IT-314 Software Engineering

Name: Harshali Dharmik

Student ID: 202201214

LAB: 09

Program Inspection and Debugging:

Q1) Armstrong

```
//Armstrong Number
class Armstrong{
    public static void main(String args[]){
        int num = Integer.parseInt(args[0]);
        int n = num; //use to check at last time
        int check=0,remainder;
        while(num > 0){
                remainder = num / 10;
                check = check + (int)Math.pow(remainder,3);
                num = num % 10;
        }
        if(check == n)
                System.out.println(n+" is an Armstrong Number");
        else
                System.out.println(n+" is not a Armstrong Number");
    }

Input: 153
Output: 153 is an armstrong Number.
```

Program Inspection:

1. Errors:

- Error 1: In the while loop, the remainder is being calculated incorrectly. It should be ‘remainder = num % 10;’ instead of ‘remainder = num / 10;’

- Error 2: Missing closing bracket for the class Armstrong ‘}’

2. **Effective Category:**

- Category B (Data Declaration Errors) and Category E (Control Flow Errors).

3. **Unidentified Error Types:**

- The program inspection did not identify potential issues like integer overflow.

4. **Applicability:**

- Program inspection helps catch syntax and some semantic errors, but it doesn't verify the correctness of the logic in the program.

Debugging:

1. **Errors:**

- Error 1: In the while loop, change the calculation of ‘remainder’ to ‘remainder = num %10;’ to correctly calculate the remainder.

2. **Breakpoints Needed:**

- At least one breakpoints are needed to address these errors.

Corrected Code:

```
class Armstrong {
    public static void main(String args[]) { int num =
        Integer.parseInt(args[0]);
        int n = num; // original number to check at last

        int check = 0, remainder; while (num > 0) {
            remainder = num % 10;
            check = check + (int)Math.pow(remainder, 3); num = num / 10;
        }
        if (check == n)
            System.out.println(n + " is an Armstrong Number"); else
            System.out.println(n + " is not an Armstrong Number");
    }
}
```

Q2) GCD_LCM

```
//program to calculate the GCD and LCM of two given numbers
import java.util.Scanner; public class
GCD_LCM
{
    static int gcd(int x, int y)
    {
        int r=0, a, b;
        a = (x > y) ? y : x; // a is greater number
        b = (x < y) ? x : y; // b is smaller number r = b;
        while(a % b == 0) //Error replace it with while(a % b != 0)
        {
            r = a % b;
            a = b;
            b = r;
        }
        return r;
    }
    static int lcm(int x, int y)
    {
        int a;
        a = (x > y) ? x : y; // a is greater number while(true)
        {
            if(a % x != 0 && a % y != 0) return a;
            ++a;
        }
   }
    public static void main(String args[])
    {
        Scanner input = new Scanner(System.in); System.out.println("Enter the
        two numbers: "); int x = input.nextInt();
        int y = input.nextInt();
        System.out.println("The GCD of two numbers is: " + gcd(x, y));
        System.out.println("The LCM of two numbers is: " + lcm(x, y));
        input.close();
    }
 }


 Input: 4 5
Output: The GCD of two numbers is 1 The GCD of two
numbers is 20
```

Program Inspection:

1. **Errors:**

- Error 1: In the ‘gcd’ method, the while loop condition is incorrect. It should be ‘while(a %b!= 0)’ instead of

  ‘while(a % b == 0)’.

2. **Effective Category:**

- Category B (Semantic Errors).

3. **Unidentified Error Types:**

- The program inspection did not identify potential issues like integer overflow or incorrect logic in the ‘lcm’

  method.

4. **Applicability:**

- Program inspection helps catch syntax and some semantic errors but does not verify the correctness of the mathematical logic in the ‘gcd’ and ‘lcm’ methods.

Debugging:

1. **Errors:**

- Error 1: In the ‘gcd’ method, change the while loop condition to ‘while(a % b != 0)’ to find the greatest common divisor correctly.

2. **Breakpoints Needed:**

- At least one breakpoints are needed to address these errors. import java.util.Scanner;

Corrected Code:

```
import java.util.Scanner;
public class GCD_LCM { static int gcd(int
    x, int y) {
        int r = 0, a, b;
        a = (x > y) ? y : x; // a is the smaller number
        b = (x < y) ? x : y; // b is the greater number r = b;
        while (a % b != 0) {  // Corrected loop condition r = a % b;
            a = b;
            b = r;
        }
        return r;
    }
    static int lcm(int x, int y) { int a;
        a = (x > y) ? x : y;  // a is the greater number while (true) {
            if (a % x == 0 && a % y == 0) {  // Condition corrected to find LCM
                return a;
            }
++a;
        }
    }
    public static void main(String args[]) {
        Scanner input = new Scanner(System.in); System.out.println("Enter the
        two numbers: "); int x = input.nextInt();
        int y = input.nextInt();
        System.out.println("The GCD of the two numbers is: " + gcd(x, y));
        System.out.println("The LCM of the two numbers is: " + lcm(x, y));
        input.close();
    }
}
```

Q3) Knapsack

```
//Knapsack
public class Knapsack {

    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);   // number of items
        int W = Integer.parseInt(args[1]);   // maximum weight of knapsack

        int[] profit = new int[N+1];
        int[] weight = new int[N+1];

        // generate random instance, items 1..N
        for (int n = 1; n <= N; n++) {
            profit[n] = (int) (Math.random() * 1000);
            weight[n] = (int) (Math.random() * W);
        }

        // opt[n][w] = max profit of packing items 1..n with weight limit w
     // sol[n][w] = does opt solution to pack items 1..n with weight limit w include item n?
        int[][] opt = new int[N+1][W+1];
        boolean[][] sol = new boolean[N+1][W+1];

        for (int n = 1; n <= N; n++) {
            for (int w = 1; w <= W; w++) {

                // don't take item n
                int option1 = opt[n++][w];

                // take item n
                int option2 = Integer.MIN_VALUE;
                if (weight[n] > w) option2 = profit[n-2] + opt[n-1][w-weight[n]];

                // select better of two options
                opt[n][w] = Math.max(option1, option2);
                sol[n][w] = (option2 > option1);
            }
        }

        // determine which items to take
        boolean[] take = new boolean[N+1];
        for (int n = N, w = W; n > 0; n--) {
            if (sol[n][w]) { take[n] = true;  w = w - weight[n]; }
            else           { take[n] = false;                    }
        }

        // print results
        System.out.println("item" + "\t" + "profit" + "\t" + "weight" + "\t" + "take");
        for (int n = 1; n <= N; n++) {
            System.out.println(n + "\t" + profit[n] + "\t" + weight[n] + "\t" + take[n]);
        }
    }
}
Input: 6, 2000
Output:

Item   Profit   Weight Take
1   336       784    false
2   674       1583   false
3   763       392    true
4   544       1136   true
5   14        1258   false
6   738       306    true
```

Program Inspection:

1. Errors:

- Error 1: In the for loop header, there is a post-increment operator (‘n++’) used instead of just incrementing ‘n’ by one (‘n++’ should be ‘n+1’).
- Error 2: The indexing of arrays should be from ‘0’ to ‘N’, but it starts from ‘1’ to ‘N’. In Java, arrays are zero- indexed.
- Error 3: In the option2 calculation, the code is using the item's profit and weight at index ‘n 2’, which is likely
incorrect. It should be using ‘n-1’.
- Error 4: The code calculates the maximum profit value using ‘opt[N][W]’, but this should be ‘opt[N][W]’ for the actual result.

2. Effective Category:

- Category B (Data Declaration Errors) and Category C (Computational Errors).

3. **Unidentified Error Types:**

- The program inspection did not identify potential logical errors, such as the correctness of the knapsack

  algorithm's implementation.

4. **Applicability:**

- Program inspection helps catch syntax and some semantic errors but does not verify the correctness of the algorithm's implementation.

Debugging:

1. **Errors:**

- Error 1: Change ‘n++’ to ‘n+1’ in the for loop header.
- Error 2: Adjust array indexing to start from ‘0’.
- Error 3: Use ‘n-1’ instead of ‘n-2’ for item profit and weight in option2 calculation.
- Error 4: Change ‘opt[N][W]’ to ‘opt[N][W]’ for the actual result.

2. **Breakpoints Needed:**

- At least four breakpoints are needed to address these errors.

Corrected Code:

```
public class Knapsack {

    public static void main(String[] args) {
    int N = Integer.parseInt(args[0]);
    int W = Integer.parseInt(args[1]);
    int[] profit = new int[N];
        int[] weight = new int[N];
         for (int n = 0; n < N; n++) {
             profit[n] = (int) (Math.random() * 1000);
             weight[n] = (int) (Math.random() * W);
             }
             int[][] opt = new int[N+1][W+1];
             boolean[][] sol = new boolean[N+1][W+1];
             for (int n = 1; n<=N; n++)
             {
             for (int w = 1; w <=W; w++) {
             int option1 = opt[n-1][w];
             int option2 = integer.MIN_VALUE;
             if (weight[n-1] <= w) {
             option2 =profit[n-1] + opt[n-1][w-weight[n-1]];
             }
             opt[n][w] = Math_max(option1, option2);
             sol[n][w] = option2 > option1;
             }
             boolean[] take = new Boolean[N];
             for (int n = N, w = W; n > 0; n--) {
             if(sol[n][w]) {
             take[n-1] = true;
             w = w-weight[n-1];
             }
             else {
             take[n-1] = false;
             }
             }
             System.out.println(“item”+”\t”+”profit”+ ”\t”+”weight”+ ”\t”+”take”);
             for (int n = 0; n<N; n++) {
             System.out.println((n+1) + ”\t” + profit[n] + ”\t”+ weight[n] + ”\t” +
             take[n]);
             }
             }
             }
```

Q4) MagicNumber

```
// Program to check if number is Magic number in JAVA
import java.util.*;
public class MagicNumberCheck
{
    public static void main(String args[])
    {
        Scanner ob=new Scanner(System.in);
        System.out.println("Enter the number to be checked."); int n=ob.nextInt();
        int sum=0,num=n;
        while(num>9)
        {
            sum=num;int s=0;
            while(sum==0)
            {
                s=s*(sum/10);
                sum=sum%10
            }
            num=s;
        }
        if(num==1)
        {
                     System.out.println(n+" is a Magic Number.");
        }
        else
        {
            System.out.println(n+" is not a Magic Number.");
        }
    }
}
```

Input: Enter the number to be checked 119 Output 119 is a Magic Number.

Input: Enter the number to be checked 199 Output 199 is not a Magic Number.

<u>Program Inspection:</u>

1. Errors:

Error 1: In the inner while loop, the loop condition is ‘while (sum == 0)’, which means the loop will only execute if ‘sum’ is initially 0, causing an infinite loop. The loop condition should likely be changed.

Error 2: Missing semicolons at the end of lines with ‘sum=sum%10’ and ‘s=s*(sum/10)’.

2. Effective Category

Category A (Data Reference Errors Errors) and Category C (Computation Errors).

3. Unidentified Error Types:

Program inspection identified lexical and computation errors. However, it might not identify potential logical errors, such as the loop condition and the computation within the loops.

4. Applicability:

Program inspection is useful for catching syntax and computation errors. To identify and fix logical errors in

the code, additional testing and debugging are required.

Debugging:

1. Errors:

- Error 1: Change the inner while loop's condition from ‘while (sum == 0)’ to ‘while (sum > 0)’ to avoid an infinite loop.
- Error 2: Add semicolons at the end of lines with ‘sum=sum%10’ and ‘s=s*(sum/10)’ to fix the syntax errors.

2. Breakpoints Needed:

- At least two breakpoints are needed to address these errors.

Corrected Codes:

```
import java.util.*;
public class MagicNumberCheck {
    public static void main(String args[]) {
        Scanner ob = new Scanner(System.in);
        System.out.println("Enter the number to be checked."); int n = ob.nextInt();
        int sum = 0, num = n; while (num >
        9) {
            sum = num; int s =
            0;
            while (sum > 0) {
                s = s + (sum % 10); sum =
                sum / 10;
            }
            num = s;
        }
        if (num == 1) {
            System.out.println(n + " is a Magic Number.");
        } else {
            System.out.println(n + " is not a Magic Number.");
        }
    }}
```

Q5) Merge Sort

```
// This program implements the merge sort algorithm for
// arrays of integers.

import java.util.*;

public class MergeSort {
    public static void main(String[] args) {
        int[] list = {14, 32, 67, 76, 23, 41, 58, 85};
        System.out.println("before: " + Arrays.toString(list));
        mergeSort(list);
        System.out.println("after:  " + Arrays.toString(list));
    }

    // Places the elements of the given array into sorted order
    // using the merge sort algorithm.
    // post: array is in sorted (nondecreasing) order
    public static void mergeSort(int[] array) {
        if (array.length > 1) {
            // split array into two halves
            int[] left = leftHalf(array+1);
            int[] right = rightHalf(array-1);

            // recursively sort the two halves
            mergeSort(left);
            mergeSort(right);

            // merge the sorted halves into a sorted whole
            merge(array, left++, right--);
        }
    }

    // Returns the first half of the given array.
    public static int[] leftHalf(int[] array) {
        int size1 = array.length / 2;
        int[] left = new int[size1];
        for (int i = 0; i < size1; i++) {
            left[i] = array[i];
        }
        return left;
    }

    // Returns the second half of the given array.
    public static int[] rightHalf(int[] array) {
        int size1 = array.length / 2;
        int size2 = array.length - size1;
        int[] right = new int[size2];
        for (int i = 0; i < size2; i++) {
            right[i] = array[i + size1];
        }
        return right;
    }

    // Merges the given left and right arrays into the given
    // result array.  Second, working version.
    // pre : result is empty; left/right are sorted
    // post: result contains result of merging sorted lists;
    public static void merge(int[] result,
                             int[] left, int[] right) {
        int i1 = 0;   // index into left array
        int i2 = 0;   // index into right array

        for (int i = 0; i < result.length; i++) {
            if (i2 >= right.length || (i1 < left.length &&
                    left[i1] <= right[i2])) {
                result[i] = left[i1];    // take from left
                i1++;
            } else {
                result[i] = right[i2];   // take from right
                i2++;
            }
        }
    }
 }


Input: before 14 32 67 76 23 41 58 85
       after 14 23 32 41 58 67 76 85
```

Program Inspection:

1. Errors:

- Error 1: In the ‘mergeSort’ method, the function calls ‘leftHalf(array+1)’ and ‘rightHalf(array 1)’. Instead, it should call ‘leftHalf(array)’ and ‘rightHalf(array)’.
- Error 2: The ‘merge’ method is missing, which is called in the ‘mergeSort’ method.

2. Effective Category:

- Category C (Computation Errors).

3. Unidentified Error Types:

- Program inspection identified computation errors, but it might not catch potential logical errors in the sorting

  algorithm.

4. Applicability:

- Program inspection is useful for catching syntax and computation errors. For more comprehensive testing and fixing logical errors, additional testing techniques (e.g., debugging and test cases) are required.

Debugging:

1. Errors:

- Error 1: In the ‘mergeSort’ method, replace ‘int[] left = leftHalf(array+1);’ with ‘int[] left = leftHalf(array);’ and ‘int[] right = rightHalf(array-1);’ with ‘int[] right = rightHalf(array);’.
- Error 2: The ‘merge’ method is called in the code but not provided. A correct implementation of the ‘merge’

  method is needed for the code to work.

2. Breakpoints Needed:

- At least two breakpoints are needed to address these errors.

Corrected Code:

```
import java.util.*;

public class MergeSort {

    public static void main(String[] args) {

        int[] list = {14, 32, 67, 76, 23, 41, 58, 85};

        System.out.println("before: " + Arrays.toString(list)); mergeSort(list);
        System.out.println("after: " + Arrays.toString(list));

    }

    public static void mergeSort(int[] array) { if (array.length > 1) {
            int[] left = leftHalf(array);

            int[] right = rightHalf(array); mergeSort(left);
            mergeSort(right);

            merge(array, left, right);

        }

    }

    public static int[] leftHalf(int[] array) { int size1 =
        array.length / 2;
        int[] left = new int[size1]; for (int i = 0; i <
        size1; i++) {
            left[i] = array[i];

        }

        return left;

    }

    public static int[] rightHalf(int[] array) { int size1 =
        array.length / 2;
        int size2 = array.length - size1; int[] right = new
        int[size2];
        for (int i = 0; i < size2; i++) { right[i] = array[i +
            size1];

        }

        return right;
```

Q6) Multiply Matrix

```
//Java program to multiply two matrices import java.util.Scanner;
class MatrixMultiplication
{
    public static void main(String args[])
    {
        int m, n, p, q, sum = 0, c, d, k;
        Scanner in = new Scanner(System.in);
        System.out.println("Enter the number of rows and columns of first matrix");
        m = in.nextInt();
        n = in.nextInt();
        int first[][] = new int[m][n];
        System.out.println("Enter the elements of first matrix"); for ( c = 0 ; c < m ; c++ )
            for ( d = 0 ; d < n ; d++ ) first[c][d] =
                in.nextInt();
        System.out.println("Enter the number of rows and columns of second
matrix");
        p = in.nextInt();
        q = in.nextInt(); if ( n !=
        p )
            System.out.println("Matrices with entered orders can't be multiplied with each
other.");
        else
        {
            int second[][] = new int[p][q];
            int multiply[][] = new int[m][q];
            System.out.println("Enter the elements of second
matrix");
            for ( c = 0 ; c < p ; c++ ) for ( d = 0 ;
                d < q ; d++ )
                    second[c][d] = in.nextInt(); for ( c = 0 ; c <
            m ; c++ )
            {
                for ( d = 0 ; d < q ; d++ )
                {
                    for ( k = 0 ; k < p ; k++ )
                    {
```

```
                    sum = sum + first[c-1][c-k]*second[k-1][k-d];
                 }
                 multiply[c][d] = sum; sum = 0;
              }
           }
           System.out.println("Product of entered matrices:-");
           for ( c = 0 ; c < m ; c++ )
           {
              for ( d = 0 ; d < q ; d++ )
                 System.out.print(multiply[c][d]+"\t"); System.out.print("\n");
           }
        }
    }
}


Input: Enter the number of rows and columns of first matrix 2 2
       Enter the elements of first matrix 1 2 3 4
       Enter the number of rows and columns of first matrix 2 2
       Enter the elements of first matrix 1 0 1 0
Output: Product of entered matrices: 3 0
         7 0
```

Program Inspection:

1. Errors:

- Error 1: In the nested loop that calculates the product of matrices, change ‘sum = sum + first[c-1][c-k]*second[k- 1][k-d];’ to ‘sum = sum + first[c][k] * second[k][d];’. The indices should start from 0.
- Error 2: In the nested loops, the variables ‘c’, ‘d’, and ‘k’ should be properly initialized within the for loops.

2. Effective Category:

- Category C (Computation Errors) Category B (Data declaration Errors).

3. Unidentified Error Types:

- Program inspection identified computation errors, but it might not catch potential logical errors in the matrix multiplication algorithm.

4. Applicability:

- Program inspection is useful for catching syntax and computation errors, but for more comprehensive testing and fixing logical errors, additional testing techniques (e.g., debugging and test cases) are required.

Debugging:

1. Errors:

- Error 1: In the nested loop that calculates the product of matrices, change ‘sum = sum + first[c-1][c-k]*second[k- 1][k-d];’ to ‘sum = sum + first[c][k] * second[k][d];’.
- Error 2: Initialize variables ‘c’, ‘d’, and ‘k’ properly within the for loops.

2. Breakpoints Needed:

- At least two breakpoints are needed to address these errors.

Corrected Code:

```
import java.util.Scanner; class
MatrixMultiplication {
    public static void main(String args[]) { int m, n, p, q, sum =
        0, c, d, k;
        Scanner in = new Scanner(System.in);
        System.out.println("Enter the number of rows and columns of the first matrix");
        m = in.nextInt();
        n = in.nextInt();
        int first[][] = new int[m][n];


        System.out.println("Enter the elements of the first matrix");
        for (c = 0; c < m; c++) { for (d = 0; d <
            n; d++) {
                first[c][d] = in.nextInt();
            }
        }
        System.out.println("Enter the number of rows and columns of the second matrix");
        p = in.nextInt();
        q = in.nextInt(); if (n != p)
        {
            System.out.println("Matrices with entered orders can't be multiplied with each
other.");
```

```
        } else {
            int second[][] = new int[p][q];
            int multiply[][] = new int[m][q];

            System.out.println("Enter the elements of the second matrix");
            for (c = 0; c < p; c++) {
                for (d = 0; d < q; d++) {
                    second[c][d] = in.nextInt();
                }
            }
            for (c = 0; c < m; c++) { for (d = 0; d <
                q; d++) {
                    sum = 0;
                        for (k = 0; k < p; k++) {
                                sum = sum + first[c][k] * second[k][d];
                    }
                    multiply[c][d] = sum;
                }
            }
            System.out.println("Product of entered matrices:-"); for (c = 0; c < m; c++) {
                for (d = 0; d < q; d++) {
                    System.out.print(multiply[c][d] + "\t");
                }
                System.out.println();
            }
        }
    }
}
```

Q7) Quadratic Probing

```
    /**

  *  Java Program to implement Quadratic Probing Hash Table
```

```
 **/

import java.util.Scanner;

/** Class QuadraticProbingHashTable **/
class QuadraticProbingHashTable
{
    private int currentSize, maxSize;
    private String[] keys;
    private String[] vals;

    /** Constructor **/
    public QuadraticProbingHashTable(int capacity)
    {
        currentSize = 0;
        maxSize = capacity;
        keys = new String[maxSize];
    vals = new String[maxSize];
}


/** Function to clear hash table **/
public void makeEmpty()
{
    currentSize = 0;
    keys = new String[maxSize];
    vals = new String[maxSize];
}


/** Function to get size of hash table **/
public int getSize()
{
    return currentSize;
}
```

```
/** Function to check if hash table is full **/
public boolean isFull()
{
    return currentSize == maxSize;
}

/** Function to check if hash table is empty **/
public boolean isEmpty()
{
    return getSize() == 0;
}

/** Fucntion to check if hash table contains a key **/
public boolean contains(String key)
{
    return get(key) !=  null;
}
```

```
/** Functiont to get hash code of a given key **/

private int hash(String key)

{

    return key.hashCode() % maxSize;

}



/** Function to insert key-value pair **/

public void insert(String key, String val)

{

    int tmp = hash(key);

    int i = tmp, h = 1;

    do

    {

        if (keys[i] == null)

        {

            keys[i] = key;
```

```
                vals[i] = val;

                currentSize++;

                return;

            }

            if (keys[i].equals(key))

            {

                vals[i] = val;

                return;

            }

            i + = (i + h / h--) % maxSize;

        } while (i != tmp);

    }



    /** Function to get value for a given key **/

    public String get(String key)

    {

        int i = hash(key), h = 1;
        while (keys[i] != null)

        {

            if (keys[i].equals(key))

                return vals[i];

            i = (i + h * h++) % maxSize;

            System.out.println("i "+ i);

        }

        return null;

    }



    /** Function to remove key and its value **/

    public void remove(String key)

    {

        if (!contains(key))

            return;



        /** find position key and delete **/
        int i = hash(key), h = 1;
        while (!key.equals(keys[i]))
            i = (i + h * h++) % maxSize;
        keys[i] = vals[i] = null;


        /** rehash all keys **/
        for (i = (i + h * h++) % maxSize; keys[i] != null; i = (i + h * h++) % maxSize)
        {
            String tmp1 = keys[i], tmp2 = vals[i];
            keys[i] = vals[i] = null;
            currentSize--;
            insert(tmp1, tmp2);
        }
        currentSize--;
    }


    /** Function to print HashTable **/
    public void printHashTable()

    {

        System.out.println("\nHash Table: ");

        for (int i = 0; i < maxSize; i++)

            if (keys[i] != null)

                System.out.println(keys[i] +" "+ vals[i]);

        System.out.println();

    }

}



/** Class QuadraticProbingHashTableTest **/

public class QuadraticProbingHashTableTest

{

    public static void main(String[] args)

    {

        Scanner scan = new Scanner(System.in);

        System.out.println("Hash Table Test\n\n");
System.out.println("Enter size");
/** maxSizeake object of QuadraticProbingHashTable **/
QuadraticProbingHashTable qpht = new QuadraticProbingHashTable(scan.nextInt() );

char ch;
/**  Perform QuadraticProbingHashTable operations  **/
do
{
    System.out.println("\nHash Table Operations\n");
    System.out.println("1. insert ");
    System.out.println("2. remove");
    System.out.println("3. get");
    System.out.println("4. clear");
    System.out.println("5. size");

    int choice = scan.nextInt();
    switch (choice)
```

```
{

case 1 :

   System.out.println("Enter key and value");

   qpht.insert(scan.next(), scan.next() );

   break;

case 2 :

   System.out.println("Enter key");

   qpht.remove( scan.next() );

   break;

case 3 :

   System.out.println("Enter key");

   System.out.println("Value = "+ qpht.get( scan.next() ));

   break;

case 4 :

   qpht.makeEmpty();

   System.out.println("Hash Table Cleared\n");

   break;
```

```
    }
```

```
          case 5 :

            System.out.println("Size = "+ qpht.getSize() );

            break;

          default :

            System.out.println("Wrong Entry \n ");

            break;

          }

          /** Display hash table **/

          qpht.printHashTable();



          System.out.println("\nDo you want to continue (Type y or n) \n");

          ch = scan.next().charAt(0);

        } while (ch == 'Y'|| ch == 'y');

      }

    }
```

Input:

Hash table test

```
Enter size: 5

Hash Table Operations

1. Insert

2. Remove

3. Get

4. Clear

5. Size


1


Enter key and value

c computer

d desktop

h harddrive


Output:

Hash Table:

c computer

d desktop
h harddrive
```

Program Inspection:

1. Errors:

- Error 1: In the ‘insert’ method, there is a logical error in the line ‘i + = (i + h / h--) %maxSize;’. The correct statement should be ‘i = (i + h * h++) % maxSize;’ to implement quadratic probing.
- Error 2: In the ‘get’ method, the loop condition should consider the case when the table is full. Change ‘while (keys[i] != null)’ to ‘for (int j = 0; j < maxSize; j++)’ to ensure that the loop will eventually stop.
- Error 3: In the ‘remove’ method, there is an issue with the loop that rehashes keys. Change ‘for (i = (i + h * h++) % maxSize; keys[i] != null; i = (i + h * h++) % maxSize)’ to ‘for (int j = 0; j < maxSize; j++)’ to ensure it rehashes properly.

2. Effective Category:

- Category C (Computation Errors) and Category E (Control-Flow Errors).

3. Unidentified Error Types:

- Program inspection has identified computation errors, but it might not catch potential logical errors in the hash table operations.

4. Applicability:

- Program inspection is useful for catching syntax and computation errors, but for more comprehensive testing and fixing logical errors, additional testing techniques (e.g., debugging and test cases) are required.

Debugging:

1. Errors:

- Error 1: In the ‘insert’ method, change ‘i + = (i + h / h--) % maxSize;’ to ‘i = (i + h * h++) % maxSize;’.
- Error 2: In the ‘get’ method, change the loop condition to ‘for (int j = 0; j < maxSize; j++)’ to ensure it doesn't run indefinitely.
- Error 3: In the ‘remove’ method, change the rehash loop to ‘for (int j = 0; j < maxSize; j++)’ to ensure it rehashes properly.

2. Breakpoints Needed:

- At least three breakpoints are needed to address these errors.

Corrected Code:

```
import java.util.Scanner;


class QuadraticProbingHashTable { private int currentSize, maxSize; private String[] keys;
private String[] vals;


public QuadraticProbingHashTable(int capacity) { currentSize = 0;
maxSize = capacity;
keys = new String[maxSize]; vals = new String[maxSize];
}


public void makeEmpty() { currentSize = 0;
keys = new String[maxSize]; vals = new String[maxSize];
}
```

```
public int getSize() {

return currentSize;
}

public boolean isFull() {
return currentSize == maxSize;
}

public boolean isEmpty() { return getSize() == 0;
}

public boolean contains(String key) { return get(key) != null;
}

private int hash(String key) {
return key.hashCode() % maxSize;
}

public void insert(String key, String val) { int tmp = hash(key);
int i = tmp, h = 1;

do {
if (keys[i] == null) { keys[i] = key;
vals[i] = val; currentSize++; return;
}
if (keys[i].equals(key)) { vals[i] = val;
return;
}
i = (i + h * h++) % maxSize;
} while (i != tmp);
}

public String get(String key) { int i = hash(key), h = 1;
for (int j = 0; j < maxSize; j++) { if (keys[i] != null) {
if (keys[i].equals(key)) { return vals[i];
}

i = (i + h * h++) % maxSize;
} else {
return null;
}
}
return null;
}

public void remove(String key) { if (!contains(key)) return;
int i = hash(key), h = 1;
while (!key.equals(keys[i])) i = (i + h * h++) % maxSize; keys[i] = vals[i] = null;
for (int j = 0; j < maxSize; j++) { if (keys[i] != null) {
String tmp1 = keys[i], tmp2 = vals[i]; keys[i] = vals[i] = null;
currentSize--;
insert(tmp1, tmp2);
} else {
currentSize--; break;
}
}
}

public void printHashTable() { System.out.println("\nHash Table: "); for (int i = 0; i < maxSize;
    i++) {
if (keys[i] != null) {
System.out.println(keys[i] + " " + vals[i]);
}
}
System.out.println();
}
}

public class QuadraticProbingHashTableTest { public static void main(String[] args) {
Scanner scan = new Scanner(System.in); System.out.println("Hash Table Test\n\n");
    System.out.println("Enter size");
QuadraticProbingHashTable qpht = new QuadraticProbingHashTable(scan.nextInt());

char ch;
do {
System.out.println("\nHash Table Operations"); System.out.println("1. insert ");
System.out.println("2. remove"); System.out.println("3. get"); System.out.println("4. clear");
```

```
    System.out.println("5. size");
int choice = scan.nextInt(); switch (choice) {
case 1:
System.out.println("Enter key and value"); qpht.insert(scan.next(), scan.next());
break; case 2:
System.out.println("Enter key"); qpht.remove(scan.next());
break; case 3:
System.out.println("Enter key");

System.out.println("Value = " + qpht.get(scan.next()));
break; case 4:
qpht.makeEmpty();
System.out.println("Hash Table Cleared\n"); break;
case 5:
System.out.println("Size = " + qpht.getSize()); break;
default:
System.out.println("Wrong Entry \n"); break;
}
qpht.printHashTable();
System.out.println("\nDo you want to continue (Type y or n) \n");
ch = scan.next().charAt(0);
} while (ch == 'Y' || ch == 'y');
}
}
```

Q8) Sorting Array

```
// sorting the array in ascending order import java.util.Scanner;

public class Ascending _Order

{

    public static void main(String[] args)

    {

        int n, temp;

        Scanner s = new Scanner(System.in);

        System.out.print("Enter no. of elements you want in array:");

        n = s.nextInt();

        int a[] = new int[n];
        System.out.println("Enter all the elements:"); for (int i = 0; i < n; i++)
        {
            a[i] = s.nextInt();
        }
        for (int i = 0; i >= n; i++);
        {
            for (int j = i + 1; j < n; j++)
            {
                if (a[i] <= a[j])
                {
                    temp = a[i]; a[i] =
                    a[j];
                    a[j] = temp;
                }
            }
        }
        System.out.print("Ascending Order:"); for (int i = 0; i < n - 1;
        i++)
        {
            System.out.print(a[i] + ",");
        }
        System.out.print(a[n - 1]);
    }
}

Input: Enter no. of elements you want in array: 5 Enter all elements:
       1 12 2 9 7
       1 2 7 9 12
```

Program Inspection:

1. **Errors**:

   - **Error 1**: In the first for loop, the condition for(int i=0; i>=n; i++); is incorrect. It uses i>= n, which will never be true, and as a result, the loop's body will not execute.
   - **Error 2**: The loop condition in the second for loop should be i<n, not i>=n. The same issue is present in this loop as well.

2. **Effective Category**:
   - Category C (Computation Errors)
   - Category D (Comparison Errors).
3. **Unidentified Error Types**:
   - The program inspection process identified computation errors, but it might not catch potential logical errors in the sorting logic.
4. **Applicability**:
   - Program inspection is useful for catching syntax and computation errors, but for more comprehensive testing and fixing logical errors, additional testing techniques (e.g., debugging and test cases) are required.

Debugging:

1. **Errors**:
   - **Error 1**: In the first for loop, change for(int i=0; i>=n; i++); to for(int i=0; i<n; i++) to correctly iterate

     through the array.
   - **Error 2**: In the second for loop, change for (int i = 0; i >= n; i++) to for (int i = 0; i < n; i++) to correctly

     iterate through the array.
2. **Breakpoints Needed**:
   - Two breakpoints are needed to address these errors.
3. **Steps Taken to Fix Errors**:
   - In the first for loop, change the condition to i < n to iterate through the array.
   - In the second for loop, change the condition to i < n to iterate through the array.

Corrected Code:

```
import java.util.Scanner;
public class Ascending_Order {
public static void main(String[] args) { int n, temp;
Scanner s = new Scanner(System.in);
System.out.print("Enter no. of elements you want in the array: ");
n = s.nextInt();

int a[] = new int[n];
System.out.println("Enter all the elements: ");
for (int i = 0; i < n; i++) { a[i] = s.nextInt();
}
for (int i = 0; i < n; i++) {
for (int j = i + 1; j < n; j++) { if (a[i] > a[j]) {
temp = a[i]; a[i] = a[j];
a[j] = temp;
}
}
}
System.out.print("Ascending Order: "); for (int i = 0; i < n - 1; i++) {
System.out.print(a[i] + ", ");
}
System.out.print(a[n - 1]);
}
}
```

Q9) Stack Implementation

```
//Stack implementation in java import
java.util.Arrays;


public class StackMethods { private int
    top;
    int size;
    int[] stack ;


    public StackMethods(int arraySize){ size=arraySize;
        stack= new int[size]; top=-1;
    }


    public void push(int value){ if(top==size-
        1){
            System.out.println("Stack is full, can't push a value");
        }
        else{


            top--;
            stack[top]=value;

        }

    }


    public void pop(){ if(!isEmpty())
            top++;
        else{
            System.out.println("Can't pop...stack is empty");

        }

    }


    public boolean isEmpty(){ return
        top==-1;
    }
    public void display(){ for(int i=0;i>top;i++){
            System.out.print(stack[i]+ " ");

        }

        System.out.println();
    }

}
public class StackReviseDemo {


    public static void main(String[] args) {

        StackMethods newStack = new StackMethods(5); newStack.push(10);
        newStack.push(1);
        newStack.push(50);
        newStack.push(20);
        newStack.push(90);


        newStack.display();
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.display();

    }

}
```

```
output:

      10

       1

       50

       20

       90

       10
```

Program Inspection:

1. Errors:

- Error 1: In the ‘push’ method, the top should be incremented before pushing the value. However, it's currently being decremented, leading to incorrect

  behavior.

- Error 2: In the ‘display’ method, the loop condition is using ‘>‘ instead of ‘<‘, which will not display the stack correctly.

2. Effective Category:

- Category C (Computation Errors) and Category D (Comparison Errors).

3. Unidentified Error Types:

- While this inspection identified computation errors, it might not catch potential logical errors in the behavior of the stack (e.g., handling overflow or underflow).

4. Applicability:

- Program inspection is useful for catching syntax and computation errors, but for more comprehensive testing, additional testing techniques are required.

Debugging:

1. Errors:

- Error 1: In the ‘push’ method, change ‘top--’ to ‘top++’ to increment the top before pushing the value.
- Error 2: In the ‘display’ method, change the loop condition from ‘for(int i=0; i > top; i++)’ to ‘for (int i = 0; i

  < top; i++)’ to correctly display the stack.

2. Breakpoints Needed:

- Two breakpoints are needed to address these errors.

3. Steps Taken to Fix Errors:

- In the ‘push’ method, change ‘top--’ to ‘top++’.- In the ‘display’ method, change the loop condition to use ‘<‘ instead of ‘>‘.

Corrected Code:

```
import java.util.Arrays;


public class StackMethods { private int top;

int size;

int[] stack;


public StackMethods(int arraySize) { size = arraySize;

stack = new int[size]; top = -1;

}


public void push(int value) { if (top == size - 1) {

System.out.println("Stack is full, can't push a value");

} else {

top++;

stack[top] = value;

}

}


public void pop() { if (!isEmpty()) {

top--;

} else {

System.out.println("Can't pop...stack is empty");

}

}


public boolean isEmpty() { return top == -1;

}
public void display() {

for (int i = 0; i <= top; i++) { System.out.print(stack[i] + " ");

}

System.out.println();

}

}


public class StackReviseDemo {

public static void main(String[] args) {

StackMethods newStack = new StackMethods(5); newStack.push(10);

newStack.push(1); newStack.push(50);


newStack.push(20); newStack.push(90); newStack.display(); newStack.pop();

newStack.pop(); newStack.pop(); newStack.pop();

newStack.display();

}
}
```

Q10) Tower of Hanoi

```
//Tower of Hanoi

public class MainClass {

    public static void main(String[] args) { int nDisks = 3;
        doTowers(nDisks, 'A', 'B', 'C');

    }

    public static void doTowers(int topN, char from, char inter, char to) {
        if (topN == 1){

            System.out.println("Disk 1 from "

            + from + " to " + to);

        }else {

            doTowers(topN - 1, from, to, inter); System.out.println("Disk "
            + topN + " from " + from + " to " + to); doTowers(topN ++,
            inter--, from+1, to+1)

        }

    }
}


Output: Disk 1 from A to C Disk 2
    from A to B
    Disk  1  from  C  to  B
    Disk  3  from  A  to  C
    Disk  1  from  B  to  A
    Disk  2  from  B  to  C
    Disk 1 from A to C
```

Program Inspection:

1. **Errors:**

- Error 1: Incorrect usage of post-increment and post- decrement operators (‘topN++’, ‘inter--’).
- Error 2: Incorrect order of parameters in the recursive call to ‘doTowers’.

2. **Effective Category:**

- Category C (Computation Errors).

3. **Unidentified Error Types:**

- More complex logic errors or algorithmic issues may not be identified by this program inspection checklist.

4. **Applicability:**

- The program inspection technique is worth applying to catch syntax and logical errors, but it should be complemented with other testing methods.

Debugging:

1. **Errors:**

- Error 1: Incorrect usage of post-increment and post- decrement operators (‘topN++’, ‘inter- ‘).
- Error 2: Incorrect order of parameters in the recursive call to ‘doTowers’.

2. **Breakpoints Needed:**

- At least two breakpoints are needed to address these errors.

3. **Steps Taken to Fix Errors:**

- Replace ‘topN++’ with ‘topN--’.- Swap the order of ‘from’ and ‘inter’ in the recursive call.

Corrected Code:

```
   public class MainClass {
    public static void main(String[] args) { int nDisks = 3;
        doTowers(nDisks, 'A', 'B', 'C');
    }


    public static void doTowers(int topN, char from, char inter, char to) {
        if (topN == 1) {
            System.out.println("Disk 1 from " + from + " to " + to);
        } else {
            doTowers(topN - 1, from, to, inter);
            System.out.println("Disk " + topN + " from " + from + " to " + to);
            doTowers(topN - 1, inter, from, to);
        }
    }
}
```