Experiment No. : 2

Title: Divide and Conquer Strategy

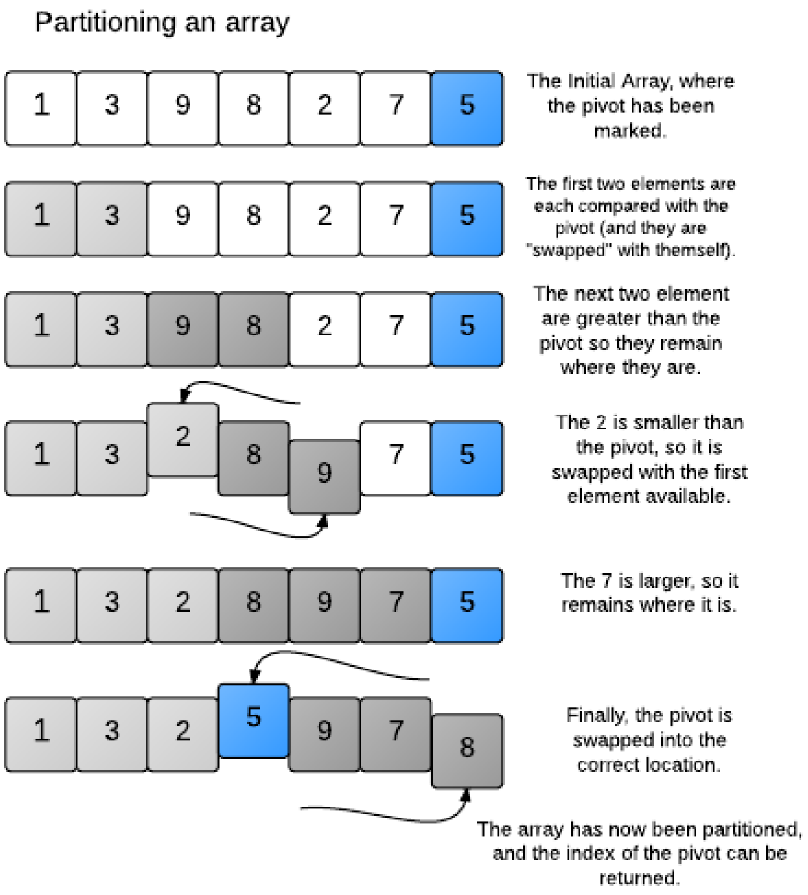
Aim: To implement and analyse time complexity of Quick-sort.

Explanation and Working of Quick sort:

Quick sort is a popular sorting algorithm that uses the divide-and-conquer strategy to sort a list of items. It works by partitioning the list into two parts, sorting each part recursively, and then merging the sorted parts together.

The basic steps for quick sort are:

- Choose a pivot element from the list. The pivot is usually the last element in the list, but it can be any element.
- Partition the list by rearranging all the elements smaller than the pivot to the left of the pivot and all the elements greater than the pivot to the right of the pivot.
- Recursively sort the left partition and the right partition.
- Merge the sorted partitions.


Partitioning an array
1 3 9 8 2 7 5
The Initial Array, where the pivot has been marked.
1 3 9 8 2 7 5
The first two elements are each compared with the pivot (and they are "swapped" with themself).
1 3 9 8 2 7 5
The next two element are greater than the pivot so they remain where they are.
1 3 2 8 9 7 5
The 2 is smaller than the pivot, so it is swapped with the first element available.
1 3 2 8 9 7 5
The 7 is larger, so it remains where it is.
1 3 2 5 9 7 8
Finally, the pivot is swapped into the correct location.
The array has now been partitioned, and the index of the pivot can be returned.

The time complexity of quick sort is O(n log n) in the average case and O(n^2) in the worst case (when the pivot is chosen poorly, and the list is already sorted or nearly sorted). However, quick sort is often faster than other sorting algorithms, such as merge sort and heap sort, in practice because it has good cache performance and does not require additional memory

Algorithm of Quick sort:

```
quickSort(arr, left, right):
    if left < right:
        pivot = partition(arr, left, right)
        quickSort(arr, left, pivot - 1)
        quickSort(arr, pivot + 1, right)

partition(arr, left, right):
    pivot = arr[left]
    i = left + 1
    j = right
    while i <= j:
        while i <= j and arr[i] <= pivot:
            i += 1
```

```
while i <= j and arr[j] > pivot: j -= 1
         if i <= j:
             arr[i], arr[j] = arr[j], arr[i]
     arr[left], arr[j] = arr[j], arr[left]
     return j
```

Derivation of Analysis Quick sort:

Worst Case Analysis

The worst-case scenario occurs when the pivot element is either the smallest or largest element in the array, resulting in one subarray of size n-1 and the other subarray of size 0. In this case, each recursive call to the quick sort function sorts an array of size n-1, and the algorithm degrades to a simple comparison-based sorting algorithm. The time complexity in the worst case is O(n^2).

Best Case Analysis

The best-case scenario occurs when the pivot element divides the array into two subarrays of equal size. In this case, each recursive call to the quick sort function divides the array into two halves of equal size, and the pivot element is chosen in the middle of the array. The time complexity in the best case is O(n log n).

Average Case Analysis

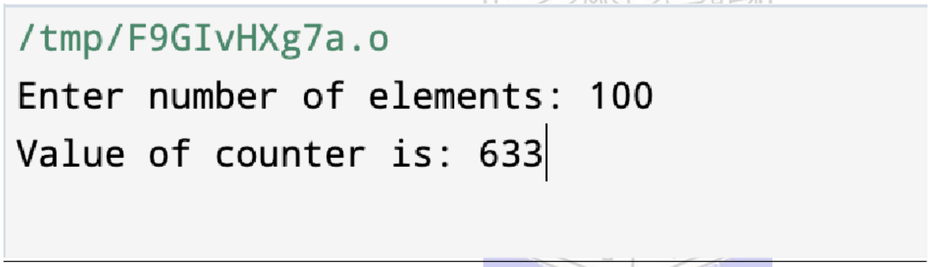
The average case occurs when the pivot element is chosen randomly, or when the median element is chosen as the pivot element. In this case, the algorithm divides the array into two subarrays of roughly equal size. Each recursive call sorts an array of size proportional to n/2, and the algorithm makes O(log n) recursive calls. The time complexity in the average case is O(n log n).

Program(s) of Quick sort:

```
#include <stdio.h>

void swap(int *a, int *b)
{

    int temp = *a;
    *a = *b;
    *b = temp;
```

```
}


int partition(int arr[], int low, int high, int *counter)
{

    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {

        (*counter)++; // increment counter for every comparison
        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, int *counter)
{

    if (low < high)
    {

        int pi = partition(arr, low, high, counter);
        quickSort(arr, low, pi - 1, counter);
        quickSort(arr, pi + 1, high, counter);
    }

}


int main()
{

    int num,i;
    printf("Enter number of elements: ");
    scanf("%d",&num);
    int arr[num];
    for(i=0;i<num;i++)
    {
        arr[i]=rand();
    }

    int n = sizeof(arr) / sizeof(arr[0]);

    int counter = 0;
    quickSort(arr, 0, n - 1, &counter);

    /*printf("Sorted array: ");
    for (int i = 0; i < n; i++)
        printf("%d ",
    arr[i]); printf("\n");*/

    printf("Value of counter is: %d", counter);
    return 0;
}
```

Output(o) of Quick sort:



```
/tmp/F9GIvHXg7a.o
Enter number of elements: 100
Value of counter is: 633
```

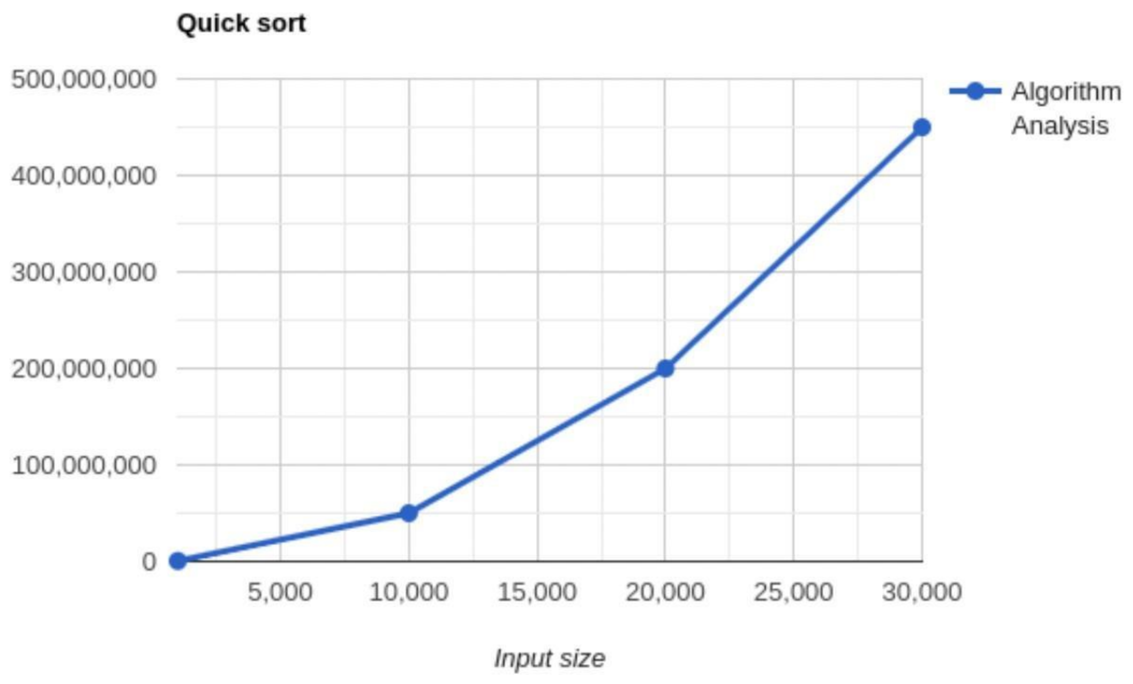
Results:

Time Complexity of Quick sort:

Worst Case Analysis:

Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
1.	1000	499500	6907
2.	10000	49995000	92103
3.	20000	199990000	198069
4.	30000	449985000	309268

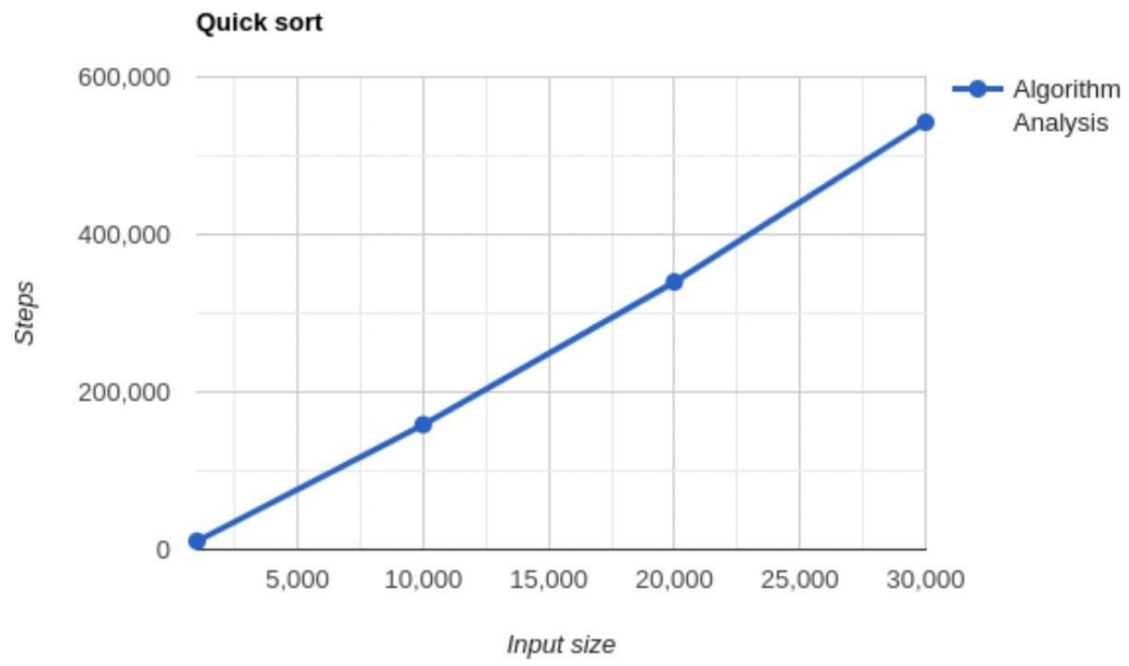
GRAPH:



Best Case Analysis:

Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
1.	1000	10676	6907
2.	10000	158562	92103
3.	20000	339756	198069
4.	30000	542423	309268

GRAPH



Conclusion: (Based on the observations):
The detailed analysis of the time complexity of quick sort and merge sort was performed

Outcome: CO1: Analyze time and space complexity of basic algorithms

References:

1. Richard E. Neapolitan, " Foundation of Algorithms ", 5th Edition 2016, Jones & Bartlett Students Edition
2. Harsh Bhasin , " Algorithms : Design & Analysis", 1st Edition 2013, Oxford Higher education, India
3. T.H. Coreman ,C.E. Leiserson,R.L. Rivest, and C. Stein, " Introduction to algorithms", 3rd Edition 2009, Prentice Hall India Publication
4. Jon Kleinberg, Eva Tardos, " Algorithm Design", 10th Edition 2013, Pearson India Education Services Pvt. Ltd.