Bhartiya Vidya Bhavan's
Sardar Patel Institute of Technology
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India
(Autonomous College Affiliated to University of Mumbai)

NAME:	Gitanjali Gangurde
UID:	2021300034
SUBJECT	Data Analysis and Algorithm
EXPERIMENT NO:	Experiment 2
AIM:	To implement the various sorting algorithms using divide and conquer technique. 1)Quick Sort 2)Merge Sort
THEORY:	**Quicksort** is the widely used sorting algorithm that makes n log n comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem. Best Case Complexity - In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. ◦ 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 quicksort is O(n*logn). ◦ The best-case time complexity of quicksort is O(n*logn). ◦ Worst Case Complexity - In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is

sorted already in ascending or descending order. The worst-case time complexity of quicksort is O(n2).

Merge sort is like the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves.

- 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).

ALGORITHM:

1]Merge Sort:

1. start
2. declare array and left, right, mid variable.
3. perform merge function.

```
if left > right
   return
mid= (left + right)/2
mergesort (array, left, mid)
mergesort (array, mid+1, right)
merge (array, left, mid, right)
```

4. stop

	2]Quick Sort: 1. QUICKSORT (array A, start, end) 2. {if (start < end) { 3. p = partition (A, start, end) 4. QUICKSORT (A, start, p - 1) 5. QUICKSORT (A, p + 1, end) 6. }}
PROGRAM:	DAA > C sort.c > partition(int [], int, int)

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void merge(int a[], int beg, int mid, int end)
{
    int i, j, k;
    int n1 = mid - beg + 1;
    int n2 = end - mid;
    int LeftArray[n1], RightArray[n2];
    for (int i = 0; i < n1; i++)
        LeftArray[i] = a[beg + i];
    for (int j = 0; j < n2; j++)
        RightArray[j] = a[mid + 1 + j];
    i = 0,
    j = 0;
    k = beg;
    while (i < n1 && j < n2)
    {
        if (LeftArray[i] <= RightArray[j])
        {
            a[k] = LeftArray[i];
            i++;
        }
        else
        {
            a[k] = RightArray[j];
            j++;
        }
        k++;
    }
```

```
    while (i < n1)
    {
        a[k] = LeftArray[i];
        i++;
        k++;
    }
    while (j < n2)
    {
        a[k] = RightArray[j];
        j++;
        k++;
    }
}
void mergeSort(int a[], int beg, int end)
{
    if (beg < end)
    {
        int mid = (beg + end) / 2;
        mergeSort(a, beg, mid);
        mergeSort(a, mid + 1, end);
        merge(a, beg, mid, end);
    }
}
void printArray(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}
```

```
int partition(int a[], int start, int end)
{
    int pivot = a[end]; // pivot element
    int i = (start - 1);
    for (int j = start; j <= end - 1; j++)
    {
        if (a[j] < pivot)
        {
            i++;
            int t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
    int t = a[i + 1];
    a[i + 1] = a[end];
    a[end] = t;
    return (i + 1);
}
void quick(int a[], int start, int end)
{
    if (start < end)
    {
        int p = partition(a, start, end);
        quick(a, start, p - 1);
        quick(a, p + 1, end);
    }
}
void printArr(int a[], int n)
{
```

```
void main()
{
    int n = 0;
    for (int k = 0; k < (100000 / 100); k++)
    {
        n = n + 100;
        int num[n];
        int quicksort[n];
        int merge[n];
        int j, min;
        clock_t start_t, end_t;
        double total_t;
        printf("%d\t", n);
        for (int i = 0; i < n; i++)
        {
            num[i] = rand() % 10;
            merge[i] = num[i];
            quicksort[i] = num[i];
        }
        start_t = clock();
        mergeSort(merge, 0, n - 1);
        end_t = clock();
        total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
        printf("%f\n", total_t);
        start_t = clock();
        quick(quicksort, 0, n - 1);
        end_t = clock();
        total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
        printf("%f\n", total_t);
    }
```

CONCLUSION:

Merge sort is more efficient as it is worst case time complexity is O(nlogn) while in case of quick sort, it remains constant throughout all operations as we can see from its graph which is linear in nature.