Name	Dhruvil Doshi
Roll No.	2021300027
Subject	Data Analysis Algorithm
Experiment	2

Aim- To implement the various sorting algorithms using divide and conquer technique.

Algorithm

Both merge sort and quick sort employ a common algorithmic paradigm based on recursion.
This paradigm, divide-and-conquer, breaks a problem into subproblems that are similar to
the original problem, recursively solves the subproblems, and finally combines the solutions
to the subproblems to solve the original problem. Because divide-and-conquer solves
subproblems recursively, each subproblem must be smaller than the original problem, and
there must be a base case for subproblems. You should think of a divide-and-conquer
algorithm as having three parts:
1. Divide the problem into a number of subproblems that are smaller instances of the
same problem.
2. Conquer the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3. Combine the solutions to the subproblems into the solution for the original problem.

You can easily remember the steps of a divide-and-conquer algorithm as divide, conquer,
combine. Here's how to view one step, assuming that each divide step creates two subproblems (though some divide-and-conquer algorithms create more than two):

Merge Sort –

Step 1: Start

Step 2: Declare array and left, right, mid variable

Step 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)
```

Step 4: Stop

Quick Sort –

This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot.

```
partition (arr[], low, high)
{
        pivot = arr[high];
        i = (low – 1)
        for (j = low; j <= high- 1; j++){
        If current element is smaller than the pivot
        if (arr[j] < pivot){
        i++;
        swap arr[i] and arr[j]
        }
        }
        swap arr[i + 1] and arr[high])
        return (i + 1)
}
```

Code-

```
#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];
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)
{
int i;
for (i = 0; i < n; i++)
printf("%d ", a[i]);
}
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 its worst case time complexity is O(logn) while in case of quick sort, it remains constant throughout all operations as we can see from its graph which is linear in nature.