Design and Analysis of Algorithms

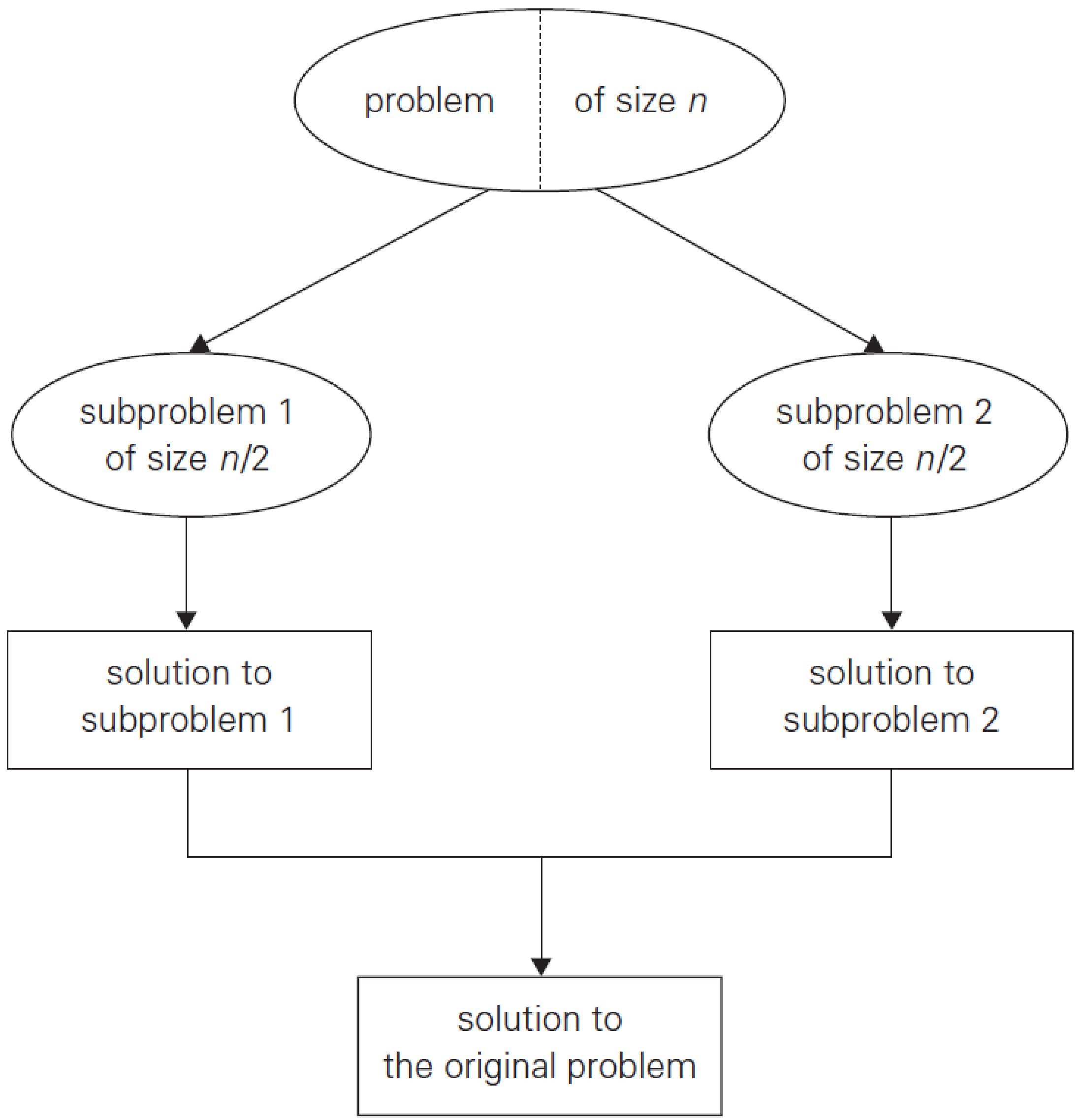
Algorithm Design Techniques

- **Brute Force and Exhaustive Search**
- **Divide-and-Conquer**
- **Decrease-and-Conquer**
- Transform-and-Conquer
- Space and Time Trade-Offs
- Dynamic Programming
- Greedy Technique
- Iterative Improvement
- Backtracking
- Branch-and-Bound

Divide-and-Conquer

- Divide-and-conquer algorithms work according to the following general plan:
  1. A problem is divided into several subproblems of the same type, ideally of about equal size.
  2. The subproblems are solved recursively.
  3. The solutions to the subproblems are combined to get a solution to the original problem.

Divide-and-Conquer



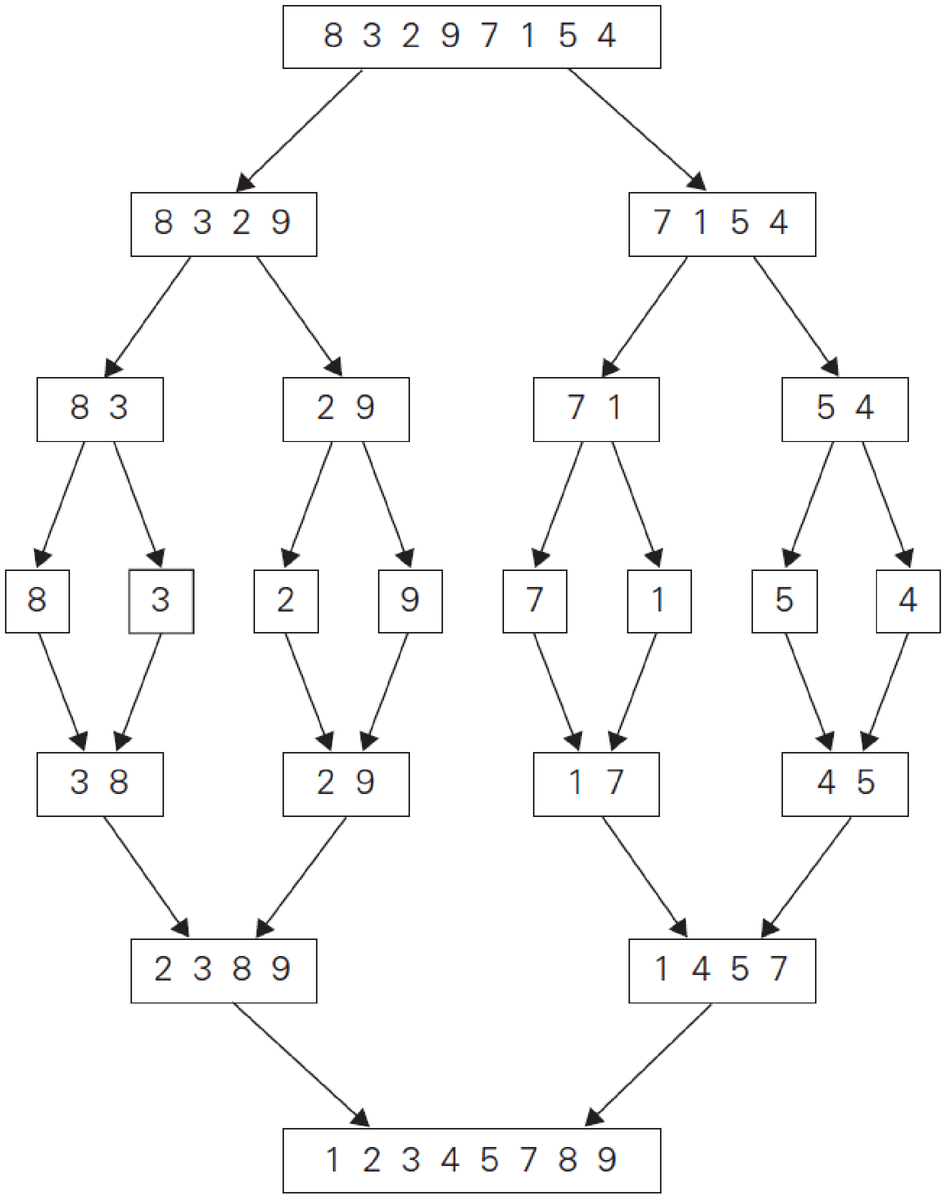
Divide-and-Conquer

- Divide-and-conquer is probably the best-known general algorithm design technique.
- Thus, not every divide-and-conquer algorithm is necessarily more efficient than even a brute-force solution.
- In fact, the divide-and-conquer approach yields some of the most important and efficient algorithms in computer science.
- The divide-and-conquer technique is ideally suited for parallel computations, in which each subproblem can be solved at the same time by its own processor.

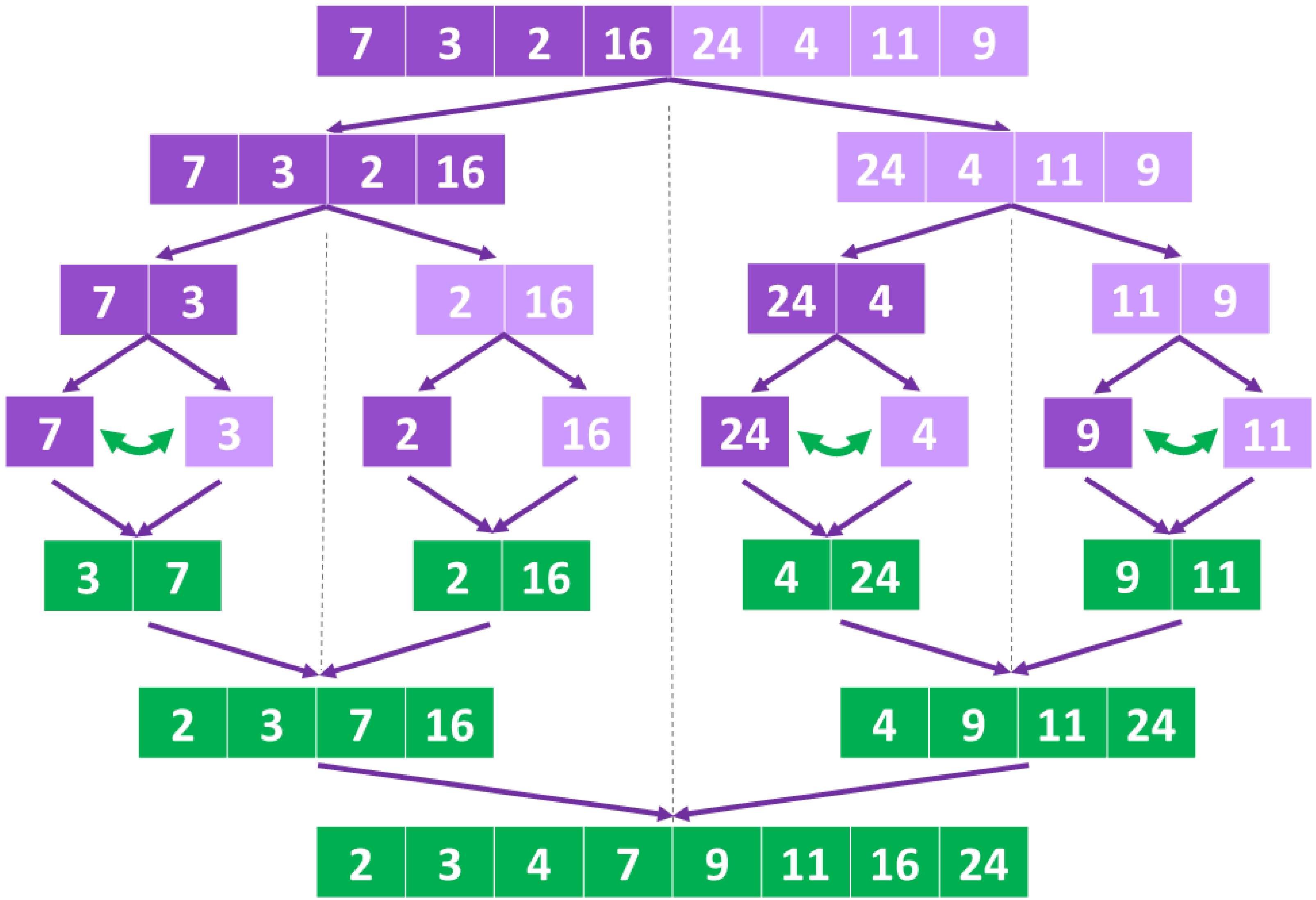
Mergesort

- Mergesort is a perfect example of a successful application of the divide-and-conquer technique.
- It sorts a given array A[0 .. $n-1$] by dividing it into two halves A[0 .. $n/2-1$] and A[$n/2$ .. $n-1$], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.
- The trick is taking advantage of the two partial solutions to construct a solution of the full problem, as we did with the merge operation.

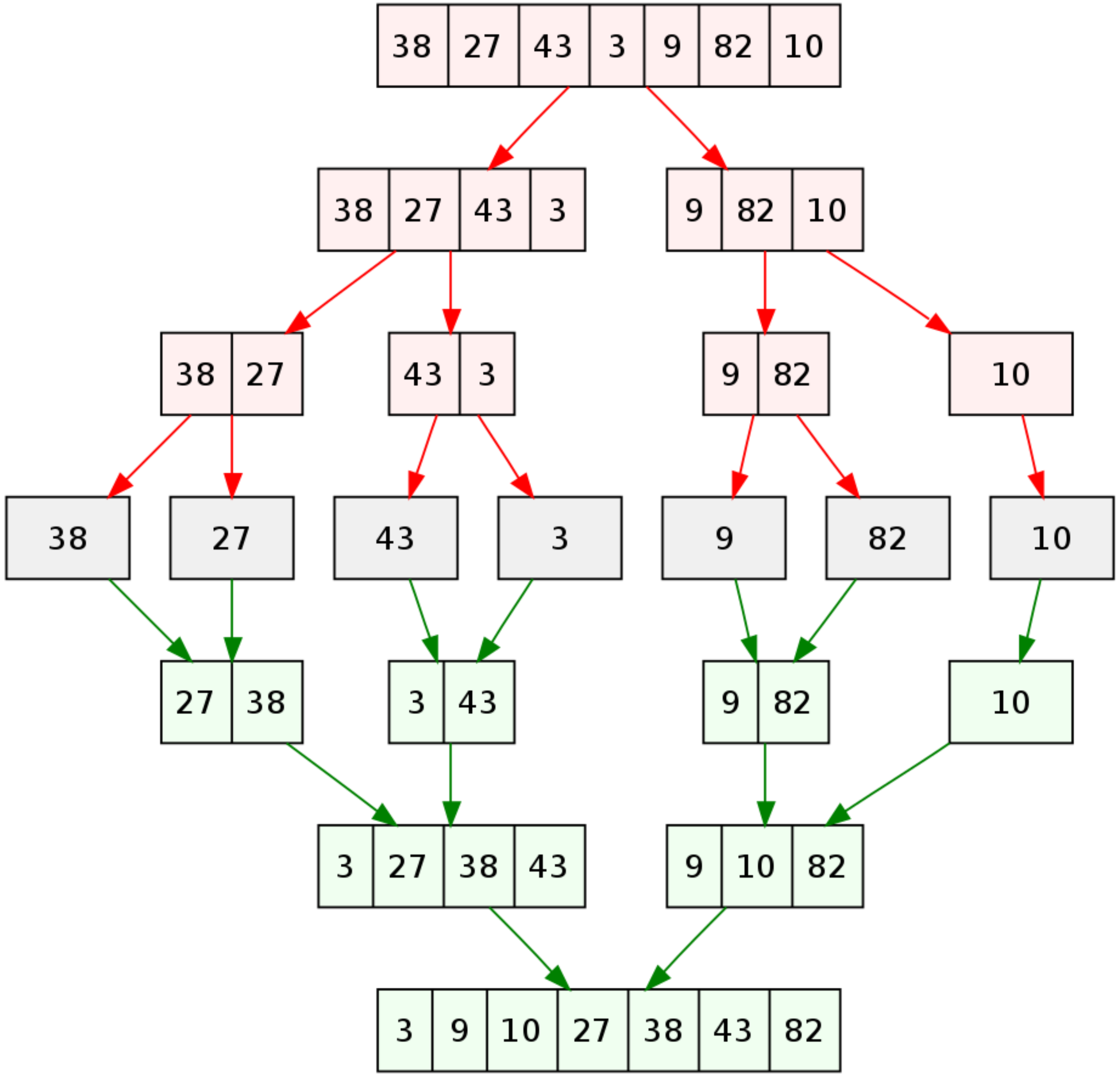
Mergesort: Example 1



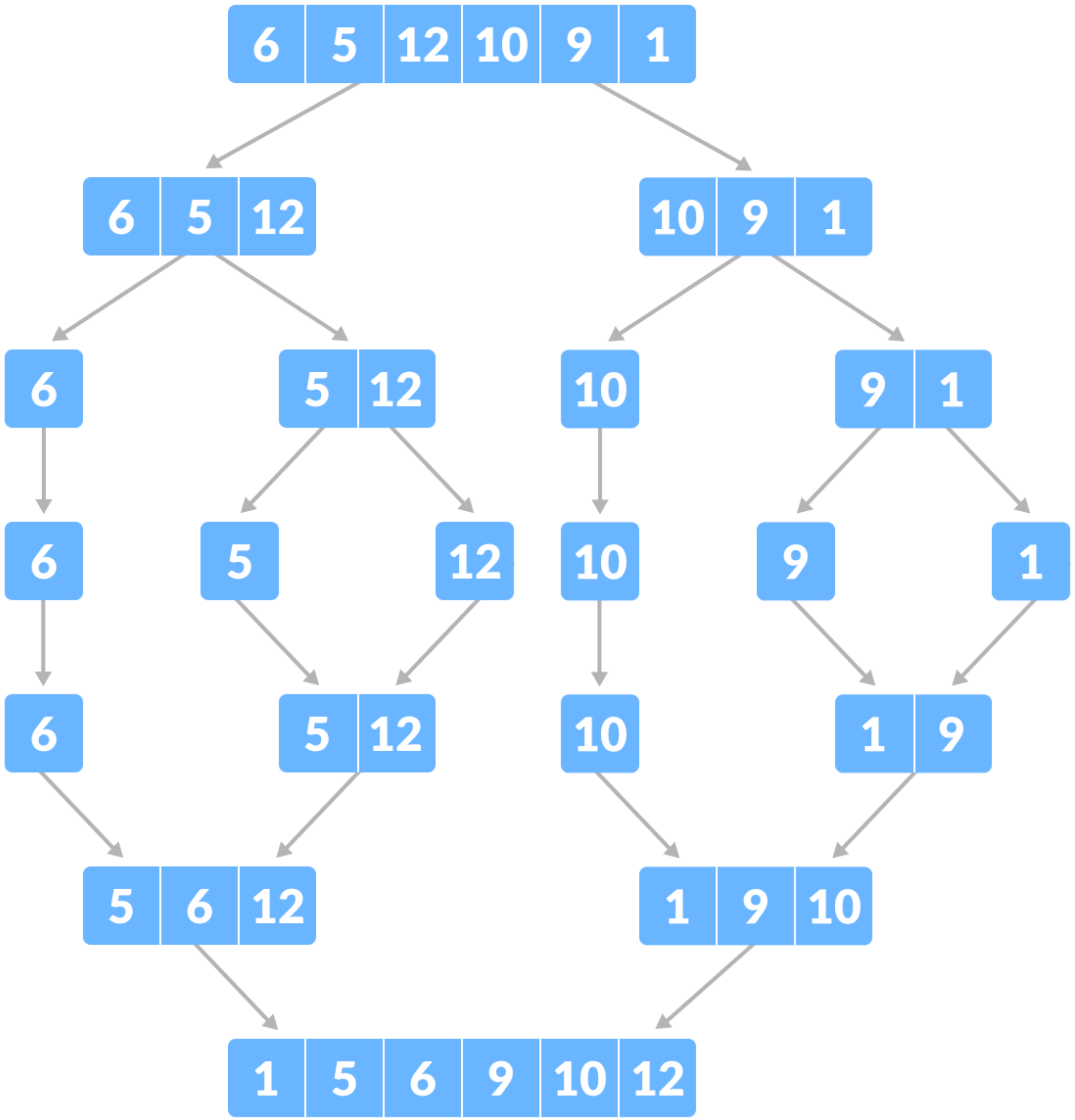
Mergesort: Example 2



Mergesort: Example 3



Mergesort: Example 4



Mergesort: Pseudocode

ALGORITHM Mergesort(A, *low*, *high*)

//Sorts array A[*low* .. *high*] by recursive mergesort

//Input: An array A[*low* .. *high*] of orderable elements

//Output: Array A[*low* .. *high*] sorted in nondecreasing order

if *low* < *high*

mid = (*low* + *high*) / 2

Mergesort(A, *low*, *mid*)

Mergesort(A, *mid* + 1, *high*)

Merge(A, *low*, *mid*, *high*)

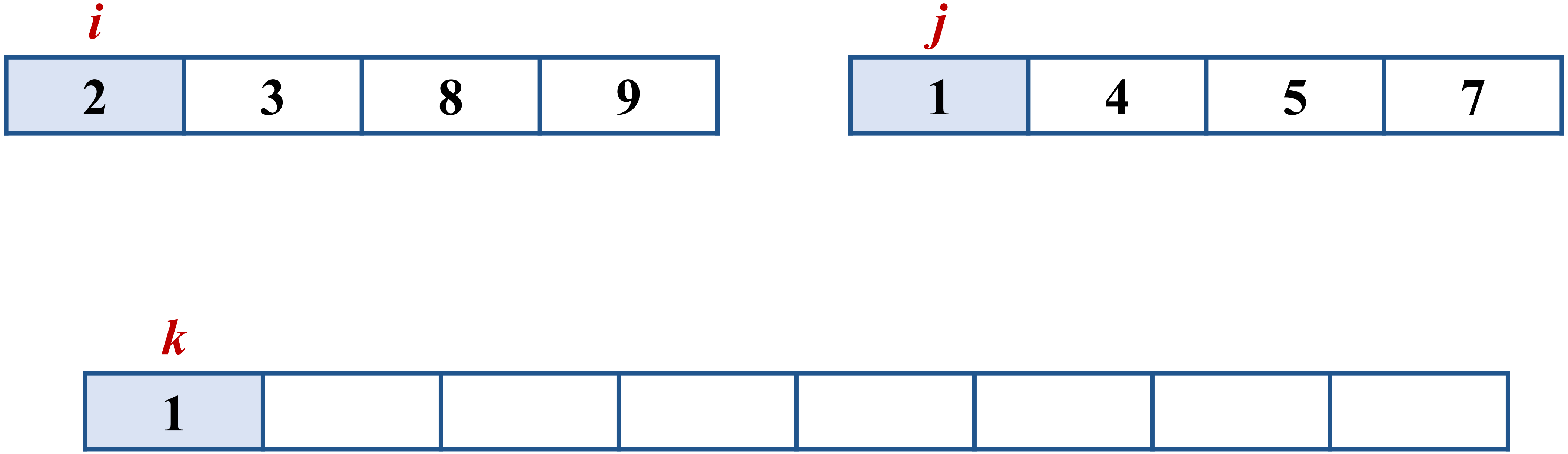
Mergesort: Merge

i			
2	**3**	**8**	**9**

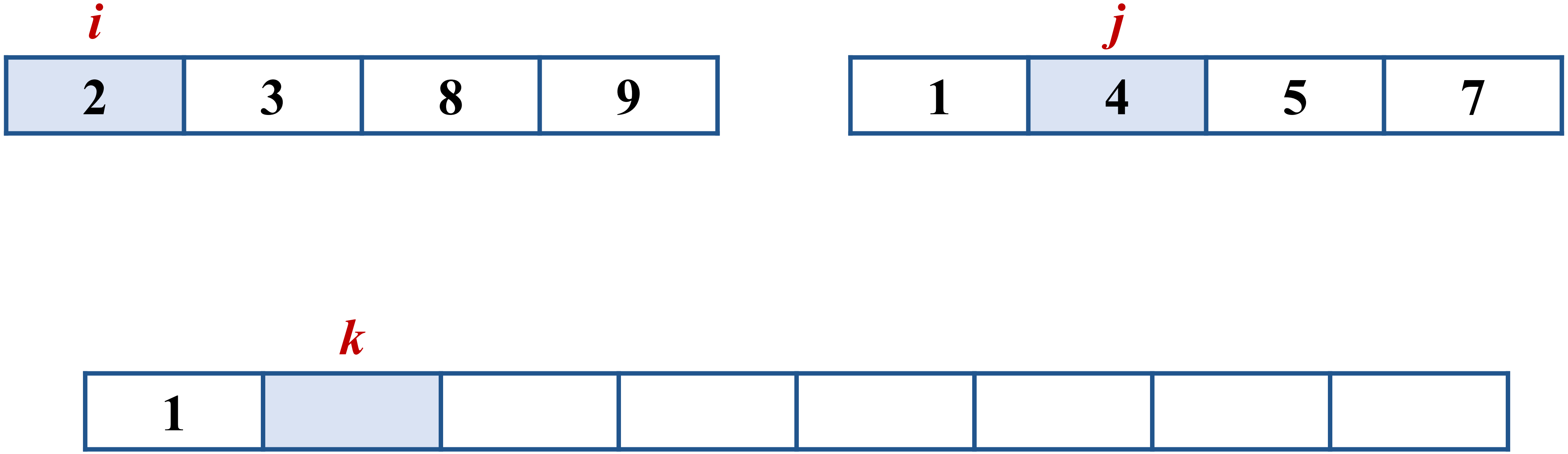
j			
1	**4**	**5**	**7**

k							

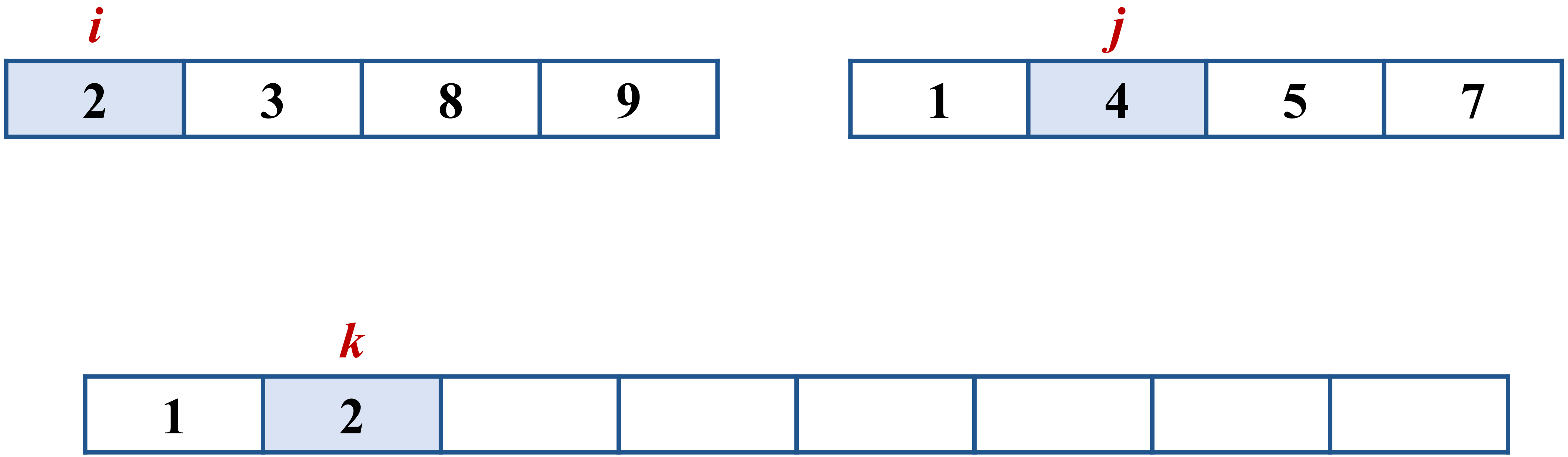
Mergesort: Merge



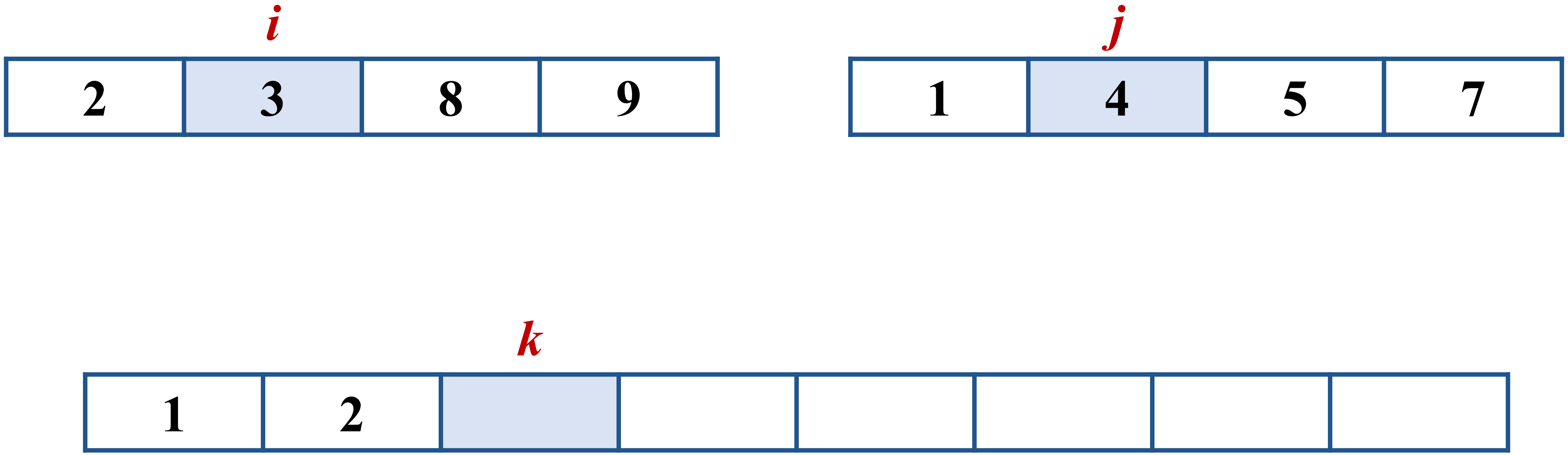
Mergesort: Merge



Mergesort: Merge



Mergesort: Merge



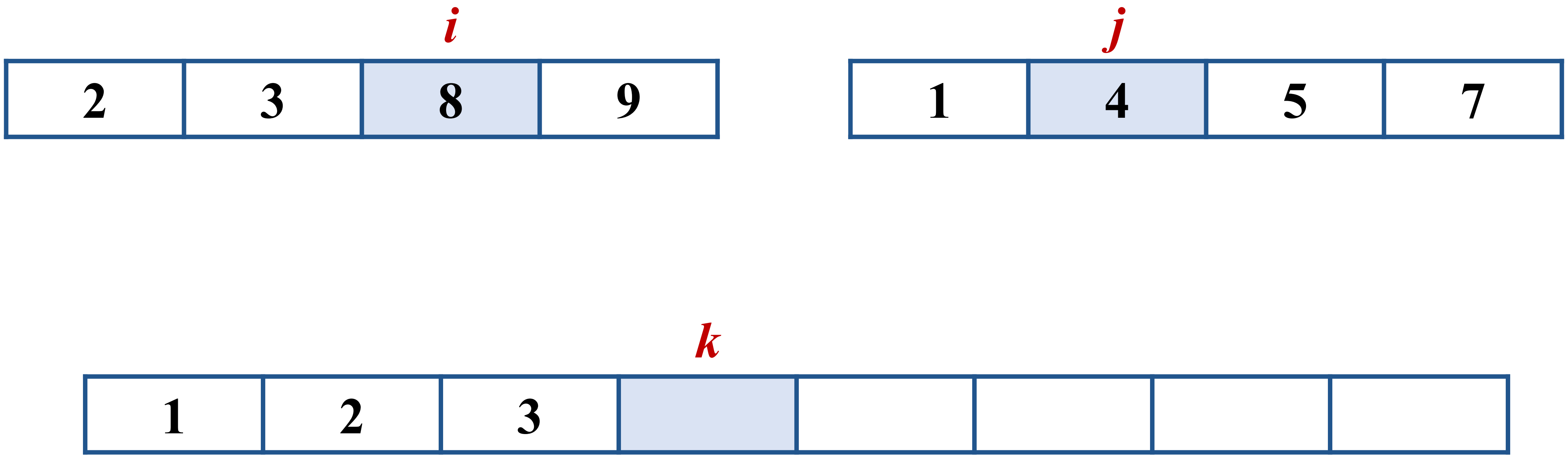
Mergesort: Merge

	i		
2	3	8	9

	j		
1	4	5	7

		k					
1	2	3					

Mergesort: Merge



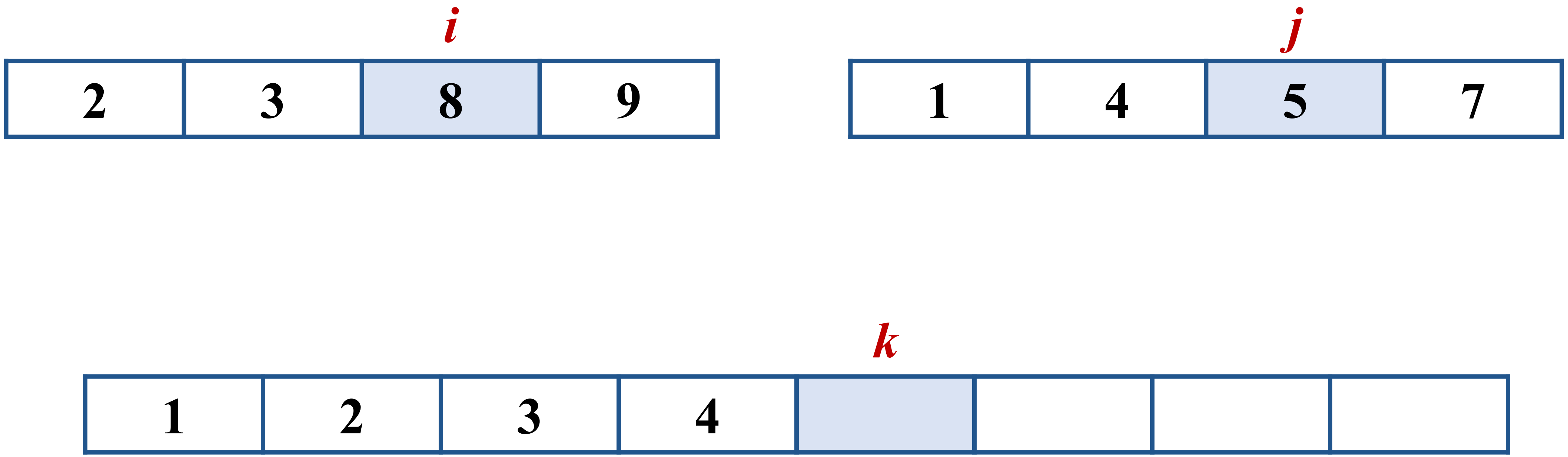
Mergesort: Merge

		i	
2	**3**	**8**	**9**

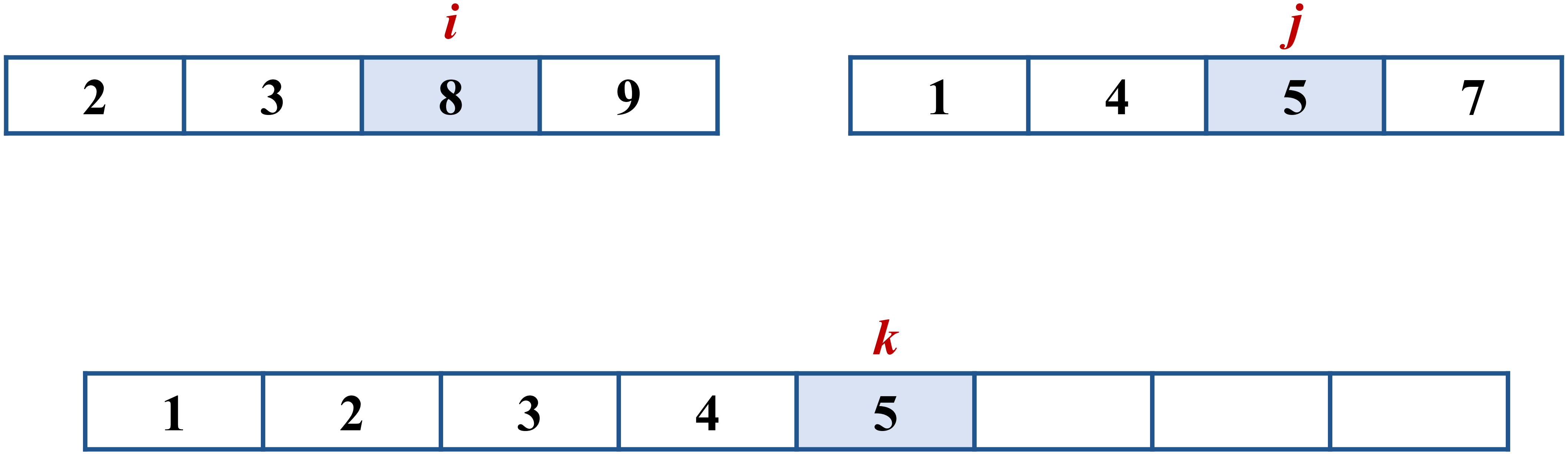
	j		
1	**4**	**5**	**7**

			k				
1	**2**	**3**	**4**				

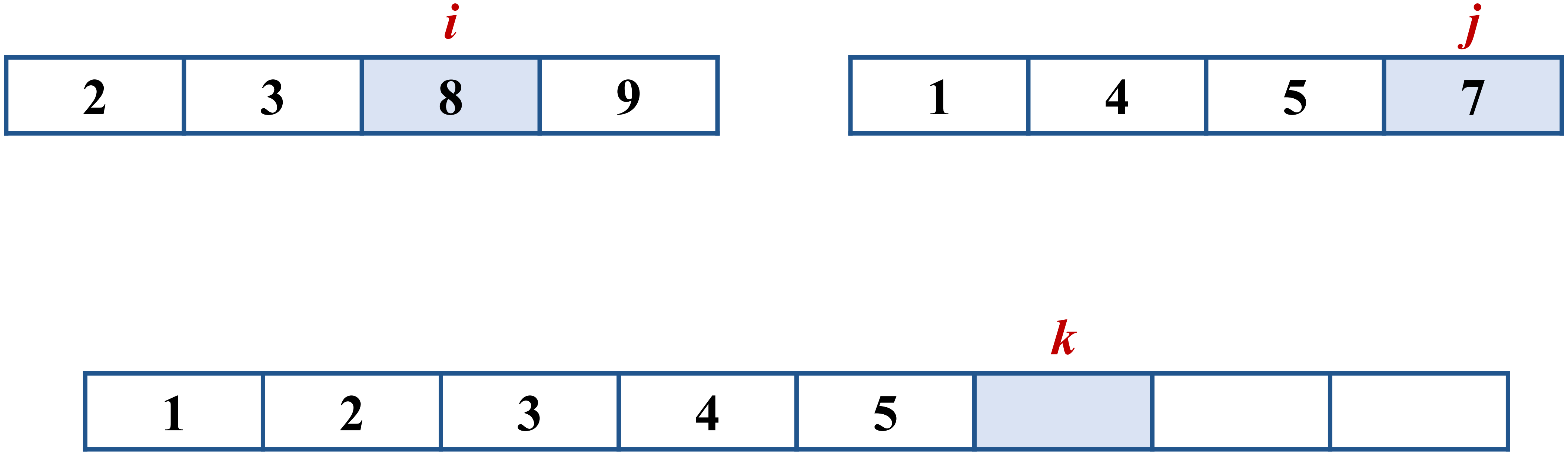
Mergesort: Merge



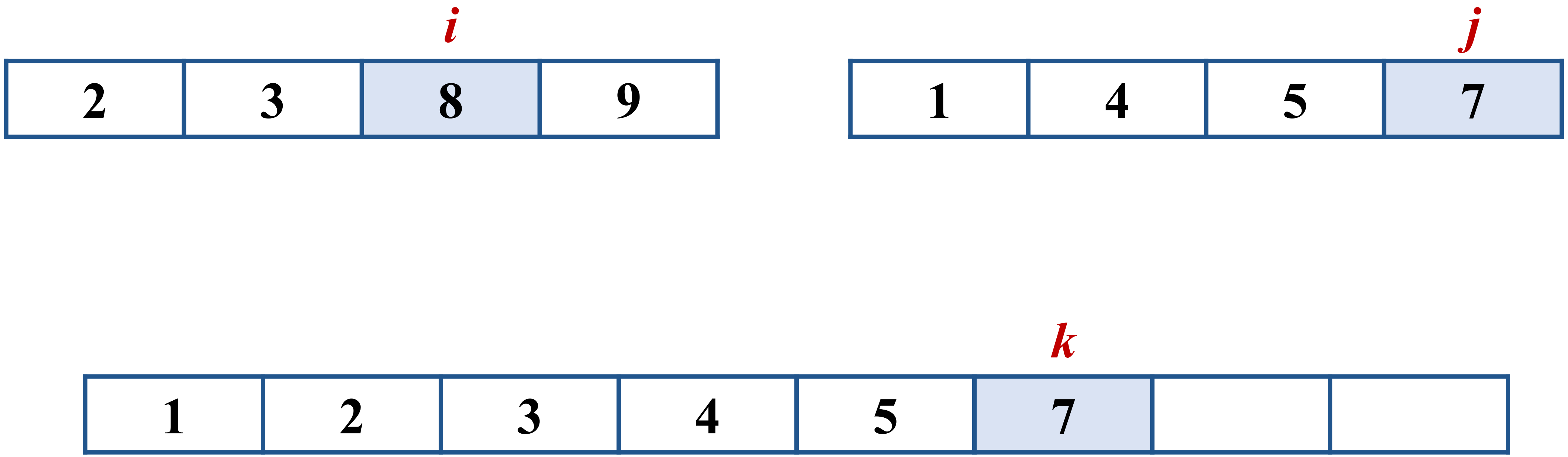
Mergesort: Merge



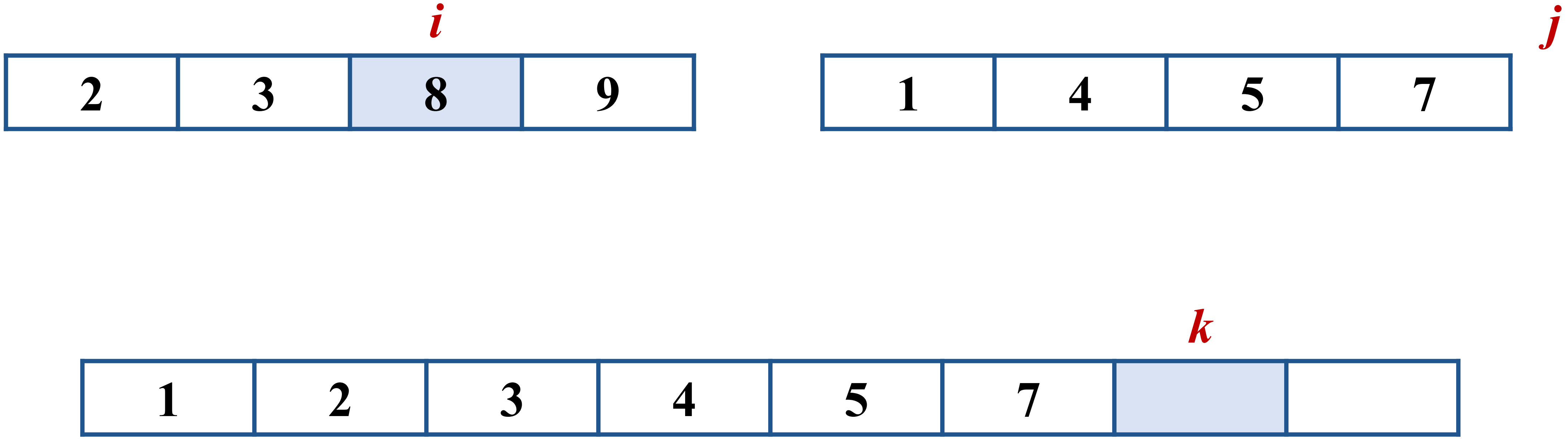
Mergesort: Merge



Mergesort: Merge



Mergesort: Merge



Mergesort: Merge

		i						*j*
2	3	8	9		1	4	5	7

						k	
1	2	3	4	5	7	8	

Mergesort: Merge

			i	
2	3	8	9	

				j
1	4	5	7	

							k
1	2	3	4	5	7	8	

Mergesort: Merge

			i
2	**3**	**8**	**9**

				j
1	**4**	**5**	**7**	

							k
1	**2**	**3**	**4**	**5**	**7**	**8**	**9**

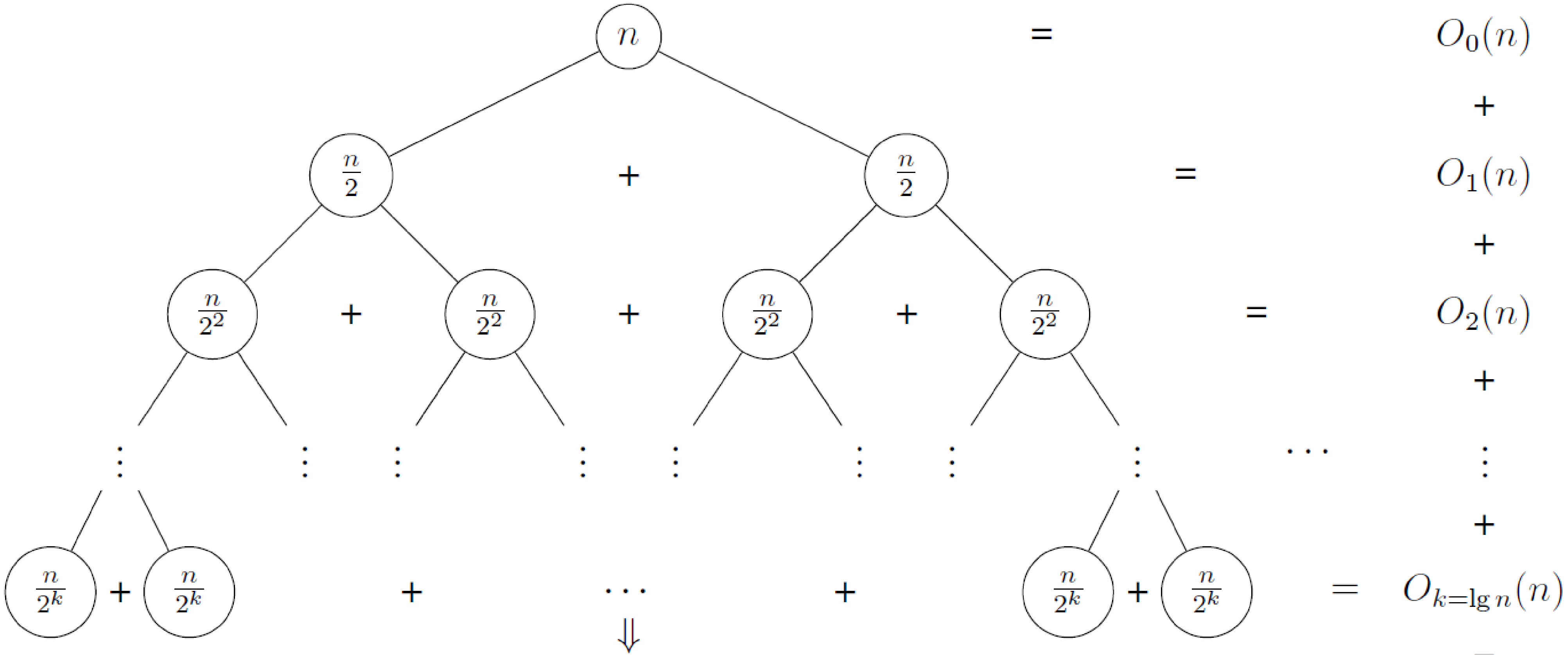
Analysis of Mergesort

- **Dividing** takes constant time, because it amounts to just computing the index *mid*.
- **Conquering** consists of the two recursive calls on subarrays, each with $n/2$ elements.
  Each of the two recursive calls takes time $T(n/2)$.
- **Combining** the results of the two recursive calls by merging the sorted subarrays takes $O(n)$ time.
- The function $T(n)$ describes the running time of merge sort

$$T(n) = 2T(n/2) + O(n)$$

Analysis of Mergesort: Recursion Tree

$$T(n) = 2T(n/2) + O(n)$$



$$\Downarrow$$

$$O\left(\sum_{i=0}^{k} 2^i \cdot \frac{n}{2^i}\right) = O\left(\sum_{i=0}^{k} n\right) = O(k \cdot n) \Leftrightarrow O(n \cdot \log n)$$

Analysis of Mergesort

- The merge sort has a running time of only $O(n \log_2 n)$ in all cases.
- When we compare its running time with the $O(n^2)$ worst-case running times of selection sort and insertion sort, we are trading a factor of n for a factor of only $\log_2(n)$.
- Merge sort does not work in place.
- It has to make complete copies of the entire input array.

Analysis of Mergesort

Time Complexity	
Best	O(n*log n)
Worst	O(n*log n)
Average	O(n*log n)
Space Complexity	O(n)
Stability	Yes

Sorting Algorithms

Algorithm	Worst-case running time	Best-case running time	Worst-case swaps	In-place?
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Insertion sort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	yes
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	no
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(n^2)$	yes