SORTING - II

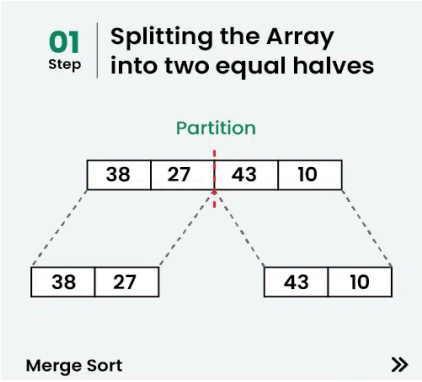
Merge Sort

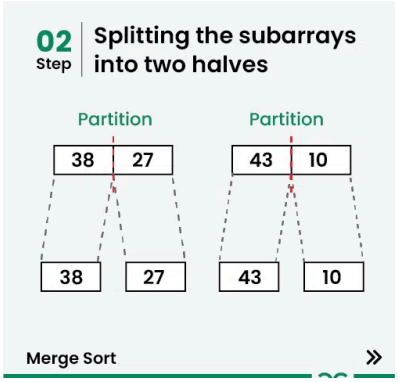
- **Merge sort** is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.
- In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.
- Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

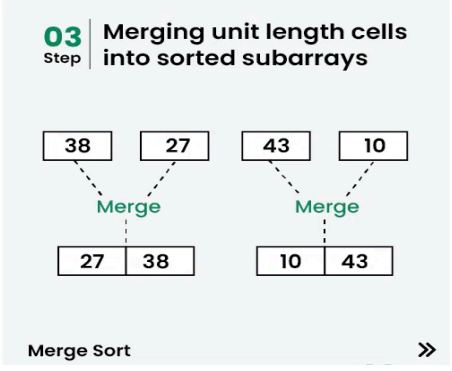
Illustration:

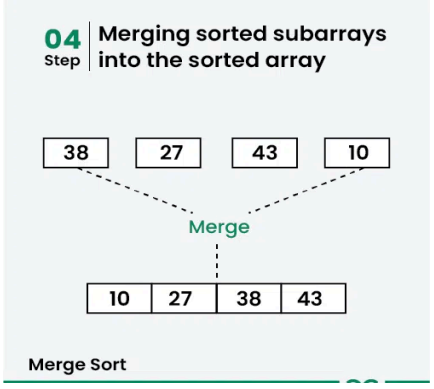
Let's consider an array **arr[] = {38, 27, 43, 10}**

- These sorted subarrays are merged together, and we get bigger sorted subarrays.
- This merging process is continued until the sorted array is built from the smaller subarrays.
- Initially divide the array into two equal halves:
- These subarrays are further divided into two halves. Now they become array of unit length that can no longer be divided and array of unit length are always sorted.
- The following diagram shows the complete merge sort process for an example array {38, 27, 43, 10}.









Complexity Analysis of Merge Sort

Time Complexity: O(N log(N)), Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$$T(n) = 2T(n/2) + \theta(n)$$

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of the Master Method and the solution of the recurrence is θ(Nlog(N)). The time complexity of Merge Sort isθ(Nlog(N)) in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Auxiliary Space: O(N), In merge sort all elements are copied into an auxiliary array. So N auxiliary space is required for merge sort.

Applications of Merge Sort:

- **Sorting large datasets:** Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of O(n log n).
- **External sorting:** Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.
- **Custom sorting:** Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.
- Inversion Count Problem

Advantages of Merge Sort:

- **Stability**: Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of O(N logN), which means it performs well even on large datasets.
- **Parallelizable**: Merge sort is a naturally parallelizable algorithm, which means it can be easily parallelized to take advantage of multiple processors or threads.

Drawbacks of Merge Sort:

- **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- **Not always optimal for small datasets:** For small datasets, Merge sort has a higher time complexity than some other sorting algorithms, such as insertion sort. This can result in slower performance for very small datasets.

Recursive Bubble Sort

- Recursive Bubble Sort has no performance/implementation advantages, but can be a good question to check one's understanding of Bubble Sort and recursion.
- If we take a closer look at the Bubble Sort algorithm, we can notice that in the first pass, we move the largest element to end (Assuming sorting in increasing order). In the second pass, we move the second largest element to the second last position and so on.

Recursion Idea.

1. Base Case: If array size is 1, return.
2. Do One Pass of normal Bubble Sort. This pass fixes the last element of the current subarray.
3. Recur for all elements except the last of current subarray.

- **Time Complexity:** O(n*n)
- **Auxiliary Space:** O(n)

Recursive Insertion Sort

- Recursive Insertion Sort has no performance/implementation advantages, but can be a good question to check one's understanding of Insertion Sort and recursion.
- If we take a closer look at the Insertion Sort algorithm, we keep processed elements sorted and insert new elements one by one in the sorted array.

Recursion Idea

- Base Case: If array size is 1 or smaller, return.
- Recursively sort first n-1 elements.
- Insert the last element at its correct position in the sorted array.

Time Complexity: $O(n^2)$
Auxiliary Space: O(n)