

4

Merge Sort

MEDIUM

■ What You Will Learn

- ✓ The Divide & Conquer strategy behind Merge Sort
- ✓ How splitting and merging works step by step
- ✓ Why it always runs in $O(n \log n)$ — no exceptions
- ✓ The tradeoff: it needs extra $O(n)$ memory

■ Intuition First

Imagine sorting a giant pile of cards by splitting it into two halves, sorting each half separately, then carefully merging both halves back together. That's merge sort. You keep splitting until each pile has 1 card (trivially sorted), then merge back up.

■ Divide & Conquer in 3 Lines

- DIVIDE: Split array in half recursively until size = 1
- CONQUER: Size-1 arrays are already sorted (base case)
- COMBINE: Merge two sorted halves into one sorted array

■ Visual — Recursion Tree

```
[38, 27, 43, 3, 9, 82, 10]
■ ■
[38,27,43,3] [9,82,10]
■ ■ ■ ■
[38,27] [43,3] [9,82] [10]
■ ■ ■ ■ ■ ■
[38] [27] [43] [3] [9] [82]
■ ■ ■ ■ ■ ■
[27,38] [3,43] [9,82]
■ ■ ■ ■
[3,27,38,43] [9,10,82]
■ ■
[3, 9, 10, 27, 38, 43, 82] ✓
```

■ The Merge Step (Most Important!)

Two sorted arrays: [3, 27, 38, 43] and [9, 10, 82]

Use two pointers — one per array. Compare front elements, pick the smaller one, advance that pointer. Repeat until one array empties, then copy rest.

```
L=[3,27,38,43] R=[9,10,82] Result=[]
3 < 9 → pick 3. Result=[3]
9 < 27 → pick 9. Result=[3,9]
10 < 27 → pick 10. Result=[3,9,10]
27 < 82 → pick 27. Result=[3,9,10,27]
38 < 82 → pick 38. Result=[3,9,10,27,38]
43 < 82 → pick 43. Result=[3,9,10,27,38,43]
Copy rest of R → Result=[3,9,10,27,38,43,82] ✓
```

■ Java Implementation

```
public static void mergeSort(int[] arr, int left, int right) {
    if (left >= right) return; // Base case: 1 element
    int mid = left + (right - left) / 2; // Avoid overflow
    mergeSort(arr, left, mid); // Sort left half
    mergeSort(arr, mid + 1, right); // Sort right half
    merge(arr, left, mid, right); // Merge both halves
}

public static void merge(int[] arr, int left, int mid, int right) {
    int n1 = mid - left + 1, n2 = right - mid;
    int[] L = new int[n1], R = new int[n2];
    // Copy data to temp arrays
    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];
    // Merge back using two pointers
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++]; // Copy remaining
    while (j < n2) arr[k++] = R[j++];
}
```

■■ Time & Space Complexity

Case	Time	Space	What happens?
Best	$O(n \log n)$	$O(n)$	Always splits and merges — no shortcuts
Average	$O(n \log n)$	$O(n)$	Consistent. No bad inputs exist for merge sort

Worst

 $O(n \log n)$ $O(n)$

Guaranteed even for reverse-sorted input

■ Why $O(n \log n)$?

- $\log n$ levels of splitting (each halving = one level of the tree)
- At each level, merging all elements costs $O(n)$
- Total: $n * \log n$ operations — that's the sweet spot

■ Common Mistakes

- Using $\text{mid} = (\text{left} + \text{right}) / 2$ can overflow for large arrays — use $\text{left} + (\text{right} - \text{left}) / 2$
- Not allocating a temp array for merge — you NEED extra $O(n)$ space
- Forgetting to copy remaining elements after one pointer exhausts

■ Pro Tips

- Merge sort is the go-to for sorting linked lists (no random access needed)
- Java's `Arrays.sort()` for objects uses TimSort — a merge sort variant
- Stable sort — equal elements maintain their relative order

■ Practice Problems

[Easy] Trace merge sort on [8, 3, 5, 1, 9, 2, 7, 4]

[Med] Count inversions in an array using merge sort

[Med] Merge k sorted arrays using merge sort logic

[Hard] Sort a linked list using merge sort

■ Quick Revision

- Divide: split in half until size 1
- Conquer: merge two sorted halves with 2-pointer technique
- $O(n \log n)$ always — best, avg, worst
- $O(n)$ extra space — NOT in-place
- Stable — great for objects and linked lists