Merge Sort

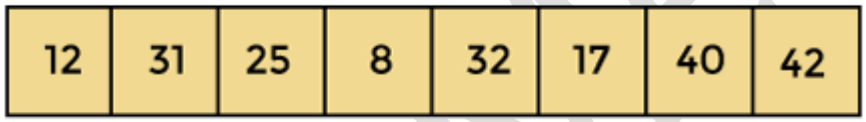
Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

Working of Merge sort Algorithm

Now, let's see the working of merge sort Algorithm.

To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

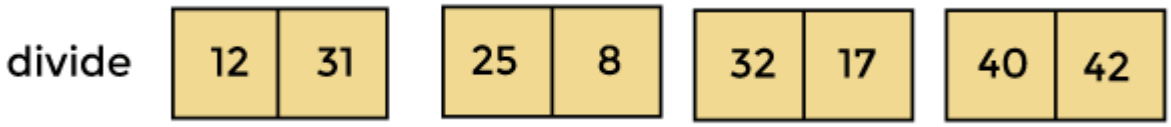
Let the elements of array are -



According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

As there are eight elements in the given array, so it is divided into two arrays of size 4.

Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.

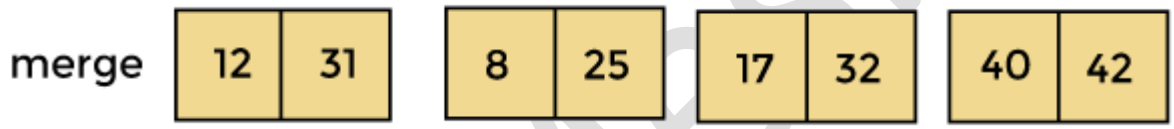


Now, again divide these arrays to get the atomic value that cannot be further divided.

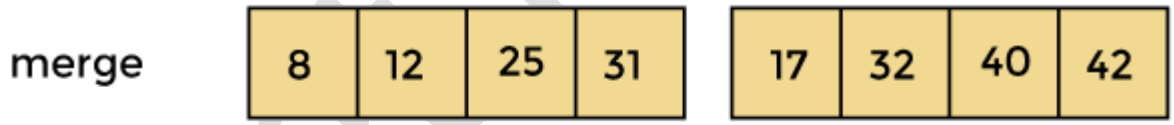
Now, combine them in the same manner they were broken.

In combining, first compare the element of each array and then combine them into another array in sorted order.

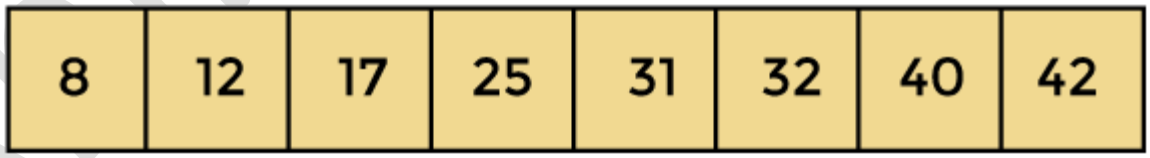
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



Now, the array is completely sorted.

Merge sort complexity

Now, let's see the time complexity of merge sort in best case, average case, and in worst case. We will also see the space complexity of the merge sort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n*logn)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **O(n*logn)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **O(n*logn)**.

2. Space Complexity

Space Complexity	O(n)
Stable	YES

Algorithm code

```
import java.util.Arrays;

public class practice {
    public static void main(String[] args) {
        int[] arr = {5,3,6,100,21,1};
//        System.out.println(Arrays.toString(merge(arr)));
        mergeInSort(arr, 0, arr.length);
        System.out.println(Arrays.toString(arr));
    }

//    public static int[] merge(int[] arr) {
//        if (arr.length == 1) {
//            return arr;
//        }
//
//        int mid = arr.length / 2;
//        int[] first = merge(Arrays.copyOfRange(arr, 0, mid));
//        int[] second = merge(Arrays.copyOfRange(arr, mid, arr.length));
//
//        return mergeSort(first, second);
//    }
//
//    private static int[] mergeSort(int[] first, int[] second) {
//        int[] mix = new int[first.length + second.length];
//
//        int i = 0;
//        int j = 0;
//        int k = 0;
//
//        while(i < first.length && j < second.length) {
//            if (first[i] < second[j]) {
//                mix[k] = first[i];
//                i++;
//                k++;
//            }
//            else {
//                mix[k] = second[j];
//                j++;
//                k++;
//            }
//        }
//
//        // it may be possible the any one of the array is completed
```

```
//        while (i< first.length) {
//            mix[k] = first[i];
//            i++;
//            k++;
//        }
//
//        while (j < second.length) {
//            mix[k] = second[j];
//            j++;
//            k++;
//        }
//        return mix;




    // another way
    public static void mergeInSort(int[] arr, int s, int e) {
        if (e - s == 1) {
            return;
        }

        int m = s + (e - s) / 2;
        mergeInSort(arr, s, m );
        mergeInSort(arr, m , e);
        mergeArr(arr, s, m, e);
    }

    public static void mergeArr(int[] arr, int start, int mid, int end) {
        int[] mix = new int[end - start];

        int i = start;
        int j = mid;
        int k = 0;

        while (i < mid && j < end) {
            if (arr[i] < arr[j]) {
                mix[k] = arr[i];
                k++;
                i++;
            }
            else {
                mix[k] = arr[j];
                k++;
                j++;
            }

        }

        // it may be possible that one of the array is completed
        while (i < mid) {
            mix[k] = arr[i];
```

```
            k++;
            i++;
        }

        while (j < end) {
            mix[k] = arr[j];
            k++;
            j++;
        }
        System.arraycopy(mix, 0, arr, start + 0, mix.length);
    }

}
```