Name : - Kunj Kheni
ID : - 202201423

IT - 314 Software Engineering

2000 Lines of code : - I have used my Numerical and Computational Lab(CS 374) codes combined to do this lab.

Code 1 : -

1. How many errors can you identify in the program? List the errors below.

Category A: Data Reference Issues

- **Uninitialized Variables:** The constructor method is incorrectly named instead _init_ of __init__, preventing it from being properly executed. Consequently, variables like self.matrix, self.vector, and self.res do not get initialized.

Category C: Computation Problems

- **Risk of Division by Zero:** In the gauss method, if the element A[i][i] equals zero, there's a risk of a division by zero occurring when executing the operation x[i] /= A[i][i].

Category D: Comparison Issues

- **Incorrect Comparison Logic:** In the diagonal_dominance method, the logic may fail if rows have repeated maximum values, potentially leading to incorrect results and unpredictable behavior.

Category E: Control Flow Mistakes

- **Off-by-One Error:** In the get_upper_permute method, the loop iterating over k is mistakenly written as for k in range(i+1, n+1) when it should be for k in range(i+1, n) to prevent accessing out-of-bounds elements.

Category F: Interface Problems

- **Incorrect Parameter Handling:** The method assumes specific input formats, such as a list of lists for matrices and a list for vectors, but lacks proper validation for these inputs and doesn't check for unexpected values.

Category G: Input/Output Issues

- **No Error Handling for Input Data:** The program does not account for cases where input matrices may not be square or where the matrix dimensions do not match the requirements for the operations being performed.

Category H: Other Considerations

- **Missing Library Imports:** Essential libraries like numpy and matplotlib.pyplot are not imported, which are necessary for the program's execution.

2. Which category of inspection would you consider more useful?

Category A: Data Reference Errors

- Uninitialized Variables: The constructor is incorrectly defined as _init_ instead of __init__, leading to , self.vector, and self.res not being properly initialized. Correcting the constructor ensures these variables are set up correctly.

Category D: Comparison Errors

- **Faulty Comparison Logic:** The diagonal_dominance method can produce incorrect results when rows contain duplicate maximum values, leading to inaccurate diagonal dominance checks and unpredictable behavior. Resolving this issue will help avoid logical inconsistencies during computations.

3. Which type of error are you not able to identify using the program inspection?

Logic Errors

- Program inspection is useful for identifying syntax and runtime issues, but it may not catch logical errors. These occur when the program runs without errors but yields incorrect results due to flaws in the algorithm or miscalculations. For instance, in methods like Jacobi or Gauss-Seidel, the algorithms may either converge slowly or fail to converge altogether under specific conditions—problems that program inspection alone may not uncover.

4. Is the program inspection technique worth applying?

Yes, the program inspection technique is certainly applicable. It aids in detecting frequent and critical errors, especially those concerning data references, computations, comparisons, and control flow. While this technique enhances code quality and reliability, it should be complemented with other testing approaches, such as unit and integration testing, to catch logical errors and verify that the program functions as expected across different scenarios.

Code 2 : -

1. How many errors are there in the program? Mention the errors you have identified.

Category A: Data Reference Errors

- **Constructor Naming Error:** The constructor is incorrectly defined as _init_ rather than __init__, preventing it from being invoked when an instance of the Interpolation class is created.
- **Potential Index Errors:** In the cubicSpline and piecewise_linear_interpolation methods, matrix elements are accessed without checking if the indices are within valid bounds, which could result in an IndexError.
- **No Type Checking:** The code lacks type validation to ensure that matrix elements are numeric (either integers or floats). If non-numeric values are passed, this could lead to runtime errors.

Category C: Computation Errors

- **Division by Zero Risk:** In the piecewise_linear_interpolation method, the calculation of the slope may result in division by zero if two consecutive x-values are the same.

2. Which category of program inspection would you find more effective?

Data Reference Errors (Category A) would be the most effective for inspection in this code. This category tackles critical problems like improper initialization and index handling, including the misnamed constructor (_init_ instead of __init__) and potential index out-of-bound errors when accessing matrix elements. Addressing these issues is essential to avoid runtime errors and ensure the program operates reliably.

3. Which type of error are you not able to identify using program inspection?

Runtime Errors:

Program inspection might not catch certain runtime errors. For example, floating-point precision issues, such as those that could lead to division by zero in the piecewise_linear_interpolation method, or errors that arise during execution due to improper handling of specific data sets may go unnoticed through inspection alone.

4. Is the program inspection technique worth applying?

Yes, program inspection is a highly valuable technique. It helps identify potential issues early in development, enhances code quality, encourages adherence to best practices, and reduces long-term costs associated with debugging and maintenance. However, it should be used alongside other testing approaches, such as unit testing and dynamic analysis, to ensure broader coverage of potential errors and ensure the program functions correctly in different scenarios.

Code 3 : -

1. How many errors are there in the program? Mention the errors you have identified.

Category A: Data Reference Errors

- **Redundant Function Definitions**: The fun and dfun functions are defined repeatedly for different equations, but it's unclear which function corresponds to which equation, leading to possible confusion.
- **Undefined Behavior Due to Variable Reuse**: The variable data is used to store results from different iterations, but it isn't clearly reset between function calls, potentially causing unexpected behavior when solving for multiple roots consecutively.

Category B: Data-Declaration Errors

- **Uninitialized Variables**: In the first iteration of the loop, next is computed before it's initialized, which could result in NaN values.
- **Incorrect DataFrame Initialization**: The DataFrame df is initialized only after the loop, meaning if the loop doesn't run (due to early convergence), the data might not be properly formed, leading to potential errors.

Category C: Computation Errors

- **Inaccurate Function Evaluation**: The line fpresent = fun(present) only checks the value of next for convergence, but it should also consider the convergence of |fun(present)|.
- **Error Calculation Inaccuracy**: The error is calculated using error.append(next - present), which might not accurately represent true convergence since it compares only the last two iterations instead of consecutive values from the iterative process.

Category D: Comparison Errors

- **Incorrect Error Condition**: The error check between next and present may not account for cases where present is close to alpha but hasn't truly converged.
- **Weak Convergence Criteria**: The convergence check only uses abs(next - present) > err, overlooking the need to verify if |fun(next)| < err to ensure proper convergence.

Category E: Control-Flow Errors

- **Risk of Infinite Loop**: If the initial guess is far from the root or if dfun(present) equals zero (e.g., vertical tangents), the loop may become infinite, preventing convergence.
- **Missing Break Conditions:** There are no break conditions to stop the loop after a fixed number of iterations or to prevent division by zero in next = present - (fpresent / dfpresent).

Category F: Input/Output Errors

- **Lack of Iteration Logging**: The program doesn't log the progress of each iteration, making it hard to track how the algorithm is evolving.
- **Confusing Plot Titles**: The plot titles don't clearly indicate which function or root they represent, causing confusion when comparing different roots from multiple functions.

Category G: Other Checks

- **Unaddressed Edge Cases**: The code does not account for edge cases, such as when the function doesn't have a root in the given domain or when the derivative creates undefined behavior.

- **Overlapping Plots**: New plots are created without clearing previous data, leading to cluttered visualizations when testing multiple functions consecutively.

2. Which category of program inspection would you find more effective?

Data Reference Errors (Category A): This category focuses on ensuring that inputs are properly defined and managed, thereby preventing numerous runtime errors and enhancing the program's overall robustness.

3. Which type of error are you not able to identify using the program inspection?

Non-obvious Logical Errors: These may involve issues like converging to the wrong root or experiencing numerical instability, which might only surface during runtime with specific input values.

4. Is the program inspection technique worth applying?

Yes, program inspection is a highly effective technique. It can reveal various errors and significantly enhance code quality. These inspection methods are especially beneficial in collaborative settings, as they improve code readability and maintainability.

Code 4 : -

1. How many errors are there in the program? Mention the errors you have identified.

Category A: Data Reference Errors

- **Inconsistent Input Structure**: The input matrix is anticipated to be a 2D array; however, the code does not validate or manage incorrect input shapes, which could result in runtime errors.
- **Variable Reuse Without Clear Definition**: The variables coef and poly_i are reused in different contexts (both inside and outside the function), leading to potential confusion about their intended use.

Category B: Data-Declaration Errors

- **Uninitialized Variables in Plotting**: The plotting function plot_fun does not consider situations where y may be empty or not properly initialized, resulting in errors when attempting to create a plot.
- **No Error Handling for Matrix Inversion**: There is no verification to ensure that ATAA^TAATA is invertible before invoking np.linalg.inv(ATA), which could cause a crash if the matrix is singular.

Category C: Computation Errors

- **Potential Loss of Precision**: The line coef = coef[::-1] reverses the coefficients, but np.poly1d requires them to be in descending order. This misalignment could lead to unexpected polynomial behavior.
- **Overwriting Coefficients**: Coefficients for each order are calculated and stored in coef within a loop, but they are not separated for each polynomial, which may cause confusion regarding which coefficients belong to which polynomial.

Category D: Comparison Errors

- **Incorrect Error Tolerance**: The hardcoded value err = 1e-3 in plot_fun might not be suitable for all datasets and lacks the flexibility to adjust dynamically based on the input ranges.
- **Inadequate Comparison Logic in Plotting**: The code does not ensure that each polynomial is clearly labeled or that the plot's legend accurately represents the lines being plotted.

Category E: Control-Flow Errors

- **Infinite Loop Risk in Plotting**: The plot_fun function could enter an infinite loop if improperly formatted data is provided, especially if there are no points available to plot.
- **Lack of Early Exit Conditions**: The leastSquareErrorPolynomial function does not implement early exit conditions to detect poorly conditioned matrices or when the polynomial degree mmm is excessively high for the number of points available.

Category F: Input/Output Errors

- **No User Feedback on Processing**: There are no print statements or logging mechanisms to indicate the progress or completion of the polynomial fitting process, making it challenging for users to track execution.
- **Misleading Variable Naming**: The variable poly_i may cause confusion, as it suggests a single polynomial while actually holding a polynomial object. A more descriptive name would enhance clarity.

Category G: Other Checks

- **No Handling of Edge Cases**: The function does not account for situations where all yyy values are identical, resulting in a constant polynomial, which could confuse the user.
- **Lack of Unit Tests or Assertions**: There are no unit tests or assertions in place to validate input parameters and ensure that the function behaves as expected across various cases.

Category H: General Code Quality

- **Redundant Code Sections**: The code for plotting multiple polynomials is repetitive and could be consolidated into a function for improved reusability.
- **Missing Function Documentation**: The functions lack proper documentation, making it difficult for other users (or even the original author) to understand their purpose and expected behavior.

2. Which category of program inspection would you find more effective?

Computation Errors (Category C): It is essential to ensure that computations are executed accurately, as this directly impacts the precision of results, particularly in numerical methods such as polynomial fitting.

3. Which type of error are you not able to identify using the program inspection?

Data-Specific Errors: Some edge cases related to input data (such as when all yyy values are identical) may only become apparent once the function is executed with particular datasets.

4. Is the program inspection technique worth applying?

Yes, program inspection is a valuable technique. It allows for systematic error identification and improvement in code structure and maintainability, making it especially valuable in complex numerical methods and data analysis tasks.

Section - 2

1) Armstrong Number :

1. How many errors are there in the program?

There are **2 errors** in the program.

2. How many breakpoints do you need to fix those errors?

We need **2 breakpoints** to fix these errors.

Steps Taken to Fix the Errors:

- **Error 1:** The division and modulus operations were swapped in the while loop.
  Fix: Adjust the code to ensure that the modulus operation retrieves the last digit, while the division operation correctly reduces the number for the subsequent iteration.
- **Error 2:** The check variable was not properly accumulated.
  Fix: Revise the logic to ensure that the check variable accurately represents the sum of each digit raised to the power of the total number of digits.

Corrected Code:

```
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 an Armstrong Number");

    }

}
```

2) GCD and LCM :

1. How many errors are there in the program?

There are **1 error** in the program.

2. How many breakpoints do you need to fix those errors?

We need **1 breakpoint** to fix these errors.

Steps Taken to Fix the Errors:

- **Error:** The condition in the while loop of the GCD method is incorrect.
  Fix: Change the condition to while (a % b != 0) instead of while (a % b == 0). This ensures the loop continues until the remainder is zero, correctly calculating the GCD.

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) ? x : y; // a is greater number

    b = (x < y) ? x : y; // b is smaller number


    r = b;

    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[]) {
```

```java
        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();

    }

}
```

3) Knapsack Problem :

1. How many errors are there in the program?

There are **3 errors** in the program.

2. How many breakpoints do you need to fix those errors?

We need **2 breakpoints** to fix these errors.

Steps Taken to Fix the Errors:

- **Error:** In the "take item n" case, the condition is incorrect.
  Fix: Change if (weight[n] > w) to if (weight[n] <= w) to ensure the profit is calculated when the item can be included.
- **Error:** The profit calculation is incorrect.
  Fix: Change profit[n-2] to profit[n] to ensure the correct profit value is used.
- **Error:** In the "don't take item n" case, the indexing is incorrect.
  Fix: Change opt[n++][w] to opt[n-1][w] to properly index the items.

Corrected Code:

```
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];
```

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

        // don't take item n
        int option1 = opt[n - 1][w];

        // take item n
        int option2 = Integer.MIN_VALUE;
        if (weight[n] <= w)
            option2 = profit[n] + 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]);

        }

    }

}
```

4) Magic Number Check :

1. How many errors are there in the program?

There are **3 errors** in the program.

2. How many breakpoints do you need to fix those errors?

We need **1 breakpoint** to fix these errors.

Steps Taken to Fix the Errors:

- **Error:** The condition in the inner while loop is incorrect.
  Fix: Change while(sum==0) to while(sum!=0) to ensure that the loop

processes digits correctly.

- **Error:** The calculation of s in the inner loop is incorrect.
  Fix: Change s=s*(sum/10) to s=s+(sum%10) to correctly sum the digits.
- **Error:** The order of operations in the inner while loop is incorrect.
  Fix: Reorder the operations to s=s+(sum%10); sum=sum/10; to correctly accumulate the digit sum.

Corrected Code:

```
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.");

        }

    }

}
```

5) Merge Sort :

1. How many errors are there in the program?

There are **3 errors** in the program.

2. How many breakpoints do you need to fix those errors?

We need **2 breakpoints** to fix these errors.

Steps Taken to Fix the Errors:

- **Error:** Incorrect array indexing when splitting the array in mergeSort.
  Fix: Change int[] left = leftHalf(array+1) to int[] left = leftHalf(array) and int[] right = rightHalf(array-1) to int[] right = rightHalf(array) to pass the array correctly.
- **Error:** Incorrect increment and decrement in merge.
  Fix: Remove the ++ and -- from merge(array, left++, right--) and instead use merge(array, left, right) to pass the arrays directly.
- **Error:** The array access in the merge function is incorrectly accessing beyond the array bounds.
  Fix: Ensure the array boundaries are respected by adjusting the indexing in the merging logic.

Corrected Code:

```
import java.util.*;
```

```java
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 + 1) / 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;

}


public static void merge(int[] result,

        int[] left, int[] right) {

    int i1 = 0;

    int i2 = 0;


    for (int i = 0; i < result.length; i++) {

        if (i2 >= right.length || (i1 < left.length &&
```

```
                    left[i1] <= right[i2])) {

                result[i] = left[i1];

                i1++;

            } else {

                result[i] = right[i2];

                i2++;

            }

        }

    }

}
```

6) Matrix Multiplication :

1. How many errors are there in the program?

There are **1 error** in the program.

2. How many breakpoints do you need to fix those errors?

We need **1 breakpoint** to fix these errors.

Steps Taken to Fix the Errors:

- **Error:** Incorrect array indexing in the matrix multiplication logic.
  Fix: Change first[c-1][c-k] and second[k-1][k-d] to first[c][k] and second[k][d]. These changes ensure that matrix elements are correctly referenced during multiplication.

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 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 < n; k++) {

                        sum += first[c][k] * second[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");
            }
        }
    }
}
```

7) Quadratic Probing Hash Table :

1. How many errors are there in the program?

There are **1 error** in the program.

2. How many breakpoints do you need to fix those errors?

We need **1 breakpoint** to fix these errors.

Steps Taken to Fix the Errors:

- **Error:** In the insert method, the line i += (i + h / h--) % maxSize; is incorrect.

- **Fix:** The correct logic should be i = (i + h * h++) % maxSize; to correctly implement quadratic probing.

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; // Fixed quadratic probing
        } while (i != tmp);
    }

    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;
        }
        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;

        currentSize--;


        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);

        }

    }


    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\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;

            }

            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');

    }

}
```

8) Sorting Array :

1. How many errors are there in the program?

There are **2 errors** in the program.

2. How many breakpoints do you need to fix those errors?

We need **2 breakpoints** to fix these errors.

Steps Taken to Fix the Errors:

- **Error 1:** The loop condition for (int i = 0; i >= n; i++); is incorrect.
- **Fix 1:** Change it to for (int i = 0; i < n; i++) to correctly iterate over the array.
- **Error 2:** The condition in the inner loop if (a[i] <= a[j]) should be reversed.
- **Fix 2:** Change it to if (a[i] > a[j]) to correctly sort the array in ascending

order.

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 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();

        }


        // Corrected sorting logic

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

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

                if (a[i] > a[j]) { // Fixed comparison

                    temp = a[i];

                    a[i] = a[j];

                    a[j] = temp;
```

```java
                }
            }
        }

        System.out.print("Ascending Order: ");
        for (int i = 0; i < n - 1; i++) {
            System.out.print(a[i] + ", ");
        }
        System.out.print(a[n - 1]);
    }
}
```

9) Stack Implementation:

1. How many errors are there in the program?

There are **2 errors** in the program.

2. How many breakpoints do you need to fix those errors?

We need **2 breakpoints** to fix these errors.

Steps Taken to Fix the Errors:

- **Error 1:** In the push method, the line top-- is incorrect.
- **Fix 1:** Change it to top++ to correctly increment the stack pointer.
- **Error 2:** In the display method, the loop condition for (int i=0; i>top; i++) is incorrect.
- **Fix 2:** Change it to for (int i=0; i<=top; i++) to correctly display all elements.

Corrected Code:

```java
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++; // Fixed increment

            stack[top] = value;

        }

    }


    public void pop() {

        if (!isEmpty()) {
```

```java
            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++) { // Corrected loop condition

            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();

    }

}
```

10) Tower of Hanoi :

1. How many errors are there in the program?

There are **1 error** in the program.

2. How many breakpoints do you need to fix those errors?

We need **1 breakpoint** to fix these errors.

Steps Taken to Fix the Errors:

- **Error:** In the recursive call doTowers(topN ++, inter--, from+1, to+1);, incorrect increments and decrements are applied to the variables.
- **Fix:** Change the call to doTowers(topN - 1, inter, from, to); for proper recursion and to follow the Tower of Hanoi logic.

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); // Corrected recursive
call

        }

    }

}
```