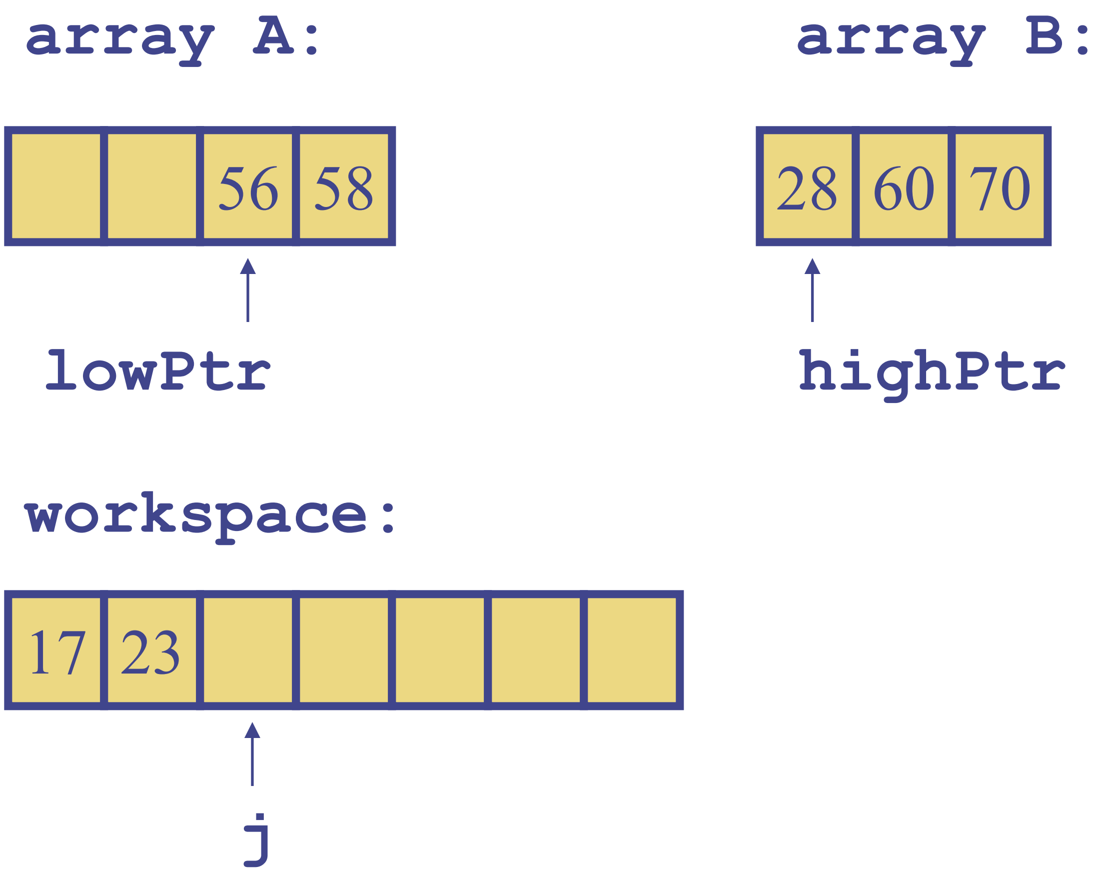


Merging sorted arrays

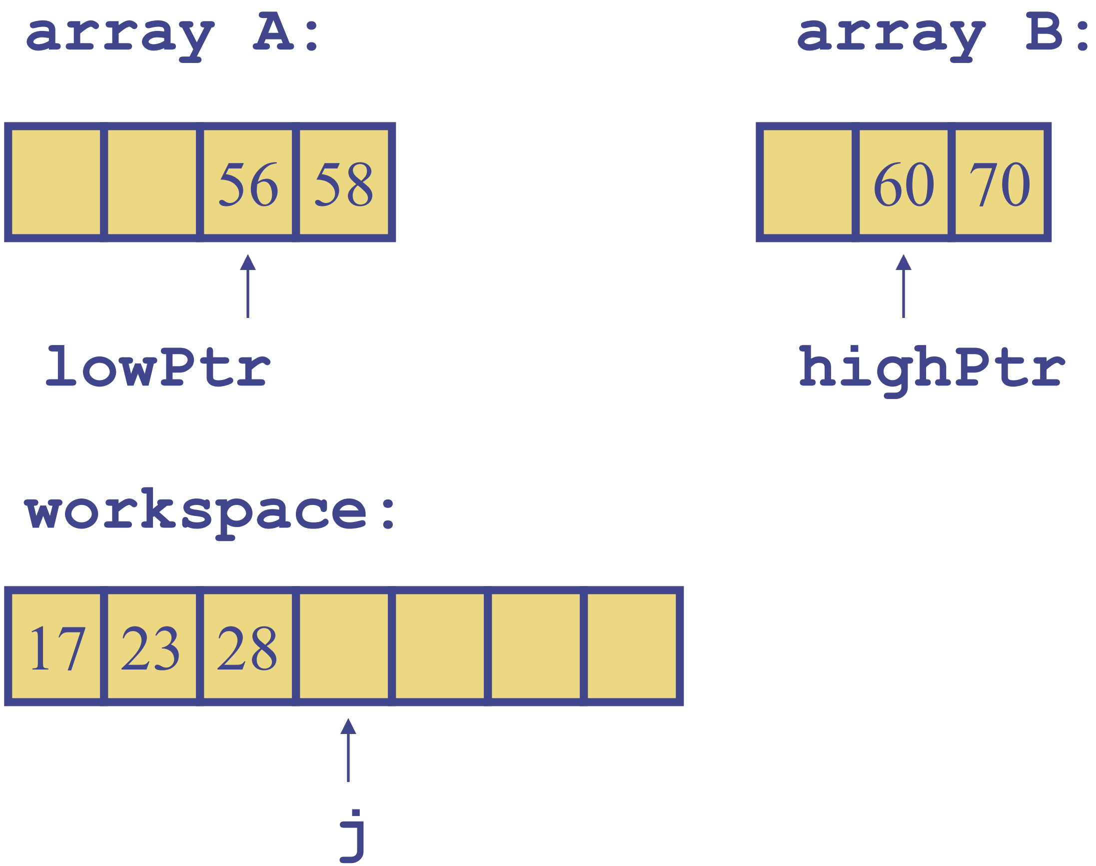


Notice the choice of 17 is only efficient and easy because the inputs are sorted and so we know we only need look at the first element!!

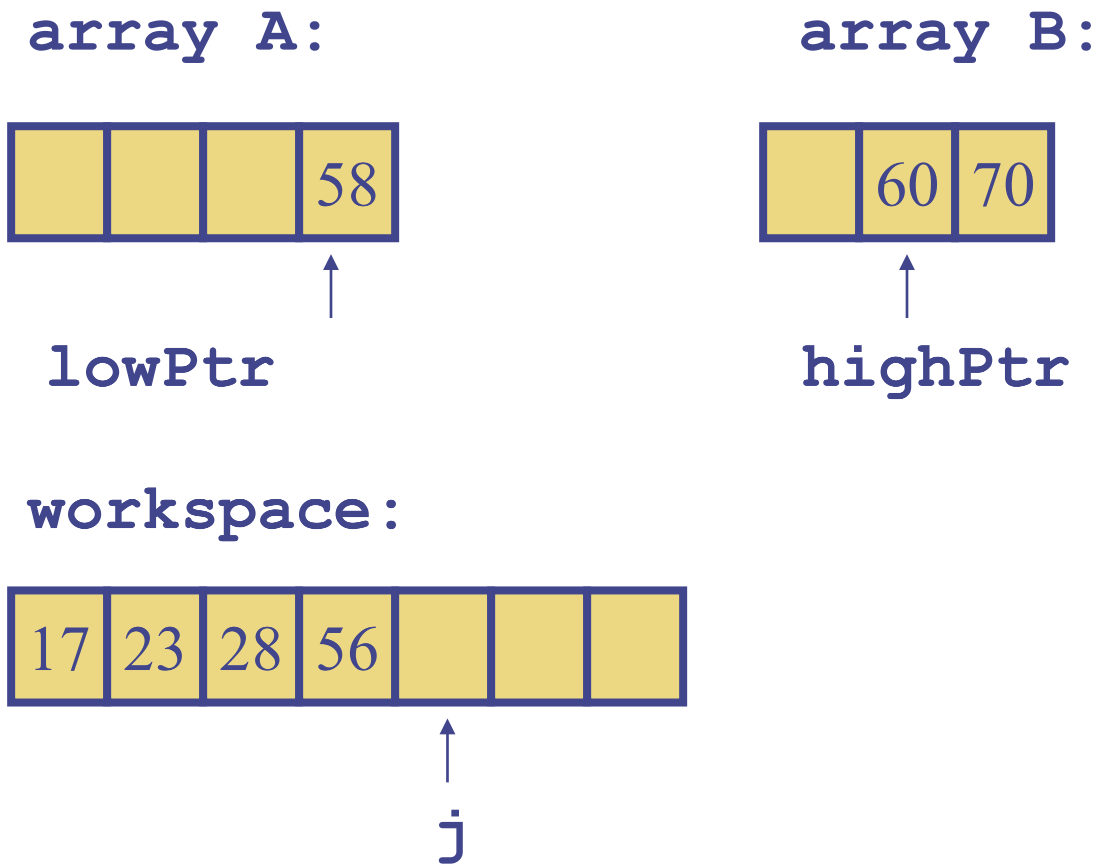
Merging sorted arrays



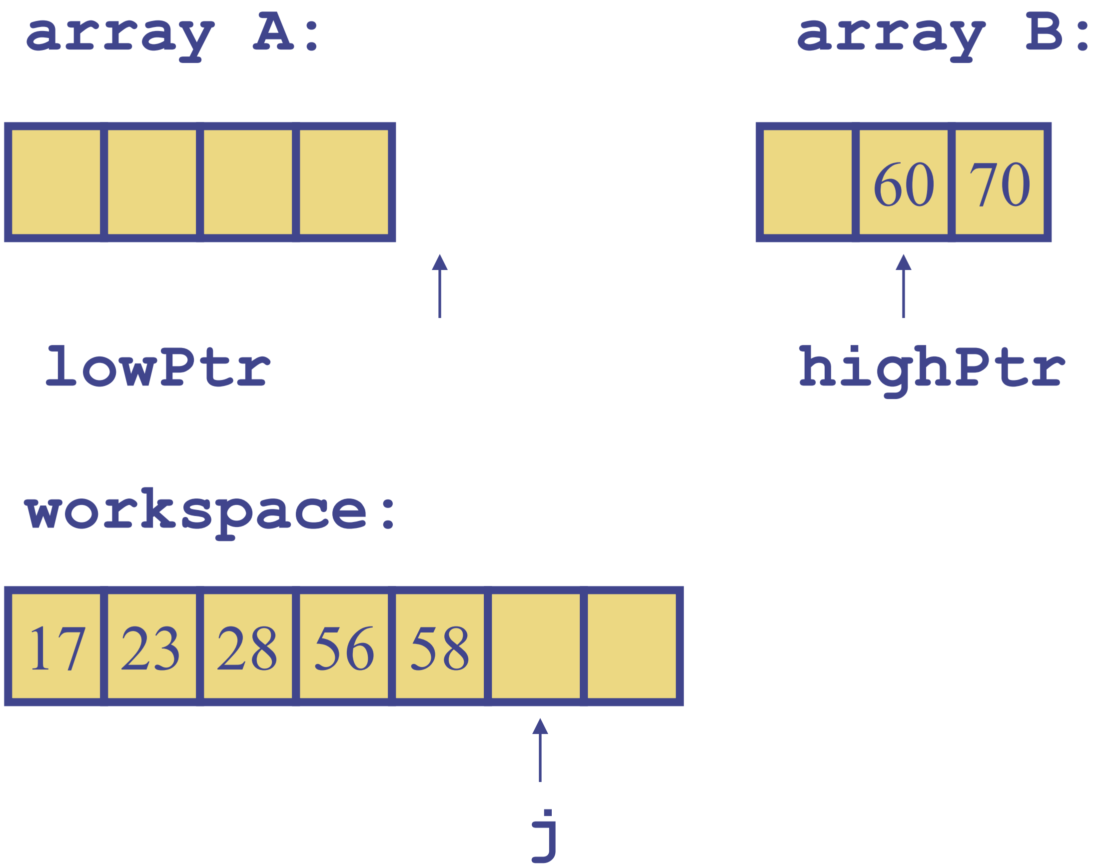
Merging sorted arrays



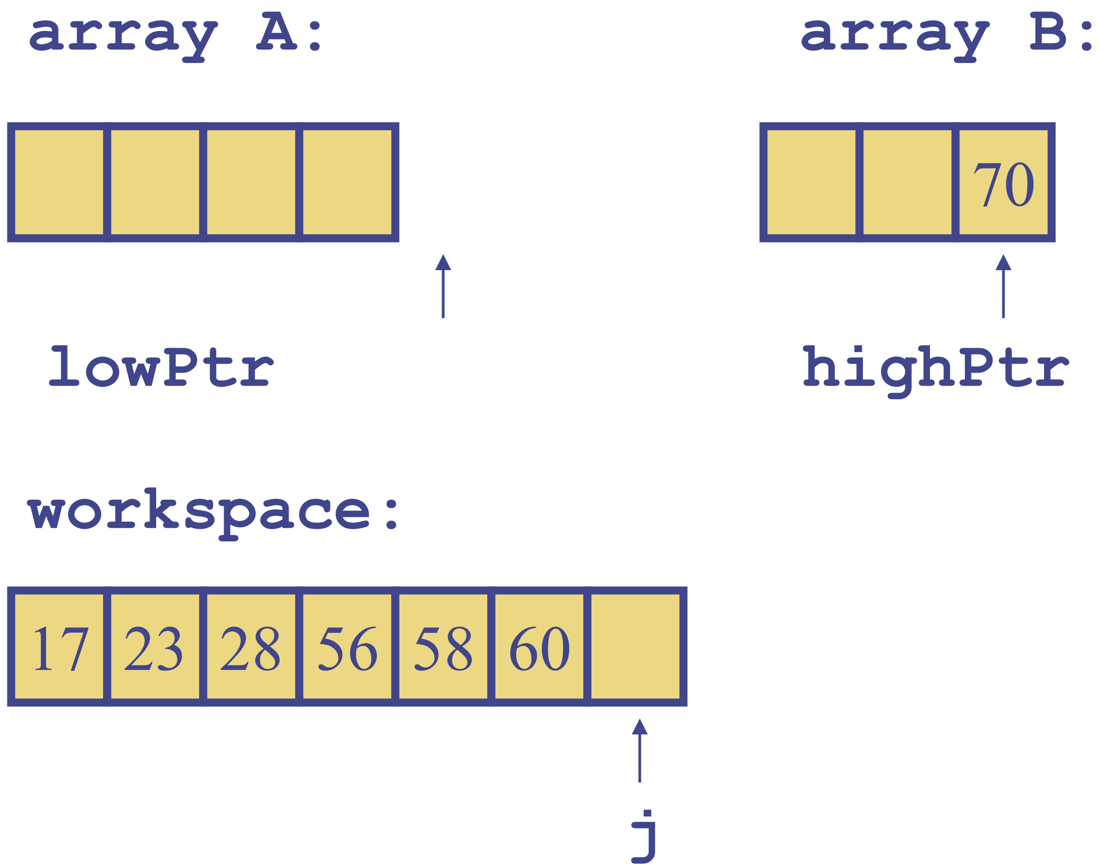
Merging sorted arrays



Merging sorted arrays



Merging sorted arrays



Merging sorted arrays

`array A:`

`lowPtr`

`array B:`

`highPtr`

`workspace:`

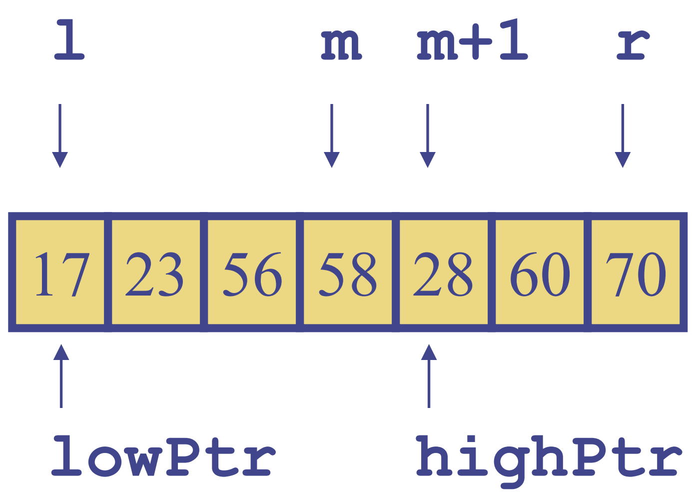
17	23	28	56	58	60	70

`j`

Runtime is $O(n)$

Merging halves of an array

- Pass boundaries of sub-arrays to the algorithm instead of new arrays:
- After merge, copy the workspace back to the original array



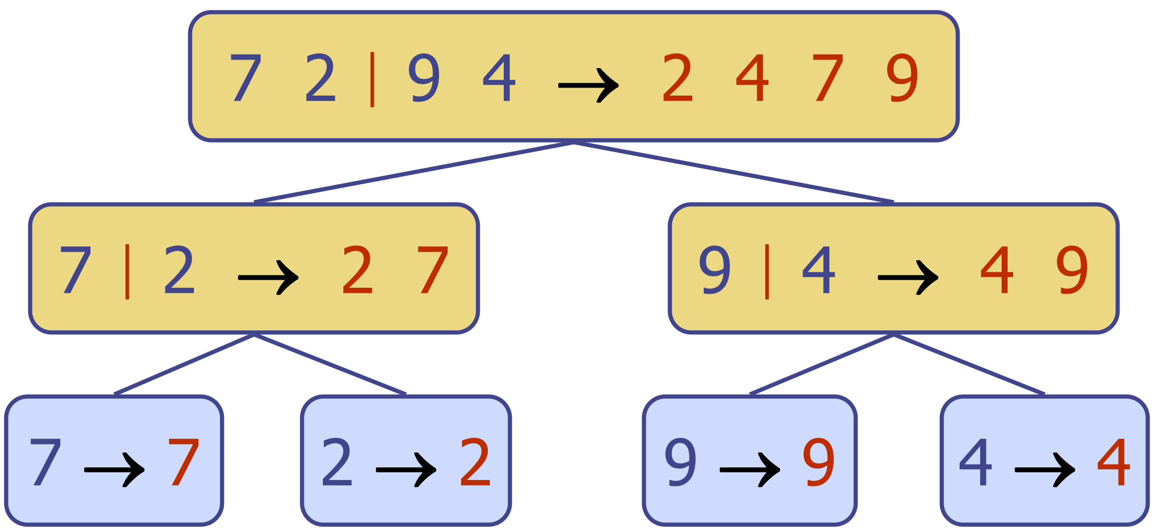
Implementation

```
public static void recMergeSort
                  (int[] arr, int[] workSpace, int l, int r) {
    if (l == r) {
        return;
    } else {
        int m = (l+r) / 2;
        recMergeSort(arr, workSpace, l, m);
        recMergeSort(arr, workSpace, m+1, r);
        merge(arr, workSpace, l, m+1, r);
    }
}
```

Merge-Sort Tree

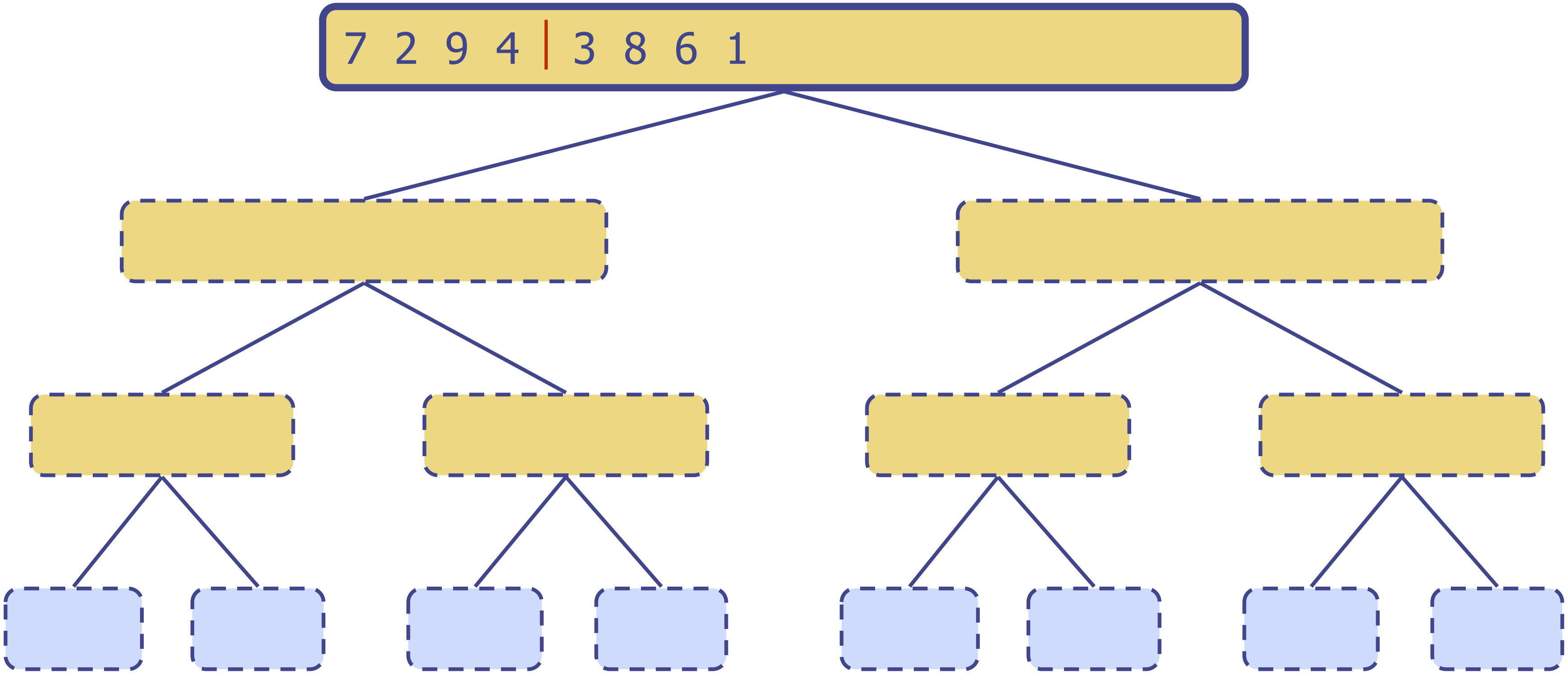
Not actually implemented this way! A `node' here is conceptual not real

- 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



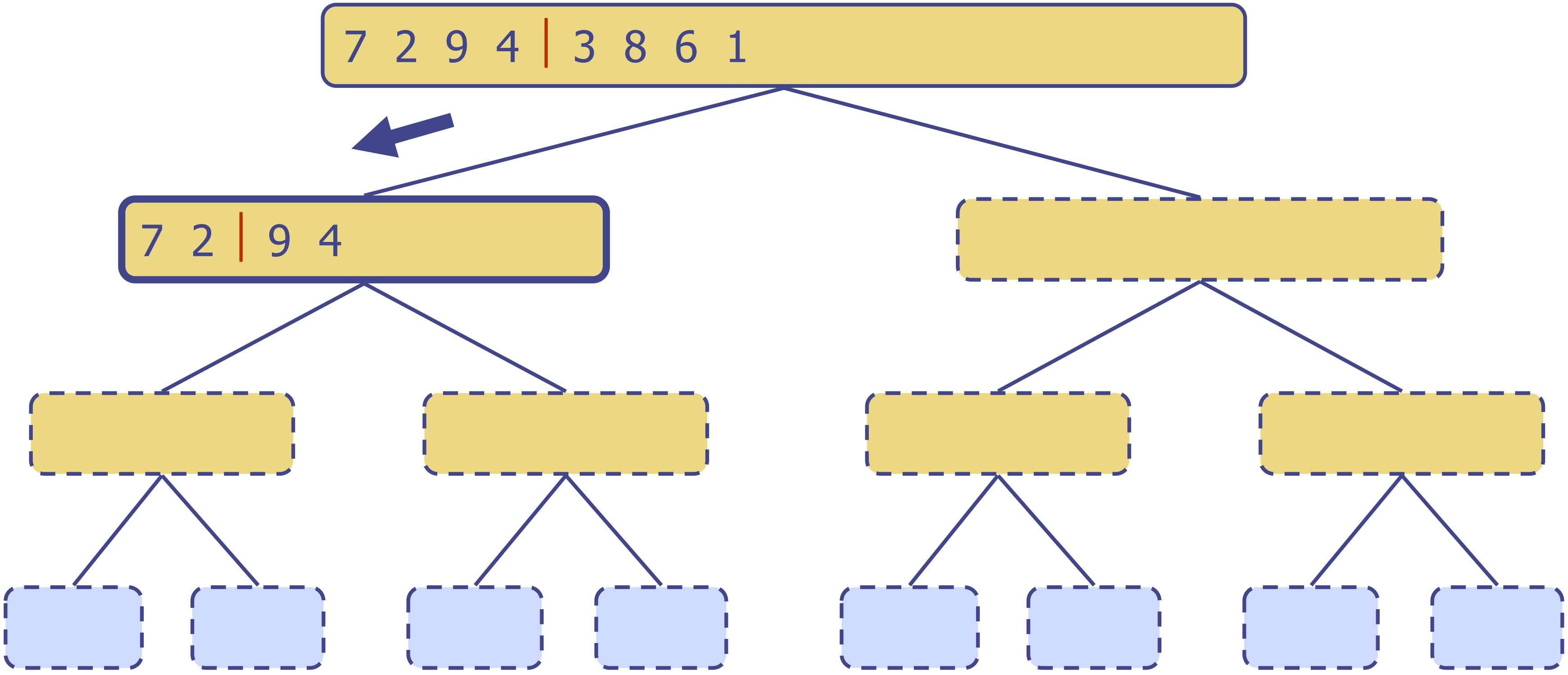
Execution Example

- Partition



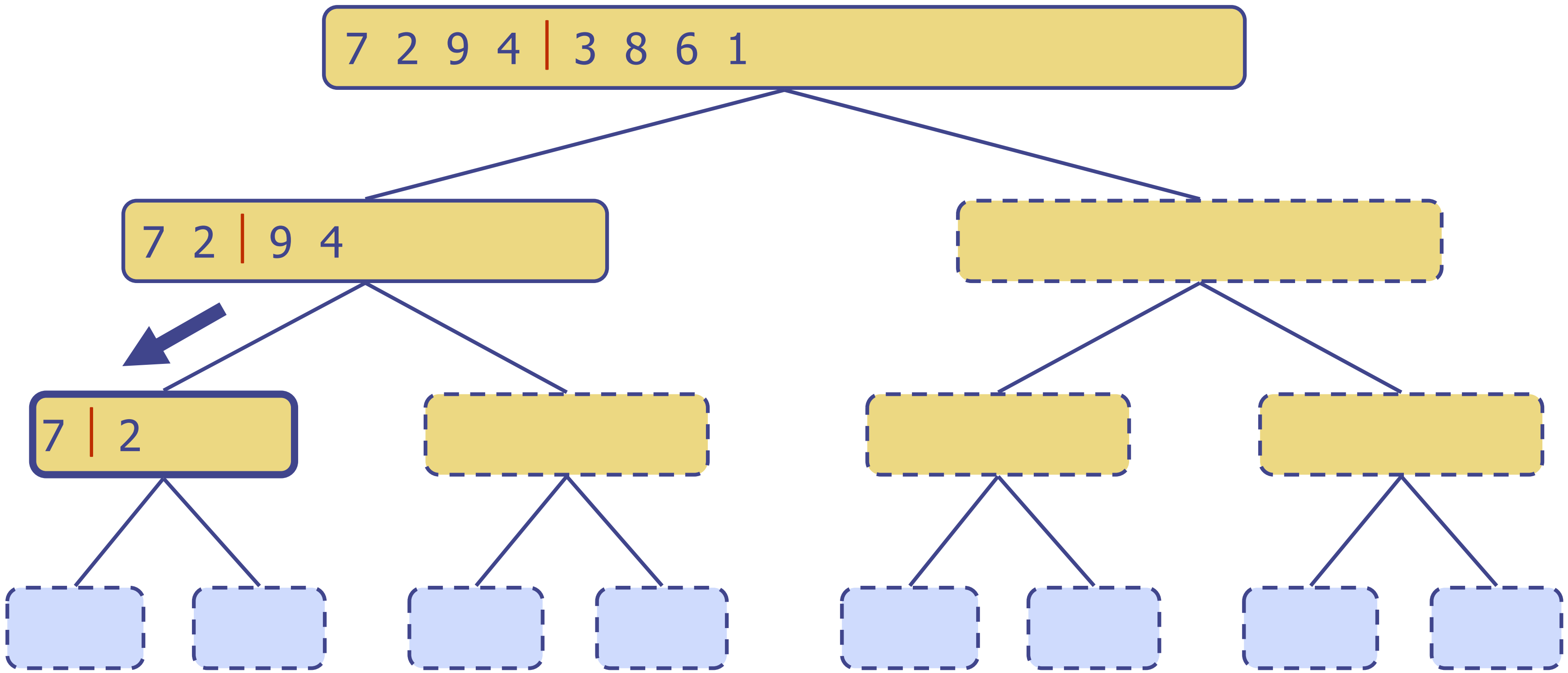
Execution Example (cont.)

- Recursive call, partition



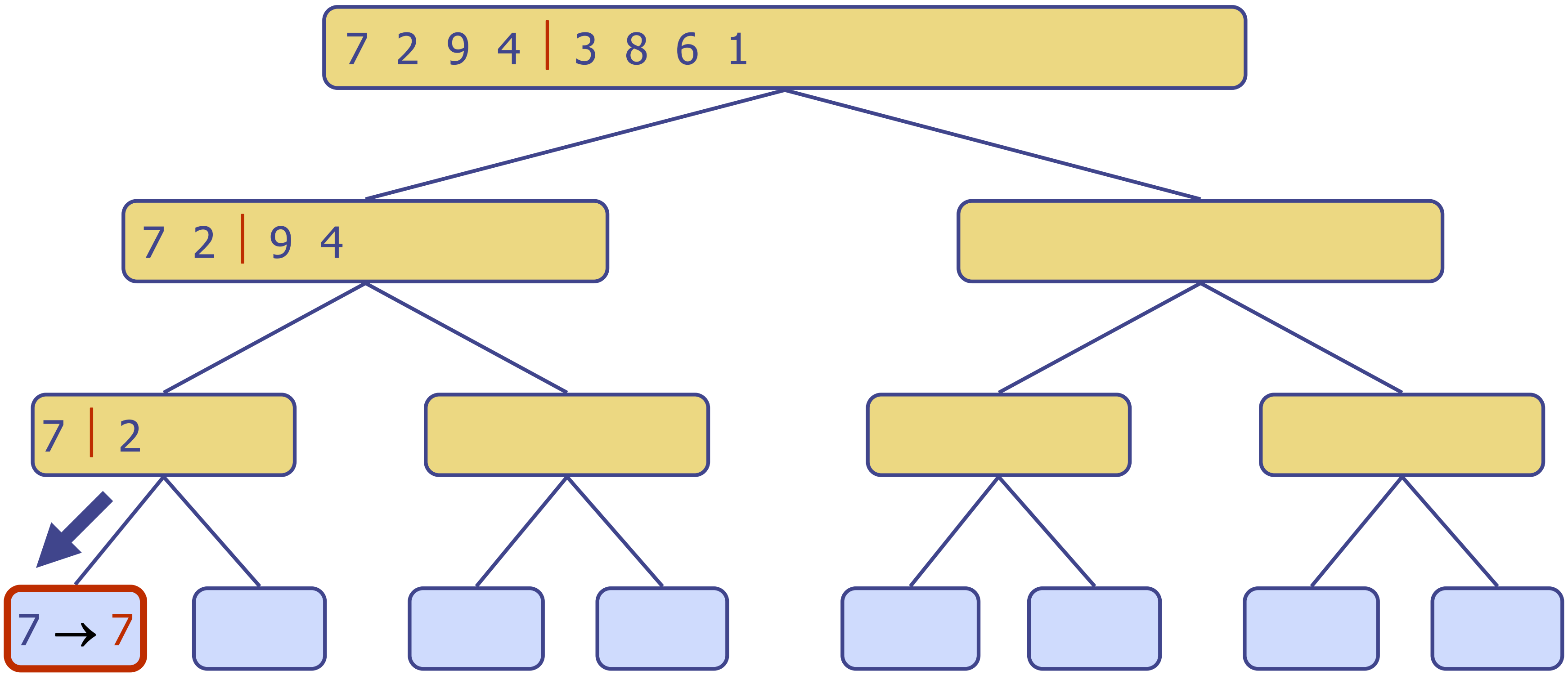
Execution Example (cont.)

- Recursive call, partition



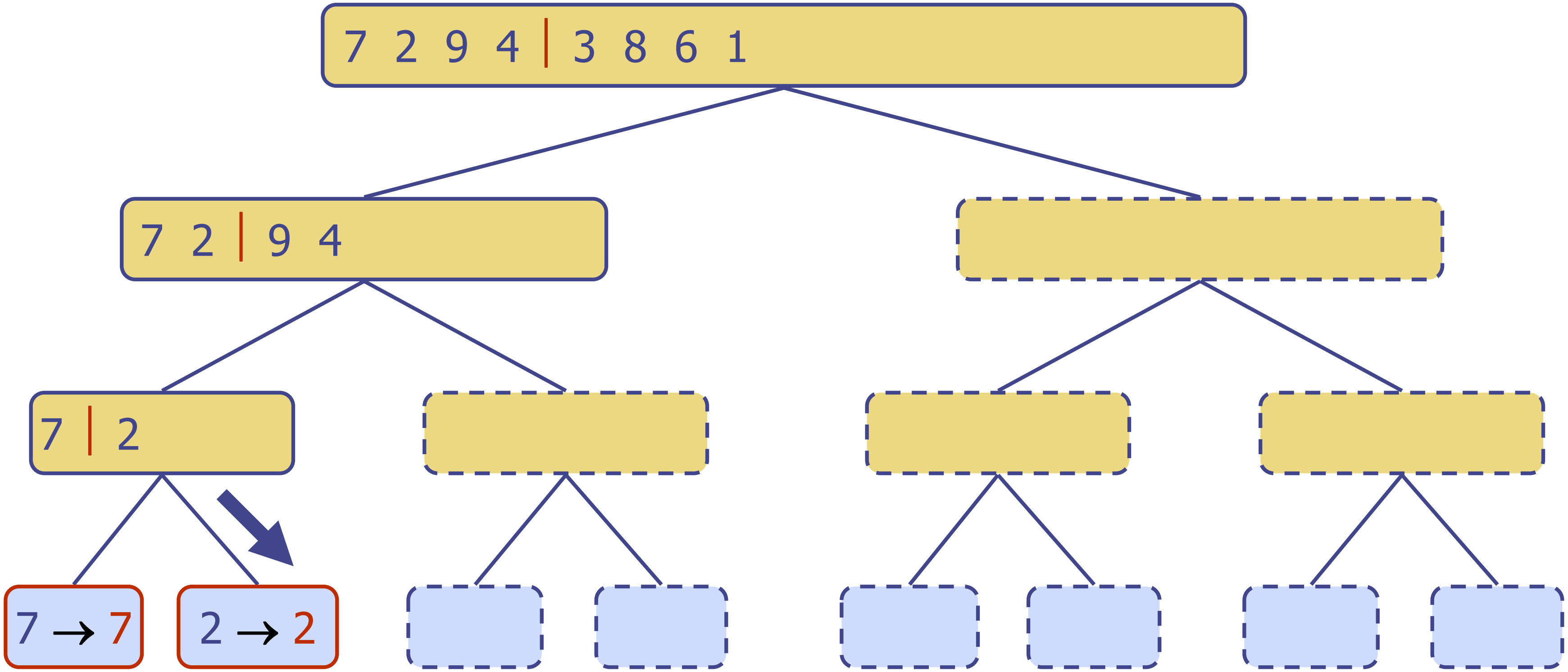
Execution Example (cont.)

- Recursive call, base case



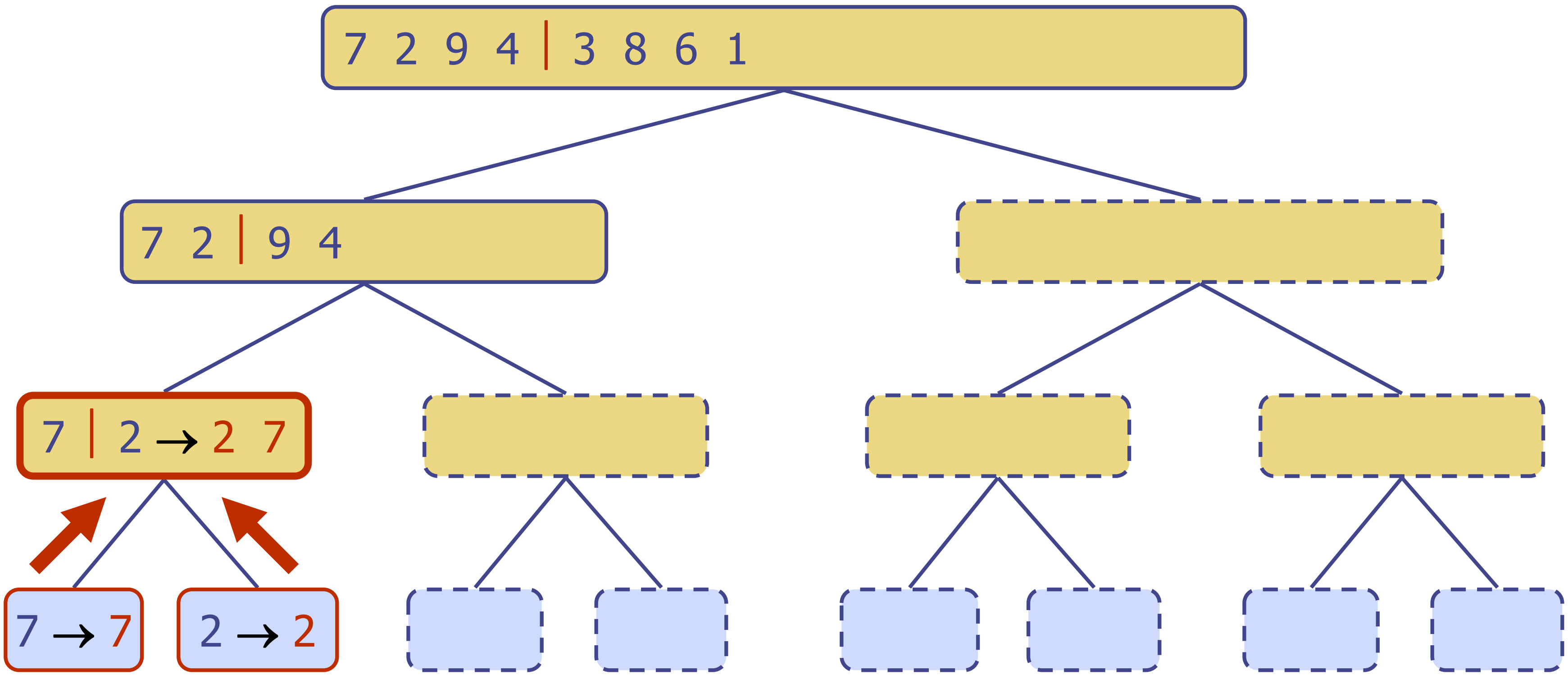
Execution Example (cont.)

- Recursive call, base case



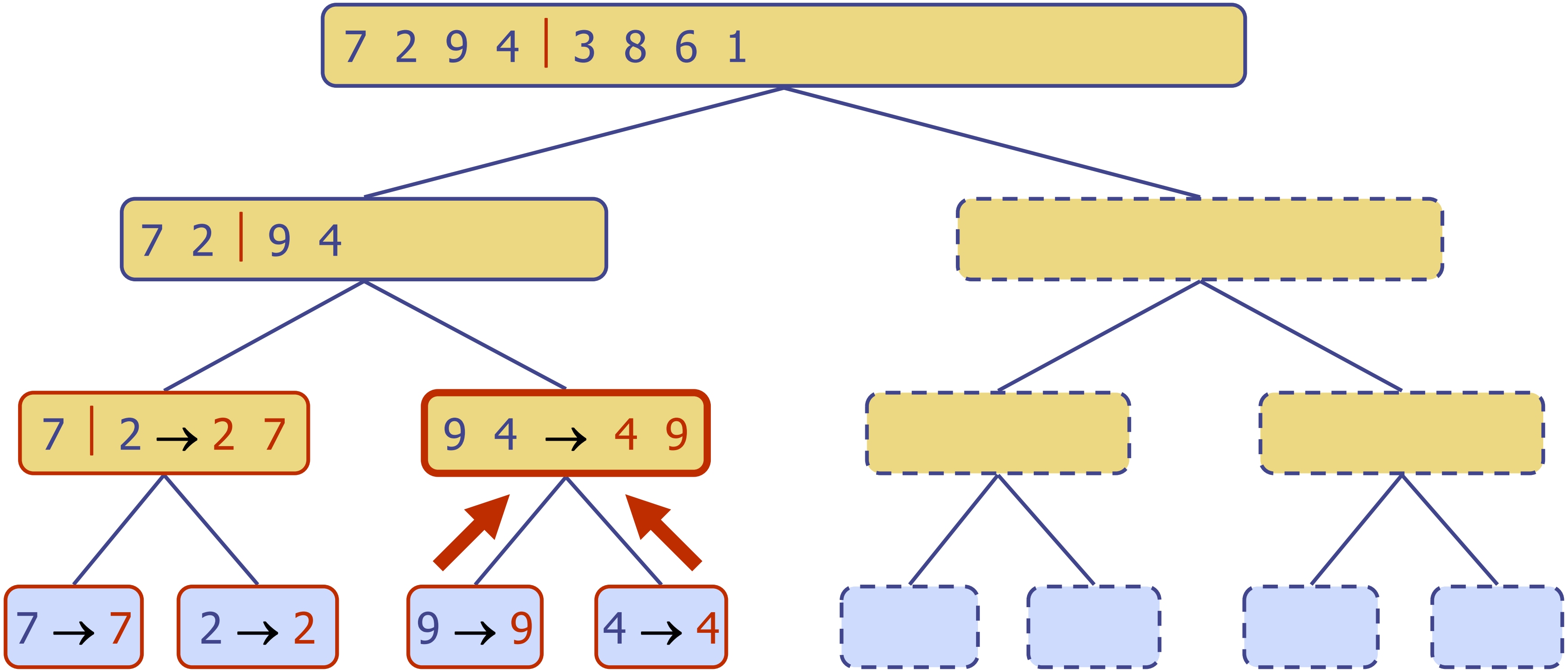
Execution Example (cont.)

- Merge



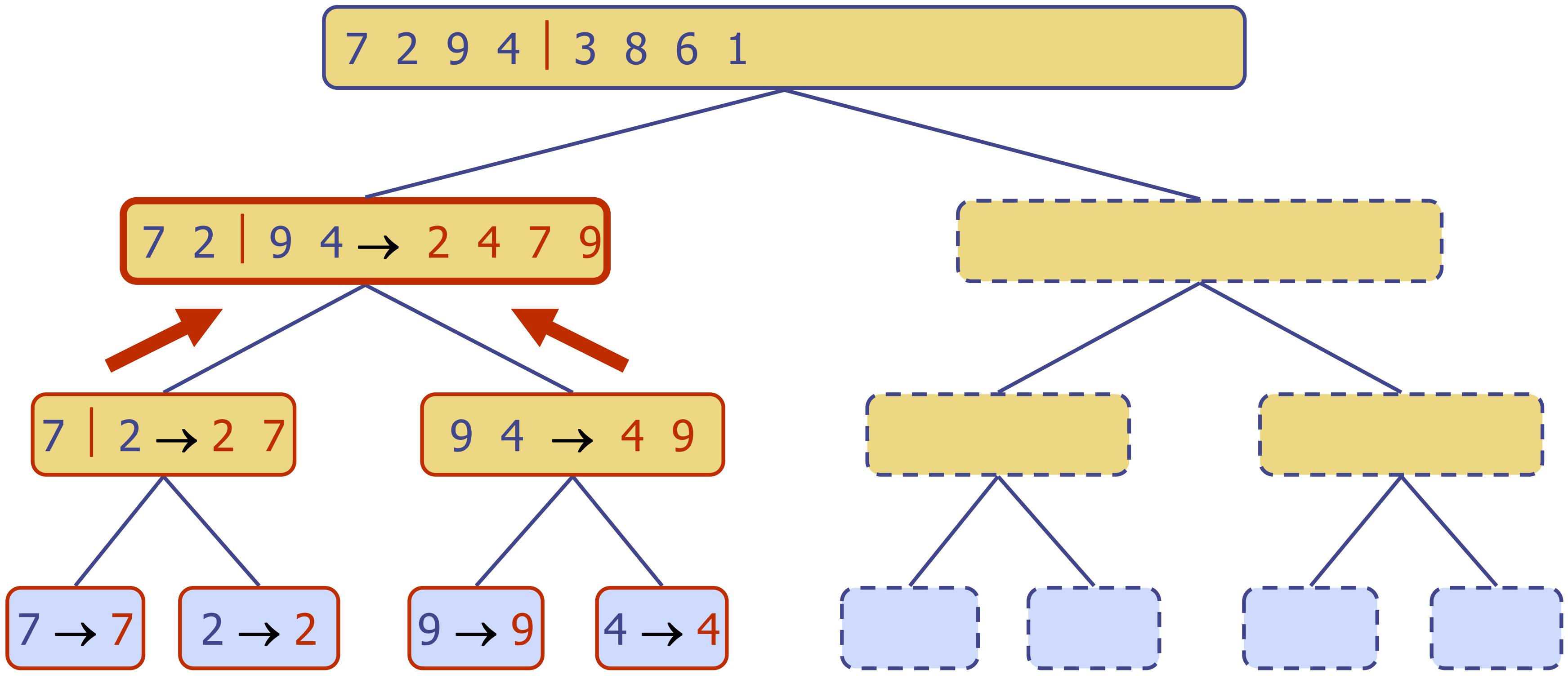
Execution Example (cont.)

- Recursive call, …, base case, merge



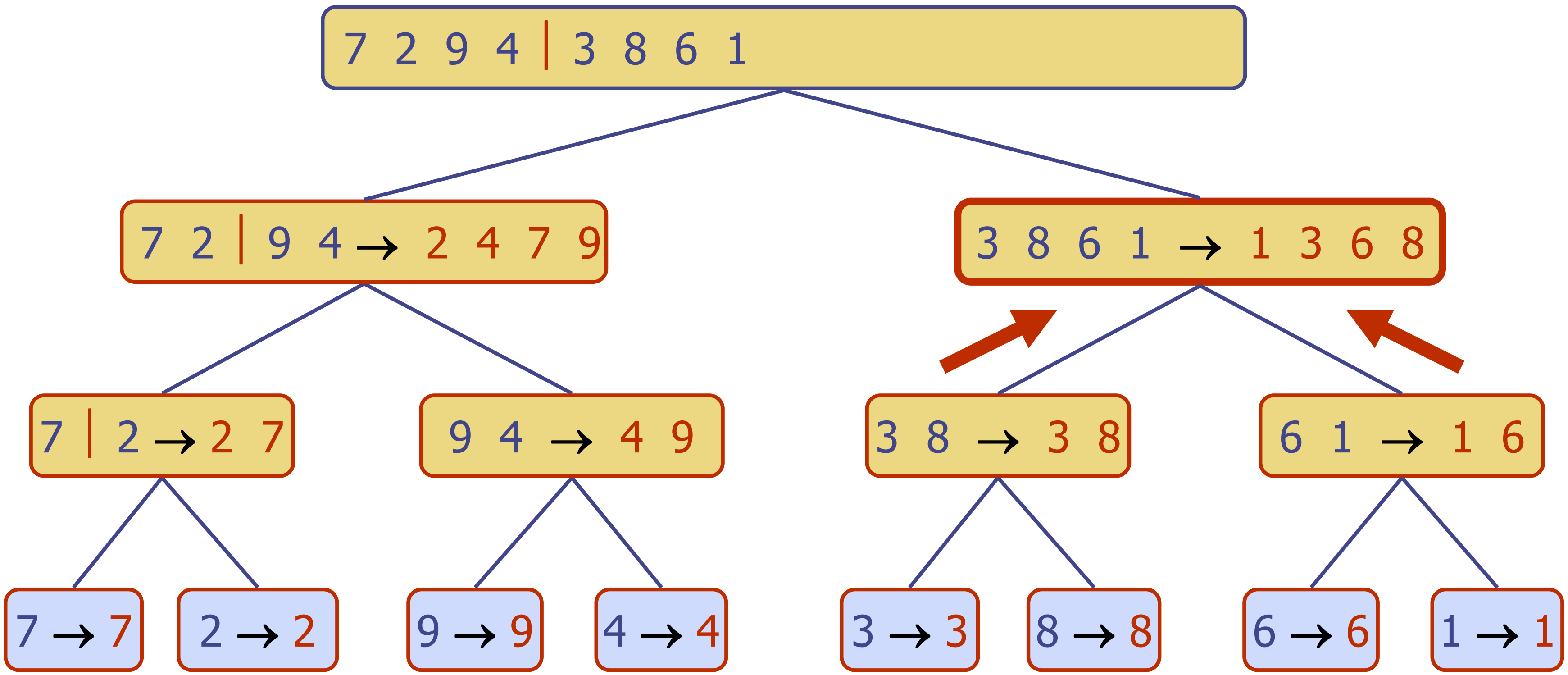
Execution Example (cont.)

- Merge



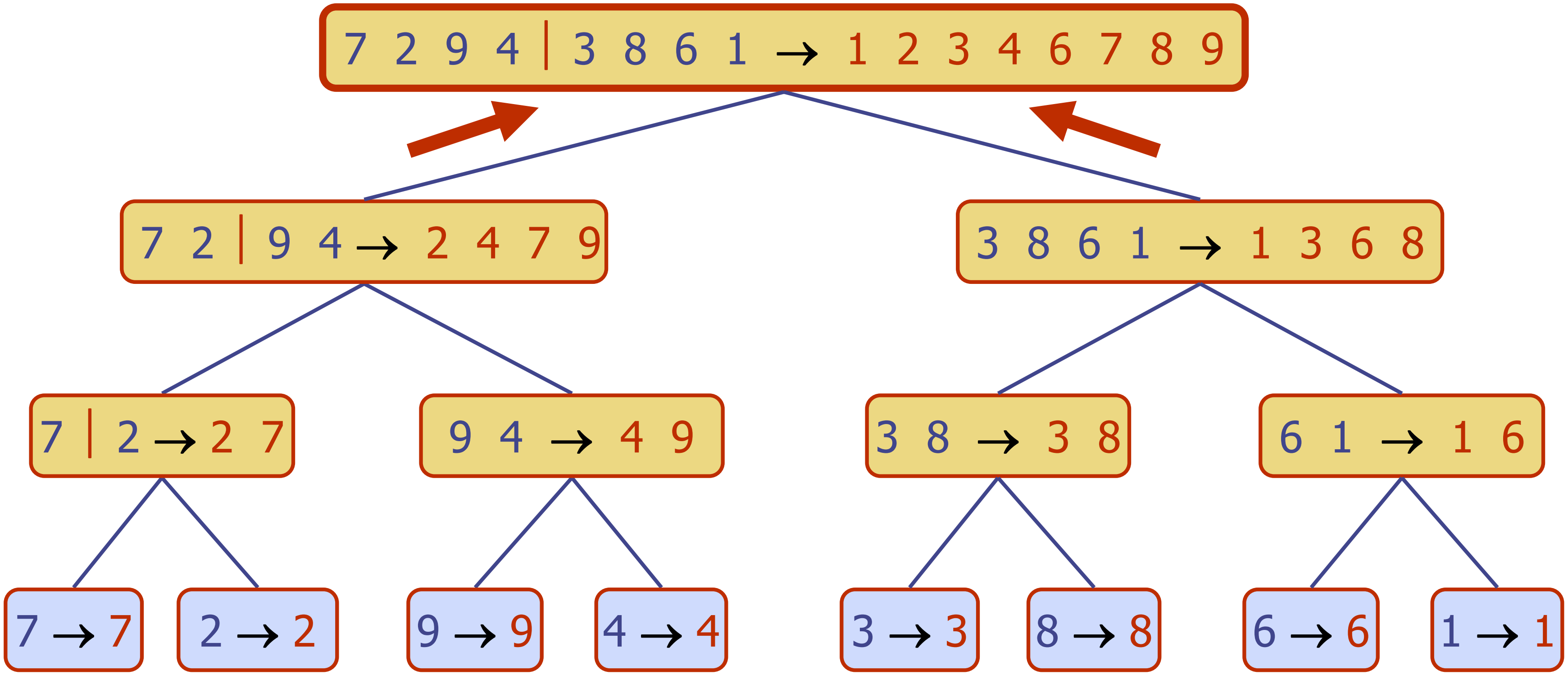
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

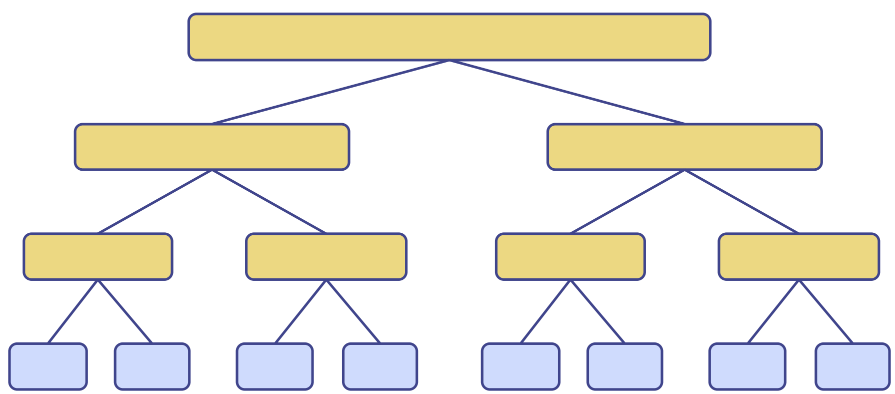
- Merge



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 all the nodes at depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
  - the numbers all occur and are used at each depth
- Thus, the total running time of merge-sort is $O(n \log n)$

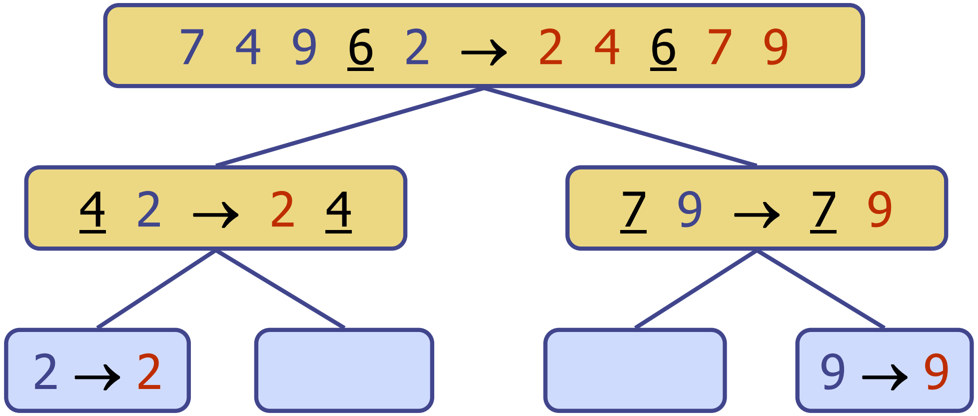
depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Using merge sort

- Fast sorting method for arrays
- Good for sorting data in external memory – because works with adjacent indices in the array (data access is sequential)
  - It accesses data in a sequential manner (suitable for sorting data on a disk)
- Not so good with lists: relies on constant time access to the middle of the sequence

Quick-Sort



Motivations

- In merge sort the `divide’ is simple, and the `merge’ (relatively) complicated
- Can we make the ‘merge’ simple?
  - Answer: make the `divide’ more complicated so that the merge becomes `concatenate’
- Analogy: sort a pack of cards by

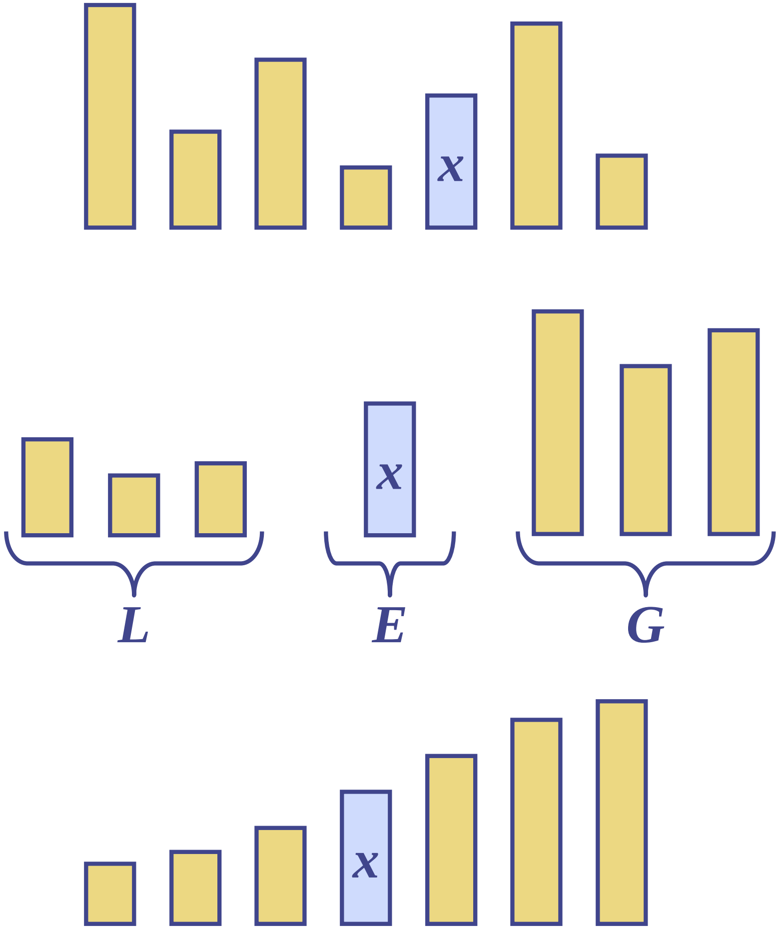
1. divide into ‘red’ and ‘black’ cards
2. divide by suit (red into hearts and diamonds,…)
3. divide by value …

When is `merge' simple?

- When the lists A and B are sorted and known to be in disjoint ordered ranges
  - all of elements of A are smaller than all those of B
- If A and B are stored as consecutive sub-arrays, then merge actually needs no work at all:
  - Just "forget the boundary"

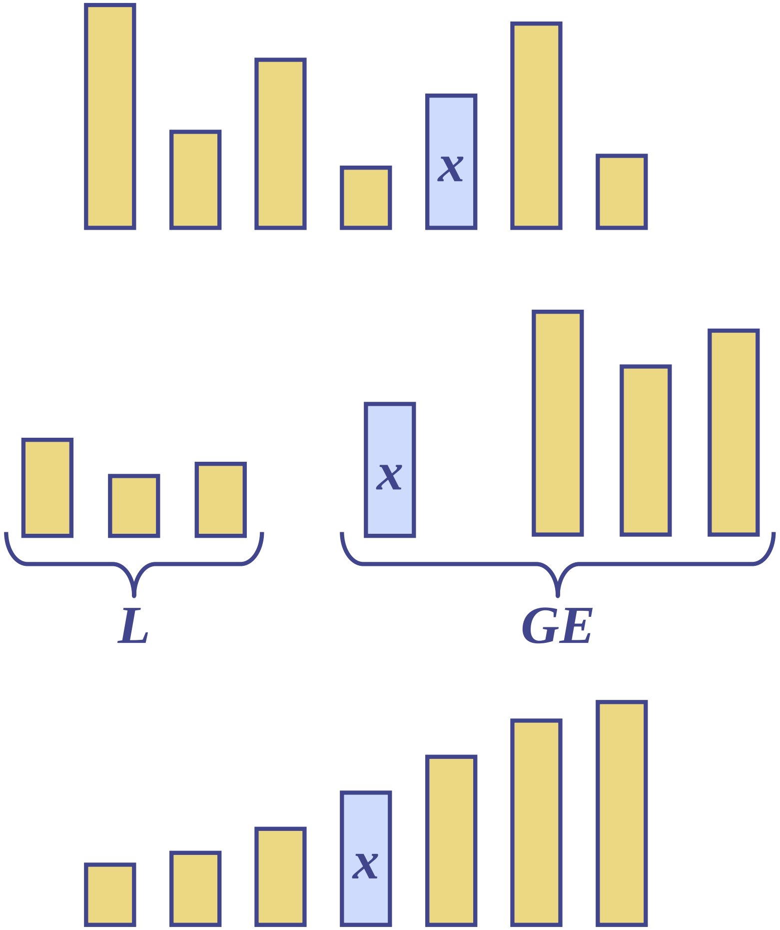
Quick-Sort (3-way split)

- 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
  - Recur: sort L and G
  - Conquer: join L, E and G



Quick-Sort (2-way split)

- 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
    - GE elements greater than or equal to x
  - Recur: sort L and GE
  - Conquer: join L, GE



Partition of lists (using extra workspace)

- Suppose store L, E and G as separate structures (e.g. as arrays, vectors or lists)
- We partition an 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 x
- Each insertion and removal is at the beginning (or end) of the sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

“In-place” or “extra workspace”?

- For sorting algorithms (and algorithms in general) an important issue can be how much extra working space they need besides the space to store the input
- “In-place” means they only a “little” extra space (e.g. O(1)) is used to store data elements.
  - The input array is also used for output, and only need a few temporary variables
  - Exercise: check that bubble-sort is “in-place”
  - Previous “merge” used extra O(n) array (can be made in-place, but messy and so we ignore this option)

Partitioning arrays "in-place"

- Perform the partition using two indices to do a "2-way split" of S into L and E+G.

j k

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 **6** 9 (pivot = 6)

- Repeat until j and k cross:
  - Scan j to the right until finding an element ≥ pivot.
  - Scan k to the left until finding an element < pivot.
  - Swap elements at indices j and k

j k

3 2 5 1 0 $\boxed{7}$ 3 5 9 $\boxed{2}$ 7 9 8 9 7 **6** 9

The scans are not done in 'lock step' but independently work inwards. Do some examples and make sure you understand how and why this works!!

Exercise (LAB)

- Write Java code to do the partition and check that it works on some examples
  - (It is only 10-20 lines of code, but will greatly help clarify the algorithm.)

Quicksort Overall Implementation

With the previous 2-way split:

```
public static void recQuickSort(int[] arr, int left, int right) {
   if (right - left <= 0) return;
   else {
      int border = partition(arr, left, right); // "crossing position"
      recQuickSort(arr, left, border);
      recQuickSort(arr, border+1, right);
   }
}
```

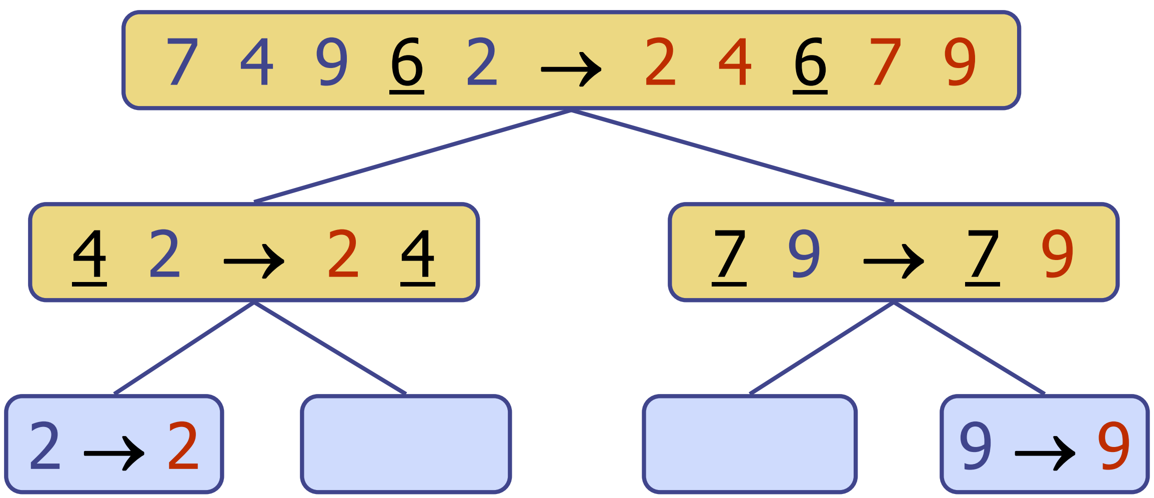
Quicksort Overall Implementation

With a 3-way split:

```
public static void recQuickSort(int[] arr, int left, int right) {
    if (right - left <= 0) return;
    else {
        int border = partition(arr, left, right); // pivot position
        recQuickSort(arr, left, border-1);
        recQuickSort(arr, border+1, right);
    }
}
```

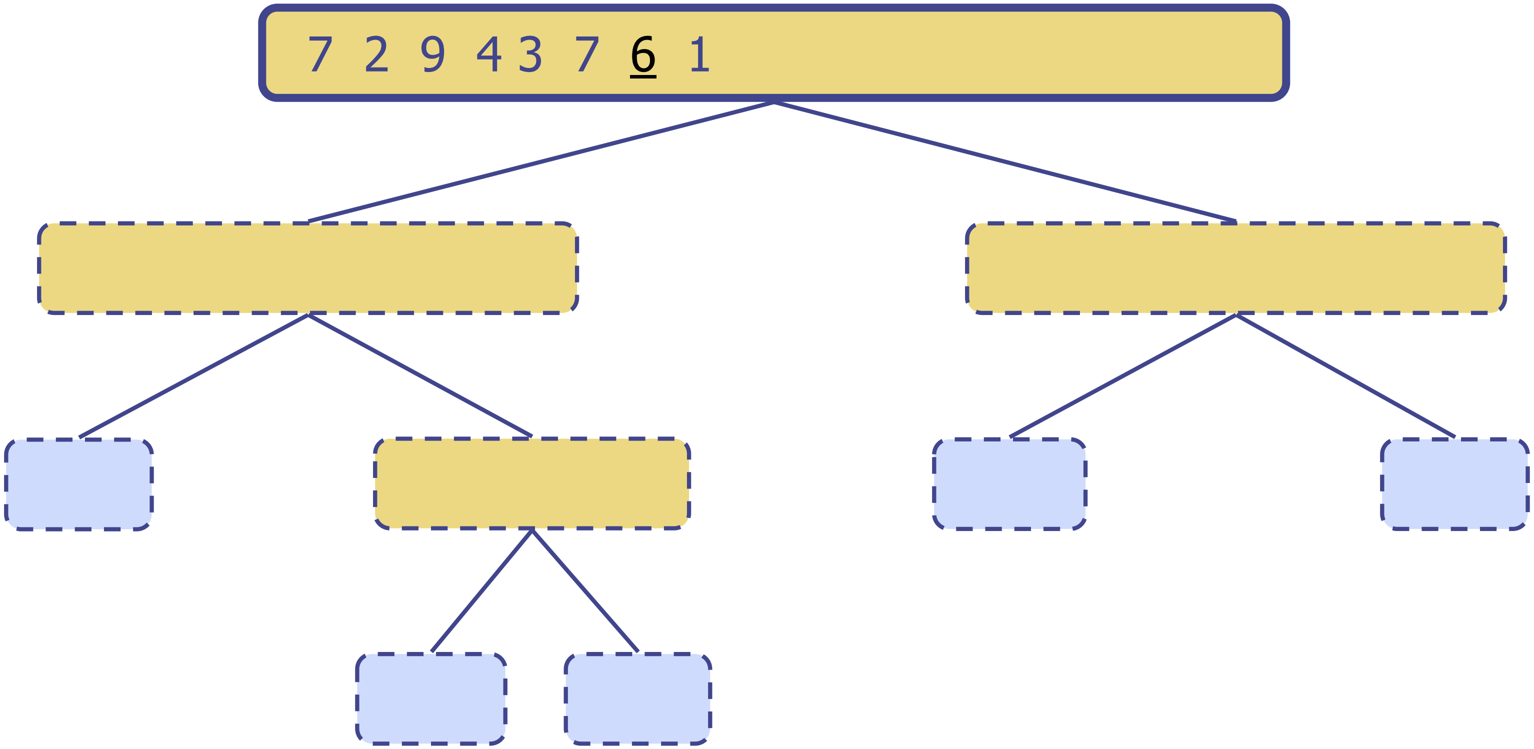
Quick-Sort Tree

- 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
  - Example shows 3-way split.
    Exercise (offline) do the same for a 2-way split.



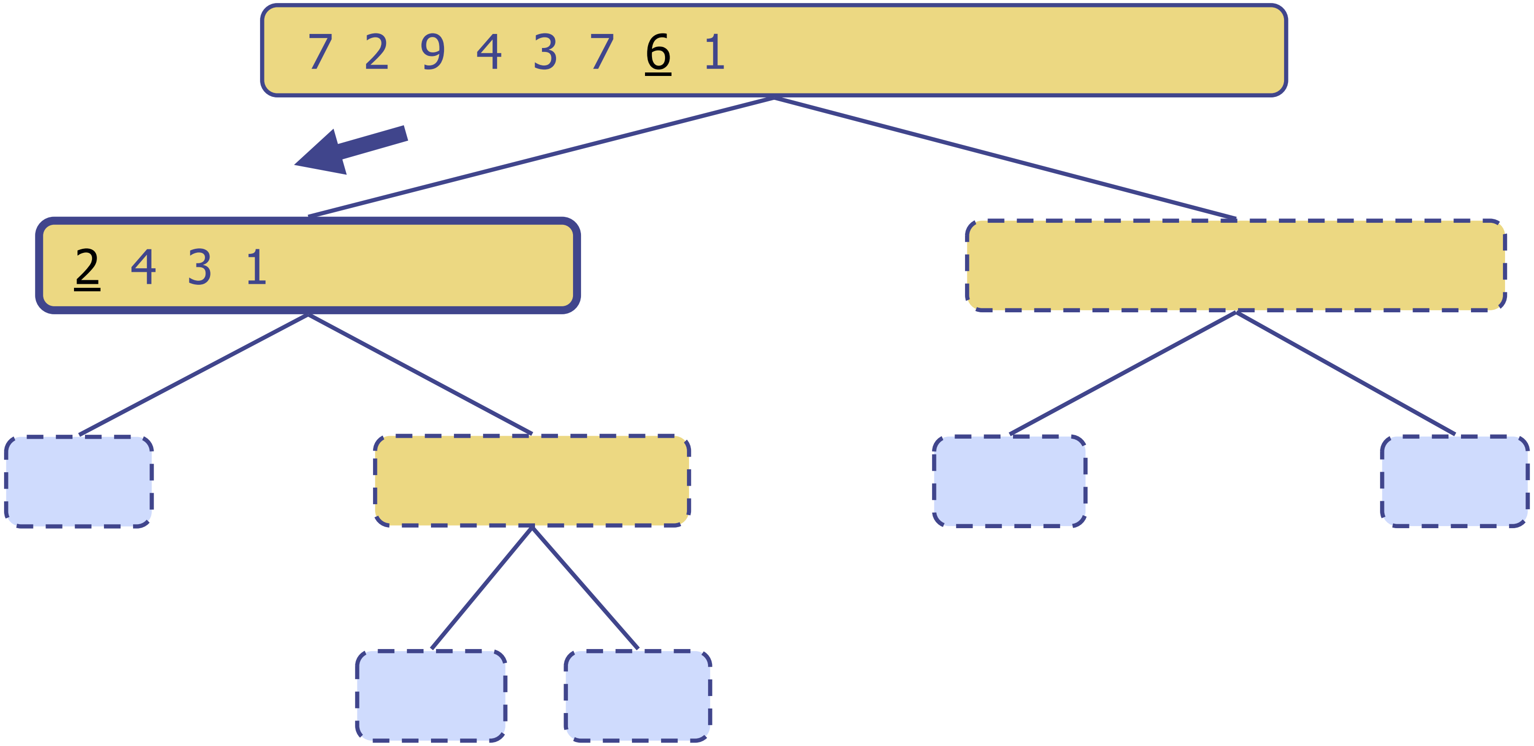
Execution Example

- Pivot selection



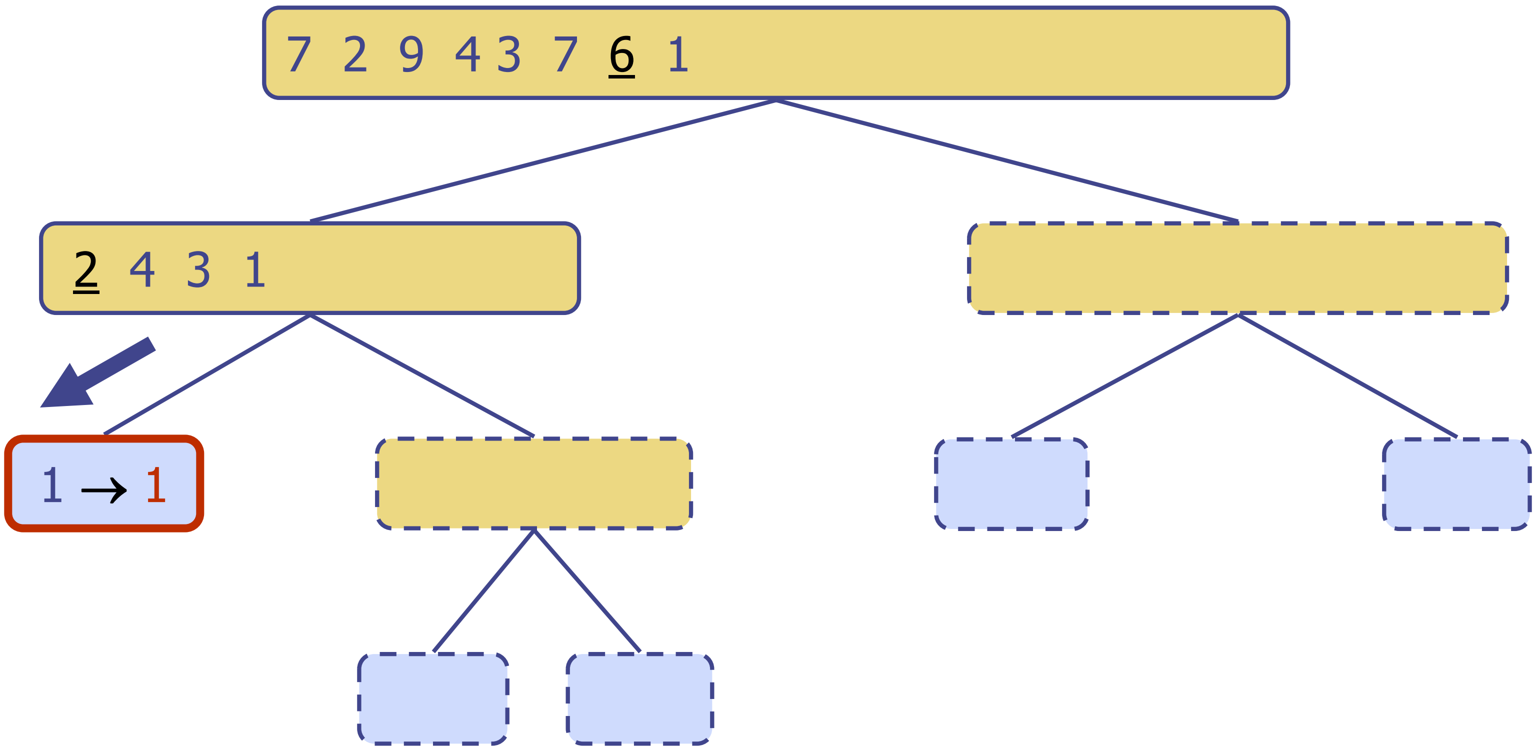
Execution Example (cont.)

- Partition, recursive call, pivot selection



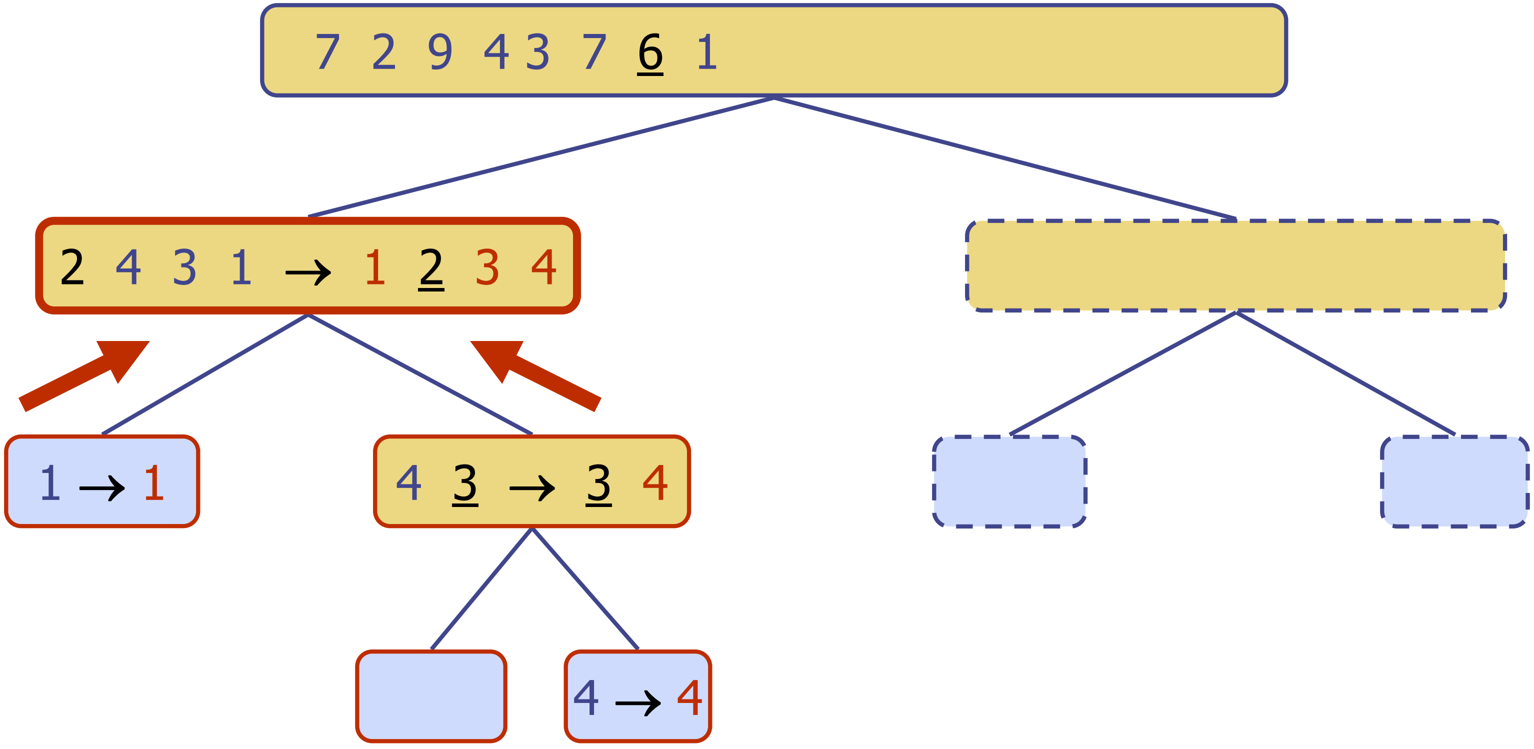
Execution Example (cont.)

- Partition, recursive call, base case



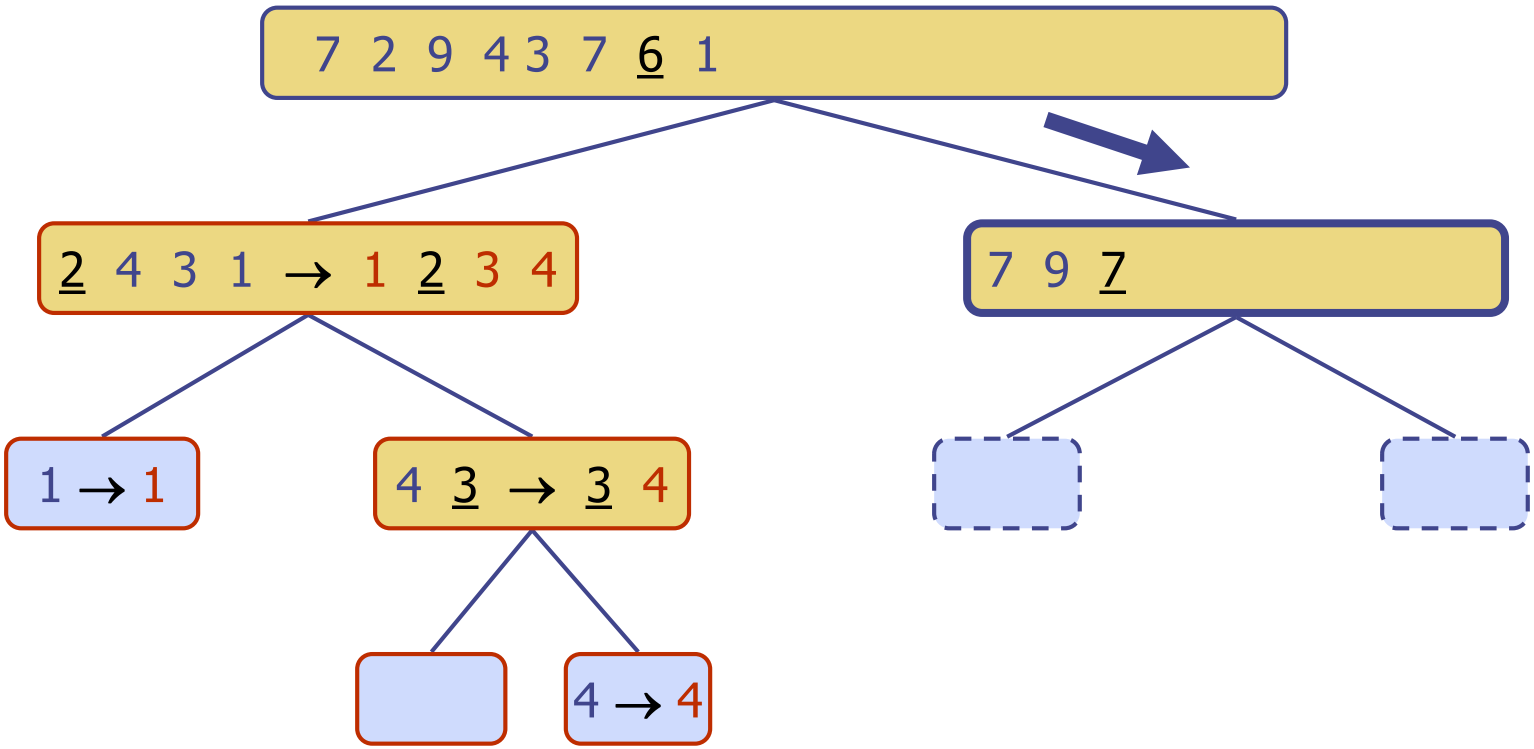
Execution Example (cont.)

- Recursive call, …, base case, join



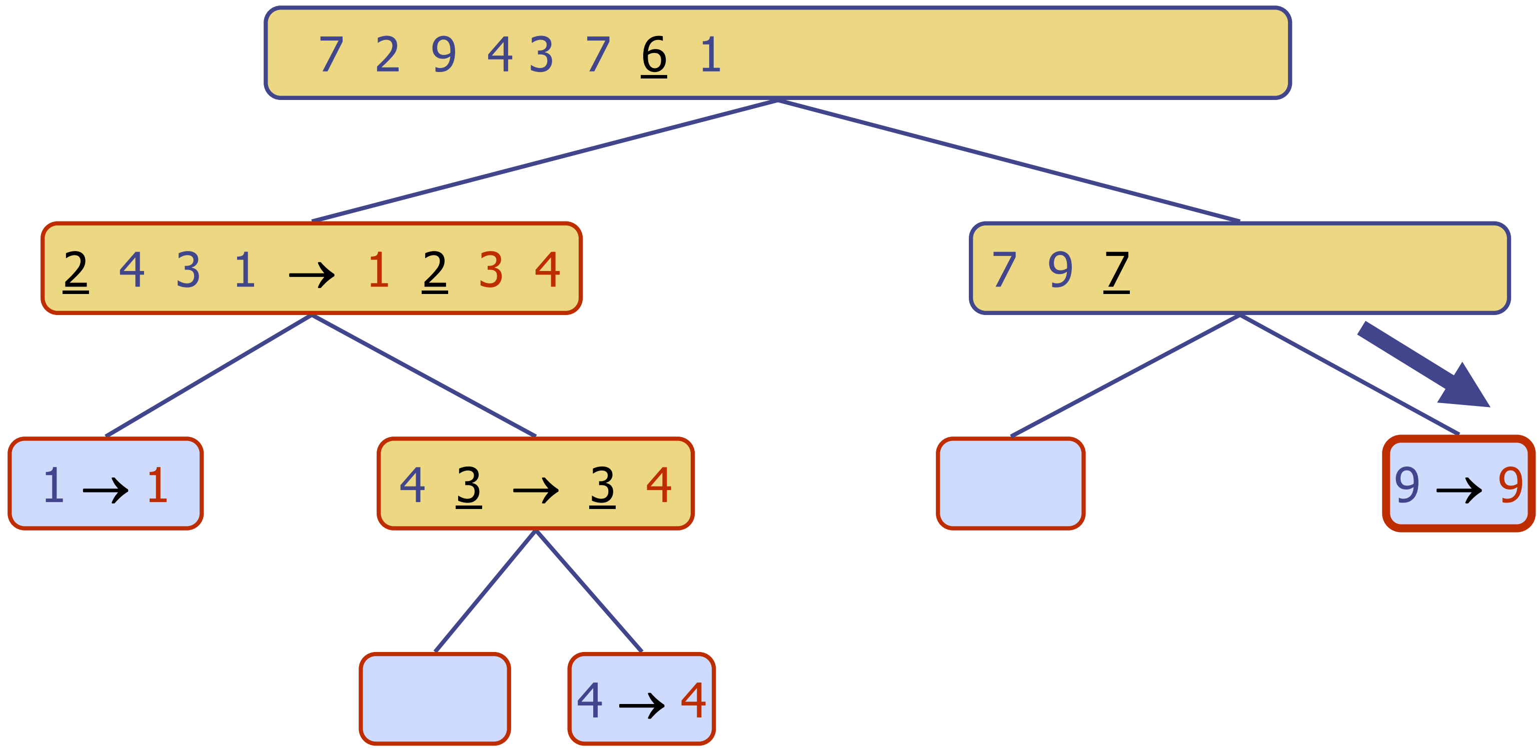
Execution Example (cont.)

- Recursive call, pivot selection



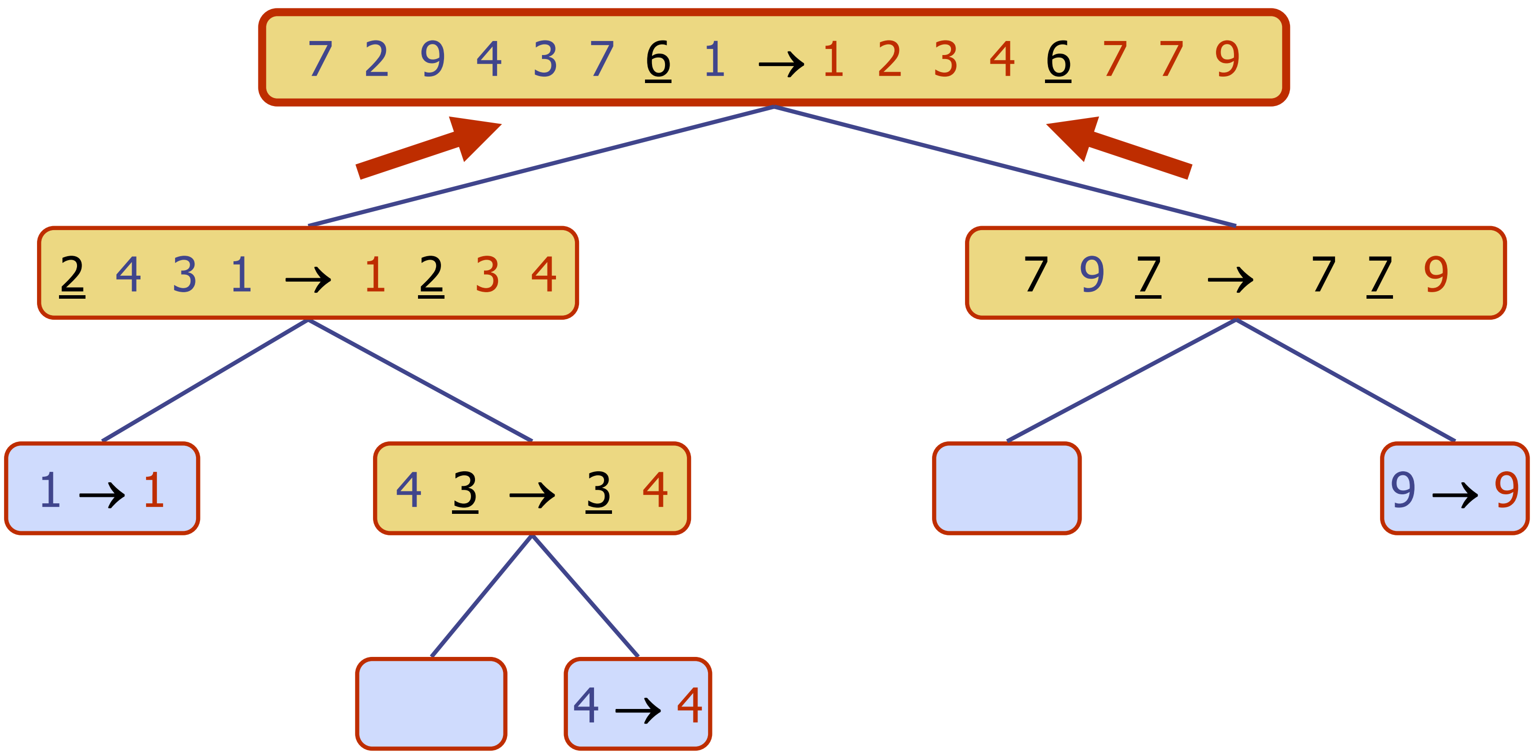
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



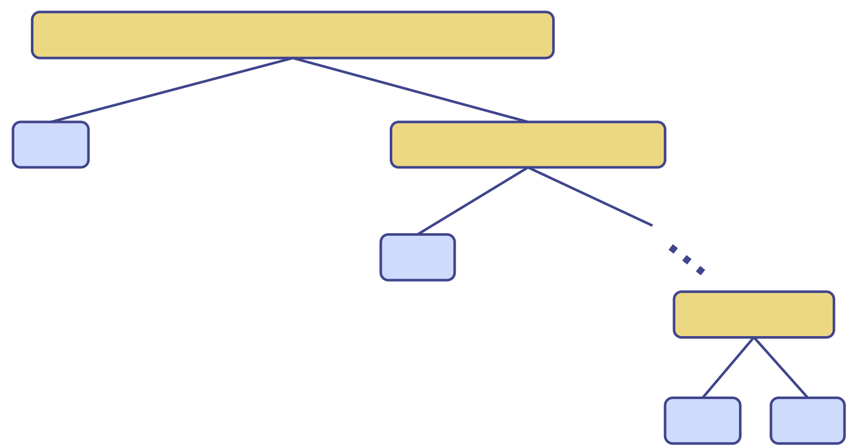
Worst-case Running Time

- 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



Best-case Running Time

- The best case for quick-sort occurs when the pivot is the median element
- The L and G parts are equal – the sequence is split in halves, like in merge sort
- Thus, the best-case running time of quick-sort is $O(n \log n)$

Average-case Running Time

- The average case for quick-sort: half of the times, the pivot is roughly in the middle
- Thus, the average-case running time of quick-sort is $O(n \log n)$ again
- Detailed proof in the textbook

Motivations for quicksort

- Why do we select a pivot? I.e. what advantages might quicksort ever have over mergesort?
  - Because it can be done “in-place”
    - Uses a small amount of workspace
    - Because the “merge” step is now a lot easier!!
  - The “split” is more complicated, and the merge “much” easier – but turns out that the quick-sort split is easier to do in-place than the merge-sort merge

Summary of Sorting Algorithms

Algorithm	Time	Notes
bubble & selection-sort	$O(n^2)$	• in-place (no extra memory) • slow (good for small inputs)
insertion-sort	$O(n^2)$	• in-place (no extra memory) • slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	• in-place (+stack), randomized • fast (good for large inputs)
merge-sort	$O(n \log n)$	• sequential data access • fast (good for huge inputs)

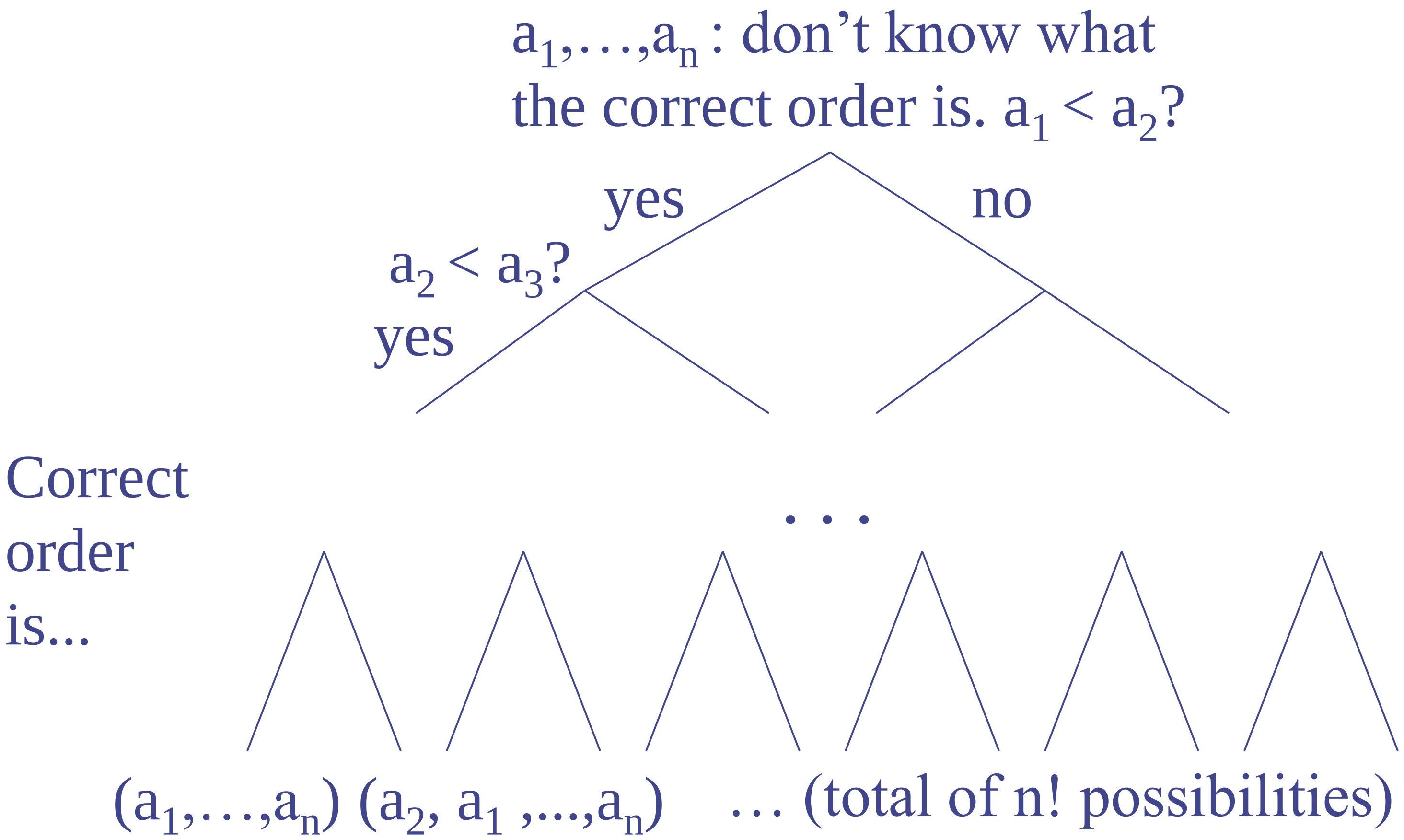
Comparison sorting

- A sorting algorithm is a comparison sorting algorithm if it uses comparisons between elements in the sequence to determine in which order to place them
- Examples of comparison sorts: bubble sort, selection sort, insertion sort, heap sort, merge sort, quicksort.
- Example of ***not*** a comparison sort: bucket sort (Ch. 11.4).
  - Runs in O(n), but relies on knowing the range of values in the sequence (e.g. "integers between 1 and 1000").

Lower bound for comparison sort

- We can model sorting which depends on comparisons between elements as a binary decision tree.
- At each node, a comparison between two elements is made; there are two possible outcomes and we find out a bit more about the correct order of items in the array.
- Finally arrive at full information about the correct order of the items in the array.

Comparison sorting



How many comparisons?

- If a binary tree has n! leaves, than the minimal number of levels (assuming the tree is perfect) is ($\log_2$ n!) +1.
- This shows that O(n log n) sorting algorithms are essentially optimal ($\log_2$n! is not equal to n $\log_2$n, but has the same growth rate modulo some hairy constants).
- **Comparison-based sorting cannot do better than O(n log n)**
- Note: you should know the result, but do not need to know the proof.

Questions to ask about sorting algorithms

- Big-Oh complexity (both time and space)?
  - Best case inputs? Worst case inputs?
- Extra workspace needed?
  Or is it `in-place'?
- Stable sorts?
- Comparison-based?
- Data access patterns?
  - Sequential? Random Access?
- Relevant and appropriate assertions

Make sure you understand these questions, and the answers for various sorting algorithms.

Minimum Expectations

- For both merge- and quicksort:
  - know the algorithm and how it works on examples
  - know and be able to justify/prove their big-Oh behaviours
- Know the meaning of ‘comparison-based sorting’ and the resulting O(n log n) lower bound on complexity (though no need to be able to prove it).