Theory: Merge sort

18 minutes 8 / 8 problems solved

Start practicing

Merge sort is an efficient comparison-based sorting algorithm. The algorithm is based on the **divide-and-conquer** technique. It divides the given unsorted array of the size **n** into **n** single-element subarrays that are already sorted, and then repeatedly merges the subarrays to produce newly sorted subarrays until there is only one subarray remaining.

In the algorithm, **merge** is the main operation. It produces a new sorted array from two input sorted arrays.

The **merge sort** algorithm and its modifications are better than primitive sorting algorithms such as bubble sort, insertion sort, and selection sort. Merge sort can be used to sort even large arrays.

§1. Implementations

The algorithm can be implemented in two ways:

- **top-down** is a recursive implementation that recursively divides the given array into two subarrays until there is only a single-element subarray remaining; it then merges the results together to produce a sorted subarray of a larger size;
- **bottom-up** is an iterative implementation that first merges pairs of adjacent single-element arrays and produces sorted subarrays of 2 elements, then merges pairs of adjacent arrays of 2 elements producing 4-elements sorted subarrays, then merges pairs of 4 elements, and so on until the whole array is merged (sorted).

We will consider both implementations.

§2. Algorithm properties

Consider the following properties of Merge sort:

- regardless of the implementation, the algorithm has a time complexity **O(n log n)** in the worst and average cases;
- merge sort is stable;
- in a typical implementation, it's not an in-place algorithm.

§3. Example

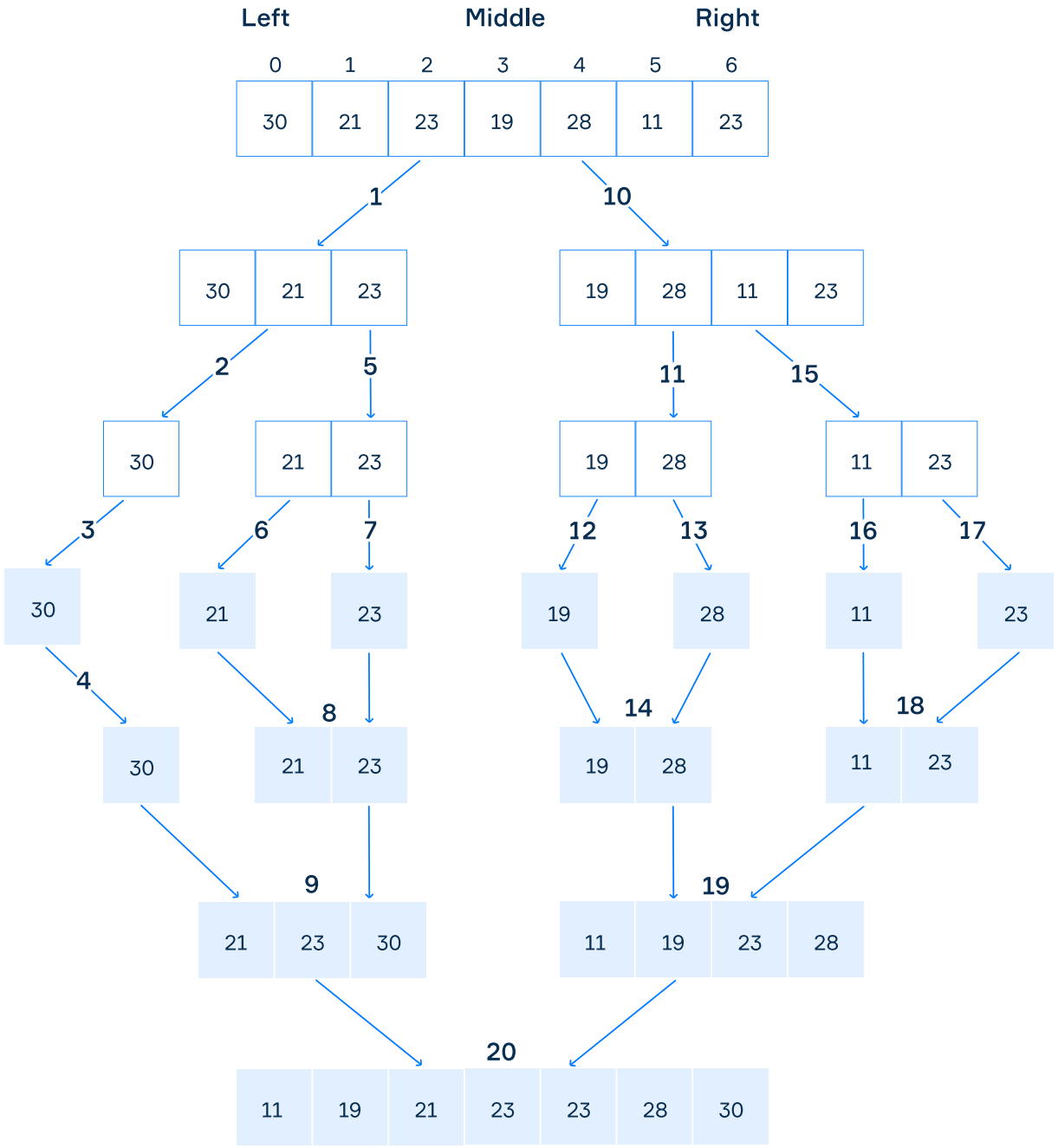
Suppose we have an unsorted seven-element array of integers:

0	1	2	3	4	5	6
30	21	23	19	28	11	23

The array elements have indexes from 0 to 6.

We have to sort the array in the ascending order using the **top-down merge sort**. The following image illustrates how it works:

Current topic:
Merge sort
Topic depends on:
Recursion basics
Divide and conquer
The sorting problem
Topic is required for:
Merge sort in Java
Merge sort in Python



In the picture, white blocks represent unsorted subarrays and blue blocks represent sorted subarrays. The number next to the arrows indicate the order of processing.

First, we calculate the index of the middle element (3) in the array and then divide the array into two subarrays. The first subarray `{ 30, 21, 23 }` contains elements from the left to the middle (exclusive), the second subarray `{ 19, 28, 11, 23 }` contains elements from the middle to the right. The subarrays have different sizes, but it's not a problem for the algorithm.

We recursively divide the first subarray to produce new subarrays using the same rule, and then do the same for the second subarray. The first subarray is `{ 30 }`, the second subarray is `{ 21, 23 }`. We do not divide the subarray `{ 30 }` because it has only a single element. Then we divide the subarray `{ 21, 23 }` into new subarrays `{ 21 }` and `{ 23 }` that are already sorted. Then we merge them to keep the ascending order, the result is `{ 21, 23 }`. Then we merge it with the array `{ 30 }`, the result is the sorted subarray `{ 21, 23, 30 }`. After, we perform the same operations for the subarray `{ 19, 28, 11, 23 }`. The result is `{ 11, 19, 23, 28 }`. See the picture to understand it better.

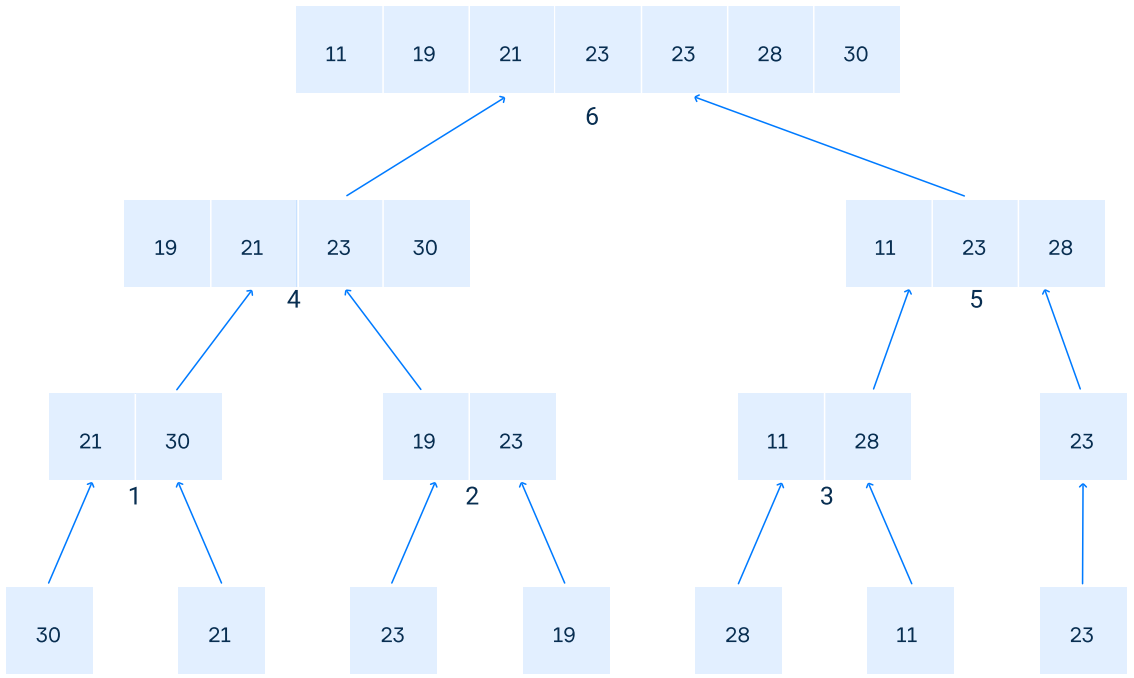
The last operation (20) merges two sorted subarrays `{ 21, 23, 30 }` and `{ 11, 19, 23, 28 }` to produce a sorted array `{ 11, 19, 21, 23, 23, 28, 30 }`.

When dividing an array this way, sometimes the second subarray is longer than the first one. If we include the middle element into the first subarray and exclude it from the second one, the situation will be reversed. Fortunately, both cases work well.

If you're still somewhat confused, see a **Merge Sort** visualization.

§4. Bottom-up implementation example

We can also sort the array using the **bottom-up merge sort**. The following image illustrates it:



We suppose the input array is separated into a sequence of one-element subarrays: `{ 30 }, { 21 }, { 23 }, { 19 }, { 28 }, { 11 }, { 23 }`. We start merging pairs of adjacent arrays to produce sorted two-element arrays and a single-element subarray. The result is `{ 21, 30 }, { 19, 23 }, { 11, 28 }, { 23 }`. The result after the following merge is `{ 19, 21, 23 , 30 }, { 11, 23, 28 }`. Then we merge two subarrays to produce a sorted input array: `{ 11, 19, 21, 23, 23, 28, 30 }`.

The merge operation works the same way as above. We reduce only recursive calls to separate the array into parts.

§5. Possible modifications

The merge sort algorithm has a lot of different modifications.

- **In-place** merge sort that is a more complex algorithm;
- **Timsort** is a hybrid stable sorting algorithm, derived from **merge sort** and **insertion sort**. It divides the input arrays into blocks of a fixed size and then sorts the block using the **insertion sort** algorithm.

Report a typo

114 users liked this theory. **5** didn't like it. **What about you?**

Start practicing

Comments (11) Hints (0) Useful links (1) Show discussion