Merge Sort: Understanding Recursion and the Merge Function

Overview of Merge Sort

Merge sort is a classic divide-and-conquer algorithm that sorts an array by recursively splitting it into smaller subarrays, sorting these subarrays, and then merging them back together. The process is broken down into three key phases:

1. **Divide:** The array is repeatedly split into two halves until each subarray contains only one element.
2. **Conquer:** The subarrays are recursively sorted, which is trivial for one-element subarrays.
3. **Combine:** The sorted subarrays are merged back together to form the final sorted array.

1. Divide

In the divide phase, the array is split recursively into two halves until the base case is reached, where each subarray contains only a single element:

$$\text{Divide: } \{38, 27, 43, 3, 9, 82, 10\} \rightarrow \{38, 27, 43\} \quad \{3, 9, 82, 10\}$$

This division continues until:

$$\{38, 27, 43\} \rightarrow \{38\}, \{27, 43\} \quad \text{and} \quad \{3, 9, 82, 10\} \rightarrow \{3, 9\}, \{82, 10\}$$

2. Conquer

During the conquer phase, the recursion reaches its base case, where the subarrays contain only one element. These one-element subarrays are inherently sorted:

$$\{38\}, \{27\}, \{43\}, \{3\}, \{9\}, \{82\}, \{10\}$$

The recursion now begins to unwind, and the merging process begins.

3. Combine (The Merge Function)

The merge function is where the real work of sorting occurs. As the recursion unwinds, the merge function combines two sorted subarrays into a single sorted array. Here's how it works:

a. Inputs to the Merge Function

The merge function takes in:

- **Left Subarray:** The first sorted half of the array.
- **Right Subarray:** The second sorted half of the array.
- **Auxiliary Array:** (Optional) Temporarily holds the merged elements.

b. Initialization

- Two pointers/indices, $i = 0$ and $j = 0$, are initialized for the left and right subarrays, respectively.
- A third pointer $k = 0$ is initialized to track the position in the merged array.

c. Merging Process

The merging process involves comparing elements from the left and right subarrays and placing the smaller one into the merged array. This is done in a loop:

- Compare the elements left[i] and right[j].
- If left[i] $\leq$ right[j], place left[i] in the merged array and move i and k forward.
- If right[j] is smaller, place right[j] in the merged array and move j and k forward.
- Continue until one subarray is fully merged.

d. Handling Remaining Elements

If there are remaining elements in either subarray after the loop:

- Copy all remaining elements from the left subarray, if any, into the merged array.
- Copy all remaining elements from the right subarray, if any, into the merged array.

e. Example of the Merge Process

Consider two sorted subarrays:

$$\text{Left: } \{27, 38, 43\}, \quad \text{Right: } \{3, 9, 10, 82\}$$

The merge function works as follows:

1. Compare 27 (left) and 3 (right): 3 is smaller, so it's placed in the merged array.
2. Compare 27 (left) and 9 (right): 9 is smaller, so it's placed in the merged array.
3. Compare 27 (left) and 10 (right): 10 is smaller, so it's placed in the merged array.
4. Compare 27 (left) and 82 (right): 27 is smaller, so it's placed in the merged array.
5. Compare 38 (left) and 82 (right): 38 is smaller, so it's placed in the merged array.
6. Compare 43 (left) and 82 (right): 43 is smaller, so it's placed in the merged array.
7. Finally, 82 (the only element left) is placed in the merged array.

The final merged array is:

$$\{3, 9, 10, 27, 38, 43, 82\}$$

Role of Recursion and Stack Unwinding

As the recursion stack unwinds, the merge function is called repeatedly, each time merging progressively larger sorted subarrays until the entire array is sorted. The division step ensures that each merge operation involves only sorted subarrays, making the merging process straightforward and efficient.

Efficiency of Merge Sort

The efficiency of merge sort lies in its ability to merge two sorted subarrays in linear time, $O(n)$, where n is the total number of elements in the subarrays. Since the overall process involves $O(\log n)$ levels of recursion, the total time complexity of merge sort is $O(n \log n)$.