Name	MAAZ SHAIKH
UID no.	2021700059 SY CSE(DS)
Experiment No.	2

AIM:	Experiment based on divide and conquer approach
PROBLEM STATEMENT :	– For this experiment, you need to implement two sorting algorithms namely Quicksort and Merge sort methods. Compare these algorithms based on time and space complexity. Time required for sorting algorithms can be performed using high_resolution_clock::now() under namespace std::chrono. You have to generate 1,00,000 integer numbers using C/C++ Rand function and save them in a text file. Both the sorting algorithms uses these 1,00,000 integer numbers as input as follows. Each sorting algorithm sorts a block of 100,200,300,...,100000 integer numbers with array indexes numbers A[0..99], A[100..199], A[200..299],…, A[99900..99999]. You need to use high_resolution_clock::now() function to find the time required for 100, 200, 300…. 100000 integer numbers. Finally, compare two algorithms namely Quicksort and Merge sort by plotting the time required to sort integers using LibreOffice Calc/MS Excel. The x-axis of 2-D plot represents the block no. of 1000 blocks. The y-axis of 2-D plot represents the tunning time to sort 1000 blocks of 100,200,300,...,100000 integer numbers.
ALGORITHM/ THEORY:	**Merge Sort Algorithm** In this article, we will discuss the merge sort Algorithm. Merge sort is the sorting technique that follows the divide and conquer approach. This article will be very helpful and interesting to students as they might face merge sort as a question in their examinations. In coding or technical interviews for software engineers, sorting algorithms are widely asked. So, it is important to discuss the topic. Merge sort is similar to 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. 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 is merged into the four-element lists, and so on until we get the sorted list.

Quick Sort Algorithm

In this article, we will discuss the Quicksort Algorithm. The working procedure of Quicksort is also simple. This article will be very helpful and interesting to students as they might face quicksort as a question in their examinations. So, it is important to discuss the topic.

Sorting is a way of arranging items in a systematic manner. 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.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

PROGRAM:

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

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 current element is smaller than the pivot
        if (a[j] < pivot)
        {
            i++; // increment index of smaller element
            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);
}
```

```c
void quickSort(int a[], int start, int end) /* a[] = array to be sorted, start =
Starting index, end = Ending index */
{
    if (start < end)
    {
        int p = partition(a, start, end); //p is the partitioning index
        quickSort(a, start, p - 1);
        quickSort(a, p + 1, end);
    }
}
```

```c
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]; //temporary arrays

    /* copy data to temp arrays */
    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; /* initial index of first sub-array */
    j = 0; /* initial index of second sub-array */
    k = beg;  /* initial index of merged sub-array */
```

```
    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);
    }
}
```

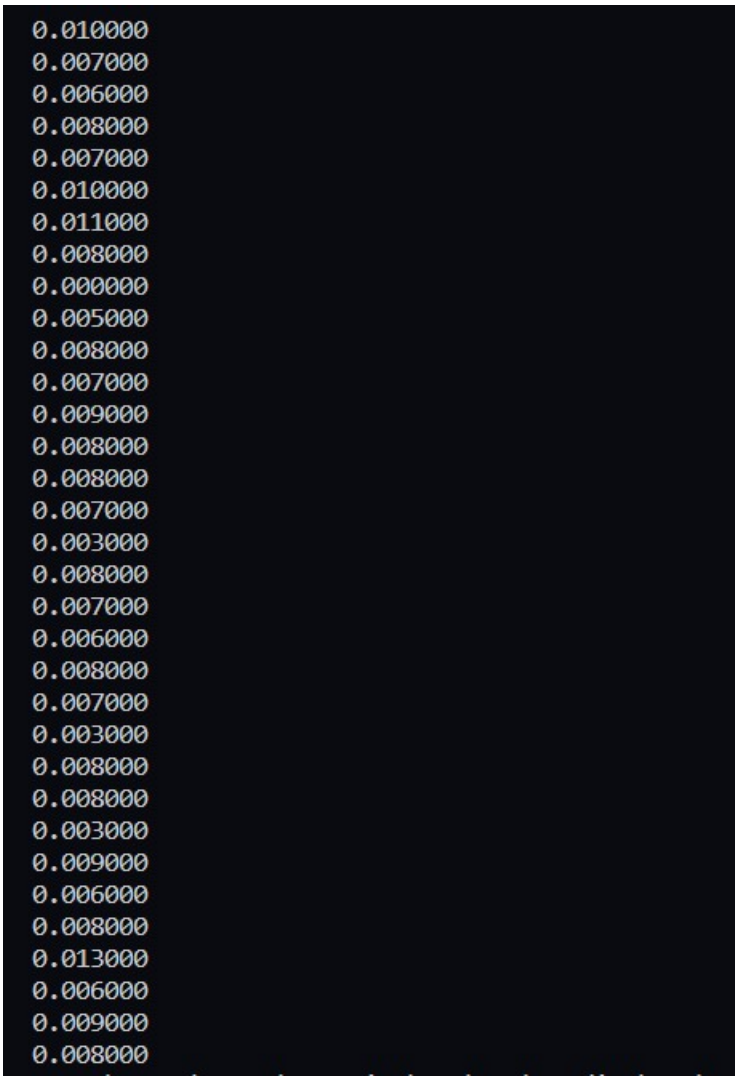
```
int main(){
    for(int i=1;i<=1000;i++){
        int j=0;
        int numberArray[100000];
        FILE *f;
        f = fopen("new.txt","r");
        for (j = 0; j < 100000; j++){
            fscanf(f, "%d,", &numberArray[j] );
            }
        fclose(f);
        clock_t t;
        t = clock();
        //mergeSort(numberArray,0,i*100);
        quickSort(numberArray,0,i*100);
        t = clock() - t;
```

```
        double time_taken = ((double)t)/CLOCKS_PER_SEC;
        printf("%f\n",time_taken);
    }
}
```

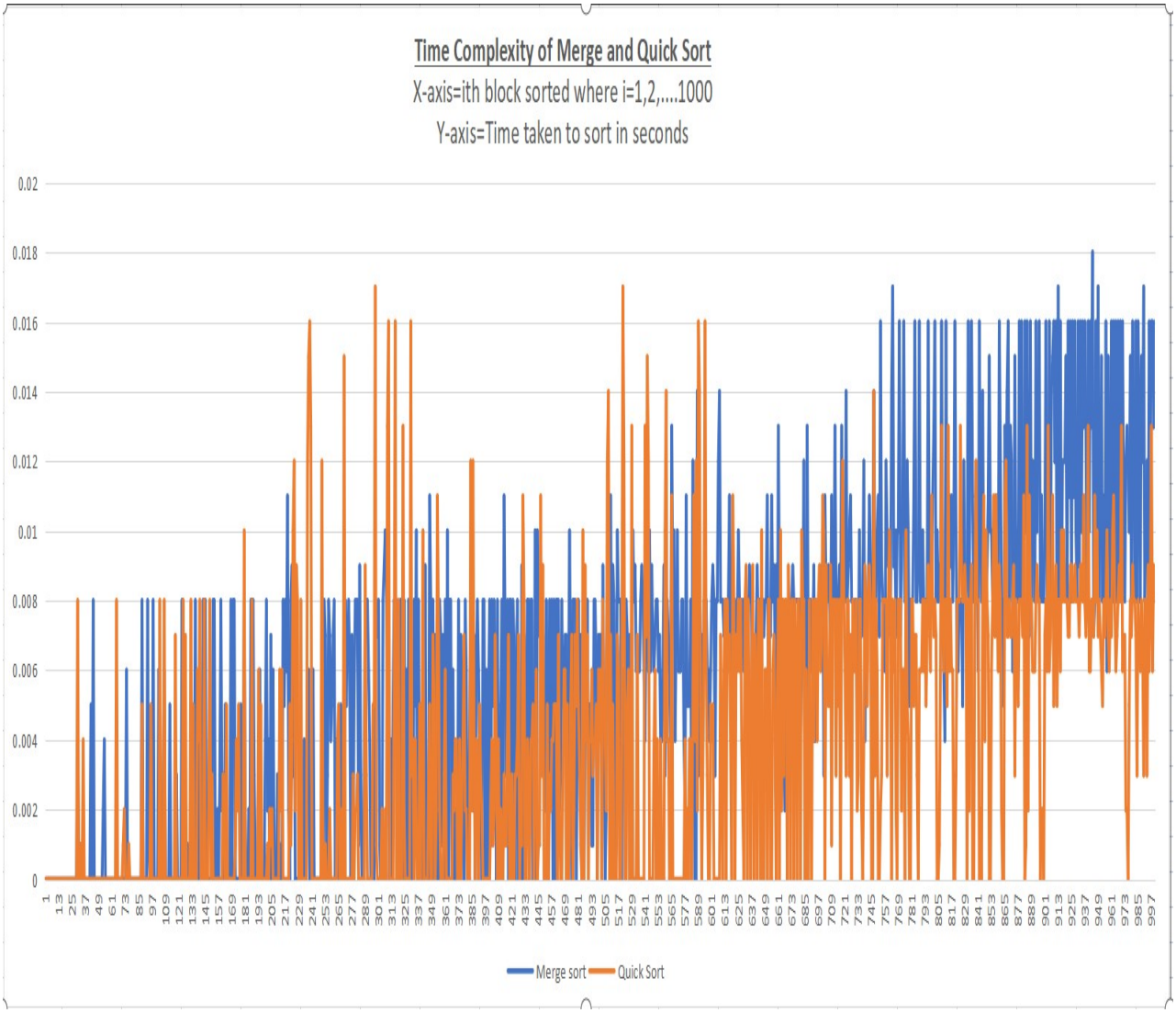
RESULT: Merge sort-

```
0.013000
0.008000
0.016000
0.010000
0.016000
0.013000
0.018000
0.007000
0.008000
0.016000
0.008000
0.017000
0.008000
0.012000
0.015000
0.008000
0.011000
0.008000
0.016000
0.006000
0.015000
0.011000
0.008000
0.016000
0.010000
0.012000
0.008000
0.016000
0.013000
0.016000
0.016000
0.013000
0.016000
```

Quick Sort-


0.010000
0.007000
0.006000
0.008000
0.007000
0.010000
0.011000
0.008000
0.000000
0.005000
0.008000
0.007000
0.009000
0.008000
0.008000
0.007000
0.003000
0.008000
0.007000
0.006000
0.008000
0.007000
0.003000
0.008000
0.008000
0.003000
0.009000
0.006000
0.008000
0.013000
0.006000
0.009000
0.008000

GRAPH:



CONCLUSION:	As the number of input size increases,the time difference of sorting also increases.From the graph we can see for large number of inputs Quick sort takes less time to sort.Hence,Quick sort is better than merge sort.Also the peak in the graph denotes the worst case of sorting