Understanding the Merge Sort Algorithm

September 16, 2025

Abstract

This document provides a clear explanation of the Merge Sort algorithm. It includes the fundamental pseudocode for both the recursive sorting function and the merge helper function, followed by two illustrative examples that trace the algorithm's execution step-by-step.

1 Merge Sort Pseudocode

Merge Sort is a classic "divide and conquer" algorithm. It operates in two main phases:

1. **Divide:** Recursively break down the array into two halves until you have subarrays containing only one element. An array with a single element is considered inherently sorted.
2. **Conquer (Merge):** Starting with the single-element arrays, merge them back together in a sorted manner until the entire original array is reassembled.

The logic is split into two functions: the main recursive function `MergeSort` and a helper function `Merge`.

Algorithm 1 MergeSort(arr, left, right)

```
1: procedure MergeSort(arr, left, right)
2:     if left < right then
3:         mid ← ⌊(left + right)/2⌋          ▷ Find the middle point
4:         MergeSort(arr, left, mid)          ▷ Recursive call on the left half
5:         MergeSort(arr, mid + 1, right)     ▷ Recursive call on the right half
6:         Merge(arr, left, mid, right)       ▷ Merge the two sorted halves
7:     end if
8: end procedure
```

Algorithm 2 Merge(arr, left, mid, right)

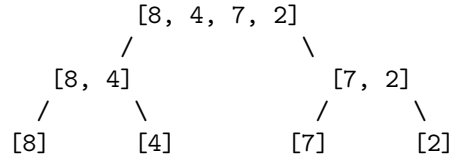
```
1:  procedure MERGE(arr, left, mid, right)
2:      n_1 ← mid − left + 1
3:      n_2 ← right − mid
4:      Create temporary arrays L[1 ... n_1] and R[1 ... n_2]
5:      for i ← 1 to n_1 do
6:          L[i] ← arr[left + i − 1]
7:      end for
8:      for j ← 1 to n_2 do
9:          R[j] ← arr[mid + j]
10:     end for
11:     i ← 1, j ← 1, k ← left
12:     while i ≤ n_1 and j ≤ n_2 do
13:         if L[i] ≤ R[j] then
14:             arr[k] ← L[i]
15:             i ← i + 1
16:         else
17:             arr[k] ← R[j]
18:             j ← j + 1
19:         end if
20:         k ← k + 1
21:     end while
22:     while i ≤ n_1 do                ▷ Copy remaining elements of L[], if any
23:         arr[k] ← L[i]
24:         i ← i + 1
25:         k ← k + 1
26:     end while
27:     while j ≤ n_2 do                ▷ Copy remaining elements of R[], if any
28:         arr[k] ← R[j]
29:         j ← j + 1
30:         k ← k + 1
31:     end while
32: end procedure
```

2 Example 1: Sorting a Small Array

Let's trace the algorithm with the array: `[8, 4, 7, 2]`.

2.1 Phase 1: Dividing

The array is recursively split until we have single-element arrays.



At this point, the dividing is complete. Each array – `[8]`, `[4]`, `[7]`, and `[2]` – is considered sorted.

2.2 Phase 2: Merging

Now, the algorithm merges these subarrays back together on its way up the recursion tree.

1. **Merge [8] and [4]:** Compare 8 and 4. Since 4 is smaller, the result is `[4, 8]`.
2. **Merge [7] and [2]:** Compare 7 and 2. Since 2 is smaller, the result is `[2, 7]`.
3. **Final Merge: [4, 8] and [2, 7]:** This is the most interesting step. We compare the first elements of each array.
   - Compare **4** and **2**. Take 2. Result: `[2]`
   - Compare **4** and **7**. Take 4. Result: `[2, 4]`
   - Compare **8** and **7**. Take 7. Result: `[2, 4, 7]`
   - Only 8 is left. Take 8. Result: `[2, 4, 7, 8]`

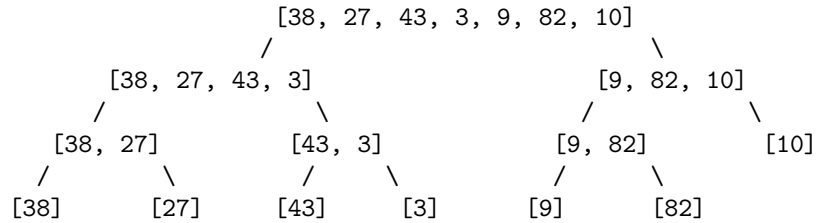
The final sorted array is `[2, 4, 7, 8]`.

3 Example 2: Sorting an Array with an Odd Number of Elements

Let's trace the algorithm with: `[38, 27, 43, 3, 9, 82, 10]`.

3.1 Phase 1: Dividing

The splits will be asymmetrical due to the odd number of elements.



3.2 Phase 2: Merging

The merges happen from the bottom up.

1. Merge `[38]` and `[27]` → `[27, 38]`
2. Merge `[43]` and `[3]` → `[3, 43]`
3. Merge `[9]` and `[82]` → `[9, 82]`

4. The array `[10]` is already a single, sorted subarray.
5. **Next Level Merge:**
    - Merge `[27, 38]` and `[3, 43]` → `[3, 27, 38, 43]`
    - Merge `[9, 82]` and `[10]` → `[9, 10, 82]`
6. **Final Merge:** `[3, 27, 38, 43]` and `[9, 10, 82]`
    - Compare 3 vs 9. Take 3.
    - Compare 27 vs 9. Take 9.
    - Compare 27 vs 10. Take 10.
    - Compare 27 vs 82. Take 27.
    - Compare 38 vs 82. Take 38.
    - Compare 43 vs 82. Take 43.
    - Only 82 is left. Take 82.

The final sorted array is `[3, 9, 10, 27, 38, 43, 82]`.