# EXPERIMENT 4

**Experiment No:** 4 **Date:** 17/04/2021

**Aim:** Implementation of Quick Sort Algorithm and obtain its step count

**Theory:**

## QUICK SORT

- Quick Sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
- A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.
- There are many different versions of Quick Sort that pick pivot in different ways.
    - Always pick first element as pivot.
    - Always pick last element as pivot (implemented below)
    - Pick a random element as pivot.
    - Pick median as pivot.
- Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.
- This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(n2), respectively.

# EXPERIMENT 4

## ALGORITHM

**Algorithm for quicksort:**

```
/* 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

   }

}
```

**Algorithm for Partition:**

```
partition (arr[], low, high)

{

   // pivot (Element to be placed at right position)

   pivot = arr[high];
```

EXPERIMENT 4

```
    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)
}
```

**Algorithm for swapping:**

```
algorithm interchange(a,i,j)
{
        temp=a[i];
        a[i]=a[j];
        a[j]=temp;
}
```

# EXPERIMENT 4

## ILLUSTRATION OF PARTITION()

arr[] = {10, 80, 30, 90, 40, 50, 70}

Indexes: 0 1 2 3 4 5 6

low = 0,

high = 6,

pivot = arr[h] = 70

Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1

j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 0

arr[] = {10, 80, 30, 90, 40, 50, 70} ***// No change as i and j***

***// are same***

j = 1 : Since arr[j] > pivot, do nothing

***// No change in i and arr[]***

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 1

arr[] = {10, 30, 80, 90, 40, 50, 70} ***// We swap 80 and 30***

# EXPERIMENT 4

j = 3 : Since arr[j] > pivot, do nothing

***// No change in i and arr[]***

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 2

arr[] = {10, 30, 40, 90, 80, 50, 70} ***// 80 and 40 Swapped***

j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

i = 3

arr[] = {10, 30, 40, 50, 80, 90, 70} ***// 90 and 50 Swapped***

We come out of loop because j is now equal to high-1

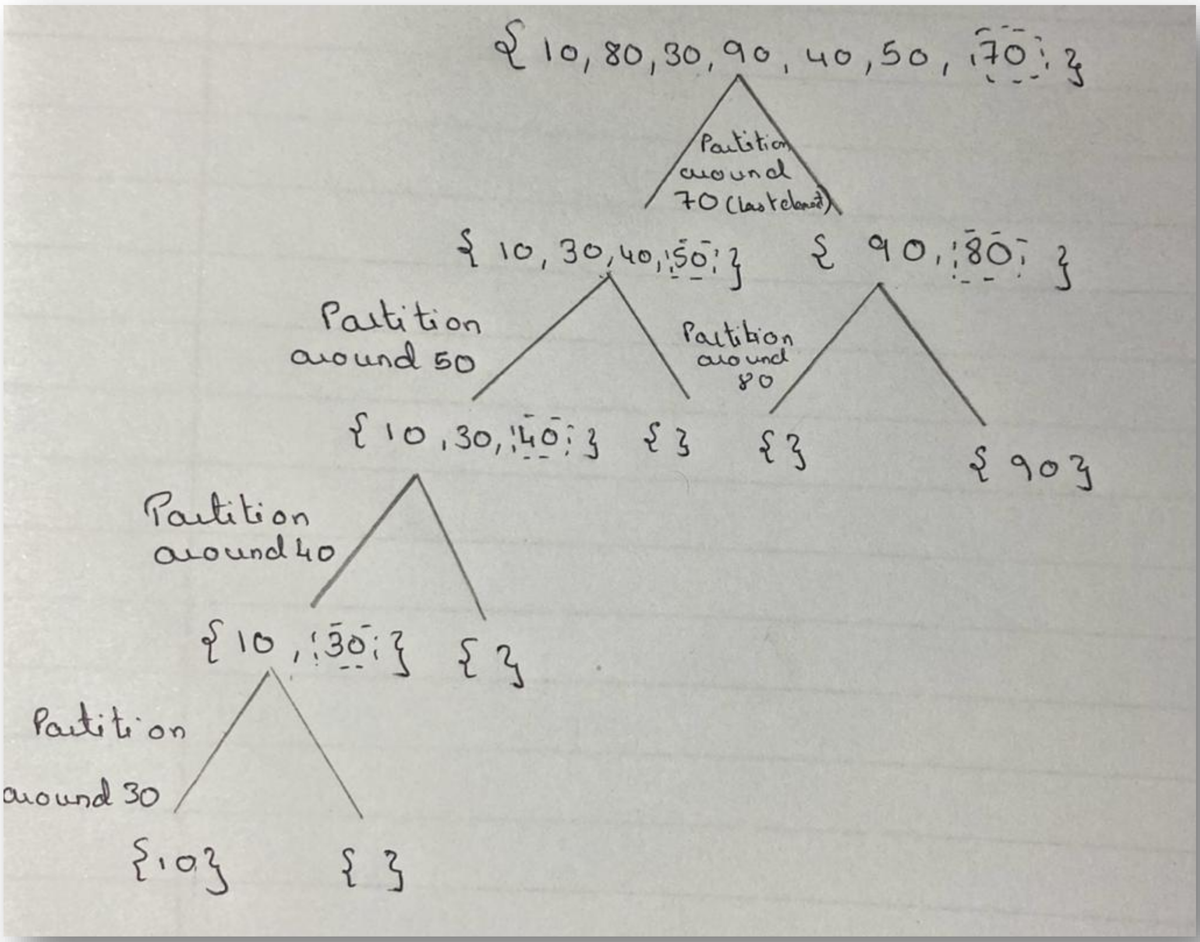
Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)

arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

# EXPERIMENT 4

## **RECURSIVE TREE CALL DIAGRAM**



# EXPERIMENT 4

## ANALYSIS OF QUICK SORT

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

***T(n) = T(k) + T(n-k-1) + O(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) + O(n)***
***which is equivalent to***
***T(n) = T(n-1) + O(n)***
***The solution of above recurrence is O(n2).***

## EXPERIMENT 4

- **Best Case:**
    - The best case occurs when the partition process always picks the middle element as pivot.
    - Following is recurrence for best case.

$$T(n) = 2T(n/2) + O(n)$$

    - The solution of above recurrence is O(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.

$$T(n) = T(n/9) + T(9n/10) + O(n)$$

    - Solution of above recurrence is also O(nLogn)

# EXPERIMENT 4

## **CODE**

```
#include<iostream>
using namespace std;

int count=0;
void swap(int &a, int &b)
{
   int t=a;
   a=b;
   b=t;
          count=count+3;
}

int partition(int arr[], int low, int high)
{
          int pivot=arr[high];
          count=count+1;
          int i=low-1;
          count=count+1;
   for(int j=low; j<=high-1; j++)
   {
      count=count+1;
      count=count+1;
      if (arr[j]<pivot)
      {
         i++;
         count=count+1;
                              swap(arr[i], arr[j]);
      }
```

## EXPERIMENT 4

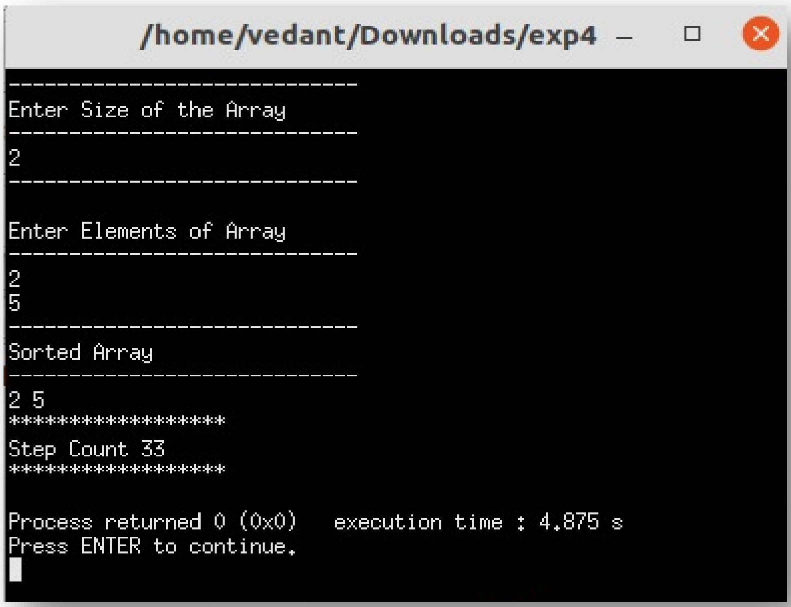
```
  }
  count=count+1;
  swap(arr[i + 1], arr[high]);
  count=count+1;
        return (i + 1);
}

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

int main()
{
        int arr[50],n;
        cout<<"-----------------------------"<<endl;
        cout<<"Enter Size of the Array "<<endl;
        cout<<"-----------------------------"<<endl;
        count=count+1;
        cin>>n;
        count=count+1;
        cout<<"-----------------------------"<<endl;
        cout<<"\nEnter "<<"Elements of Array"<<endl;
        cout<<"-----------------------------"<<endl;
        count=count+1;
        for(int i=0;i<n;i++)
        {
                count=count+1;
                cin>>arr[i];
                count=count+1;
        }
        count=count+1;
        quickSort(arr,0,n-1);
        cout<<"-----------------------------"<<endl;
        cout<<"Sorted Array "<<endl;
        cout<<"-----------------------------"<<endl;
        count=count+1;
        for(int i=0;i<n;i++)
        {
                count=count+1;
                cout<<arr[i]<<" ";
                count=count+1;
        }
        count=count+1;
        count=count+1;
        count=count+1;
        cout<<"\n*****************"<<endl;
        cout<<"Step Count "<<count;
        cout<<"\n*****************"<<endl;
        return 0;
}
```

# EXPERIMENT 4

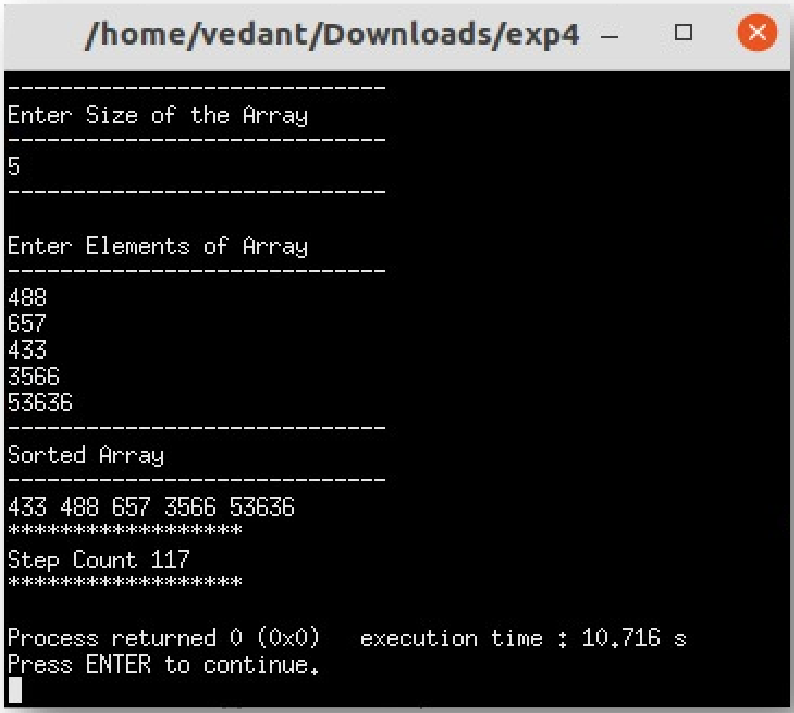
## **OUTPUT**



```
-----------------------------
Enter Size of the Array
-----------------------------
2
-----------------------------

Enter Elements of Array
-----------------------------
2
5
-----------------------------
Sorted Array
-----------------------------
2 5
******************
Step Count 33
******************

Process returned 0 (0x0)   execution time : 4.875 s
Press ENTER to continue.
```

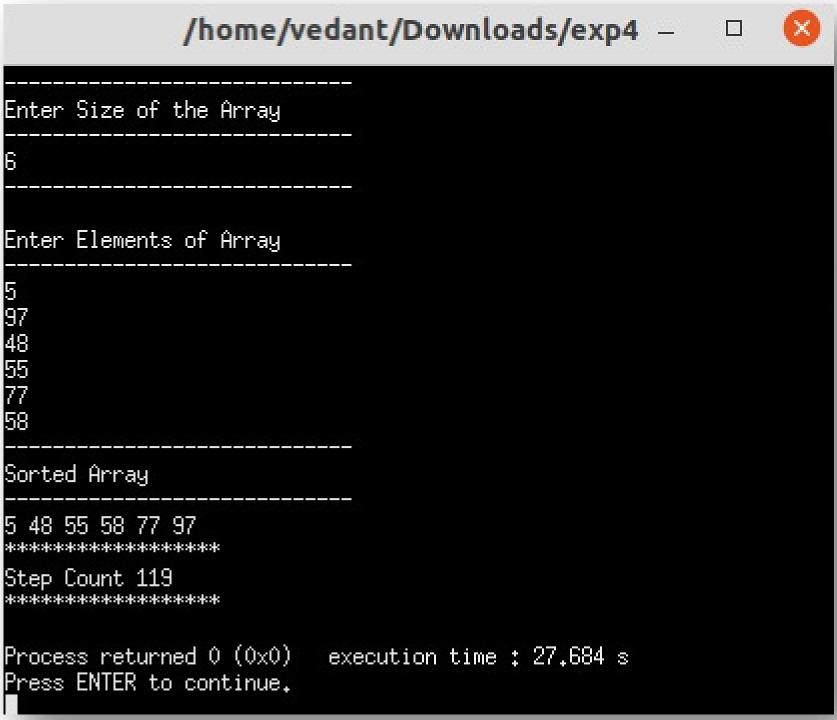


```
-----------------------------
Enter Size of the Array
-----------------------------
5
-----------------------------

Enter Elements of Array
-----------------------------
488
657
433
3566
53636
-----------------------------
Sorted Array
-----------------------------
433 488 657 3566 53636
******************
Step Count 117
******************

Process returned 0 (0x0)   execution time : 10.716 s
Press ENTER to continue.
```

# EXPERIMENT 4



/home/vedant/Downloads/exp4

```
------------------------------
Enter Size of the Array
------------------------------
6
------------------------------

Enter Elements of Array
------------------------------
5
97
48
55
77
58
------------------------------
Sorted Array
------------------------------
5 48 55 58 77 97
******************
Step Count 119
******************

Process returned 0 (0x0)   execution time : 27.684 s
Press ENTER to continue.
```

## CONCLUSION

- Detailed concept of Quick Sort algorithm was studied successfully.
- Quick Sort program were executed successfully.
- The step count for the Quick Sort algorithm was obtained for different cases.