Merge Sort

- Merge sort is a popular sorting algorithm based on the divide-and-conquer strategy.
- It works by dividing the input array into smaller subarrays, recursively sorting them, and then merging them to produce a sorted output.
- The main idea behind merge sort is that it's easier to merge two sorted arrays than to sort an unsorted array.
- Merge sort has a time complexity of O(n log n), making it efficient for large datasets.

Steps of Merge Sort:

1. **Divide**: The array is divided into two roughly equal halves until the subarrays contain only one or zero elements.
2. **Conquer**: The subarrays are recursively sorted using the merge sort algorithm. This step continues until the base case is reached.
3. **Merge**: The sorted subarrays are merged back together to create a single sorted array. This is done by comparing elements from the two subarrays and selecting the smaller one to be placed in the merged array. This process is repeated until all elements are merged.

Key Concepts and Advantages:

- **Merge sort is a stable sorting algorithm, meaning that it maintains the relative order of elements with equal values.**
- It is an efficient algorithm for sorting linked lists due to its sequential access pattern.
- **Merge sort has a consistent time complexity of O(n log n) in all cases, making it suitable for large datasets.**
- **It does not rely on the initial order of the input, making it suitable for various data distributions.**
- **Merge sort requires additional space for the temporary arrays used during the merging process. The space complexity is O(n).**

Note:

- Visualize the divide-and-conquer approach: Imagine splitting the array into halves, sorting them individually, and then merging them back together.

- Understand the recursive nature: Each recursive call handles a smaller part of the array until the base case is reached.
- Focus on the merging process: Pay attention to how two sorted subarrays are merged to produce a larger sorted array.

Visualize Array:

1. Initial Array: [55, 99, 10, 5, 0, 96, 45, 66, 78, 3]
2. Divide the array into two halves:
    - Left Half: [55, 99, 10, 5, 0]
    - Right Half: [96, 45, 66, 78, 3]
3. Recursive Sorting:
    - Sorting the Left Half:
        - Divide the left half further:
            - Left Half: [55, 99]
            - Right Half: [10, 5, 0]
        - Sorting the subarrays:
            - Sorting [55, 99]:
                - Left Half: [55]
                - Right Half: [99]
                - Merge: [55, 99]
            - Sorting [10, 5, 0]:
                - Left Half: [10]
                - Right Half: [5, 0]
                - Sorting [5, 0]:
                    - Left Half: [5]
                    - Right Half: [0]
                    - Merge: [0, 5]
                - Merge: [0, 5, 10]
            - Merge: [0, 5, 10, 55, 99]
        - Merge: [0, 5, 10, 55, 99] (Left Half)

- Sorting the Right Half:
    - Divide the right half further:
        - Left Half: [96, 45]
        - Right Half: [66, 78, 3]
    - Sorting the subarrays:
        - Sorting [96, 45]:
            - Left Half: [96]
            - Right Half: [45]
            - Merge: [45, 96]
        - Sorting [66, 78, 3]:
            - Divide the right half further:
                - Left Half: [66]
                - Right Half: [78, 3]
            - Sorting the subarrays:
                - Sorting [78, 3]:
                    - Left Half: [78]
                    - Right Half: [3]
                    - Merge: [3, 78]
                - Merge: [3, 66, 78]
            - Merge: [3, 66, 78] (Right Half)
        - Merge: [3, 45, 66, 78, 96]
    - Merge: [3, 45, 66, 78, 96] (Right Half)

4. Merge the sorted halves:
    - Merging [0, 5, 10, 55, 99] (Left Half) and [3, 45, 66, 78, 96] (Right Half):
        - Comparing the elements from the left and right halves, and merging them in sorted order:
            - Merged Array: [0, 3, 5, 10, 45, 55, 66, 78, 96, 99]
5. Final Sorted Array: [0, 3, 5, 10, 45, 55, 66, 78, 96, 99]

Approach 1: Recursive function to perform merge sort

1. The **merge** function takes in the vector **arr**, the starting index **start**, the middle index **mid**, and the ending index **end**. It merges two sorted subarrays **[start, mid]** and **[mid+1, end]** into a single sorted array.
2. In the **merge** function, the sizes of the left and right subarrays are calculated using **leftArraySize** and **rightArraySize**.
3. Temporary arrays **leftArray** and **rightArray** are created to hold the elements of the left and right subarrays, respectively. The elements are copied from the main array **arr** into these temporary arrays.
4. The merging process starts by initializing indices:
    - **mainArrayIndex** represents the current index of the main array **arr** being updated.
    - **leftArrayIndex** represents the current index of the left subarray.
    - **rightArrayIndex** represents the current index of the right subarray.
5. The while loop compares the elements at **leftArray[leftArrayIndex]** and **rightArray[rightArrayIndex]** and merges them in sorted order into the main array **arr**. This process continues until either the left subarray or the right subarray is exhausted.
6. After the while loop, if there are any remaining elements in the left subarray or right subarray, they are copied into the main array.
7. Finally, the dynamically allocated memory for the temporary arrays is freed using **delete[]**.
8. The **mergeSort** function performs the merge sort algorithm using recursion. It takes in the vector **arr**, the starting index **start**, and the ending index **end**.
9. The base case checks if the subarray has one or fewer elements, in which case it is already sorted, so the function returns.
10. If the base case is not satisfied, the **mergeSort** function proceeds to divide the array into two halves. The middle index **mid** is calculated as the average of **start** and **end**.
11. The **mergeSort** function is recursively called on the left half of the array (**start** to **mid**) and the right half of the array (**mid+1** to **end**).
12. After the left and right halves are sorted, the **merge** function is called to merge the two sorted halves into a single sorted array.
13. The **print** function is used to display the elements of the sorted array.
14. In the **main** function, an example array is created (**arr**) with values [55, 99, 10, 5, 0, 96, 45, 66, 78, 3]. The **mergeSort** function is called to sort this array, passing in the starting index as 0 and the ending index as **arr.size() - 1**.

15. Finally, the **print** function is called to display the sorted array.