**Experiment Number: 03**

**Problem Statement: Time complexity analysis derivation of deterministic and randomized quick sort,**

**and its implemetation.**

**Observe that: Both the matrices of comparison count (nxn), are almost same. and conclude that Avg case time c. Of deterministic qs with expected avg case t.c of randomized qs is same by comparing the matrix of both the variants.**

**Similarity increases as we run more number of test cases.....**

**Also compare practical time estimate with statistical one.**

**NAME:** Harshwardhan Patil  **ROLLNO:** 50

**CLASS:** TY-IT-B **BATCH:** B1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

A piece of paper with writing on it

AI-generated content may be incorrect.A notebook with writing on it

AI-generated content may be incorrect.A piece of paper with writing on it

AI-generated content may be incorrect.A piece of paper with writing on it

AI-generated content may be incorrect.A piece of paper with writing on it

AI-generated content may be incorrect.

Code:  
import java.util.Random;

import java.util.Arrays;

public class Main {

    static final int N = 9; // 9x9 matrix (values from 1 to 9)

    static final int RUNS = 362880; // 9! possible permutations

    static int[][] comparisonMatrixRandomized = new int[N + 1][N + 1];

    static int[][] comparisonMatrixDeterministic = new int[N + 1][N + 1];

    static double[][] expectedMatrix = new double[N + 1][N + 1];

    public static void main(String[] args) {

        initializeMatrices();

        int[] arr = new int[N];

        for (int i = 0; i < N; i++) arr[i] = i + 1;

        int count = 0;

        do {

            int[] tempArr1 = Arrays.copyOf(arr, N);

            int[] tempArr2 = Arrays.copyOf(arr, N);

            randomizedQuickSort(tempArr1, 0, N - 1);

            deterministicQuickSort(tempArr2, 0, N - 1);

            count++;

        } while (nextPermutation(arr) && count < RUNS);

        computeExpectedMatrix();

        printMatrices();

    }

    static void initializeMatrices() {

        for (int i = 1; i <= N; i++) {

            for (int j = 1; j <= N; j++) {

                comparisonMatrixRandomized[i][j] = 0;

                comparisonMatrixDeterministic[i][j] = 0;

                expectedMatrix[i][j] = 0.0;

            }

        }

    }

    /\*\* Randomized QuickSort Implementation \*/

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

        if (low < high) {

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

            randomizedQuickSort(arr, low, pi - 1);

            randomizedQuickSort(arr, pi + 1, high);

        }

    }

    static int randomizedPartition(int[] arr, int low, int high) {

        Random rand = new Random();

        int pivotIndex = rand.nextInt(high - low + 1) + low;

        int pivot = arr[pivotIndex];

        swap(arr, pivotIndex, high);

        int i = low - 1;

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

            comparisonMatrixRandomized[arr[j]][pivot]++;

            comparisonMatrixRandomized[pivot][arr[j]]++;

            if (arr[j] < pivot) {

                i++;

                swap(arr, i, j);

            }

        }

        swap(arr, i + 1, high);

        return i + 1;

    }

    /\*\* Deterministic QuickSort Implementation (choosing last element as pivot) \*/

    static void deterministicQuickSort(int[] arr, int low, int high) {

        if (low < high) {

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

            deterministicQuickSort(arr, low, pi - 1);

            deterministicQuickSort(arr, pi + 1, high);

        }

    }

    static int deterministicPartition(int[] arr, int low, int high) {

        int pivot = arr[high]; // Always choose last element as pivot

        int i = low - 1;

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

            comparisonMatrixDeterministic[arr[j]][pivot]++;

            comparisonMatrixDeterministic[pivot][arr[j]]++;

            if (arr[j] < pivot) {

                i++;

                swap(arr, i, j);

            }

        }

        swap(arr, i + 1, high);

        return i + 1;

    }

    static void swap(int[] arr, int i, int j) {

        int temp = arr[i];

        arr[i] = arr[j];

        arr[j] = temp;

    }

    static boolean nextPermutation(int[] arr) {

        int i = arr.length - 2;

        while (i >= 0 && arr[i] >= arr[i + 1]) i--;

        if (i < 0) return false;

        int j = arr.length - 1;

        while (arr[j] <= arr[i]) j--;

        swap(arr, i, j);

        reverse(arr, i + 1);

        return true;

    }

    static void reverse(int[] arr, int start) {

        int end = arr.length - 1;

        while (start < end) swap(arr, start++, end--);

    }

    /\*\* Compute Expected Comparison Matrix \*/

    static void computeExpectedMatrix() {

        for (int i = 1; i <= N; i++) {

            for (int j = i + 1; j <= N; j++) {

                expectedMatrix[i][j] = 2.0 / (j - i + 1);

                expectedMatrix[j][i] = expectedMatrix[i][j];

            }

        }

    }

    /\*\* Print Matrices \*/

    static void printMatrices() {

        System.out.println("Randomized QuickSort Comparison Matrix:");

        printMatrix(comparisonMatrixRandomized);

        System.out.println("\nDeterministic QuickSort Comparison Matrix:");

        printMatrix(comparisonMatrixDeterministic);

        System.out.println("\nExpected Comparison Matrix:");

        printMatrix(expectedMatrix);

    }

    static void printMatrix(int[][] matrix) {

        for (int i = 1; i <= N; i++) {

            for (int j = 1; j <= N; j++) {

                System.out.printf("%.4f ", matrix[i][j] / (double) RUNS);

            }

            System.out.println();

        }

    }

    static void printMatrix(double[][] matrix) {

        for (int i = 1; i <= N; i++) {

            for (int j = 1; j <= N; j++) {

                System.out.printf("%.4f ", matrix[i][j]);

            }

            System.out.println();

        }

    }

}

Output:  
