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

Experiment No.	2
Aim	Experiment on finding the running time of an algorithm.
Name	Aarush Kinhikar
UID No.	2021300063
Class & Division	SE Comps A, Batch D

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.

One thing that you might wonder is what is the specialty of this algorithm. We already have a number of sorting algorithms then why do we need this algorithm? One of the main advantages of merge sort is that it has a time complexity of O(n log n), which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.

Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement. It is often used in conjunction with other algorithms, such as quicksort, to improve the overall performance of a sorting routine.

2. Quick Sort

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.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e., select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

Algorithm:

1. Merge Sort

Step 1: Find the middle index of the array.
Middle = 1 + (last – first)/2
Step 2: Divide the array from the middle.
Step 3: Call merge sort for the first half of the array
MergeSort(array, first, middle)
Step 4: Call merge sort for the second half of the array.
MergeSort(array, middle+1, last)
Step 5: Merge the two sorted halves into a single sorted array.

2. Quick Sort

Step 1 - Consider the first element of the list as pivot (i.e., Element at first position in the array).
Step 2 - Define two variables i and j. Set i and j to first and last elements of the list respectively.
Step 3 - Increment i until arr[i] > pivot then stop.
Step 4 - Decrement j until arr[j] < pivot then stop.
Step 5 - If i < j then exchange list[i] and list[j].
Step 6 - Repeat steps 3,4 & 5 until i > j.
Step 7 - Exchange the pivot element with arr[j] element..

Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<math.h>
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);
    }
}
void quicksort(int number[25],int first,int last){
    int i, j, pivot, temp;
    if(first<last){
        pivot=first;
        i=first;
        j=last;
        while(i<j){
            while(number[i]<=number[pivot]&&i<last)
```

```
                i++;
            while(number[j]>number[pivot])
                j--;
            if(i<j){
                temp=number[i];
                number[i]=number[j];
                number[j]=temp;
            }
        }
        temp=number[pivot];
        number[pivot]=number[j];
        number[j]=temp;
        quicksort(number,first,j-1);
        quicksort(number,j+1,last);
    }
}
void getNumbers(int arr[],int x){
    for(int i=0;i<x*100;i++){
        arr[i] = rand()%1000 + 1;
    }
}
int main(){
    int i, count, arr[100000];
    int x=1;
    clock_t start,end;
    printf("Merge Sort:\n");
    for(x=1;x<=1000;x++){
        getNumbers(arr,x);
        start = clock();
        mergeSort(arr, 0, x*100 - 1);
        end = clock();
        printf("%f\n",((float)(end-start))/CLOCKS_PER_SEC);
    }
    printf("Quick Sort:\n");
    for(x=1;x<=1000;x++){
        getNumbers(arr,x);
        start = clock();
        quicksort(arr,0,x*100-1);
        end = clock();
        printf("%f\n",((float)(end-start))/CLOCKS_PER_SEC);
    }
    return 0;
}
```

Random Pivot Quick Sort:

Program:

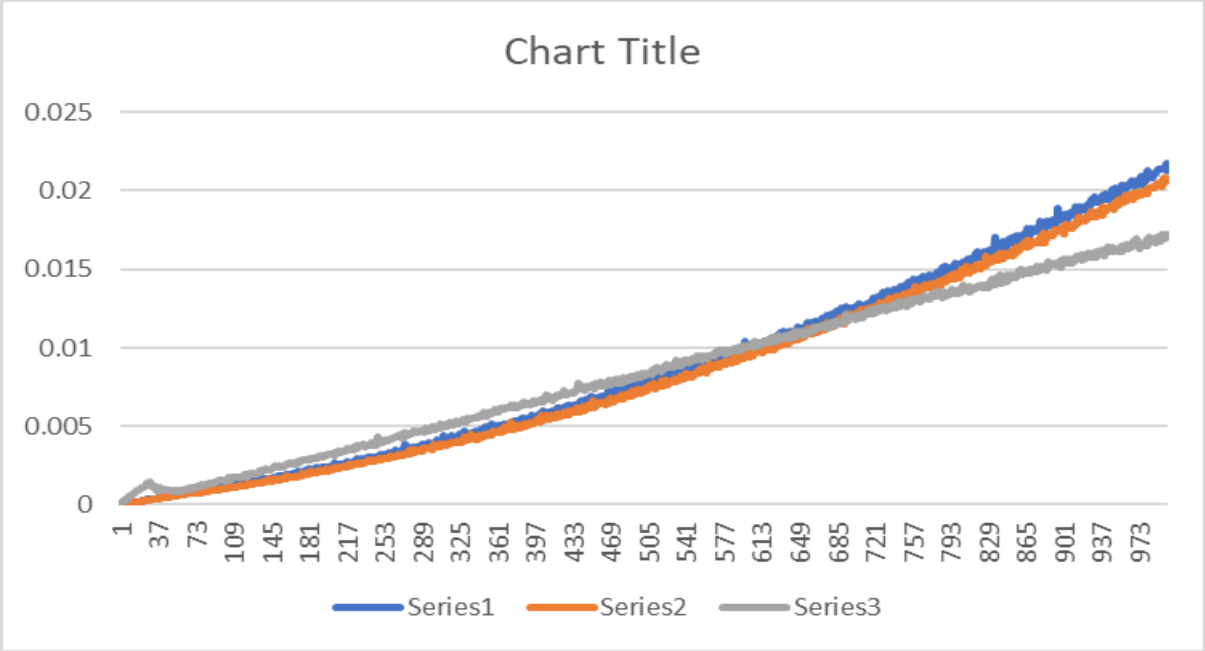
```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<math.h>
void quicksort(int number[],int first,int last){
    int i, j, pivot, temp;
    if(first<last){
        pivot=first+(rand()%(last-first+1));
        i=first;
        j=last;
        while(i<j){
            while(number[i]<=number[pivot]&&i<last)
                i++;
            while(number[j]>number[pivot]&&j>first)
                j--;
            if(i<j){
                if(i==pivot){
                    pivot=j;
                }
                if(j==pivot){
                    pivot=i;
                }
                temp=number[i];
                number[i]=number[j];
                number[j]=temp;
            }
        }
        temp=number[pivot];
        number[pivot]=number[j];
        number[j]=temp;

        quicksort(number,first,j-1);
        quicksort(number,j+1,last);
    }
}
void getNumbers(int arr[],int x){
    srand(time(0));
    for(int i=0;i<x*100;i++){
        arr[i] = rand()%1000 + 1;
    }
}
void printArray(int arr[],int size){
```

```
    int i;
    for(i=0;i<size;++i){
        printf("%d ",arr[i]);
    }
}
int main(){
    int i, count, arr[1000000];
    int x=1;
    __clock_t start,end;
    printf("Quick Sort:\n");
    for(x=1;x<=1000;x++){
        getNumbers(arr,x);
        start = clock();
        quicksort(arr,0,x*100-1);
        end = clock();
        printf("%f\n",((float)(end-start))/CLOCKS_PER_SEC);
    }
    return 0;
}
```

Observations:

The graph of the obtained results when plotted is as follows:



Series 1: Merge Sort
Series 2: Quick Sort (Fixed Pivot)
Series 3: Quick Sort (Random Pivot)

Conclusion:

The results obtained for the execution time for both the algorithms are quite good. This shows that both the algorithms are quite efficient. When compared by plotting a graph of the results it can be concluded that both algorithms are quite good but merge sort is slightly better than quick sort when the number of inputs increases to a large amount.