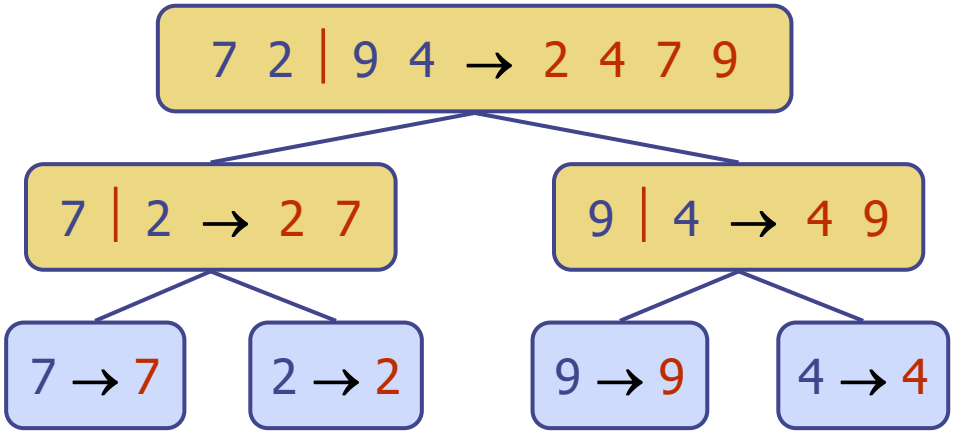
COMP2054-ADE

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

Mergesort



Divide-and-Conquer

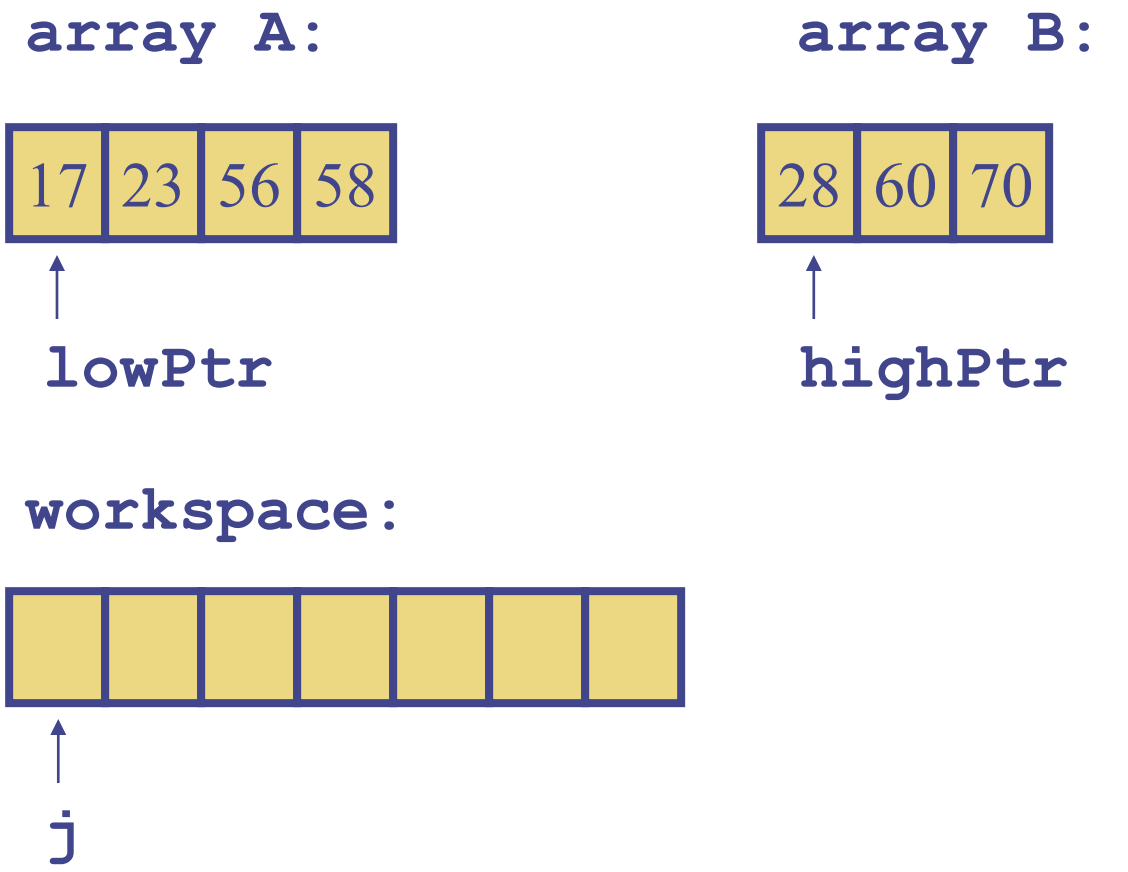
- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two disjoint subsets S_1 and S_2
  - Recur: solve the subproblems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S
- The base case for the recursion 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

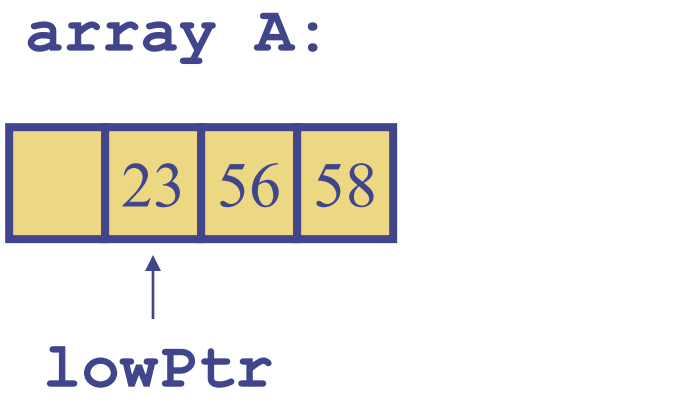
Merge-Sort

- Merge-sort on an input sequence (array/list) 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
- First questions:
  Is the merge easy?
  What is the big-Oh of the merge?

Merging sorted arrays

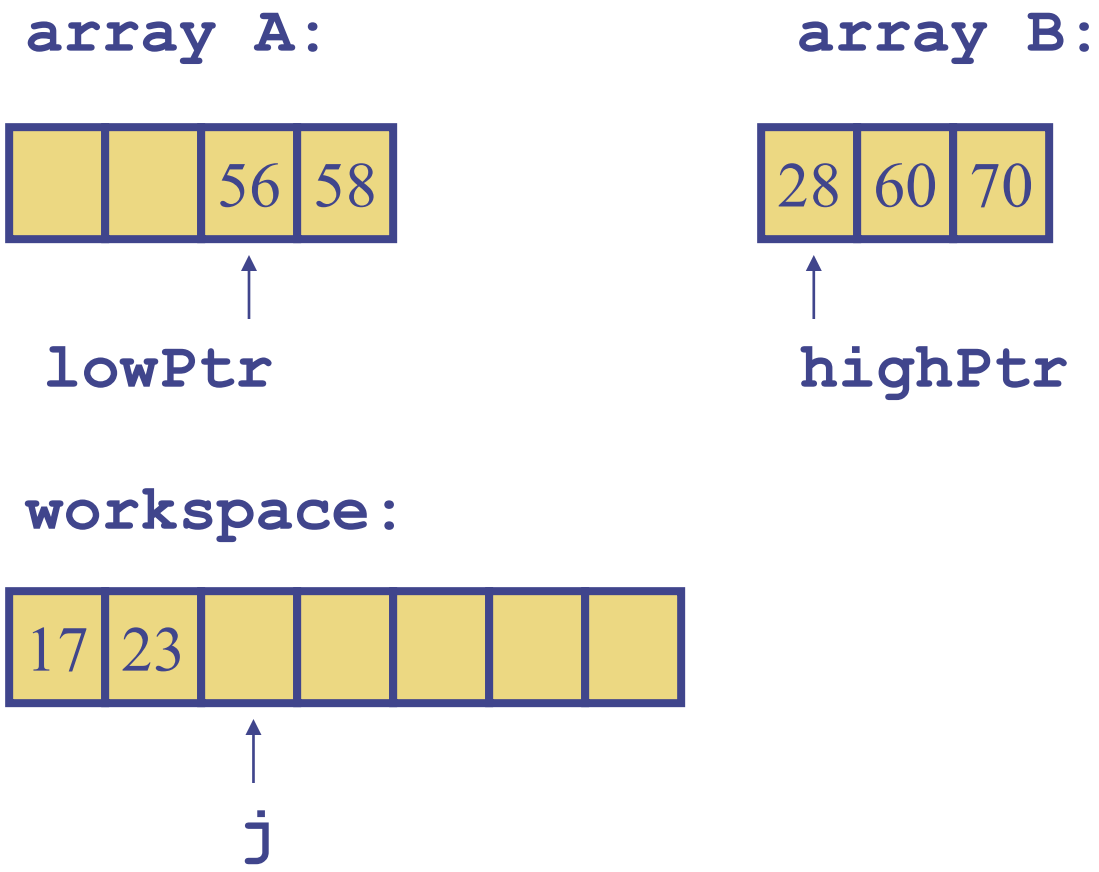


Merging sorted arrays

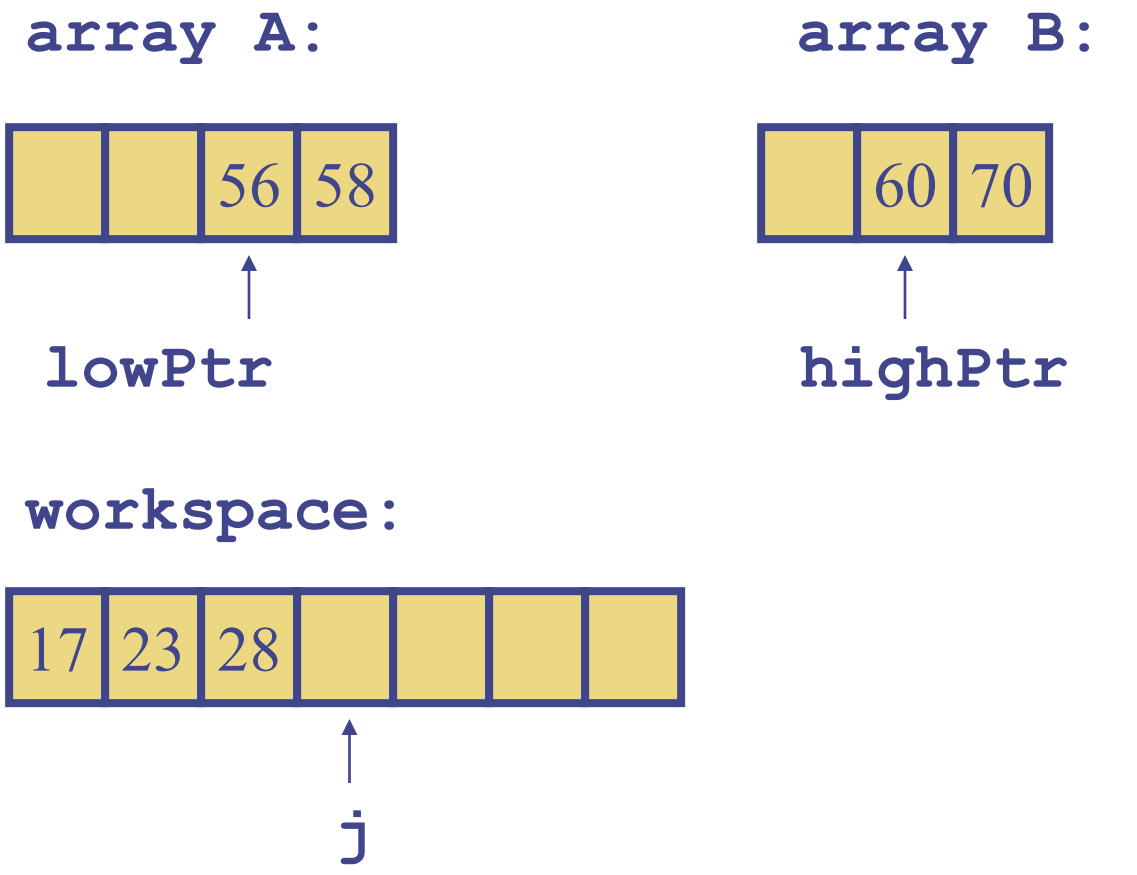


Notice the choice of 17 is only efficient and easy because the inputs are sorted and so we know we only need look at the first element!!

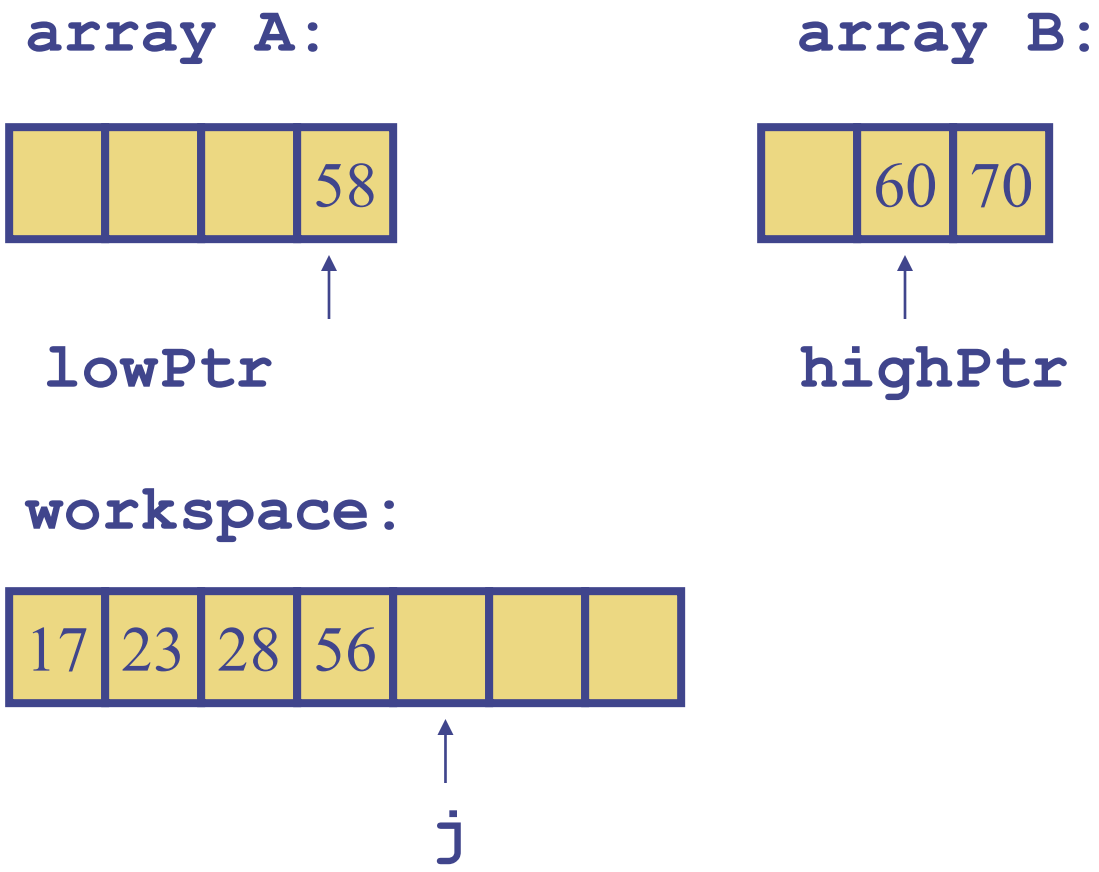
Merging sorted arrays



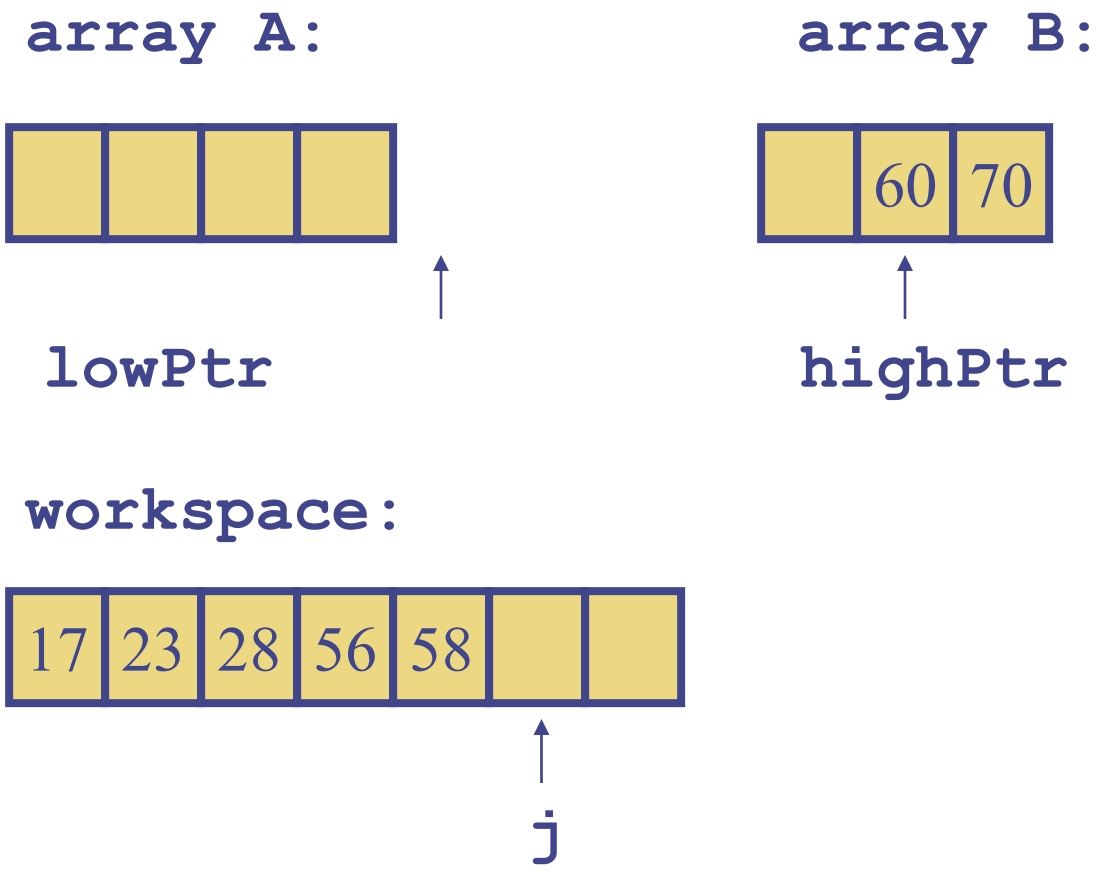
Merging sorted arrays



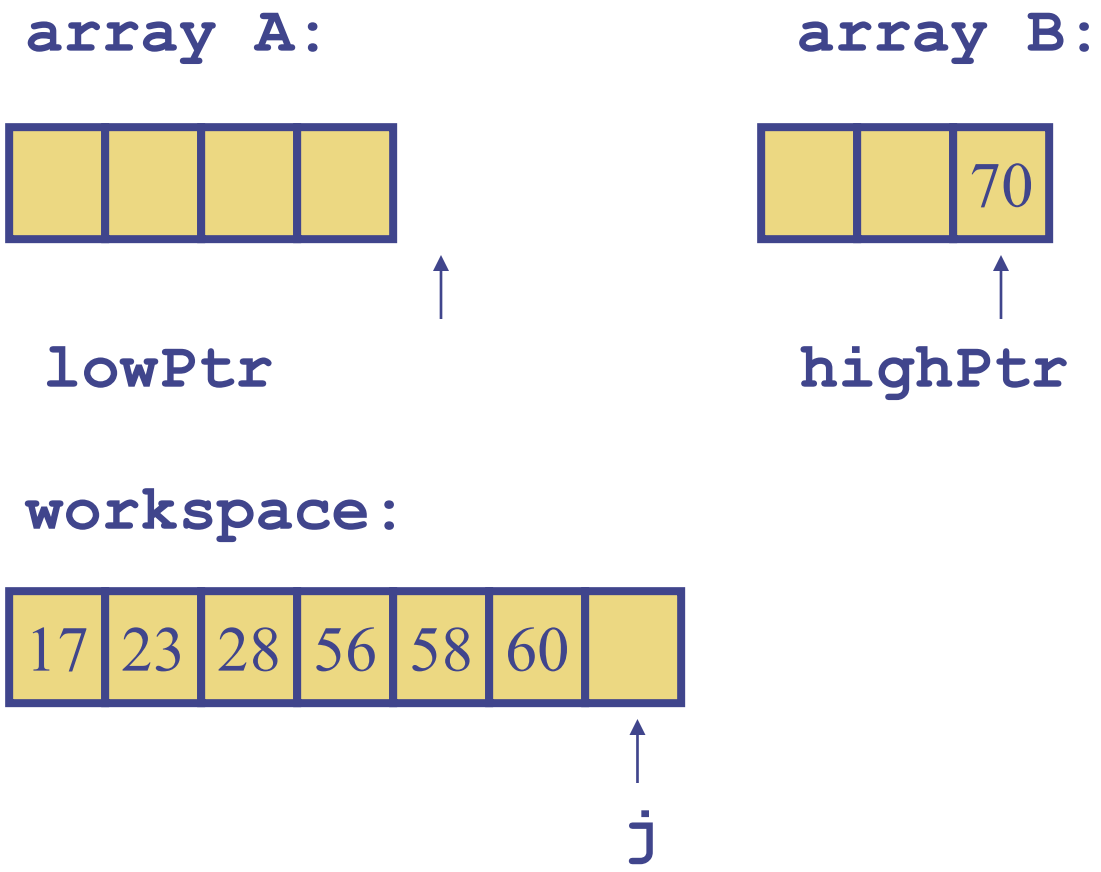
Merging sorted arrays



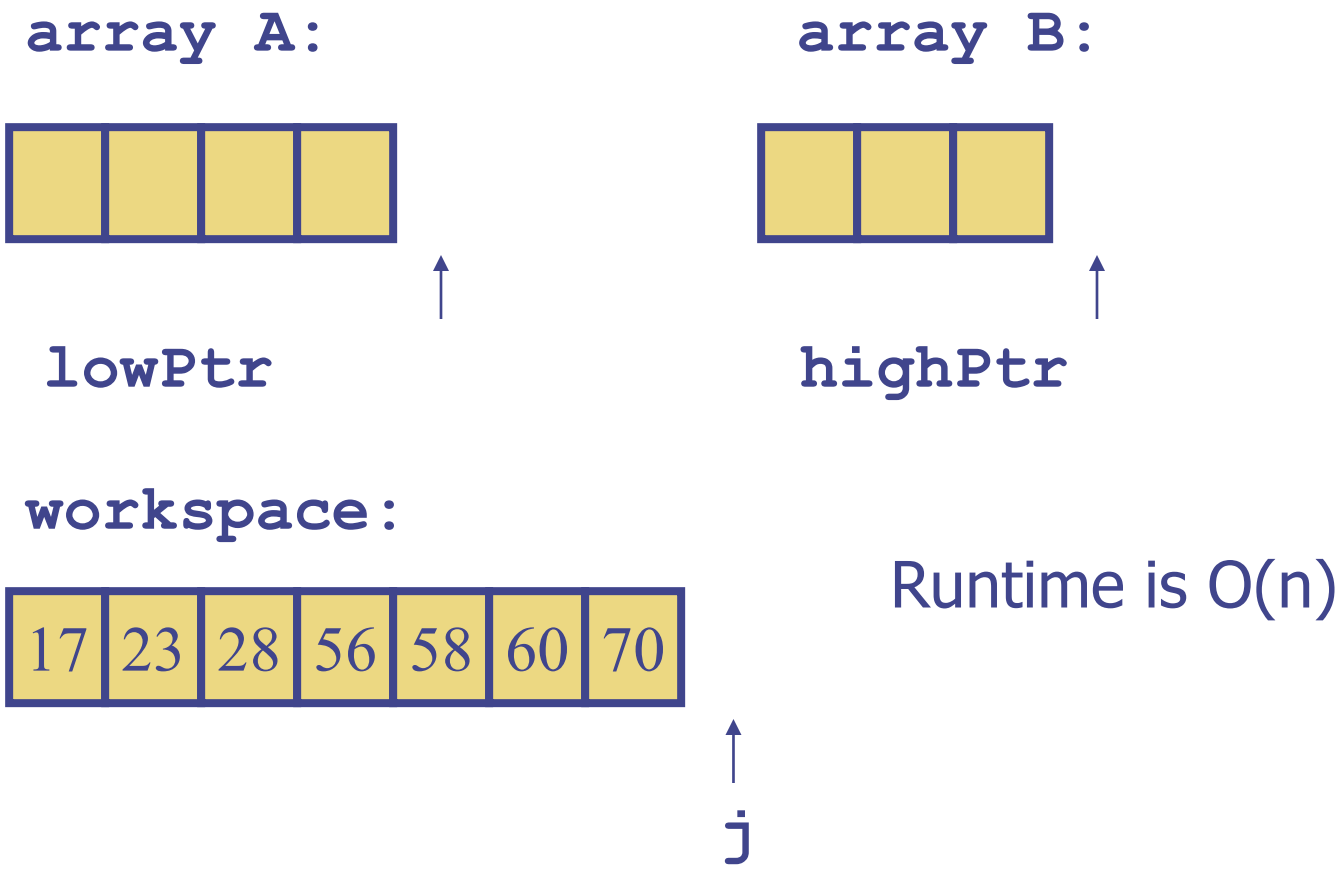
Merging sorted arrays



Merging sorted arrays



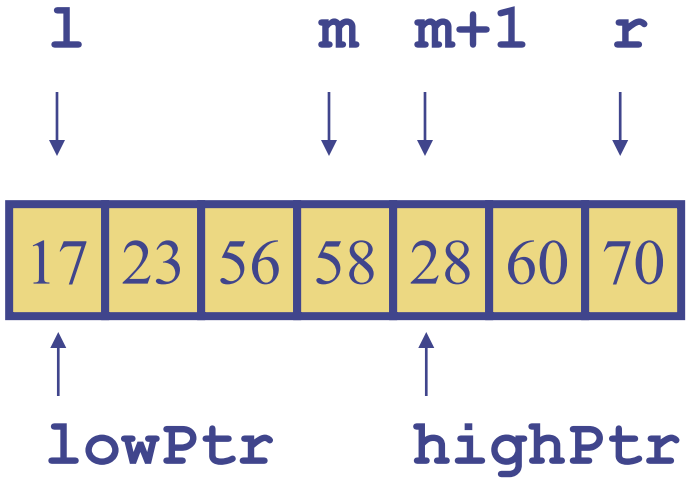
Merging sorted arrays



Runtime is O(n)

Merging halves of an array

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



Implementation

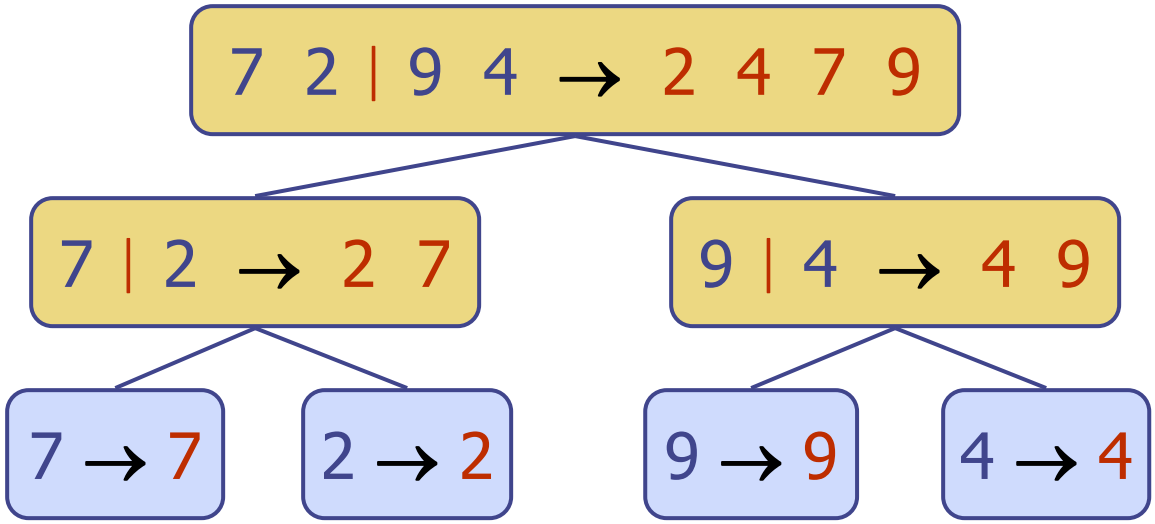
```
public static void recMergeSort
                    (int[] arr, int[] workSpace, int l, int r) {
    if (l == r) {
        return;
    } else {
        int m = (l+r) / 2;
        recMergeSort(arr, workSpace, l, m);
        recMergeSort(arr, workSpace, m+1, r);
        merge(arr, workSpace, l, m+1,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

Initial call is with l=0 and r to be end of the array

Merge-Sort Tree

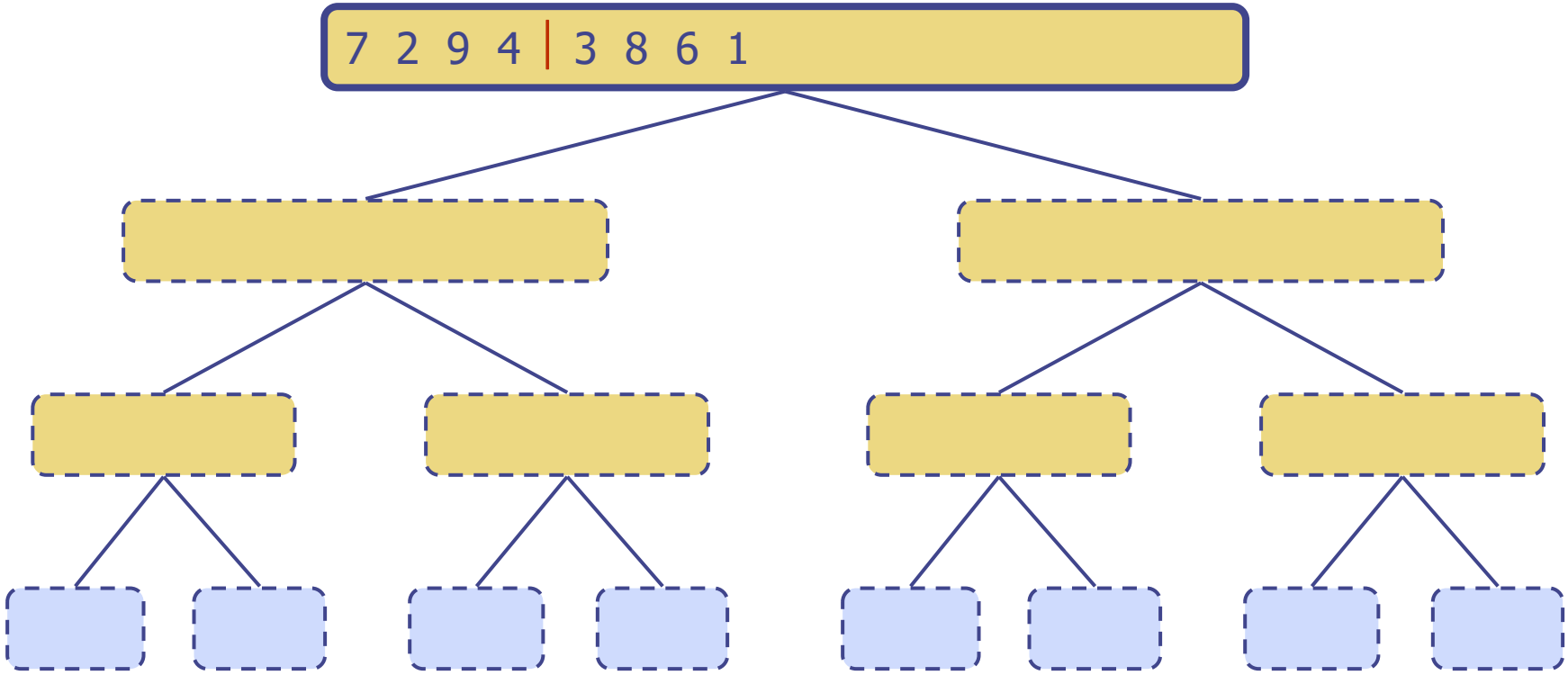
Not actually implemented this way! A `node' here is conceptual not real

- 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



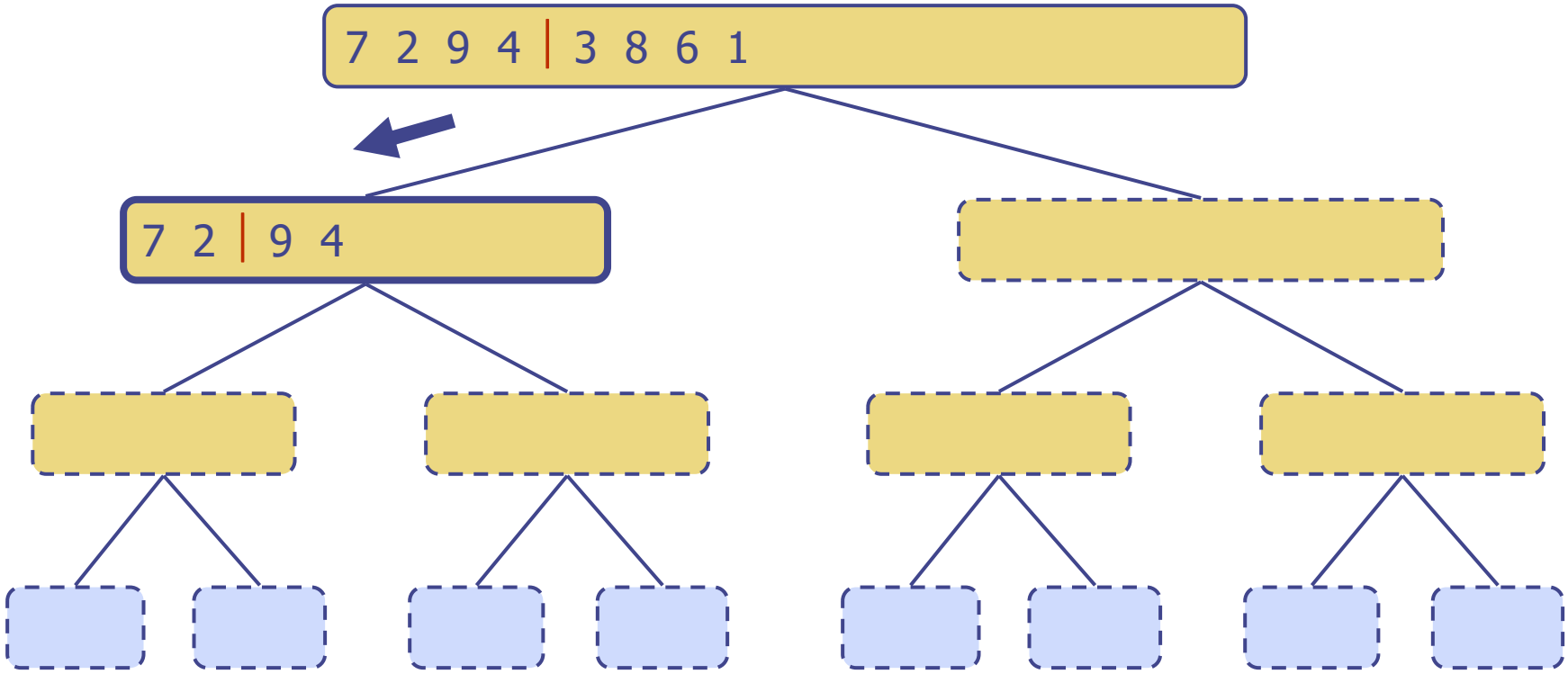
Execution Example

- Partition



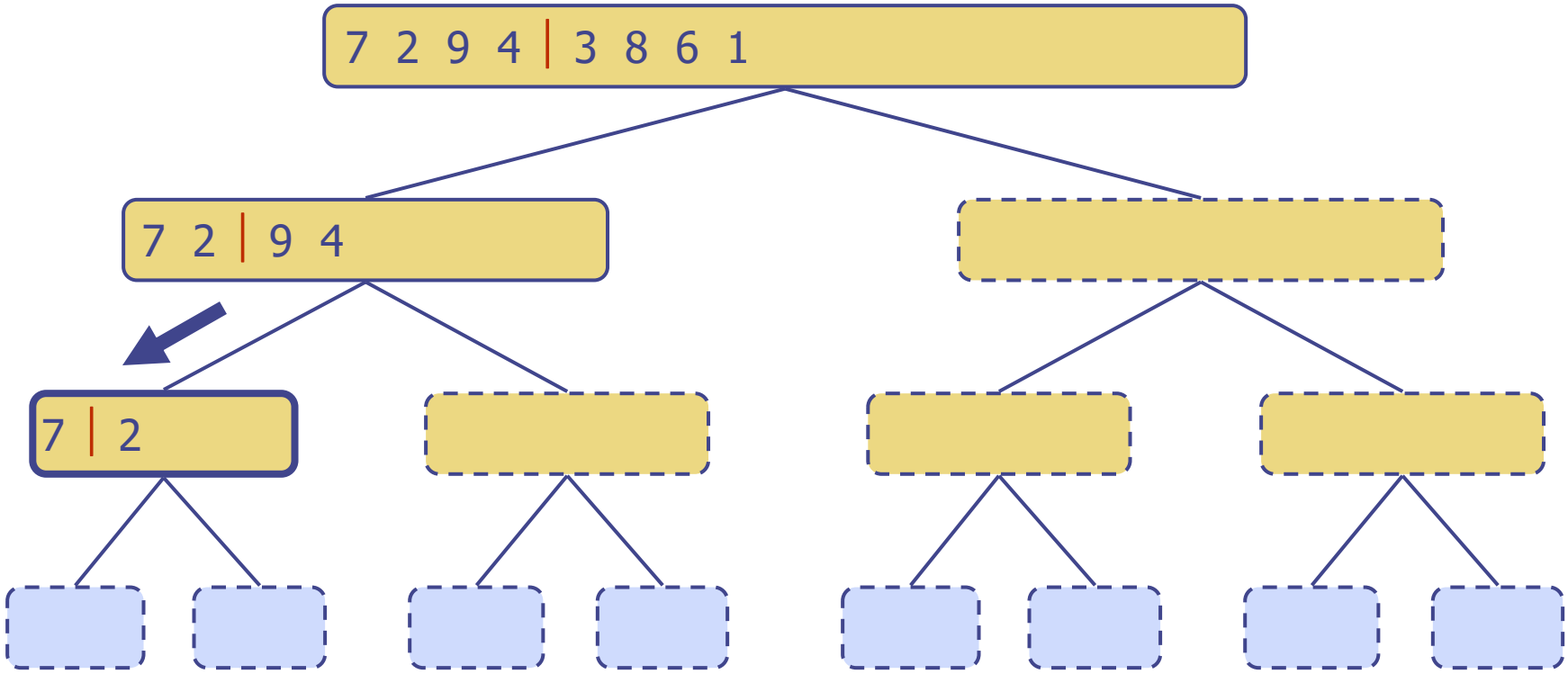
Execution Example (cont.)

- Recursive call, partition



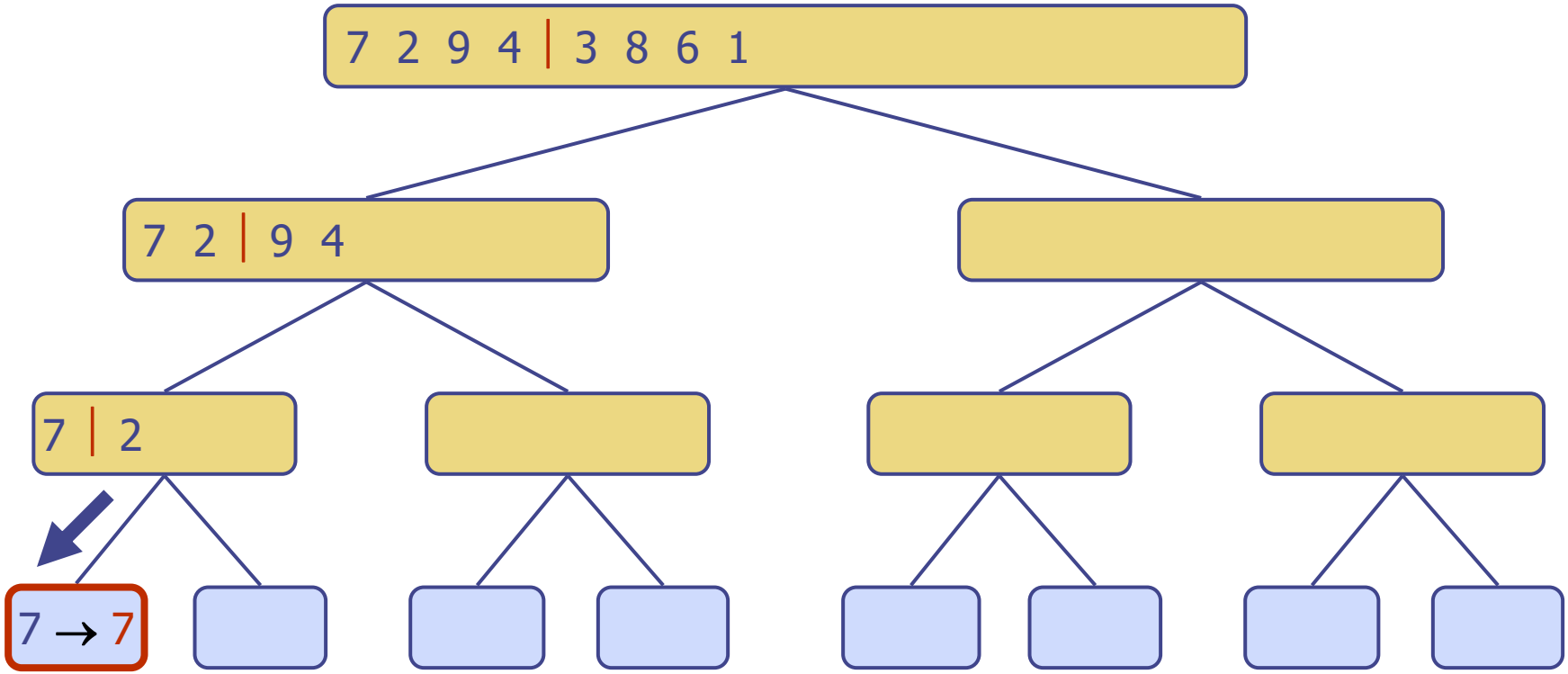
Execution Example (cont.)

- Recursive call, partition



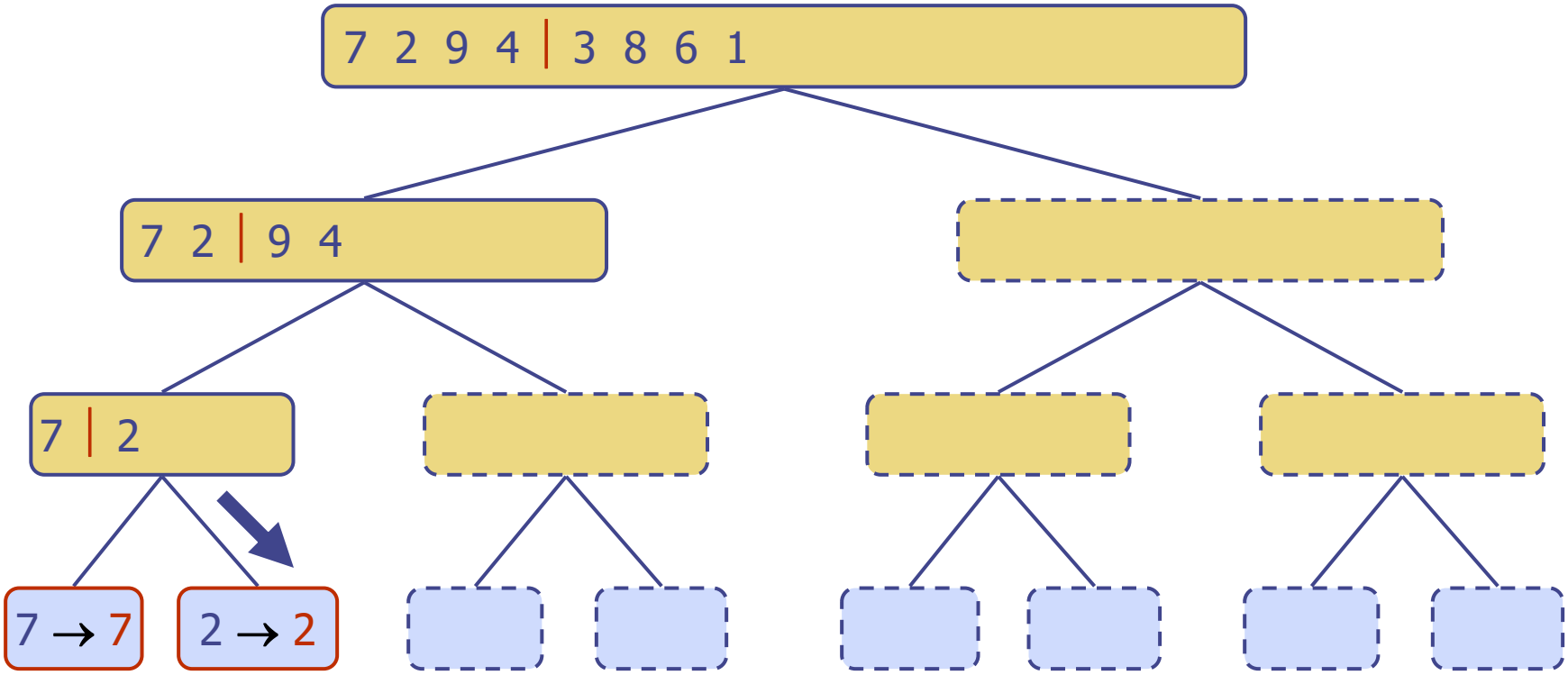
Execution Example (cont.)

- Recursive call, base case



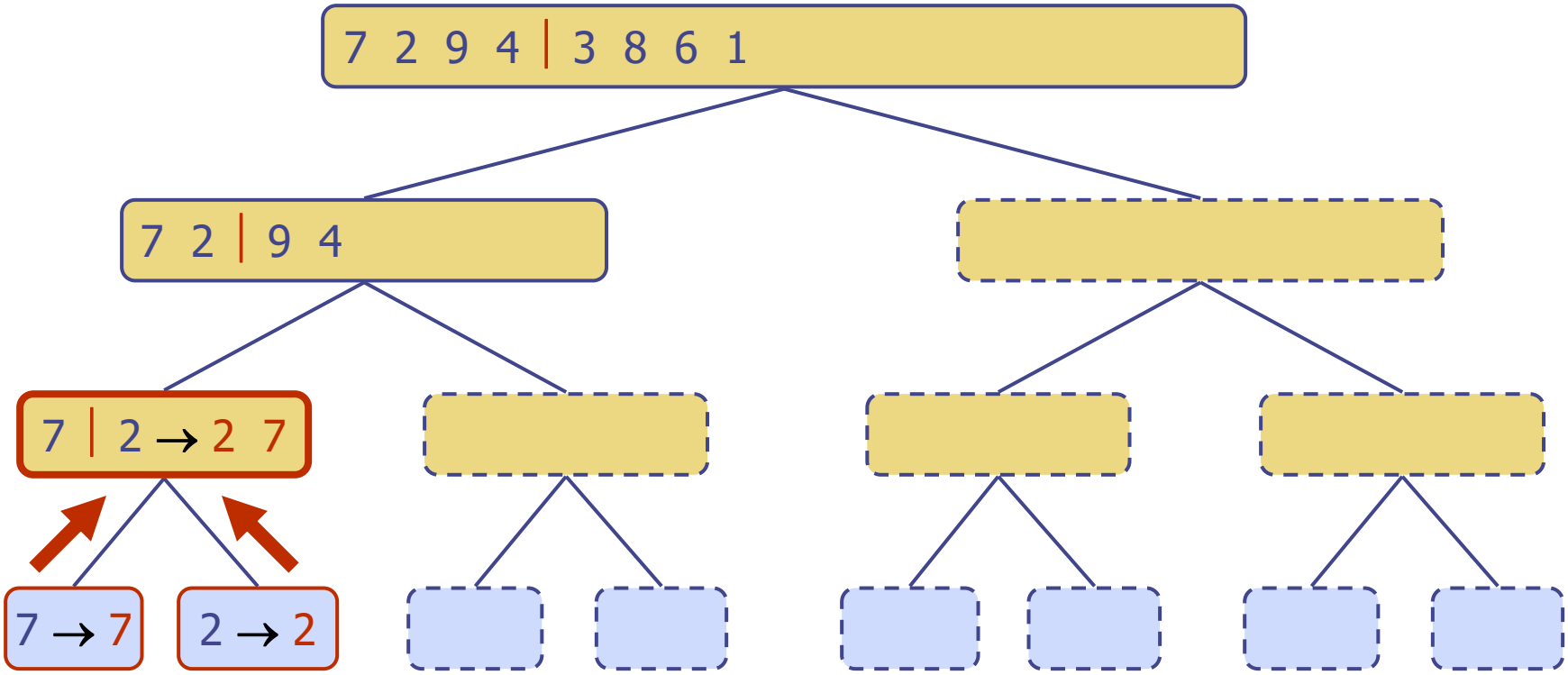
Execution Example (cont.)

- Recursive call, base case



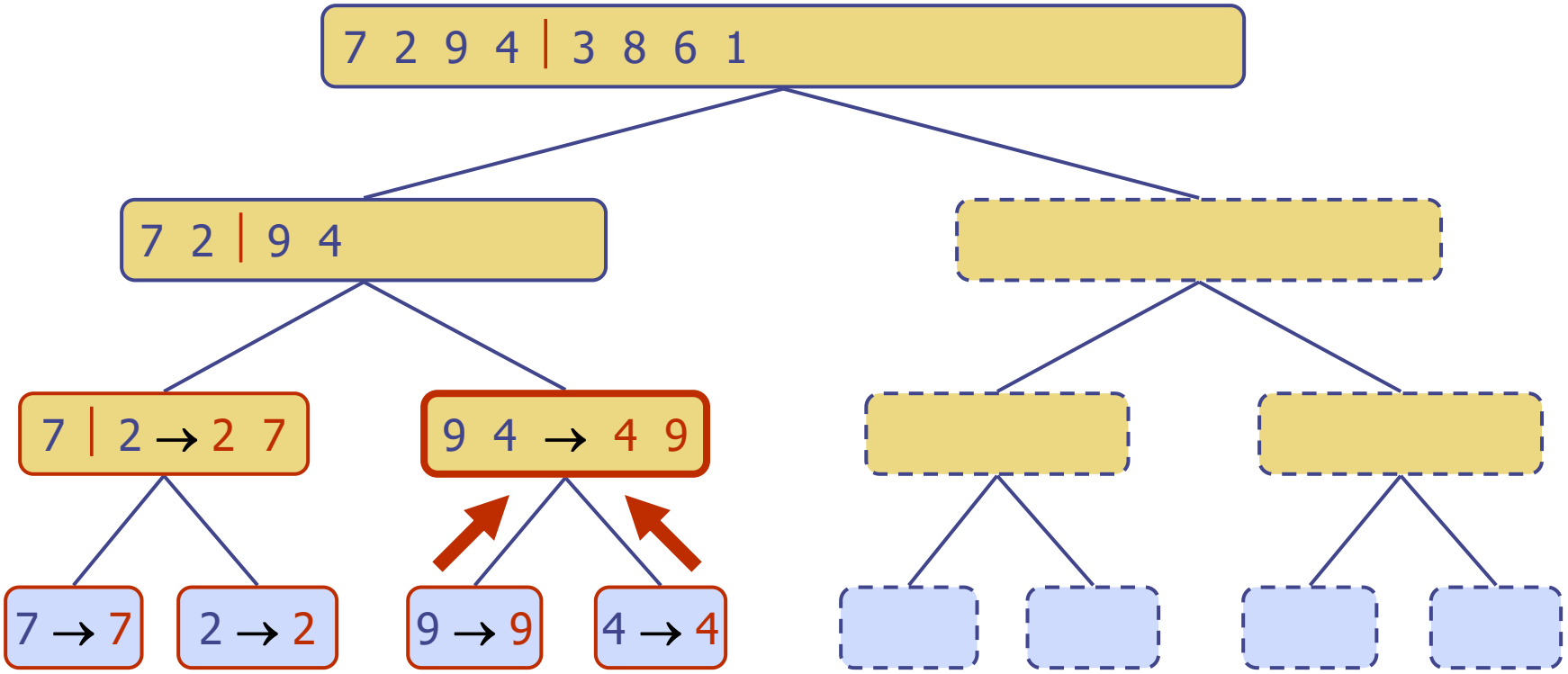
Execution Example (cont.)

- Merge



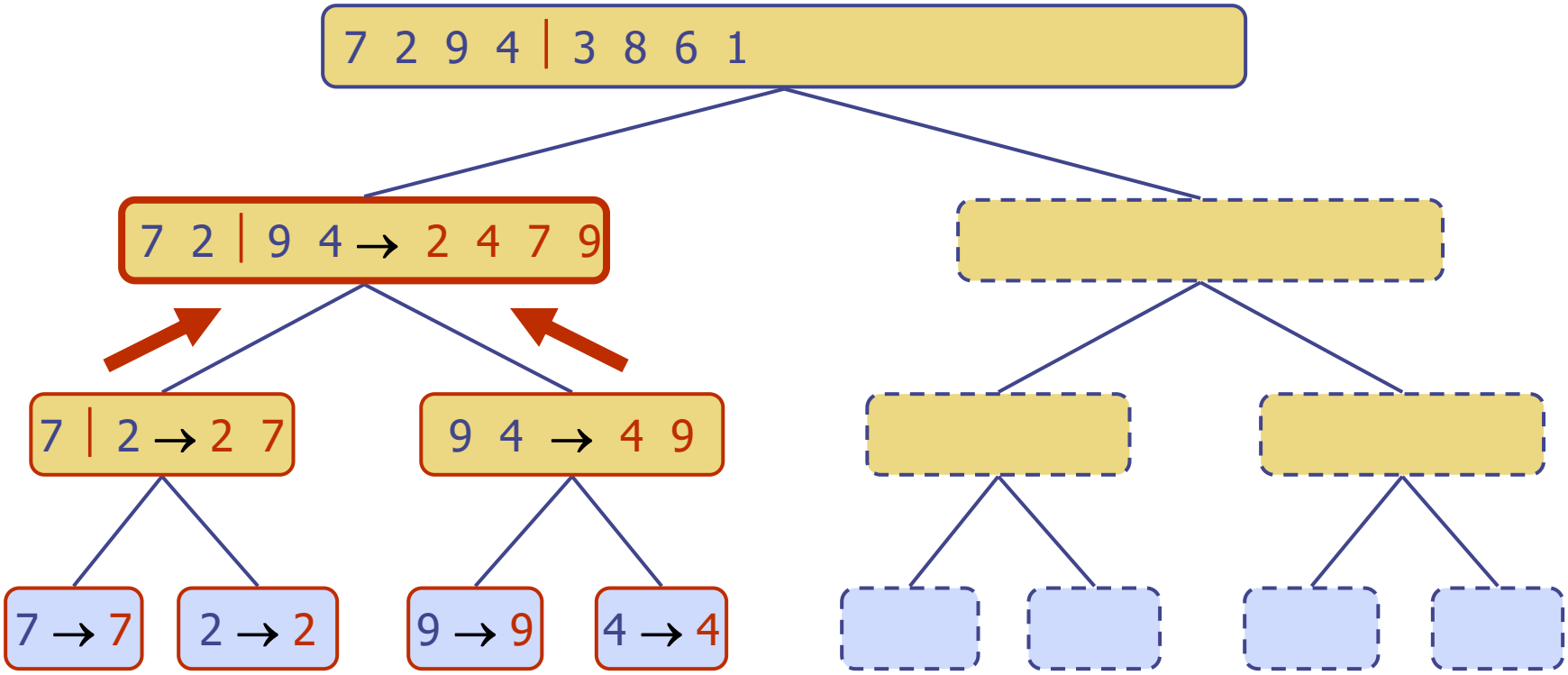
Execution Example (cont.)

- Recursive call, …, base case, merge



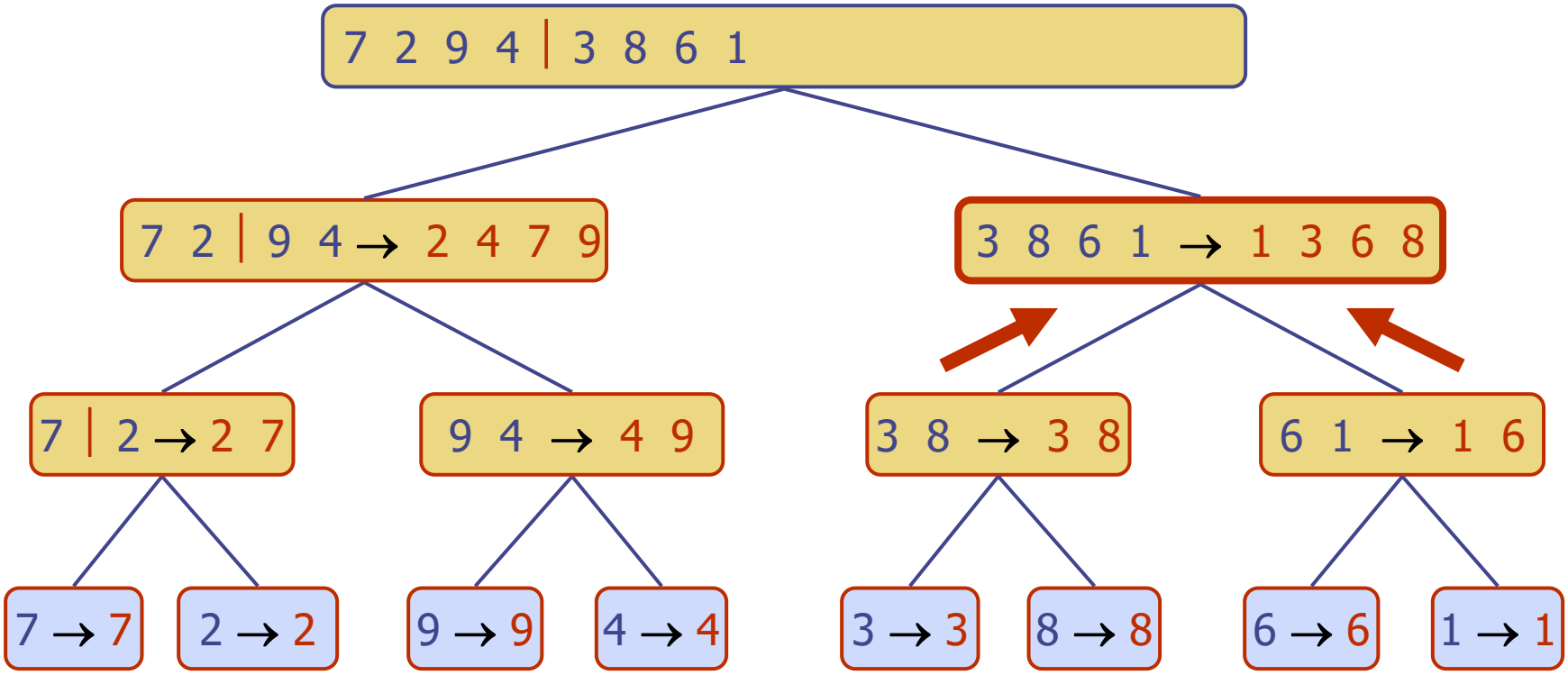
Execution Example (cont.)

- Merge



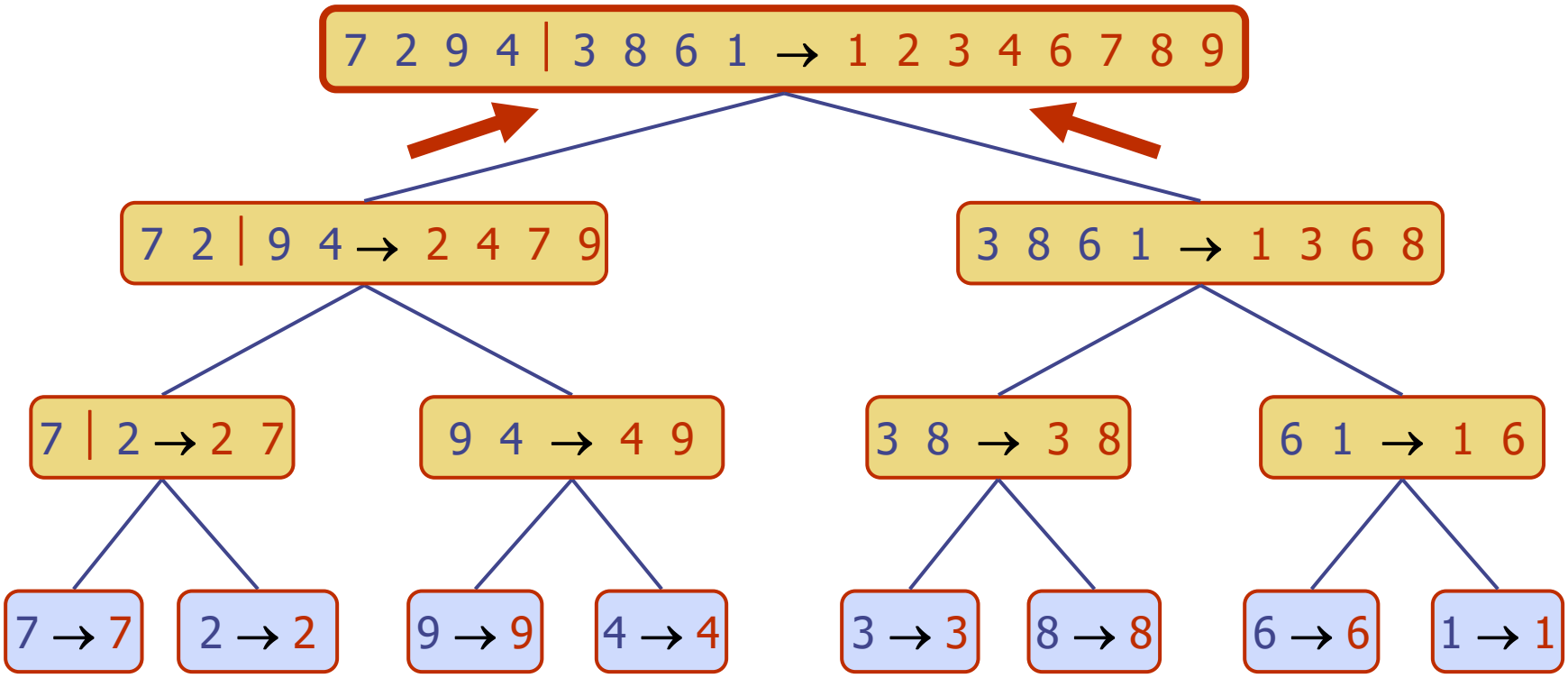
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

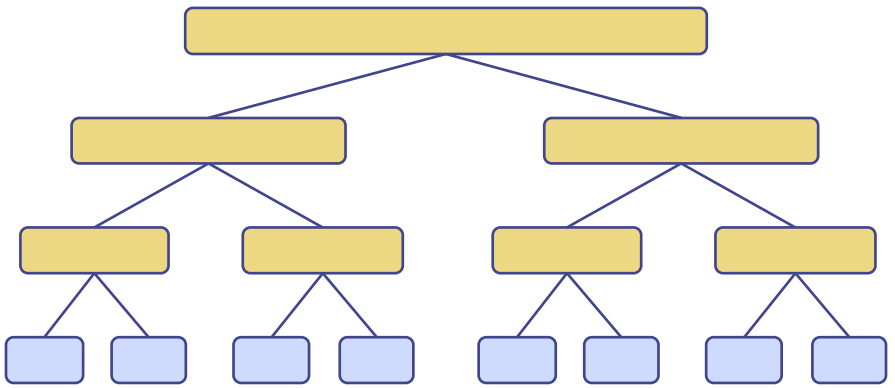
- Merge



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 all "used" at each depth
    - So, each depth uses O(n) work
- 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

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?
- "Dynamic sorting" – how well does it do if the data is already "nearly sorted"
- Data access patterns?
  - Sequential? Random Access?
- Relevant and appropriate assertions

Aim to understand these issues for various sorting algorithms.

Minimum Expectations

- Know the Mergesort algorithm and how it works on examples
- Know and be able to justify/prove the big-Oh family of behaviours of Mergesort