

Pre Requisites:

- Basic JAVA syntax
- Recursion concepts

List of concepts involved :

- Introduction to divide and conquer strategy
- Merge sort Algorithm
- Working of merge sort algorithm
- time and space complexity analysis of merge sort algorithm
- Quick sort Algorithm
- working of Quick sort algorithm
- Time and space complexity analysis of quick sort algorithm
- Randomized quick sort algorithm

A divide and conquer algorithm is a strategy of solving a large problem by

- Breaking the problem into smaller sub-problems
- Solving the sub-problems, and
- Combining them to get the desired output.
- To use the divide and conquer algorithm, recursion is used.

Here are the steps involved:

- Divide: Divide the given problem into subproblems using recursion.
- Conquer: Solve the smaller subproblems recursively. If the subproblem is small enough, then solve it directly.
- Combine: Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

Merge sort :

Merge sort uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithms. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs are merged into the four-element lists, and so on until we get the sorted list.

Algorithm of merge sort :

In the following algorithm, **array** is the given array, **begin** is the starting element, and **end** is the last element of the array.

```

MERGE_SORT(array, begin, end)
if begin < end
set mid = begin + (end-begin)/2
MERGE_SORT(array, begin, mid)
MERGE_SORT(array, mid + 1, end)
MERGE (array, begin, mid, end)
end of if
END MERGE_SORT

```

working of merge sort algorithm :

Now, let's look at the working of merge sort algorithm on an example :

let the given array be,



To sort this array divide,



Again divide it until the size does not become 1 for each smaller part.



divide again,



now that we have each subarray of size 1. we will now merge them into one single array.



Merge again,

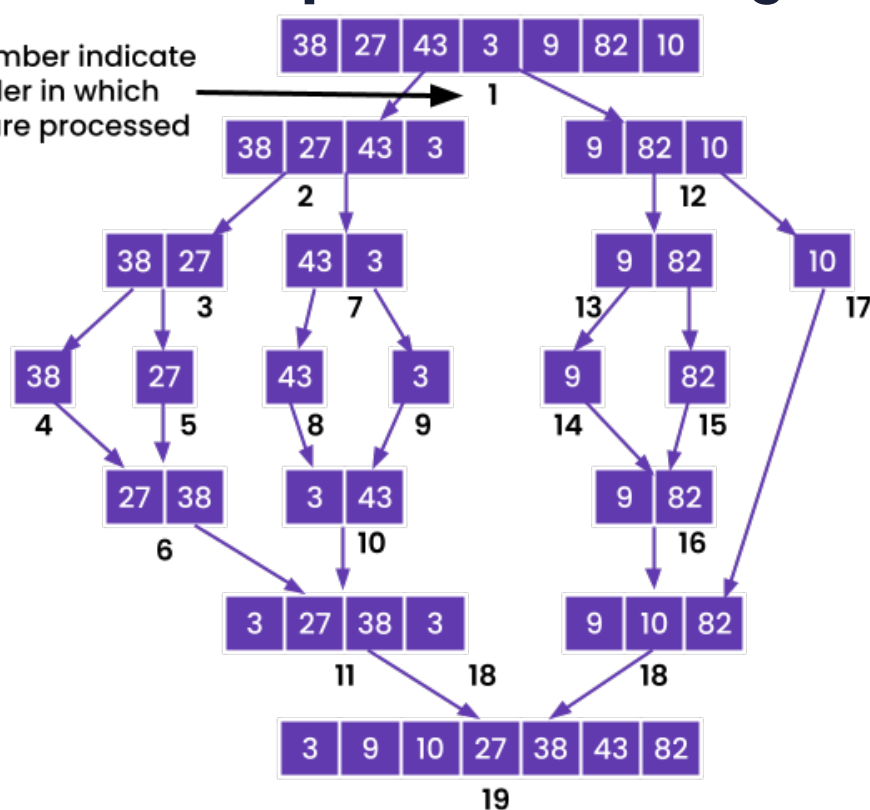


merge one last time ,



Another example and tree diagram for merge sort

The number indicate
The order in which
Steps are processed



Merge Sort Time and space complexity analysis:

Now, let us calculate time complexity with the steps. our very own first step was to divide the input into two halves which comprised us of a logarithmic time complexity ie. $\log(N)$ where N is the number of elements in the array.

Our second step was to merge back the array into a single array, so if we observe it in all the number of elements to be merged N , and to merge back we use a simple loop which runs over all the N elements giving a time complexity of $O(N)$.

finally, total time complexity will be - step -1 + step-2

$$T(n) = 2T(n/2) + O(n)$$

The solution of the above recurrence can be written as $O(n \log n)$.

The array of size N is divided into a max of $\log n$ parts, and the merging of all subarrays into a single array takes $O(N)$ time, the worst-case run time of this algorithm is $O(n \log n)$.

The time complexity of MergeSort is $O(n \log n)$ in all the 3 cases (worst, average and best) as the mergesort always divides the array into two halves and takes linear time to merge two halves.

Now let's look at how we can calculate the space complexity for merge sort.

Let us take an example again.



Once we call merge sort for the entire array ($N = 6$), the array is divided into two parts, (each of size $N / 2 = 3$).



However, we must note that function calls are not running in parallel. Everytime during the divide phase, we are making a single function call, first with the left part and then waiting for its return (with sorted array in place) to call for the right part.

The same happens for the next function calls, till we get a single element, in which case we return.

So, the first time we encounter a single element in a function call, it would be as follows:

At any point, in the call stack, we would be having a maximum of $O(\log N)$ function calls.

Now, once we are returned from the left and the right call, the merge phase for that function call would begin.

So, if the size of the subarray for that function call is N with its left and right part of size $N/2$, we would be creating a new array of size N to do the merge.

Also note that once we return from any function, the extra space we take to merge the two sorted subarrays is also freed. So, we need not sum up the extra space taken in each function call since maximum space taken at any point is something we are concerned about.

Hence, during the divide phase, the maximum space we take in the recursion call stack is $O(\log N)$ and while merging, out of all the function calls, the maximum space we take is $O(N)$ when we merge two subarrays of the entire array. Since $O(N)$ is dominating, we would consider $O(N)$ to be the space complexity of the merge sort.

Code : [LP_Code1.java](#)

Output :

```
Before sorting array elements are -
6 4 2 7 9 11 12 5
After sorting array elements are -
2 4 5 6 7 9 11 12
```