# Chandpur Science And Technology University


## Lab Report

**Course Code :** CSE 2201

**Course Title :** Algorithm Design and Analysis

**Experiment no :** 03

**Experiment Name:** Quick Sort & Step Analysis

**Submitted By:**

**Name: Turja Chakraborty**

**ID: B210101001**

**Program: B.Sc. in CSE**

**Submitted To:**

**Name: Mustafizur Rahaman**
**Lecturer Dept. of CSE**

**Chandpur science and technology university**

# 1) Objective:

The main objective of this task is to understand the working of the Quick Sort algorithm and analyze its performance in terms of time complexity. Additionally, we aim to measure the number of computational steps taken by the algorithm on a given dataset to gain insights into its behavior during execution.

# 2) Algorithm

**Quick Sort** is a **divide-and-conquer** algorithm that works as follows:

- Choose a **pivot** element from the array.
- Partition the array such that:
  - Elements less than the pivot are on the left.
  - Elements greater than the pivot are on the right.
- Recursively apply the same process to the subarrays on the left and right.

# 3) Theoretical Solution of Given Problem

| Case | Time Complexity |
|---|---|
| Best Case | O(n log n) |
| Average Case | O(n log n) |
| Worst Case | $O(n^2)$ |

| Case | Time Complexity |
|---|---|
| Space | O(log n) |

- The **worst case** occurs when the pivot is always the smallest or largest element.
- The **average and best case** occurs when the pivot divides the array into two almost equal halves.

# 4) Practical Work

## a. Pseudocode:

```
QuickSort(arr, low, high):
    if low < high:
        pi = Partition(arr, low, high)
        QuickSort(arr, low, pi - 1)
        QuickSort(arr, pi + 1, high)

Partition(arr, low, high):
    pivot = arr[high]
```

```
    i = low - 1
    for j = low to high - 1:
        if arr[j] < pivot:
            i = i + 1
            swap arr[i] with arr[j]
    swap arr[i + 1] with arr[high]
    return i + 1
```

## b. Source Code in C (with Step Count)

```c
#include <stdio.h>
int steps = 0;



void swap(int* a, int* b) {
    steps++;
    int temp = *a;
    *a = *b;
    *b = temp;
}



int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
```

```
    for (int j = low; j <= high - 1; j++) {
        steps++;
        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) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}



void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}



int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    printf("Original array:\n");
    printArray(arr, n);

    quicksort(arr, 0, n - 1);

    printf("Sorted array:\n");
    printArray(arr, n);

    printf("Total steps (comparisons + swaps): %d\n", steps);

    return 0;
}
```

## Output:

**Original array:**

**10 7 8 9 1 5**

**Sorted array:**

**1 5 7 8 9 10**

**Total steps (comparisons + swaps): 16**

## 5) Analysis Table

| Input Size (n) | Array Before Sorting | Array After Sorting | Steps Count |
|---|---|---|---|
| 6 | 10, 7, 8, 9, 1, 5 | 1, 5, 7, 8, 9, 10 | (Depends on execution) |

## Observations

- Quick Sort is generally very fast for large arrays.
- The number of steps depends heavily on pivot choice.
- Fewer steps are needed when the pivot splits the array evenly.

---

## Challenges

- Handling the worst-case time complexity $O(n^2)$ when the pivot is poorly chosen.
- Ensuring the algorithm is stable (though by default, Quick Sort is not).
- Managing recursion stack depth in constrained memory environments.