Name	**Siddhesh Sonar**
UID no.	**2021700063**
Subject	**DAA**
Experiment No.	**2**

AIM:	To find and compare running time of merge sort and quick sort algorithm.
Program	
PROBLEM STATEMENT :	For this experiment, you need to implement two sorting algorithms namely Merge and Quick sort methods. Compare these algorithms based on time and space complexity. Time required to sorting algorithms can be performed using high_resolution_clock::now() under namespace std::chrono. You have togenerate1,00,000 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 integers numbers with array indexes numbers A[0..99], A[0..199], A[0..299],..., A[0..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 Merge and Quick by plotting the time required to sort 100000 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 representsthe tunning time to sort 1000 blocks of 100,200,300,...,100000 integer numbers. Note – You have to use C/C++ file processing functions for reading and writing randomly generated 100000 integer numbers.
ALGORITHM:	**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
	
	<u>Quick Sort</u> :-
	• **Step 1**: if (start < end)
	• **Step 2**: p = partition(A, start, end)
	• **Step 3**: QUICKSORT (A, start, p - 1)
	• **Step 4**: QUICKSORT (A, p + 1, end)
	• **Step 5**: Stop
	• **Partition Algorithm**:
	○ **Step 1**: pivot ? A[end]
	○ **Step 2**: i ? start-1
	○ **Step 3**: for j ? start to end -1 {
	○ **Step 4**: do if (A[j] < pivot) {
	○ **Step 5**: then i ? i + 1
	○ **Step 6**: swap A[i] with A[j]
	○ **Step 7**: swap A[i+1] with A[end]
	○ **Step 8**: return i+1

PROGRAM:

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

void rand_filling(int a1[], int a2[], int n)
{
    for (int i = 0; i < n; i++)
    {
        int r = rand();
        a1[i] = a2[i] = r;
    }
    FILE *fp = fopen("./mq_random.txt", "w+");
    for (int i = 0; i < n; i++)
    {
        fprintf(fp, "%d\n", a1[i]);
    }
}

void merge(int a[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = a[l + i];
    for (j = 0; j < n2; j++)
        R[j] = a[m + 1 + j];
    i = j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            a[k] = L[i];
            i++;
        }
        else
        {
            a[k] = R[j];
            j++;
        }
        k++;
    }
```

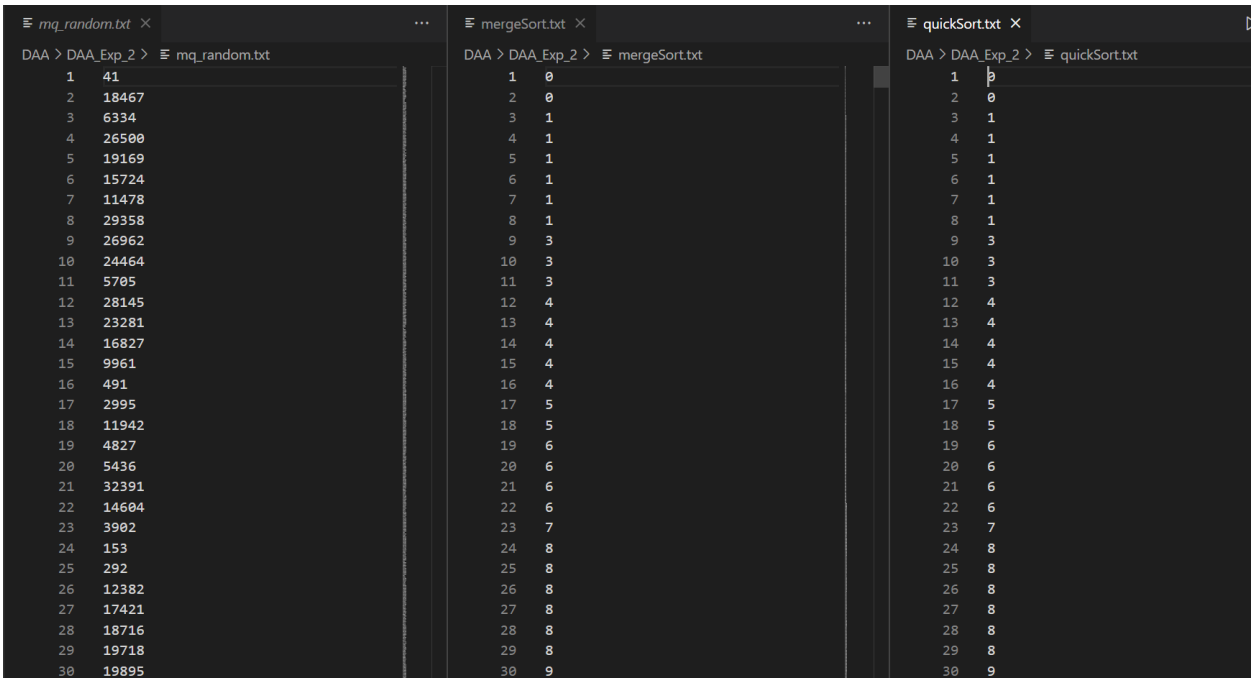
```
    while (i < n1)
    {
        a[k] = L[i];
        i++;
        k++;
    }

    while (j < n2)
    {
        a[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int a[], int l, int r)
{
    if (l < r)
    {
        int m = (l + r) / 2;
        mergeSort(a, l, m);
        mergeSort(a, m + 1, r);
        merge(a, l, m, r);
    }
}

double merge_calculation(int a[], int n)
{
    FILE *fp = fopen("./mergeSort.csv", "w+");
    double total_time = 0;
    fprintf(fp, "n, time\n");
    for (int i = 99; i <= n; i += 100)
    {
        clock_t start, end;
        double time_taken;
        start = clock();
        mergeSort(a, 0, i);
        end = clock();
        time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
        total_time += time_taken;
        fprintf(fp, "%d, %f\n", i + 1, time_taken);
        printf("Sorted from 0 to %d in %.2fs\n", i,
time_taken);
```

```
    }
    fclose(fp);
    fp = fopen("./mergeSort.txt", "w+");
    for (int i = 0; i < n; i++)
    {
        fprintf(fp, "%d\n", a[i]);
    }
    fclose(fp);
    return total_time;
}

void swap(int *x, int *y)
{
    int t = *x;
    *x = *y;
    *y = t;
}

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], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

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

```c
double quick_calculation(int a[], int n)
{
    FILE *fp = fopen("./quickSort.csv", "w+");
    double total_time = 0;
    fprintf(fp, "n, time\n");
    for (int i = 99; i <= n; i += 100)
    {
        clock_t start, end;
        double time_taken;
        start = clock();
        quickSort(a, 0, i);
        end = clock();
        time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
        total_time += time_taken;
        fprintf(fp, "%d, %f\n", i + 1, time_taken);
        printf("Sorted from 0 to %d in %.2fs\n", i,
time_taken);
    }
    fclose(fp);
    fp = fopen("./quickSort.txt", "w+");
    for (int i = 0; i < n; i++)
    {
        fprintf(fp, "%d\n", a[i]);
    }
    fclose(fp);
    return total_time;
}

int main()
{
    int n = 100000;
    int a1[n], a2[n];
    rand_filling(a1, a2, n);
    double merge_time = merge_calculation(a1, n);
    printf("\nTime taken by  Merge Sort: %f\n", merge_time);
    double quick_time = quick_calculation(a2, n);
    printf("\nTime taken by  Quick Sort: %f\n", quick_time);
    return 0;
}
```

RESULT:

Files of the random numbers to be sorted and the number sorted by Merge and Quick Sort:



mq_random.txt

DAA > DAA_Exp_2 > mq_random.txt

```
41
18467
6334
26500
19169
15724
11478
29358
26962
24464
5705
28145
23281
16827
9961
491
2995
11942
4827
5436
32391
14604
3902
153
292
12382
17421
18716
19718
19895
```

mergeSort.txt

DAA > DAA_Exp_2 > mergeSort.txt

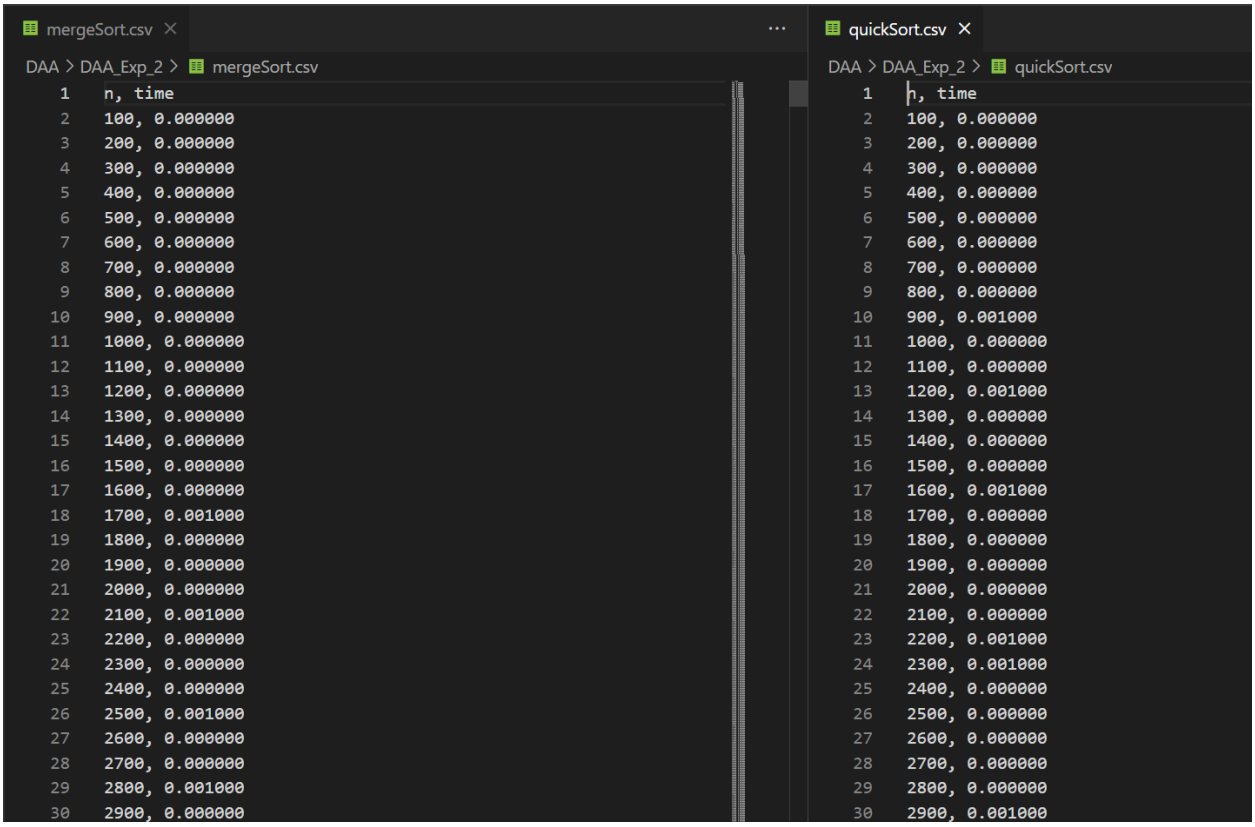
```
0
0
1
1
1
1
1
1
3
3
3
4
4
4
4
4
5
5
6
6
6
6
7
8
8
8
8
8
8
9
```

quickSort.txt

DAA > DAA_Exp_2 > quickSort.txt

```
0
0
1
1
1
1
1
1
3
3
3
4
4
4
4
4
5
5
6
6
6
6
7
8
8
8
8
8
8
9
```

Time taken by Merge and Quick Sort:



mergeSort.csv

DAA > DAA_Exp_2 > mergeSort.csv

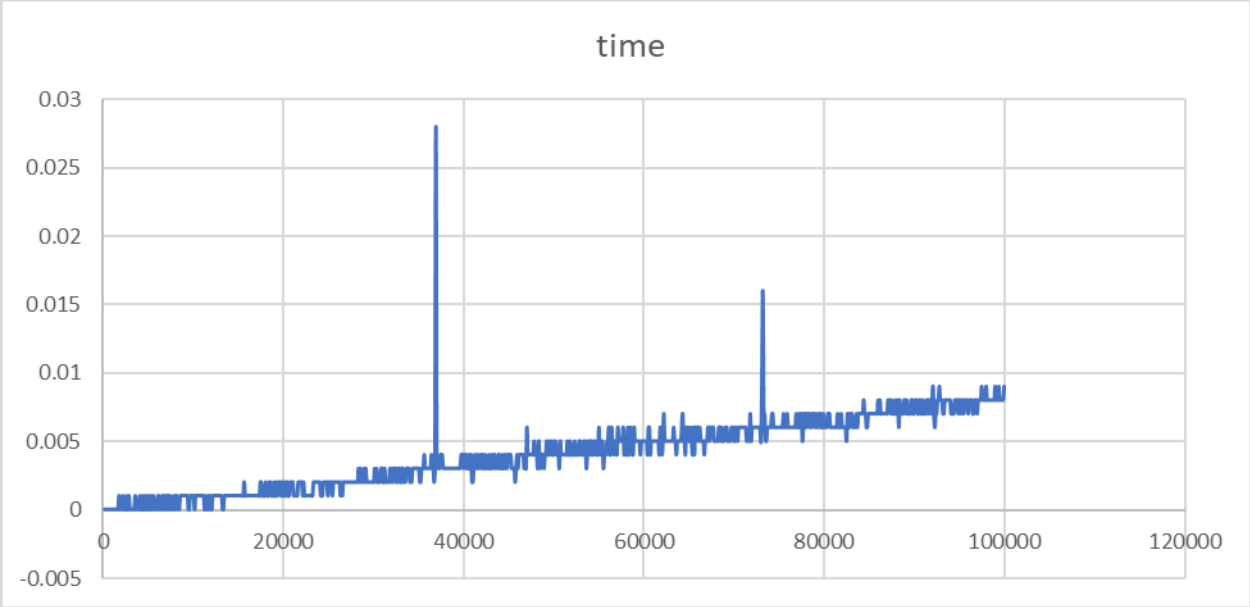
```
n, time
100, 0.000000
200, 0.000000
300, 0.000000
400, 0.000000
500, 0.000000
600, 0.000000
700, 0.000000
800, 0.000000
900, 0.000000
1000, 0.000000
1100, 0.000000
1200, 0.000000
1300, 0.000000
1400, 0.000000
1500, 0.000000
1600, 0.000000
1700, 0.001000
1800, 0.000000
1900, 0.000000
2000, 0.000000
2100, 0.001000
2200, 0.000000
2300, 0.000000
2400, 0.000000
2500, 0.001000
2600, 0.000000
2700, 0.000000
2800, 0.001000
2900, 0.000000
```

quickSort.csv

DAA > DAA_Exp_2 > quickSort.csv

```
n, time
100, 0.000000
200, 0.000000
300, 0.000000
400, 0.000000
500, 0.000000
600, 0.000000
700, 0.000000
800, 0.000000
900, 0.001000
1000, 0.000000
1100, 0.000000
1200, 0.001000
1300, 0.000000
1400, 0.000000
1500, 0.000000
1600, 0.001000
1700, 0.000000
1800, 0.000000
1900, 0.000000
2000, 0.000000
2100, 0.000000
2200, 0.001000
2300, 0.001000
2400, 0.000000
2500, 0.000000
2600, 0.000000
2700, 0.000000
2800, 0.000000
2900, 0.001000
```

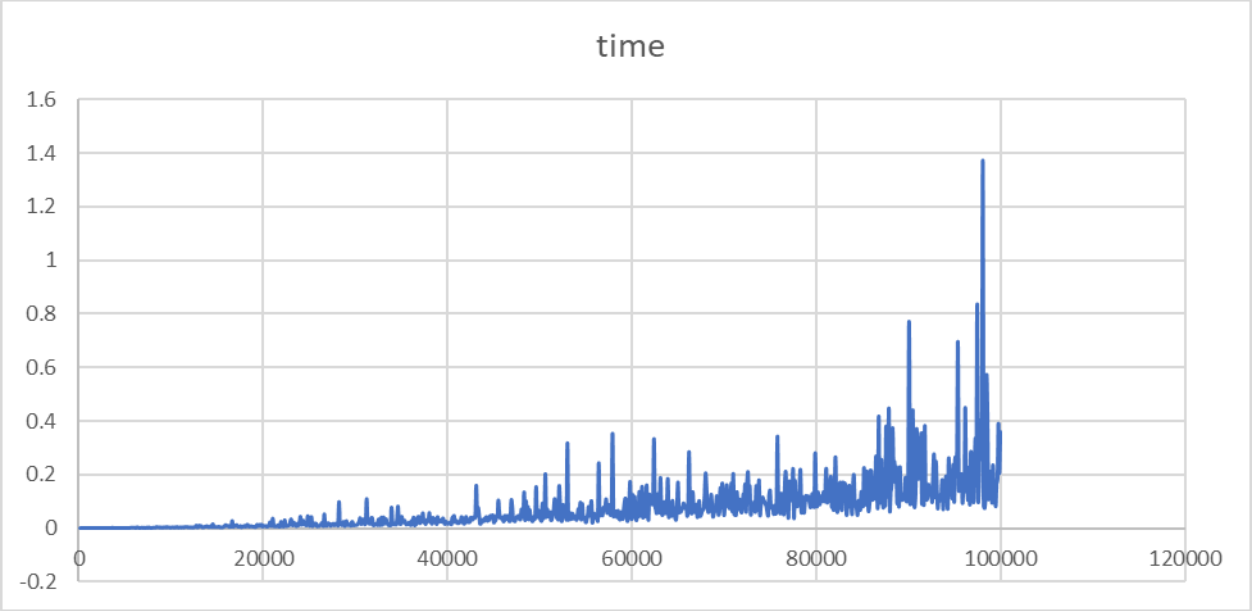
Graph for Merge Sort:



Best-Case Complexity: O(nlog(n))

Worst-Case Complexity: O(nlog(n))

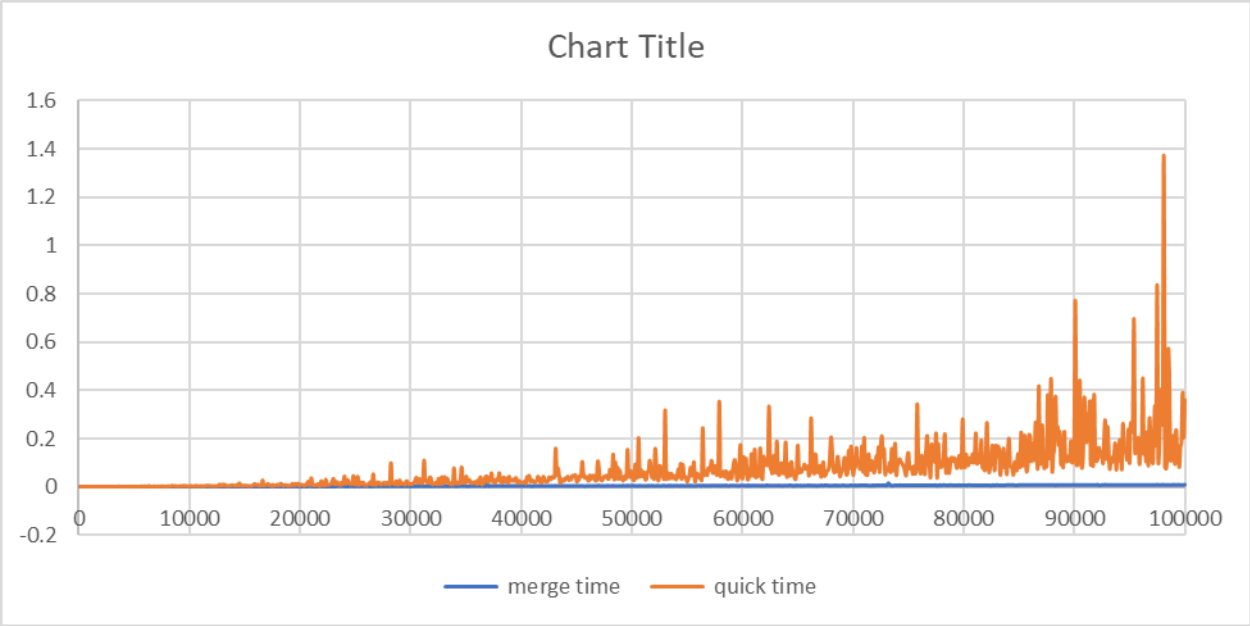
Graph for Quick Sort:



Best-Case Complexity: O(nlog(n))

Worst-Case Complexity: O(n2)

Graph of Merge Sort and Quick Sort Combined:



What we observe from the graphs is that Merge Sort is faster and more efficient than Quick Sort method. Merge Sort is equally efficient for datasets of any size whereas Quick Sort is efficient for small datasets.

CONCLUSION:	Successfully performed Merge Sort and Quick Sort for sorting of 1 lakh random numbers and plotted and compared the graphs for the same.