CS263

ASSIGNMENT 4

NAME:

ARCHIT AGRAWAL

ROLL NO.:

202052307

SECTION:

A

Problem 1

Consider two scenarios:

Case(a): Consider sorting a list of 1000 persons by the month they were born. There are only 12 possible months; if they are evenly spread throughout our array, we can anticipate each month to appear several times. As a result, we have an array with a lot of duplicate values. Situations like these occur frequently in real-life circumstances.

Think about how partitioning works in the quicksort for a moment – in particular, think about how elements equal to the pivot move during this process.

For example, 1, 4, 2, 4, 2, 4, 1, 2, 4, 1, 2, 2, 6, 9, 8, 9, 2, 2, 4, 1, 4, 4, 4.

If 4 is picked as a pivot in Simple Quick Sort, we fix only one 4 and recursively process remaining occurrences.

After one partition, we will have 1, 2, 2, 1, 2, 1, 2, 2, 2, 2, 1, 4, 4, 4, 4, 4, 4, 4, 4, 6, 9, 8, 9

Now think about the potential offered by modifying our partitioning technique to group array elements into three different groups: that is called 3-way partition.

Case(b): Consider another scenario, where we can have unique elements, can we apply a 3-way partition. If yes what will be our partition technique?

For example: 24, 8, 42, 75, 29, 77, 38, 57

Take two pivot elements and partition in 3 sub-lists. Suppose pivots are 24 and 57. Then we have three partition:- 8, 24, 42, 29, 38, 57, 75, 77

Algorithm (Case (a))

The three-way quicksort is similar, but there are three sections. array arr[1 to n] is divided into three parts.

1. arr[1 to i]
2. arr[i + 1, j]
3. arr[j + 1, n]

Partition in this case can be done in the below mentioned way:

- Choose a pivot (say the last element everytime)
- Now, partition the array into three subarrays, one that contains all elements smaller than the pivot element (array index 1 to i), second which contains all the elements equal to the pivot element (array index i + 1 to j)and third which contains all the elements that are greater than the pivot element (array index j + 1 to n).
- Now, we have three subarrays, out of which one is sorted and placed correctly. Now, we can recur down in the remaining two subarrays (one that has elements smaller than pivot and other that has elements greater than the pivot).

Code (Case (a))

```
import java.util.*;
public class QuickSort3Way {

    static int i, j;

    //method to swap two elements of array
    public static void swap(int[] arr, int i, int j){
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    //method to create partition for quicksort

    public static void partition(int[] arr, int l, int r){
        i = l - 1; j = r;
```

```
        if (r - l <= 1) {
            if (arr[r] < arr[l])
                swap(arr, r ,l);
            i = l;
            j = r;
            return;
        }
        int mid = l;
        int pivot = arr[r];
        while (mid <= r) {
            if (arr[mid]<pivot)
                swap(arr, l++, mid++);
            else if (arr[mid]==pivot)
                mid++;
            else if (arr[mid] > pivot)
                swap(arr, mid, r--);
        }
        i = l-1;
        j = mid;
    }

    //method to quicksort
    public static void quicksort(int[] arr, int left, int right) {
        if (left >= right)
            return;

        partition(arr, left, right);
        //recursive calls
        quicksort(arr, left, i);
        quicksort(arr, j, right);
    }

    public static void display(int[] arr){
        for(int x = 1; x <= arr.length; x++){

            System.out.print(arr[x-1] + " ");
        }
        System.out.println();
    }

    public static void main(String[] args){
        Scanner Sc = new Scanner(System.in);
        int[] a = new int[1000];
        Random Ob = new Random();
        for(int i = 0; i < 1000; i++){
            a[i] = Ob.nextInt(12) + 1;
        }
```

```
        quicksort(a, 0, a.length-1);
        System.out.println("SORTED ARRAY");
        System.out.println();
        display(a);
    }
}
```

Output (Case(a))

Analysis (Case(a))

The Recurrence relation for the 3-way implementation QuickSort is

If (low > high) return - 1 time (Base condition)

Inside the loop

If (element < pivot) swap – n times

If (element > pivot) swap – n times

Recursive part – 2T(n/2) times.

=> Recurrence relation = T(n) = 2T(n/2) + 2n + 1

Time Complexity (Using master theorem)

Recurrence relation = T(n) = 2T(n/2) + 2n + 1 or T(n) = 2T(n/2) + nlogn (neglecting 1 and 2)

$$\log_b a = \log_2 2 = 1 \; and \; k = 1, p = 0$$

Solving using Master's Method gives

$$T(n) = \theta(n \log n)$$

The worst case still has a time complexity of $O(n^2)$. But in the average the case the complexity reduces to O($n \log n$). This algorithm serves good when there are a large number of repeated elements. In such cases the complexity approaches O(n), however, it is O(n) only in the best case i.e the array has only 1 unique element.

Algorithm (Case (b))

The idea of dual pivot quick sort is to take two pivots, one in the left end of the array and the second, in the right end of the array. The left pivot must be less than or equal to the right pivot, so we swap them if necessary.

Then, we begin partitioning the array into three parts: in the first part, all elements will be less than the left pivot, in the second part all elements will be greater or equal to the left pivot and also will be less than or equal to the right pivot, and in the third part all elements will be greater than the right pivot. Then, we shift the two pivots to their appropriate positions as we see in the below bar, and after that we begin quicksorting these three parts recursively, using this method.

Code (Case (b))

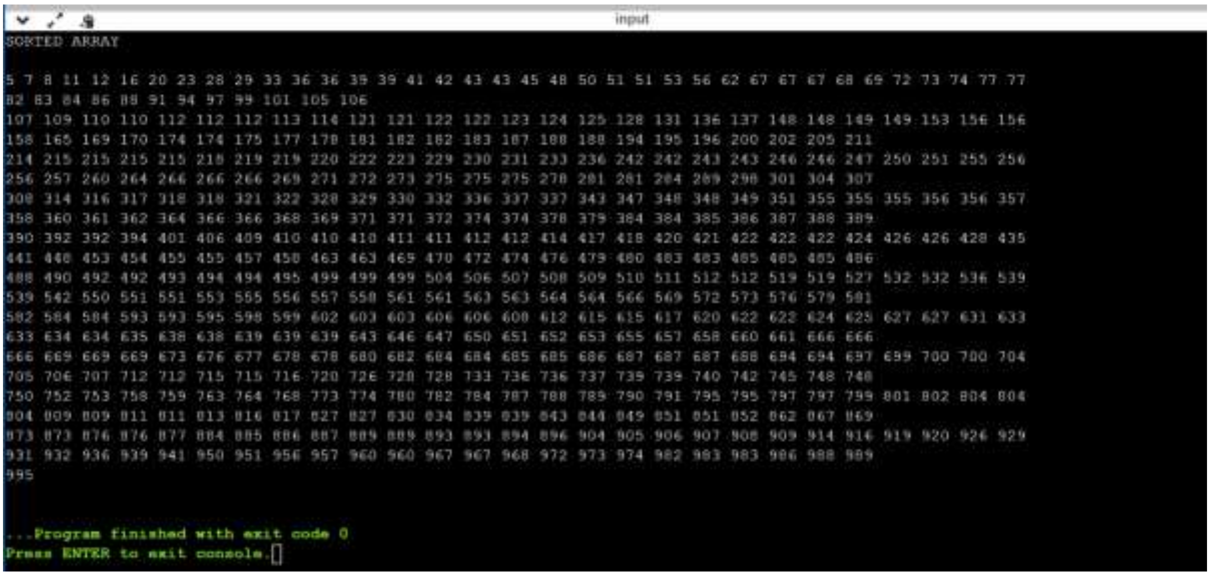
```
import java.util.*;
public class DualPivotQuickSort {

    //method to swap two elements of array
    public static void swap(int[] arr, int i, int j){
        int temp = arr[i];
        arr[i] = arr[j];
```

```
        arr[j] = temp;
    }


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

        if (low < high){

            // piv[] stores left pivot and right pivot.
            // piv[0] means left pivot and
            // piv[1] means right pivot
            int[] piv;
            piv = partition(arr, low, high);

            dualPivotQuickSort(arr, low, piv[0] - 1);
            dualPivotQuickSort(arr, piv[0] + 1, piv[1] - 1);
            dualPivotQuickSort(arr, piv[1] + 1, high);
        }
    }



    //method to create partition for quicksort

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

        //if lower pivot is at higher index, then swap the pivots
        if (arr[low] > arr[high])
            swap(arr, low, high);

        // p is the left pivot, and q
        // is the right pivot.
        int j = low + 1;
        int g = high - 1, k = low + 1;
        int p = arr[low], q = arr[high];

        while (k <= g) {

            // If elements are less than the left pivot
            if (arr[k] < p){
                swap(arr, k, j);
                j++;
            }

            // If elements are greater than or equal
            // to the right pivot
            else if (arr[k] >= q){

                while (arr[g] > q && k < g) g--;
```

```
                swap(arr, k, g);
                g--;

                if (arr[k] < p){
                    swap(arr, k, j);
                    j++;
                }
            }

            k++;
        }
        j--;
        g++;

        // Bring pivots to their appropriate positions.
        swap(arr, low, j);
        swap(arr, high, g);

        // Returning the indices of the pivots
        // because we cannot return two elements
        // from a function, we do that using an array.
        return new int[] {j, g};
    }


    public static void display(int[] arr){
        for(int i = 1; i <= arr.length; i++){
            if(i % 50 == 0) System.out.println();
            System.out.print(arr[i-1] + " ");
        }
        System.out.println();
    }

    public static void main(String[] args){
        Scanner Sc = new Scanner(System.in);
        int[] a = new int[500];

        Random rand = new Random();

        for(int i = 0; i < 500; i++){
            a[i] = rand.nextInt(1000) + 1;
        }

        dualPivotQuickSort(a, 0, a.length-1);
        System.out.println("SORTED ARRAY");
        System.out.println();
        display(a);
    }
```

```
}
```

Output (Case(b))



Analysis (Case (b))

The recurrence relation for dualPivotQuickSort is:

$$T(n) = 3T\left(\frac{n}{3}\right) + n^2$$

Solving using Master's Theorem:

$$\log_3 3 = 1 \; and \; k = 2 \; and \; p = 0$$

$$Therefore, T(n) = O(n^2)$$

In worst case the time complexity is O(n^2) but there is a very rare chance that worst case takes place. In average and best case, the time complexity is O(n $\log n$).

Problem 2

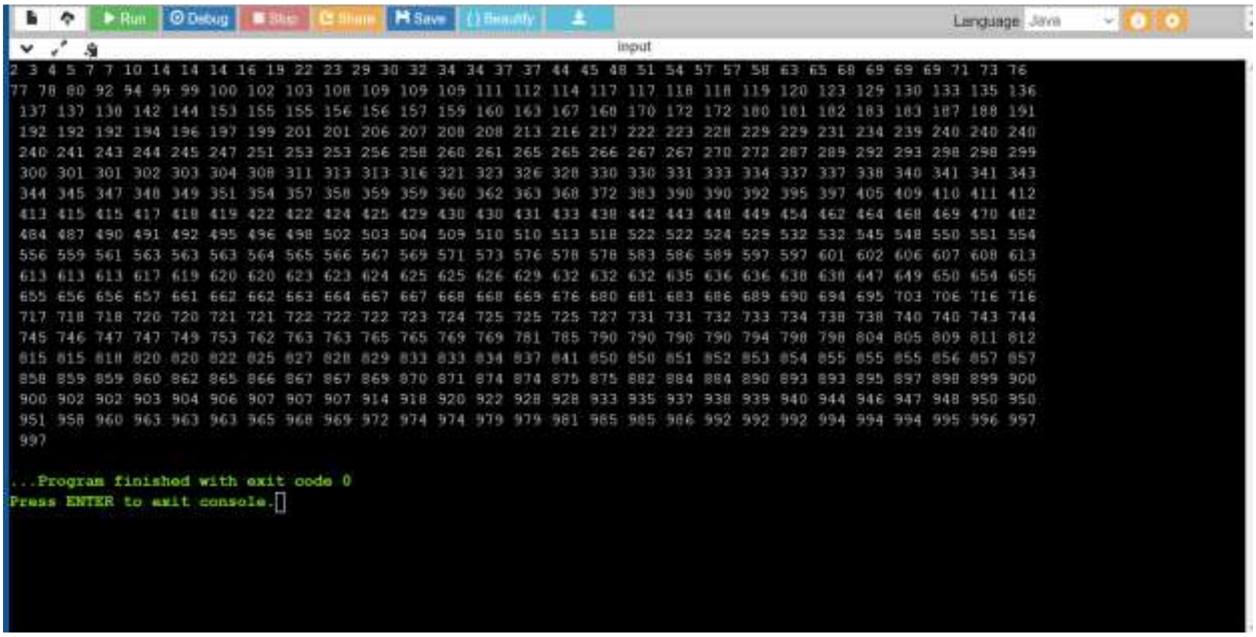
Implement Randomized Quicksort, and analyse its best, worst, and average-case complexity. Show the graph with varying its input size

Code

```
import java.util.*;

public class Main{

    public static void swap(int arr[], int i, int j){
        int temp = arr[j];
        arr[j] = arr[i];
        arr[i] = temp;
    }

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

        Random rand = new Random();
        int pivot = rand.nextInt(high - low) + low;

        swap(arr, pivot, high);
    }

    public static int partitionRandom(int arr[], int l, int h){

        random(arr, l, h);
        int pivot = arr[l];

        int i = h + 1;

        for(int j = h; j > l; j--){
            if(arr[j] > pivot){
                i--;
                swap(arr, i, j);
            }
        }

        swap(arr, i - 1, l);
        return i - 1;
    }

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

        if(low < high){
            int x = partitionRandom(arr, low, high);
            randomisedQuickSort(arr, low, x - 1);
```

```java
            randomisedQuickSort(arr, x + 1, high);
        }

    }

    public static void main(String[] args){

        int[] arr  = new int[500];
        Random rand = new Random();

        for(int i = 0; i < 500; i++){
            arr[i] = rand.nextInt(1000);
        }

        randomisedQuickSort(arr, 0, 499);

        for(int i = 0; i < 500; i++){

            System.out.print(arr[i] + " ");
        }

    }

}
```

Output



Analysis

Number of elements in array	Time taken in milliseconds(ms)
100	*0.102*
1000	0.277
10000	3.882
100000	22.451

Best Case	***Average Case***	***Worst Case***
O(nlogn)	O(nlogn)	$O(n^2)$