Bharatiya 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	**Rajas Baadkar**
UID No.	**2021300007**
Subject	**Design And Analysis Of Algorithm**
Class	**Comps A**
Experiment No.	**3**
AIM	Experiment on finding the running time of merge sort and quick sort algorithm

Theory –

1. **Merge Sort -** Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array. In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

2. **Quick Sort -** Like Merge Sort, Quicksort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quicksort that pick pivot in different ways.

    a. Always pick the first element as a pivot.

    b. Always pick the last element as a pivot (implemented below)

    c. Pick a random element as a pivot.

    d. Pick median as the pivot.

    The key process in quickSort is a partition (). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a

sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Algorithm –

1. 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.

2. Quick Sort –

a. Quick Sort Algo

```
QUICKSORT (array A, start, end)
{
    if (start < end)
    {
            p = partition(A, start, end)
        QUICKSORT (A, start, p - 1)
        QUICKSORT (A, p + 1, end)
    }
}
```

b. Partition Algo

PARTITION (array A, start, end)

```
{
    pivot ? A[end]
    i ? start-1
    for j ? start to end -1 {
        do if (A[j] < pivot) {
            then i ? i + 1
            swap A[i] with A[j]
        }
    }
 swap A[i+1] with A[end]
 return i+1
}
```

Program:

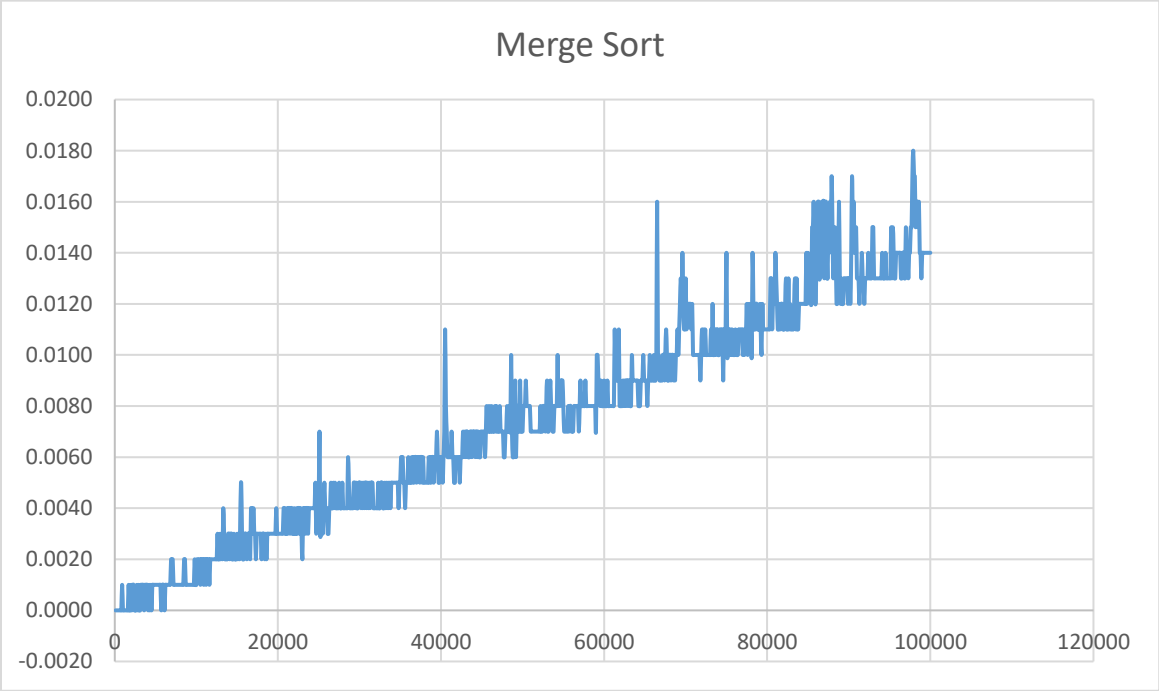
```c
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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

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

```
    }
}

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


int main(){

    FILE *fptr, *sPtr;
    int index=99;
    int arrNums[100000];
    clock_t t;
    fptr = fopen("Random.txt", "r");
    sPtr = fopen("mTimes.txt", "w");
    for(int i=0; i<=999; i++){
        for(int j=0; j<=index; j++){
            fscanf(fptr, "%d", &arrNums[j]);
        }
        t = clock();
        mergeSort(arrNums, 0, index); //have to use merge sort and quick sort
both hence do both once and find the output
        t = clock() - t;
        double time_taken = ((double)t)/CLOCKS_PER_SEC;
        fprintf(sPtr, "%lf\n", time_taken);
        printf("%d\t%lf\n", (i+1), time_taken);
        index = index + 100;
        fseek(fptr, 0, SEEK_SET);
    }
    fclose(sPtr);
    fclose(fptr);

    return 0;
}
```

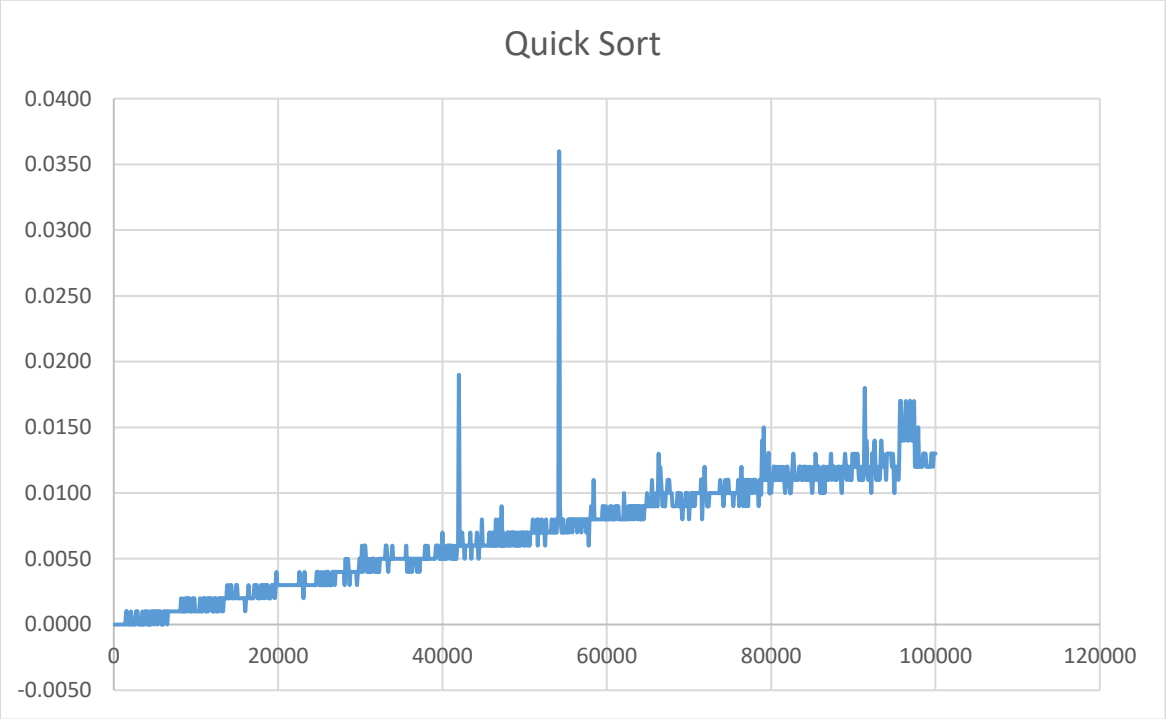
Result Analysis:

1. Merge Sort –



- Time complexity is O (n logn)
- It is a quite fast process, even for sorting 100000 numbers using merge sort it took in total 0.013 secs.

2. Quick Sort –



- Time complexity is O(n^2).
- It is a quite fast process, even for sorting 100000 numbers using merge sort it took in total 0.013 secs.

3. Comparison between both the Sorting Algorithms

- As such both the algorithms for 100000 numbers didn't show much of a difference, we can say both had their best-case time complexity but it will surely deviate more at a more larger scale.
- There were few spikes in the middle but they were all because while sorting as the data was random, the numbers were too mixed and long separated.

SE – COMP (SE-A) **Sub- DAA Lab**