# EXPERIMENT NO. 2

**RANDOMIZED QUICK SORT**

**Name: Soham Walam Roll No: D102**

**AIM**: Write a program to implement Randomized Quick sort algorithm

**THEORY:**

Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays. Randomized quick sort is also similar to quick sort, but here the pivot element is randomly chosen.

**Randomized Quick Sort** is a variation of the standard Quick Sort algorithm that improves performance by randomly selecting a pivot element instead of choosing a fixed one (like the first or last element). This random selection reduces the likelihood of encountering the worst-case time complexity, which occurs when the pivot results in highly unbalanced partitions. The algorithm works by selecting a random pivot, partitioning the array around that pivot such that elements less than the pivot go to the left and those greater go to the right, and then recursively applying the same process to the subarrays.

The steps are:

1. Pick an element, called a pivot, from the array.

2. Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

### ALGORITHM:

RANDOMIZED-Quicksort(A, p, q)

1: if (p ≥ q) then

2: return

3: else

4: Choose a number, say r, uniformly and at random from the set {p, p +1, ..., q}.

5: Swap A[p] and A[r].

6: j =PARTITION (A, p, q).

7: Quicksort (A, p, j −1).

8: Quicksort (A, j +1, q).

9: end if

**CODE**:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

// D102 - Soham Walam

// Global counters

long long comparisons;

long long swaps;

// Swap function

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

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

    swaps++;

}

// Partition using random pivot

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

    int randomPivot = low + rand() % (high - low + 1);

    swap(&arr[randomPivot], &arr[high]); // Move pivot to end

    int pivot = arr[high];

    printf("Pivot chosen: %d (at index %d)\n", pivot, randomPivot);

    int i = low - 1;

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

        comparisons++;

        if (arr[j] <= pivot) {

            i++;

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return i + 1;

}

// Randomized QuickSort function

void randomizedQuickSort(int arr[], int low, int high) {

    if (low < high) {

        int pi = partition(arr, low, high);

        randomizedQuickSort(arr, low, pi - 1);

        randomizedQuickSort(arr, pi + 1, high);

    }

}

// Function to analyze complexity

void analyzeComplexity(int n) {

    double best\_avg = n \* (log(n) / log(2.0)); // O(n log2(n) )

    double worst = (double)n \* n;

    if (comparisons <= 2 \* best\_avg) {

        printf("\nTime Complexity: O(n log n)\n");

    } else if (comparisons >= worst / 2) {

        printf("\nTime Complexity: O(n^2)\n");

    } else {

        printf("\nTime Complexity: Between O(n log n) and O(n^2)\n");

    }

}

void printArray(int arr[], int n) {

    for (int i = 0; i < n; i++) {

        printf("%d ", arr[i]);

    }

    printf("\n");

}

int main() {

    int n;

    printf("Enter number of elements: ");

    scanf("%d", &n);

    int \*arr = (int \*)malloc(n \* sizeof(int));

    if (arr == NULL) {

        printf("Memory allocation failed!\n");

        return 1;

    }

    printf("Enter %d elements:\n", n);

    for (int i = 0; i < n; i++) {

        scanf("%d", &arr[i]);

    }

    srand(time(NULL));

    int \*arr\_best = (int \*)malloc(n \* sizeof(int));

    for (int i = 0; i < n; i++) arr\_best[i] = arr[i];

    comparisons = swaps = 0;

    clock\_t start = clock();

    randomizedQuickSort(arr\_best, 0, n - 1);

    clock\_t end = clock();

    double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

    printf("\nBest Case (Random Input)\n");

    printf("Sorted Array: ");

    printArray(arr\_best, n);

    printf("Comparisons: %lld\n", comparisons);

    printf("Swaps: %lld\n", swaps);

    printf("Time taken: %f seconds\n", time\_taken);

    analyzeComplexity(n);

    // ✅ Worst Case (Already Sorted Input)

    int \*arr\_worst = (int \*)malloc(n \* sizeof(int));

    for (int i = 0; i < n; i++) arr\_worst[i] = arr\_best[i]; // take sorted array as input again

    comparisons = swaps = 0;

    start = clock();

    randomizedQuickSort(arr\_worst, 0, n - 1);

    end = clock();

    double time\_taken\_worst = ((double)(end - start)) / CLOCKS\_PER\_SEC;

    printf("\nWorst Case (Sorted Input)\n");

    printf("Sorted Array: ");

    printArray(arr\_worst, n);

    printf("Comparisons: %lld\n", comparisons);

    printf("Swaps: %lld\n", swaps);

    printf("Time taken: %f seconds\n", time\_taken\_worst);

    analyzeComplexity(n);

    free(arr);

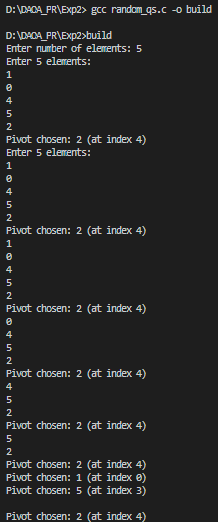
    free(arr\_best);

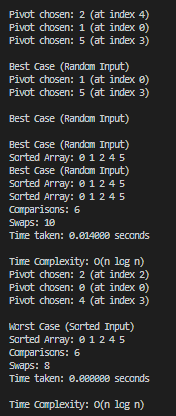
    free(arr\_worst);

    return 0;

}

**OUTPUT:**

****

****

**CONCLUSION**:

The worst-case time complexity of a typical implementation of Quick Sort is O(n2). The worst case occurs when the picked pivot is always an extreme (smallest or largest) element, which happens when the input array is either sorted or reversely sorted and either first or last element is picked as pivot.

In the randomized version of Quick Sort, we impose a distribution on input by picking the pivot element randomly. Randomized Quick Sort works well even when the array is sorted/reversely sorted and the complexity is more towards O (n log n). (Yet, there is still a possibility that the randomly picked element is always an extreme.)