Experiment No. : 2

Title: Divide and Conquer Strategy

Batch: SY-IT(B3) **Roll No.: 16010423076** **Experiment No.:2**

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

Explanation and Working of Quick sort:

Quick Sort is a sorting algorithm based on the Divide and Conquer technique.
It selects a pivot element and partitions the array into two parts.
Elements smaller than the pivot go to the left.
Larger elements go to the right.
The process repeats recursively for the left and right parts until the array is sorted.

Algorithm of Quick sort:

Choose a pivot element from the array.
Partition the array into two subarrays:
Left subarray: Elements smaller than the pivot.
Right subarray: Elements larger than the pivot.
Recursively apply steps 1 and 2 to the subarrays.
Combine the results to get the sorted array.

Derivation of Analysis Quick sort:

Worst Case Analysis
Occurs when the pivot is the smallest or largest element.
Results in an unbalanced partition with one part having no elements.
Time complexity: $O(n^2)$.

Best Case Analysis
Occurs when the pivot splits the array into two equal parts.
Results in balanced partitions for every recursive step.
Time complexity: O(n log n).

Average Case Analysis
Happens for random pivots that divide the array in a reasonably balanced way.
Most realistic scenario.
Time complexity: O(n log n).

Program(s) of Quick sort:

```
#include <stdio.h>

// Function to partition
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
```

```
            // swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    // swap arr[i+1] and pivot
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return i + 1;
}

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

// main function
int main() {
    int n;
    printf("number of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    quickSort(arr, 0, n - 1);
    printf("sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```


Output(o) of Quick sort:

```
Output
number of elements: 8
Enter the elements: 12 4 8 99 0 14 33 65
sorted array: 0 4 8 12 14 33 65 99

=== Code Execution Successful ===
```

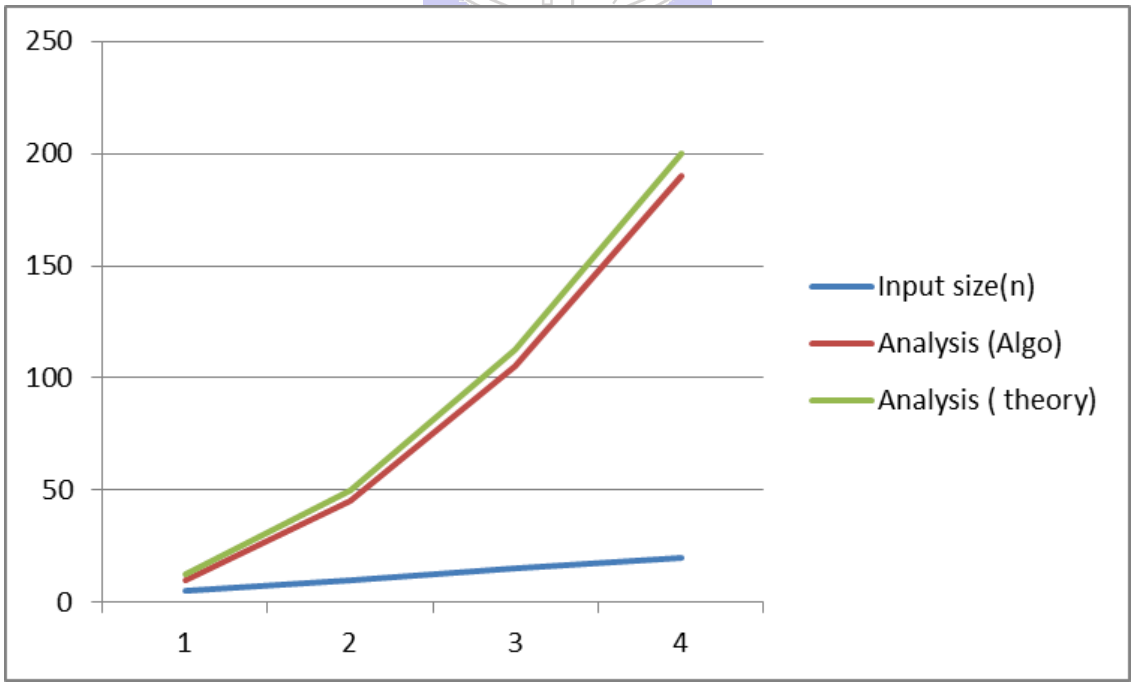
Results:

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	5	10	12.5
2	10	45	50
3	15	105	112.5
4	20	190	200

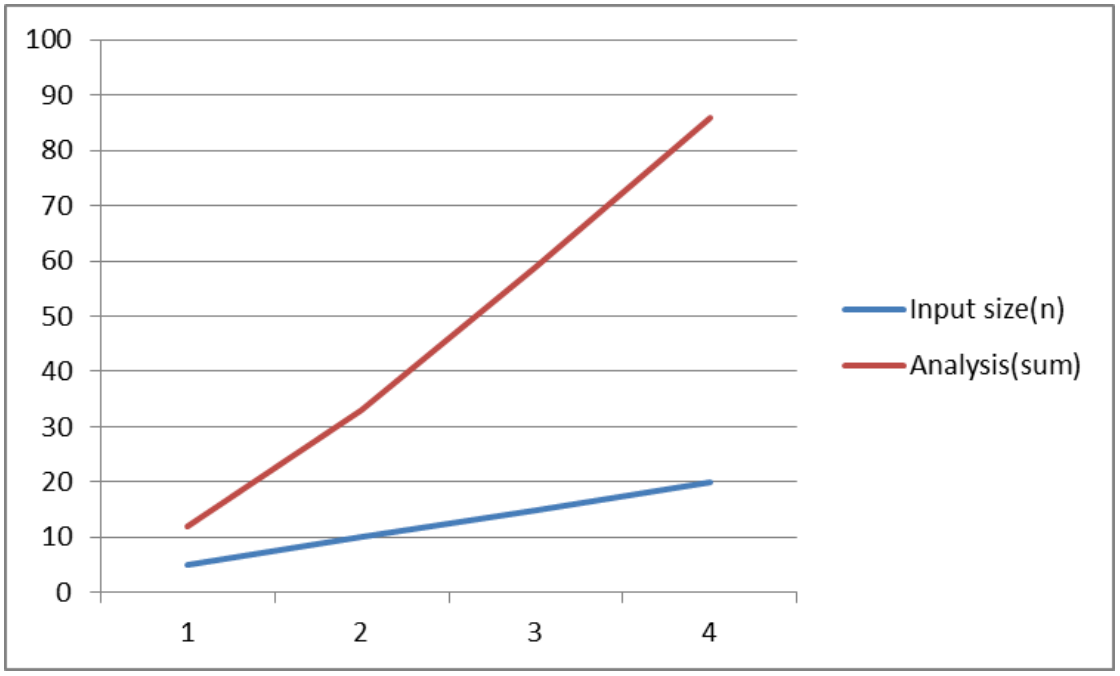
GRAPH:



Best Case Analysis:

Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
1	5	12	12
2	10	33	33
3	15	59	59
4	20	86	86

GRAPH



Conclusion: (Based on the observations):
From this experiment I learned that Quick Sort is an efficient algorithm for sorting arrays due to its average-case time complexity of O(n log n). It works well for most scenarios but can degrade to $O(n^2)$ in the worst case when the pivot selection is poor. Implementing Quick Sort helped me understand the importance of partitioning and recursion in solving complex problems.

Outcome:
CO2 : Implement Divide & Conquer algorithms & derive its time and space complexity.

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.