



**Bharatiya Vidya Bhavan's**  
**SARDAR PATEL INSTITUTE OF TECHNOLOGY**  
(Autonomous Institute Affiliated to University of Mumbai)  
Munshi Nagar, Andheri (W), Mumbai – 400 058.  
Department of Master of Computer Application

<b>Experiment</b>	1
<b>Aim</b>	Understand sorting algorithms on the basis of Divide and Conquer approach
<b>Objective</b>	1) Learn Divide and Conquer strategy in sorting algorithms 2) Learn Merge Sort and Quick Sort 3) Compare the Time complexity of Merge Sort and Quick Sort
<b>Name</b>	Durgesh Dilip Mandge
<b>UCID</b>	2023510032
<b>Class</b>	FYMCA
<b>Batch</b>	B
<b>Date of Submission</b>	30-01-2-24

<b>Algorithm and Explanation of the technique used</b>	<p>1. <b>MergeSort :</b> <u>PseudoCode :</u> function mergeSort(array)     If length of array <math>\leq 1</math>         return array     middle = length of array / 2     leftArray = mergeSort(first half of array)     rightArray = mergeSort(second half of array)     return merge(leftArray, rightArray)</p> <p>2. <b>QuickSort:</b> <u>PseudoCode:</u> function quickSort(arr, l, r)     if <math>l &lt; r</math>         pivotIndex = partition(arr, l, r)         quickSort(arr, l, pivotIndex - 1)         quickSort(arr, pivotIndex + 1, r) function partition(arr, l, r)     pivot = arr[r]     i = l - 1     for j = l to r - 1         if <math>arr[j] &lt; pivot</math>             i = i + 1             swap arr[i] and arr[j]     swap arr[i + 1] and arr[r]     return i + 1</p>
--	--

**Program(Code)****1.Merge Sort**

```
package Lab1;

import java.util.Arrays;

public class MergeSort {

    // r + (r+1)*i
    // 32 + 33*i
    // = [ 64 96 128 160 192 224 256 288 320 352 ]
    public static void main(String[] args) {
        int[] arr = { 128, 192, 64, 288, 352, 160, 96, 256, 320, 224};
        int[] ans = mergeSort(arr);
        System.out.println(Arrays.toString(ans));
    }

    private static int[] mergeSort(int[] arr) {
        if(arr.length<=1){
            return arr;
        }
        int s = 0, e = arr.length;
        int m = (e+s)/2;
        int[] left = mergeSort(Arrays.copyOfRange(arr, 0, m));
        int[] right = mergeSort(Arrays.copyOfRange(arr, m, e));
        return merge(left,right);
    }

    private static int[] merge(int[] left, int[] right) {
        int[] ans = new int[left.length+right.length];
        int i=0, j=0, k=0;
        while(i < left.length && j < right.length){
            if(left[i] < right[j]){
                ans[k]=left[i]; i++; k++;
            }else{
                ans[k]=right[j] ; j++; k++;
            }
        }
        while(i < left.length){
            ans[k] = left[i]; i++; k++;
        }
        while (j < right.length) {
            ans[k] = right[j]; j++; k++;
        }
        return ans;
    }
}
```

## 2.QuickSort

```
package Lab1;

import java.util.Arrays;

public class QuickSort {

    public static void main(String[] args){
        int[] arr = { 10, 7, 8, 9, 1, 5 };
        int N = arr.length;

        quickSort(arr, 0, N - 1);
        System.out.println(Arrays.toString(arr));
    }

    static void swap(int[] arr, int i, int j){
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    static int partition(int[] arr, int low, int high){
        int pivot = arr[high];
        int i = (low - 1);
        for (int j = low; j <= high - 1; j++) {
            if (arr[j] < pivot) {
                i++;
                swap(arr, i, j);
            }
        }
        swap(arr, i + 1, high);
        return (i + 1);
    }

    static void quickSort(int[] arr, int low, int high){
        if (low < high) {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }
}
```

Output

1.mergeSort(128, 192, 64, 288, 352, 160, 96, 256, 320, 224);

```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\smart\Documents\SPIT-lab\Sem 2\DAA\Lab1> & 'C:\Program Files\Java\jdk-21\bin\j
ilsInExceptionMessages' '-cp' 'C:\Users\smart\AppData\Roaming\Code\User\workspaceStorage\86
_ws\jdt.ls-java-project\bin' 'Lab1.MergeSort'
[64, 96, 128, 160, 192, 224, 256, 288, 320, 352]
PS C:\Users\smart\Documents\SPIT-lab\Sem 2\DAA\Lab1>
```

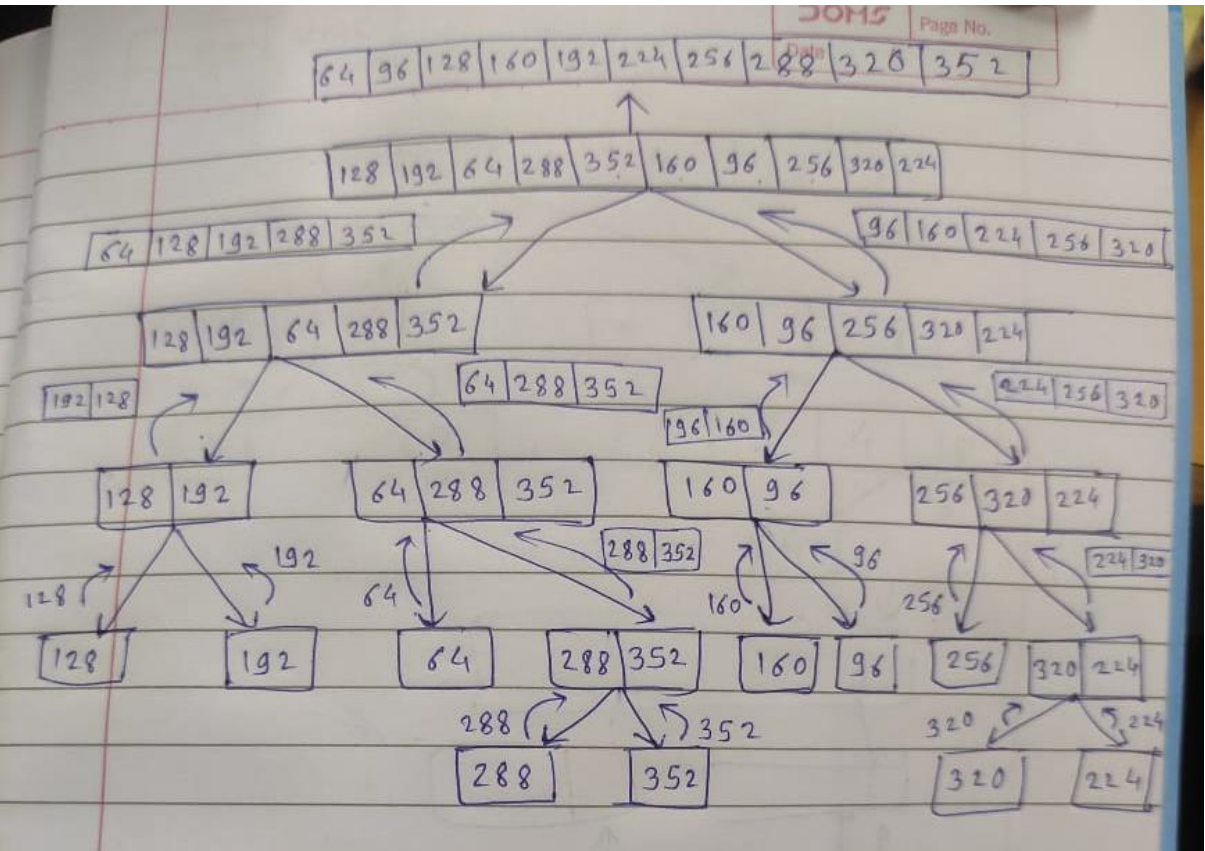
2.quickSort(128, 192, 64, 288, 352, 160, 96, 256, 320, 224)

```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS

PS C:\Users\smart\Documents\SPIT-lab\Sem 2\DAA\Lab1> c:: cd 'c:\Users\smart\Documents\S
\jdk-21\bin\java.exe' '--enable-preview' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp'
aceStorage\86a17b1b4c0252b9bf3938210292e98a\redhat.java\jdt_ws\jdt.ls-java-project\bin'
[64, 96, 128, 160, 192, 224, 256, 288, 320, 352]
PS C:\Users\smart\Documents\SPIT-lab\Sem 2\DAA\Lab1> |
```

Justification of the complexity calculated

Merge Sort (128, 192, 64, 288, 352, 160, 96, 256, 320, 224)



### ① Merge Sort

- Best Case: Sorted Array

It will still divide array into halves and merges them back.

∴ Complexity:  $O(n \log n)$

- Average Case: Randomly Shuffled Array

Average case scenario will be randomly shuffled array

∴ Complexity:  $O(n \log n)$

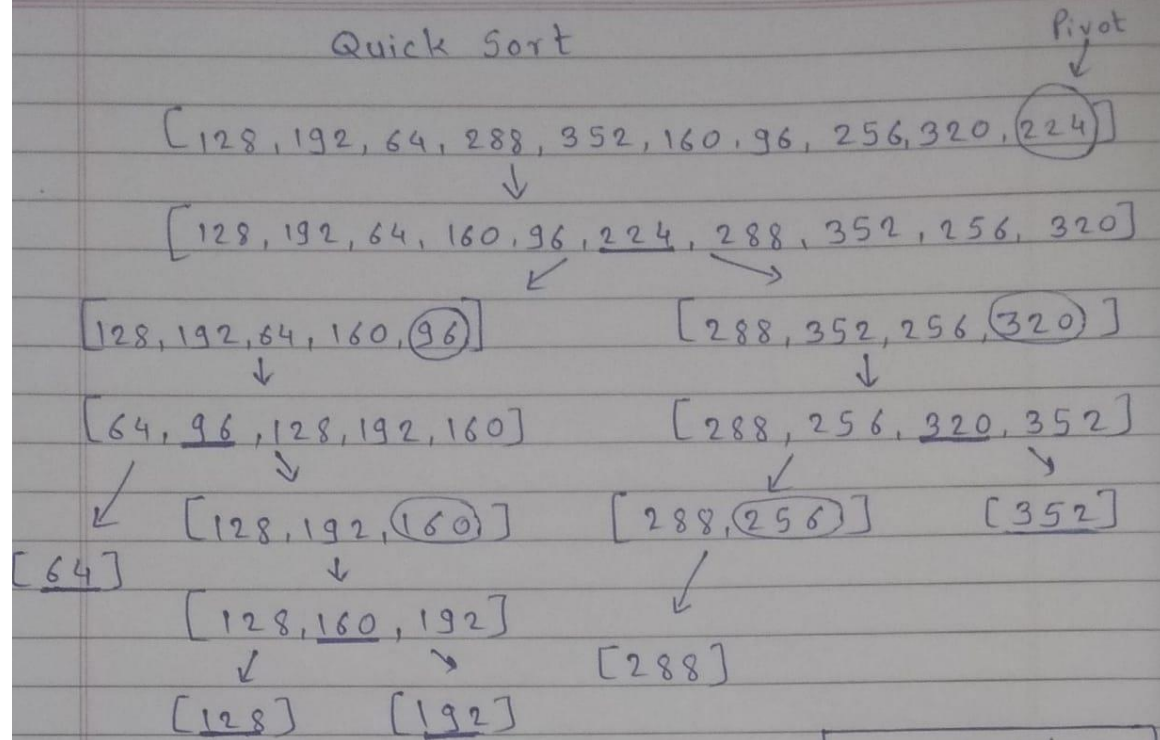
- Worst Case: When Reverse Sorted

Each element is to be compared to be moved to the end

∴ Complexity:  $O(n \log n)$

**QuickSort (128, 192, 64, 288, 352, 160, 96, 256, 320, 224)**

## Quick Sort



After swapping :

[64, 96, 128, 160, 192, 224, 256, 288, 320, 352]



### Quick Sort:

- Best Case (Pivot is median)  
When pivot divides array roughly into two equal halves. For this, pivot should be median.

Complexity:  $O(n \log n)$

- Average Case  
When pivot is not always median but it divides array into nearly halves.  
Complexity:  $O(n \log n)$

- Worst Case  
When pivot is always smallest or highest element in array.  
Complexity:  $O(n^2)$

### Complexity Comparison

#### Complexity Comparison

	Merge Sort	Quick Sort
Best	$O(n \log n)$	$O(n \log n)$
Avg	$O(n \log n)$	$O(n \log n)$
Worst	$O(n \log n)$	$O(n^2)$

<b>Conclusion</b>	<p>Worst case complexity of MergeSort (Array is reverse sorted) is less than the Worst case complexity of QuickSort (Chooosed Pivot is either highest or smallest element).</p> <p>For sorted Array given Merge Sort follows all steps alike unsorted array, but QuickSort do not follow all computations if the array is already sorted.</p> <p>For practically use Quick Sort will more effective than the merge sort as its worst case complexity do not depend on the input it depends on selection of pivot.</p>