ADVANCE ALGORITHM
Experiment 3

Ayush Jain 60004200132 B3

Aim : To implement Randomized Quick Sort.

Theory:

Randomized QuickSort is a sorting algorithm that uses the divide-and-conquer approach to sort an array of elements. The basic idea of QuickSort is to partition the array into two sub-arrays, where all elements in the left sub-array are less than or equal to a pivot element, and all elements in the right sub-array are greater than the pivot.

The algorithm then recursively sorts the left and right sub-arrays. The pivot element is chosen randomly in the array to ensure that the algorithm's worst-case time complexity is avoided.

Here's the theory for the randomized QuickSort algorithm:

1. Partitioning:
    - Choose a random pivot element from the array.
    - Rearrange the elements in the array such that all elements less than or equal to the pivot element are on the left, and all elements greater than the pivot element are on the right.
    - Return the index of the pivot element.
2. Recursive Sorting:
    - Recursively sort the left sub-array using the partition function.
    - Recursively sort the right sub-array using the partition function.
3. Termination:
    - If the sub-array to be sorted has length 1 or 0, then it is already sorted, and no further action is needed.

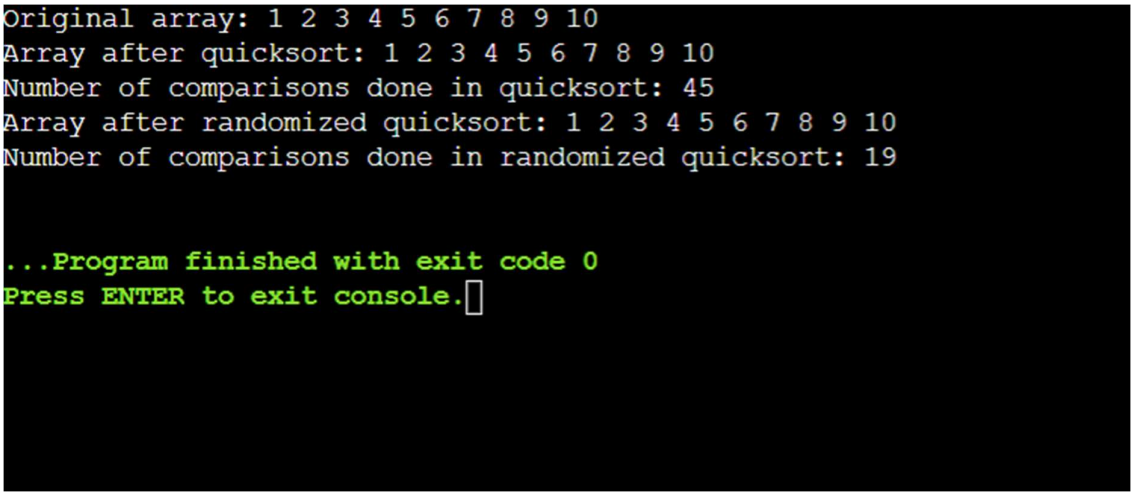
The worst-case time complexity of QuickSort is O(n^2), but the randomized version of QuickSort has an average-case time complexity of O(nlogn). This is because the randomized pivot selection ensures that the probability of choosing a bad pivot is low, leading to a more balanced partition of the array.

Code:

```
#include <iostream>
#include <algorithm>
#include <random>

using namespace std;

int partition(int arr[], int low, int high, int& comparisons) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j <= high - 1; j++) {
        comparisons++;
        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& comparisons) {
    if (low < high) {
        int pi = partition(arr, low, high, comparisons);
        quicksort(arr, low, pi - 1, comparisons);
        quicksort(arr, pi + 1, high, comparisons);
    }
}

int randomized_partition(int arr[], int low, int high, int& comparisons) {
    int i = rand() % (high - low + 1) + low;
    swap(arr[i], arr[high]);
    return partition(arr, low, high, comparisons);
}


void randomized_quicksort(int arr[], int low, int high, int& comparisons) {
    if (low < high) {
        int pi = randomized_partition(arr, low, high, comparisons);
        randomized_quicksort(arr, low, pi - 1, comparisons);
        randomized_quicksort(arr, pi + 1, high, comparisons);
    }
}
int main()
{
     int arr[] = {1,2,3,4,5,6,7,8,9,10};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    cout << "Original array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    int comparisons = 0;
    quicksort(arr, 0, n - 1, comparisons);

    cout << "Array after quicksort: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    cout << "Number of comparisons done in quicksort: " << comparisons <<
endl;

    comparisons = 0;
    randomized_quicksort(arr, 0, n - 1, comparisons);

    cout << "Array after randomized quicksort: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    cout << "Number of comparisons done in randomized quicksort: " <<
comparisons << endl;

    return 0;
}
```

Output:



Conclusion: In conclusion, we learned that the number of swaps can be decreased by randomization of elements in array while quicksort.