PART A

(PART A: TO BE REFFERED BY STUDENTS)

Experiment No.03

A.1 Aim:

Write a program to implement Quick sort using Divide and Conquer Approach and analyze its complexity.

A.2 Prerequisite: -

A.3 Outcome:

After successful completion of this experiment students will be able to analyze the time complexity of various classic problems.

A.4 Theory:

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot

Algorithm:

```
/* low --> Starting index, high --> Ending index */
quickSort(arr[], low, high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[pi] is now
         at right place */
      pi = partition(arr, low, high);
```

```
        quickSort(arr, low, pi - 1); // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}

/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
   array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1) // Index of smaller element and indicates the
                  // right position of pivot found so far

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Analysis of QuickSort:

Time taken by QuickSort, in general, can be written as following.

$T(n) = T(k) + T(n-k-1) + \quad (n)$

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

$T(n) = T(0) + T(n-1) + \quad (n)$

Which is equivalent to

$T(n) = T(n-1) + \quad (n)$

The solution of above recurrence is (n^2).
Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

$T(n) = 2T(n/2) + \quad (n)$

The solution of above recurrence is (nLogn). It can be solved using case 2 of Master Theorem.
Average Case:
To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

$T(n) = T(n/9) + T(9n/10) + \quad (n)$

Solution of above recurrence is also O(nLogn)

PART B

(PART B: TO BE COMPLETED BY STUDENTS)

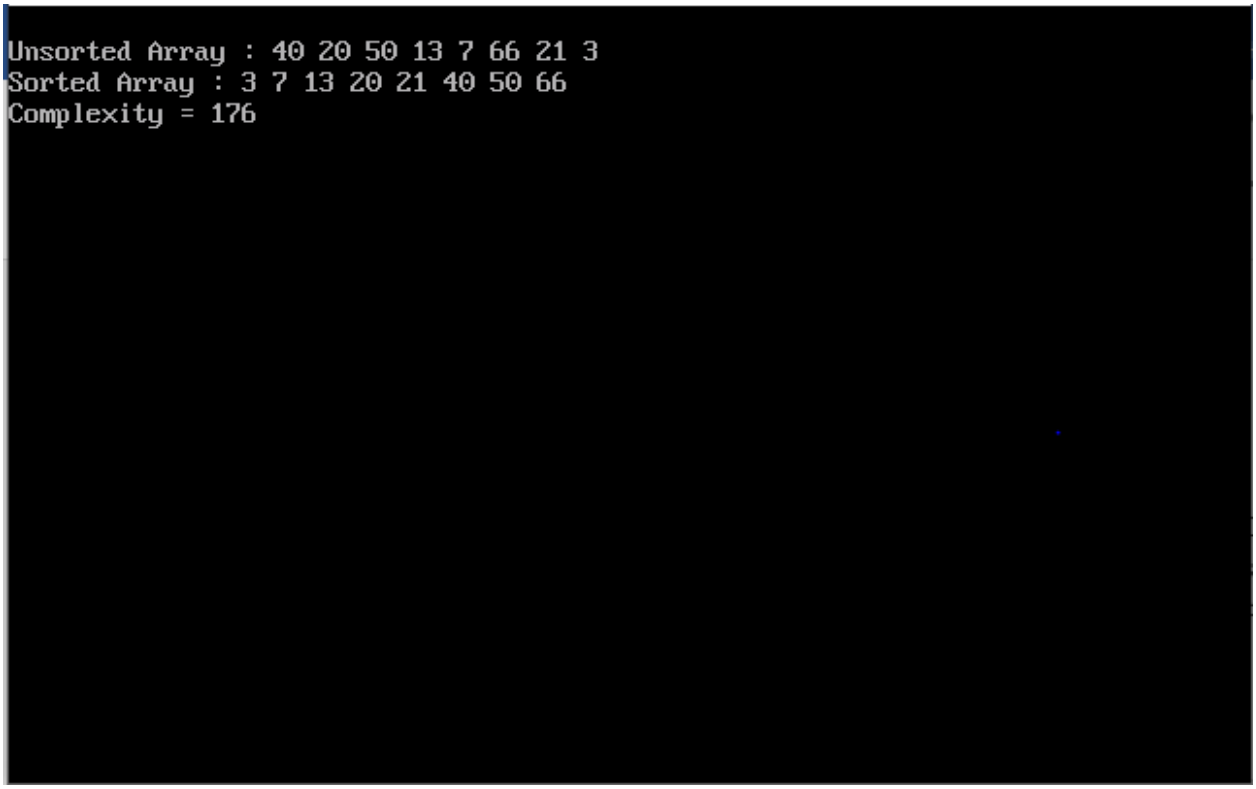
Roll No.: 5	Name: Tavishaa Jaiswal
Class: SE(COMP)	Batch: C1
Date of Experiment:	Date of Submission:
Grade:	

B.1 Software Code written by student:

```
#include<conio.h>
#include<stdio.h>
int a[8]={40,20,50,13,7,66,21,3};  c++;
void quick(int low,int high)
{
  int j;  c++;
  if(low<high)
  {
    c++;
    j=partition(low,high+1);  c++;
    quick(low,j-1); c++;
    quick(j+1,high);  c++;
  }
}
int partition(int low,int high)
{
  int key,i,j,temp;
  i=low; c++;
  j=high; c++;
  key=a[low];  c++;
  while(i<j)
  {
    c++;
    i++;  c++;
    while((a[i]<=key)&&(i<high))
    {
      c++;
      i++;  c++;
    }
    j--;  c++;
    while((a[j]>key)&&(j>low))
    {
      c++;
```

```
    j--;  c++;
   }
   if(i<j)
   {
    c++;
    temp=a[i];  c++;
    a[i]=a[j]; c++;
    a[j]=temp;  c++;
   }
   else
   {
    c++;
    temp=a[low];  c++;
    a[low]=a[j]; c++;
    a[j]=temp;    c++;
   }
  }
  return j;
}
void main()
{
  int i;
  clrscr();
  printf("\nUnsorted Array : "); c++;
  for(i=0,c++;c++,i<8;i++,c++)
  {
    printf("%d ",a[i]);  c++;
  }
  quick(0,7);
  printf("\nSorted Array : ");  c++;
  for(i=0,c++;c++,i<8;i++,c++)
  {
    printf("%d ",a[i]);  c++;
  }
  printf("\nComplexity = %d",c);  c++;
  getch();
}
```

B.2 Input and Output:



B.3 Observations and learning:

Quick Sort is a sorting algorithm that uses Divide and Conquer approach. The name itself portrays Quick Sort which portrays that it is faster than all other algorithms. Quick Sort algorithm picks a pivot element and performs sorting based on the pivot element. In Divide and Conquer approach we break down the array into sub-arrays and conquer them. Quick Sort is the most efficient algorithm among all other sorting algorithms, as sorting can be done in O(nlogn) time. In this experiment we have successfully implemented quick sort using the divide and conquer approach using C language.

B.4 Conclusion:

Implement Quick sort using Divide and Conquer Approach using C language.

B.5 Question of Curiosity

Q1: Derive time complexity of Quick sort?

Q1. In last n-1 pass, there is array of size 2 in a single comparision.
∴ Total no. of comparisions will be

$$\text{Total comp} = (n-1) + (n-2) + \dots + 1.$$

$$= \frac{(n-1)\,n}{2}$$

$$T(n) = O(n^2).$$

Q2: What is worst case and best case time complexity of Quick sort?

Q2. Worst case complexity : $T(n) = O(n^2)$
Best case Complexity : $T(n) = O(n \log n)$.

Q3: How many comparisons are done in Quick sort?

Q3. Number of comparisions = $\frac{N(N-1)}{2}$

Q4: Can we say Quick sort works best for large n? Yes or no? Reason?

Q4. Yes, in a quick sort is faster and also does not require large memory. Therefore, quick sort is better suited for large data sets.

Q5: Why quick sort is better than merge sort?

Q5. Quick sort is better than merge sort :-
i) Auxillary Space : Quick sort does not require additional space (in place algo). Merge sort requires large memory.
ii) Worst case of quick sort is $O(A)$ can be avoided by using randomized quick sort.
iii) Locality of reference.

****