Sardar Patel Institute of Technology

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

Experiment No.	1-c
Aim	Experiment on finding the running time of an algorithm of Merge Sort and Quick Sort.
Name	Harsh Mukesh Jain
UID No.	2021300048
Class & Division	SE Comps A (C – Batch)
Date of Performance	12-02-2023
Date of Submission	19-02-2023

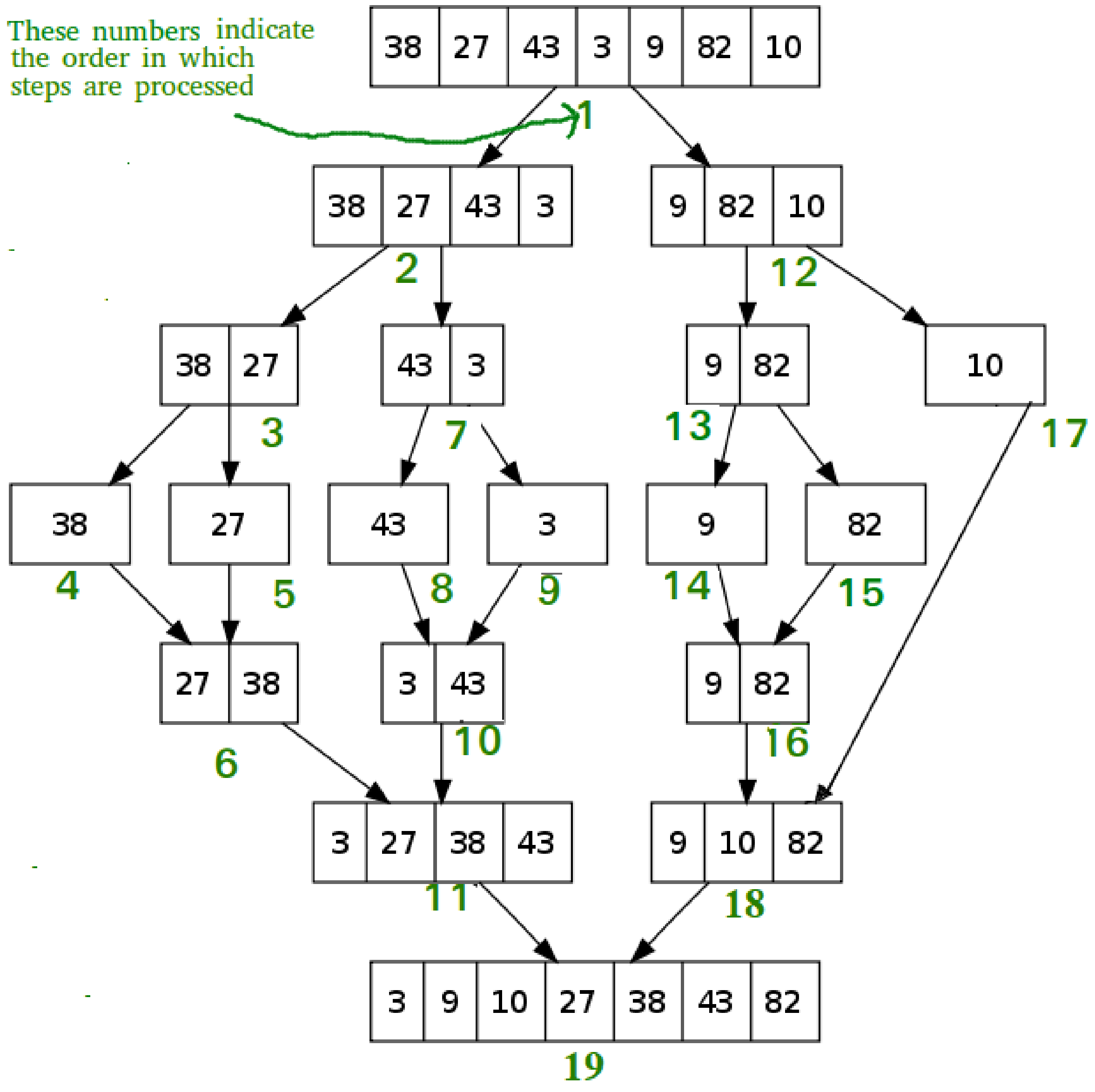
Theory/Experiment:

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 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.


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
2
9 82 10
12
38 27
3
43 3
7
9 82
13
10
17
38
4
27
5
43
8
3
9
9
14
82
15
27 38
6
3 43
10
9 82
16
3 27 38 43
11
9 10 82
18
3 9 10 27 38 43 82
19

PSEUDO CODE:-

```
merge(arr , s , e)
    mid = s + (e - s) / 2
    for i = 0 to (mid – s + 1)
        arr1[i] = arr[k++]
    end for
    for i = 0 to (e – mid)
        arr2[i] =arr[k++]
    end for
    k = s , ptr1 = 0 , ptr2 = 0

    while (ptr1 < l1 && ptr2 < l2)
        if(arr1[ptr1] < arr2[ptr2])
            arr[k++] = arr1[ptr1++]
        else arr[k++] = arr2[ptr2++]
    end while

    while(ptr1 < l1)
        arr[k++] = arr1[ptr1++];
    end while

    while(ptr2 < l2)
        arr[k++] = arr2[ptr2++];
    end while

    free(arr1)
    free(arr2)


mergeSort(arr , s , e)
    if(s >= e) return
    mid = s + (e - s) / 2
    mergeSort(arr , s , mid)
    mergeSort(arr , mid + 1 , e)
    merge(arr , s , e)
```

CODE :-

```
void merge(int *arr , int s , int e)
{
	int mid = s + (e - s) / 2;

	int l1 = mid - s + 1;
	int l2 = e - mid;

	int *arr1 = (int *)malloc(sizeof(int) * l1);
	int *arr2 = (int *)malloc(sizeof(int) * l2);

	int k = s;
	for(int i = 0 ; i < l1 ; i++)
	{
		arr1[i] = arr[k++];
	}
	for(int i = 0 ; i < l2 ; i++)
	{
		arr2[i] = arr[k++];
	}

	int ptr1 = 0 , ptr2 = 0;
	k = s;
	while(ptr1 < l1 && ptr2 < l2)
	{
		if(arr1[ptr1] < arr2[ptr2])
		{
			arr[k++] = arr1[ptr1++];
		}
		else arr[k++] = arr2[ptr2++];
	}
	while(ptr1 < l1)
	{
		arr[k++] = arr1[ptr1++];
	}
	while(ptr2 < l2)
```

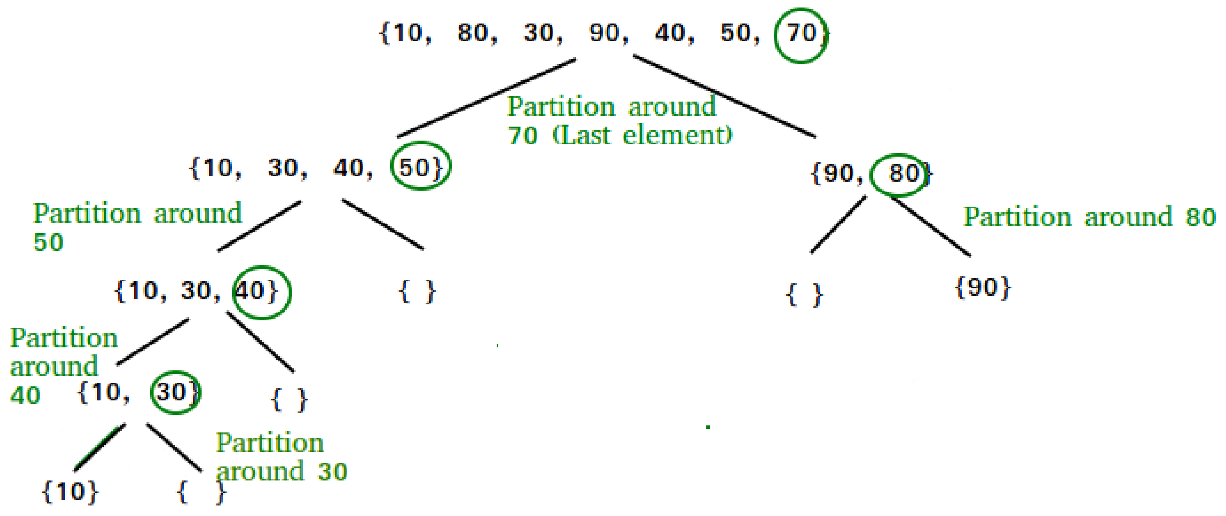
```
        {
                arr[k++] = arr2[ptr2++];
        }
        free(arr1);
        free(arr2);

}
void mergeSort(int *arr , int s , int e)
{
        if(s >= e) return;

        int mid = s + (e - s) / 2;

        mergeSort(arr , s , mid);
        mergeSort(arr , mid + 1 , e);

        merge(arr , s , e);

}
```

2.QUICK SORT

Like Merge Sort , **QuickSort** is a Divide and Conquer Method. 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.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- 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.



PSEUDO CODE:-

```
quickSort(arr , s , e)
    if(s >= e) return

    pivot = arr[s] , cnt = 0 , i = s , j = e
    for i = s + 1 to i <= e
        if(arr[s] <= pivot) cnt++
    end for

    pivotIndex = s + cnt
    swap(arr[pivotIndex] , s)

    while(i < pivotIndex && j > pivotIndex)
        while(arr[i] <= pivot) i++
        while(arr[j] > pivot) j--
    swap(arr[i++] , arr[j--])
    end while

    quickSort(arr, s, pivotIndex - 1);
    quickSort(arr, pivotIndex + 1, e);
```

CODE :-

```
void quickSort(int arr[], int s, int e)
{
    if(s >= e) return ;

    int pivot = arr[s];
    int cnt = 0 , i = s, j = e;
    for(int i = s+1; i<=e; i++)
            {
        if(arr[i] <=pivot) cnt++;
    }

    int pivotIndex = s + cnt;
    int temp = arr[pivotIndex];
    arr[pivotIndex] = arr[s];
    arr[s] = temp;


    while(i < pivotIndex && j > pivotIndex) {

        while(arr[i] <= pivot)
        {
            i++;
        }

        while(arr[j] > pivot) {
            j--;
        }

        if(i < pivotIndex && j > pivotIndex) {

                            int temp = arr[i];
                            arr[i++] = arr[j];
                            arr[j--] = temp;
        }

    }
    quickSort(arr, s, pivotIndex - 1);

    quickSort(arr, pivotIndex + 1, e);

}
```

CODE FOR GIVEN QUESTION:-

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

void quickSort(int arr[], int s, int e)
{
   if(s >= e) return ;


         int pivot = arr[s];

   int cnt = 0 , i = s, j = e;
   for(int i = s+1; i<=e; i++)
         {
      if(arr[i] <=pivot) cnt++;
   }

   int pivotIndex = s + cnt;
   int temp = arr[pivotIndex];
   arr[pivotIndex] = arr[s];
   arr[s] = temp;


   while(i < pivotIndex && j > pivotIndex) {

      while(arr[i] <= pivot)
      {
         i++;
      }

      while(arr[j] > pivot) {
         j--;
      }

      if(i < pivotIndex && j > pivotIndex) {

                        int temp = arr[i];
                        arr[i++] = arr[j];
                        arr[j--] = temp;
```

```
        }

    }

    quickSort(arr, s, pivotIndex - 1);

    quickSort(arr, pivotIndex + 1, e);

}

void merge(int *arr , int s , int e)
{
            int mid = s + (e - s) / 2;

            int l1 = mid - s + 1;
            int l2 = e - mid;

            int *arr1 = (int *)malloc(sizeof(int) * l1);
            int *arr2 = (int *)malloc(sizeof(int) * l2);

            int k = s;
            for(int i = 0 ; i < l1 ; i++)
            {
                        arr1[i] = arr[k++];
            }
            for(int i = 0 ; i < l2 ; i++)
            {
                        arr2[i] = arr[k++];
            }

            int ptr1 = 0 , ptr2 = 0;
            k = s;
            while(ptr1 < l1 && ptr2 < l2)
            {
                        if(arr1[ptr1] < arr2[ptr2])
                        {
                                    arr[k++] = arr1[ptr1++];
                        }
                        else arr[k++] = arr2[ptr2++];
            }
```

```
	while(ptr1 < l1)
	{
		arr[k++] = arr1[ptr1++];
	}
	while(ptr2 < l2)
	{
		arr[k++] = arr2[ptr2++];
	}
	free(arr1);
	free(arr2);

}
void mergeSort(int *arr , int s , int e)
{
	if(s >= e) return;

	int mid = s + (e - s) / 2;

	mergeSort(arr , s , mid);
	mergeSort(arr , mid + 1 , e);

	merge(arr , s , e);

}

int main()
{
	//Generating random numbers in "Random_No.txt"

	FILE *fptr = fopen("Random_No1.txt" , "w");

	for(int i = 0 ; i < size ; i++) {
		r = rand()%100000 + 1;
		fprintf(fptr , "%d\n" , r);
	}
	fclose(fptr);

	//Reading random numbers from "Random_No.txt"
	int size = 100000 , r;
	int block = 100;
	//Storing sorted numbers in "Sorted.txt"
	FILE *fptr1 = fopen("Sorted.txt" , "w");
	for(int i = block ; i <= size ; i = i + 100)
	{
		printf("\nSorting Block number :%d\n",(i/block));

		FILE *fptr = fopen("Random_No1.txt" , "r");
		int *arr = (int *)malloc(sizeof(int) * i);

		for(int j = 0 ; j < i ; j++)
		{
			fscanf(fptr , "%d" , &r);
			arr[j] = r;
			//printf("%d\t",arr[j]);

		}
		clock_t start , end;
		start = clock();
		quickSort(arr, 0 , i - 1);
		//mergeSort(arr , 0 , i - 1);
		end = clock();
		printf("%f\n", ((double)(end - start)/CLOCKS_PER_SEC));

		printf("\n");
		for(int j = 0 ; j < i ; j++)
		{

			//printf("%d\t",arr[j]);
			fprintf(fptr , "%d\n" , arr[j]);

		}
		printf("\n");
		free(arr);
		fclose(fptr);
	}
	fclose(fptr1);
}
```

INPUT:-

Generated random 1,00,000 numbers in Random_No.txt file using rand() function and used this input as 1000 blocks of 100000 integer numbers to Merge and Quick sorting algorithms.

OUTPUT:-

1) Stored the randomly generated 100000 integer numbers to a text file named Random_No1.txt.
2) Below represent the graph of 2 function i.e Merge Sort and Quick Sort plot on y axis versus the number of blocks marked on x axis.

3)OBSERVATION:-

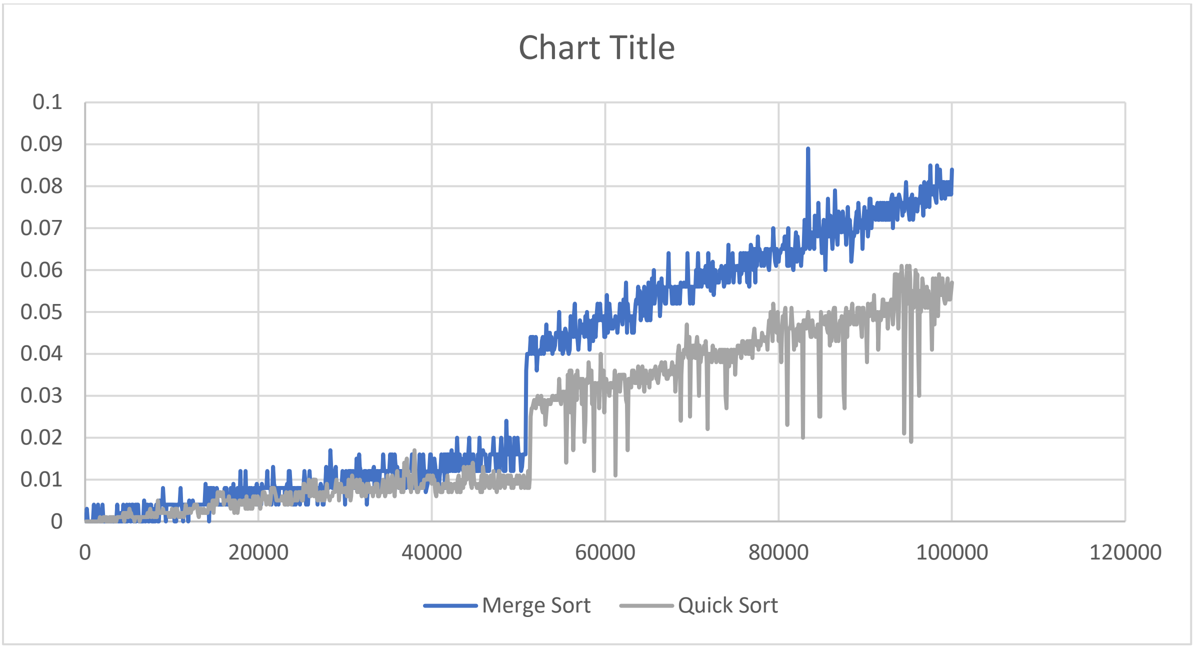
i.) Merge Sort v/s Quick Sort (in terms of time complexity)

Merge sort has a time complexity of O(nlogn) in the worst, average, and best cases.

Merge sort is a divide-and-conquer algorithm that recursively divides the input array into smaller subarrays until each subarray contains just one element. It then merges the subarrays in a sorted order to produce the final sorted array.

Quicksort has a time complexity of O(nlogn) in the average case, but in the worst case, it can have a time complexity of O(n^2) if the pivot selection is not done carefully.

Quicksort is a divide-and-conquer algorithm that recursively partitions the input array into smaller subarrays based on the pivot element, which is chosen as a "median" element. This helps to speed up the sorting process, but if the pivot is chosen poorly, it can lead to the worst-case time complexity of O(n^2).



The above graph depicts the time required to sort a block of 100 elements is less for merge sort as compared to quick sort.

ii.) <u>**Merge Sort v/s Quick Sort (in terms of space complexity)**</u>

Merge sort has a space complexity of O(n) as it requires additional space to store the divided subarrays.

Quicksort has a space complexity of O(logn) due to the recursion stack used in the sorting process.

CONCLUSION:-

By performing this experiment , I understood the concept of two sorting algorithms i.e Merge Sort and Quick Sort.
Using time.h header file , I calculated time taken by both algorithms to sort 1,000 blocks i.e 1,00,000 elements and plotted a graph in Excel.
It was clear from the graph that the performance of merge sort is best as compared to quick sort.