IT314 – Software Engineering Lab-07
Program Inspection, Debugging and Static Analysis

Ayush Popshetwar – 202201412

1. Program Inspection

The code on which I performed program inspection: https://github.com/kdave12/15112-TermProject-2000-lines-of-Python-code/blob/master/termProject.py

Category A: Data Reference errors

1. Unset or Uninitialized Variables: Several variables (data.finalImg, data.finalPuzzle, data.solverFinal, data.solverShuffled, data.pieceSelected) may be used without checking if they are None.
2. Array Subscript Bounds: Variables (data.Anim1X1, data.Anim2Y1, data.Anim3Y1, data.Anim4X1) are modified

without verifying if they are within defined bounds after updates.
3. Integer Subscript References: No explicit array subscripts are present, avoiding non-integer subscript issues.
4. Dangling References: Potential dangling references exist for data.solverFinal, data.solverShuffled, data.finalImg, and data.finalPuzzle, as these may not be allocated before use.
5. Aliasing Issues: No aliasing problems are found, as there are no multiple names referencing the same memory area.
6. Type/Attribute Mismatch: Potential type mismatches for data.finalImg, data.finalPuzzle, solverShuffled, solverFinal, and others, which may lead to issues if expected types do not match.
7. Addressing Problems: No addressing issues related to memory allocation vs. addressability are observed.
8. Pointer/Reference Variable Expectations: No pointer or reference variables are present, as Python manages memory abstraction.
9. Consistent Data Structure Definitions: There may be potential inconsistencies in the structure of data across various functions, risking undefined behavior.
10. String Indexing Errors: No string indexing operations are present, eliminating off-by-one error risks.
11. Inheritance Requirements: The code does not implement classes or inheritance, making it impossible to assess these requirements.

Category B: Data-Declaration Errors

1. Variable Declarations:

- data.mode = "grid" - Confirm data.nextX1 and data.nextX2 are defined.
- data.conBack = PhotoImage(file=filename, master=canvas) - Verify filename exists.
- data.finalImg = PhotoImage(file=filename, master=canvas) - Validate the existence of this file.
- data.puzzleName = "puzzle1_x2.gif" - Ensure this filename exists.

2. Default Attributes:
   - Accessing data.nextX1 and data.nextX2 without initialization may cause an AttributeError.
   - Using data.conBack without initialization can lead to an error.
   - Accessing data.finalImg without initialization may also cause an AttributeError.
   - Assigning to data.puzzleName without prior definition might raise an error.
3. Variable Initialization:
   - data.mode = "grid" - Proper initialization should be confirmed.
   - data.conBack = PhotoImage(file=filename, master=canvas) - Verify that filename is set correctly.
   - data.finalImg = PhotoImage(file=filename, master=canvas) - Check that filename is defined and valid.
   - data.puzzleName = "puzzle1_x2.gif" - Verify that this variable is initialized in the right context.
4. Correct Length and Data Type:
   - Confirm data.conBack is initialized with a valid PhotoImage object.

    - data.finalImg should be a valid PhotoImage type, not an invalid file.
    - data.puzzleName should be confirmed as a string.
5. Memory Type Consistency:
    - data.mode should be a string.
    - data.conBack must be correctly initialized as a PhotoImage object.
    - data.finalImg should be consistently assigned as a PhotoImage.
6. Similar Variable Names: No similar variable names were found that could lead to confusion.

Category C: Computation errors

1. Inconsistent Data Types: Ensure data.timer is an integer, and check if data.puzzleWidth and data.puzzleHeight are consistently used in arithmetic. Verify that x1 and y1 are integers in calculations.
2. Mixed-Mode Computations: No mixed-mode computations detected.
3. Variable Lengths: Confirm that computations in canvas.create_rectangle use compatible lengths.
4. Data Type Mismatch: No explicit mismatches detected; maintain compatibility in assignments, especially involving image dimensions and numeric values.
5. Overflow/Underflow Risks: Watch for potential overflow in arithmetic operations with large integers or floats, particularly with data.timer.
6. Division by Zero: Division in gameTime is safe as long as data.timer is not zero; confirm that data.timer is initialized and non-zero.

7. Base-2 Inaccuracies: No significant issues with base-2 representation noted in Python.
8. Value Range Validation: Variables like currPos, finalPos, data.timer, imgWidth, and imgHeight must be validated to prevent out-of-bounds values.
9. Operator Precedence: The order of operations follows the expected precedence rules.
10. Integer Arithmetic Validity: No invalid uses of integer arithmetic detected.

Category D: Comparison Errors

1. Different Data Types: No comparisons between different data types detected.
2. Mixed-Mode Comparisons: No mixed-mode comparisons found.
3. Comparison Operators: All comparison operators are used correctly.
4. Boolean Expressions: All Boolean expressions are stated correctly.
5. Boolean Operands: No incorrect mixing of comparison and Boolean operators observed.
6. Floating-Point Comparisons: No issues found with floating-point comparisons represented in base-2.
7. Operator Precedence: The order of evaluation for multiple Boolean operators is clear and correct.
8. Compiler Evaluation Effects: No problematic evaluations affecting the program identified.

The code contains no errors in Category D. It adheres to correct logic regarding comparisons and Boolean expressions, with no critical issues identified.

Category E: Control Flow Errors

1. Multiway Branching: No issues found regarding the potential for the index variable to exceed branch possibilities.
2. Loop Termination: Each loop is structured to eventually terminate.
3. Program Termination: The program, module, or subroutine is designed to eventually terminate.
4. Loop Execution: There are no conditions that prevent loops from executing, avoiding oversight issues.
5. Loop Fall-Through: There are no consequences of fall-through in loops that would cause infinite iterations.
6. Off-by-One Errors: No off-by-one errors detected in loop iterations.
7. Statement Grouping: All statement groups and code blocks are correctly matched with their corresponding controls.
8. Non-Exhaustive Decisions: There are no non-exhaustive decisions in the logic of the code.

The code contains no errors in Category E. It adheres to correct logic regarding control flow, with no critical issues identified.

Category F: Interface Errors

1. Parameter and Argument Matching: The number of parameters received by each module matches the number of arguments sent, and the order is correct.
2. Attribute Matching: The attributes (data type and size) of each parameter align with the corresponding arguments.

3. Units System Consistency: The units system for each parameter corresponds with that of each argument, avoiding mismatches.
4. Arguments to Other Modules: The number of arguments transmitted to another module equals the number of parameters expected by that module.
5. Attribute Consistency for Transmitted Arguments: The attributes of each argument transmitted match those of the corresponding parameters in the receiving module.
6. Units System Consistency for Transmitted Arguments: The units system for transmitted arguments aligns with that of the corresponding parameters in the receiving module.
7. Built-in Function Calls: The number, attributes, and order of arguments in built-in function calls are correct.
8. Input Parameter Alteration: No subroutines alter parameters intended as input values.
9. Global Variables Consistency: Global variables have consistent definitions and attributes across all referencing modules.

The code adheres to correct interface practices, with no major interface errors identified. The function calls are well-structured, and the data types appear consistent throughout the code. Additional context on data structures and external modules would enhance the assessment.

Category G: Input/Output Errors

1. File Attributes: The attributes for files like "finalSea.jpg" and "shuff.jpg" are not explicitly declared in the code. However, their usage indicates they are image files.

2. OPEN Statement Attributes: The Image.open() method is used correctly for reading image files, with appropriate attributes for opening in read mode.
3. Memory Availability: There is no explicit check for sufficient memory availability before reading files.
4. File Opening: All files appear to be opened before use, as images are opened with Image.open() right before processing or displaying.
5. File Closing: The code does not explicitly close the image files after use.
6. End-of-File Handling: There is no explicit handling for end-of-file conditions.
7. I/O Error Handling: The code lacks error handling for I/O operations. If a file fails to open (e.g., due to an incorrect path), the program would raise an exception and terminate.
8. Spelling/Grammatical Errors: There are no apparent spelling or grammatical errors in the text printed or displayed by the program.

Category H: Other Checks

1. Unused Variables: The code does not contain variables that are never referenced or are referenced only once.
2. Variable Attributes: No unexpected default attributes have been found for the variables based on the attribute listing.
3. Compiler Warnings/Informational Messages: The program compiled successfully without producing any warning or informational messages that would indicate questionable validity or impede optimization.
4. Robustness: The program lacks checks for input validity, which affects its robustness.

5. Missing Functions: No specific functions have been identified as missing from the program.

Overall, the code shows no critical errors in this category, although improving input validation could enhance its robustness.

Total Errors Identified: The code has approximately 51 potential errors categorized throughout the program. These are termed potential errors because the code can execute even if the variables involved in the flagged lines remain unchanged.

Preferred Inspection Approach: I find utilizing a debugger within the IDE to pinpoint errors and closely examine specific code segments to be the most effective method.

Undetected Errors from Inspection Alone: Solely depending on program inspection may result in missing:

- Performance Bottlenecks: Inefficiencies in memory use or slow performance may not become apparent until the code is run.
- Integration Issues: The way this code interacts with other components of the application cannot be fully evaluated through inspection.
- Concurrency Issues: Potential race conditions or threading problems arising from concurrent event handling would not be caught during code inspection.

Importance of Program Inspection: Engaging in program inspection is definitely valuable for several reasons:

- Early Detection of Logical Errors: It allows for the identification of problems such as inconsistent return values or improper edge case handling, thereby preventing larger complications in the future.
- Improvement of Code Quality: Inspections enhance the code's maintainability, readability, and compliance with best practices.
- Encouragement of Collaboration: Peer review fosters valuable feedback and shared insights that can help in spotting errors that an individual might miss.
- Cost-Effectiveness: Identifying errors early through inspection minimizes the time and resources needed for debugging later on.

2. Code Debugging

a) Armstrong Number

The program contains two errors. The first error is an incorrect computation of the remainder, where the code uses remainder = num / 10, calculating the quotient instead of the last digit; it should use remainder = num % 10. The second error is in the update of the num variable; the code incorrectly sets num = num % 10, which retains only the remainder instead of removing the last digit. To address these issues, two breakpoints are needed: one at the line calculating the remainder to check its value, and another at

the line updating num to observe how it changes after each iteration.

Corrected Code:

```
class Armstrong {
    public static void main(String args[]) {
        int num = Integer.parseInt(args[0]);
        int n = num, check = 0, remainder;

        while (num > 0) {
            remainder = num % 10;
            check += Math.pow(remainder, 3);
            num /= 10;
        }

        if (check == n) {
            System.out.println(n + " is an Armstrong
Number");
        } else {
            System.out.println(n + " is not an Armstrong
Number");
        }
    }
}
```

b) GCD and LCD

The program includes two critical breakpoints for error tracking. The first breakpoint is set at the line int option1 = opt[n++][w] to monitor the value of n and ensure it isn't altered prematurely. The second breakpoint is at int option2 = profit[n-2] + opt[n-1][w-weight[n]], where it checks that the correct profit value for the current item is being utilized.

To correct the identified errors, the following steps were taken: First, the index access for option1 was updated from int option1 = opt[n++][w] to int option1 = opt[n - 1][w]. Second, the condition for checking the item's weight was modified from if (weight[n] > w) to if (weight[n] <= w). Lastly, the profit calculation for including item n was corrected by changing option2 = profit[n - 2] + opt[n - 1][w - weight[n]] to option2 = profit[n] + opt[n - 1][w - weight[n]].

Corrected Code:

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class GCD_LCM {
    static int gcd(int x, int y) {
        while (y != 0) {
            int temp = y;
            y = x % y;
            x = temp;
        }
        return x;
    }

    static int lcm(int x, int y) {
        return (x * y) / gcd(x, y);
    }

    public static void main(String args[]) {
        Scanner input = new Scanner(System.in);
        int x = 0, y = 0;
        boolean validInput = false;
        while (!validInput) {
            try {
                System.out.print("Enter the first number:
");
```

```
                x = input.nextInt();
                System.out.print("Enter the second number:
");
                y = input.nextInt();
                validInput = true; // Set to true if inputs
are valid
            } catch (InputMismatchException e) {
                System.out.println("Invalid input. Please
enter integers only.");
                input.nextLine(); // Clear the invalid input
            }
        }

        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();
    }
}
```

c) Knapsack Number

Two main issues were identified. Firstly, there was an error related to the incrementing of n in the loop, which could potentially lead to incorrect indexing when accessing the profit and weight arrays. Secondly, the arrays for profit and weight were declared but not initialized with any values, which could result in runtime exceptions or incorrect calculations.

To facilitate debugging, two breakpoints were suggested. The first breakpoint should be set at the line where int option1 = opt[n - 1][w]; is declared to monitor the value of n and ensure it remains within the bounds of the array. The second breakpoint was recommended at the line calculating option2

to verify the values of weight[n], profit[n], and w before performing any calculations.

To correct these issues, it was proposed to initialize the profit and weight arrays with meaningful values prior to their usage in the loops. Additionally, modifications to the loop conditions were suggested to ensure that n does not exceed the length of the arrays. Input handling was also emphasized to read weights and profits, whether through command-line arguments or user input.

Corrected Code:

```
import java.util.Scanner;
public class Knapsack {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter number of items (N): ");
        int N = scanner.nextInt();
        System.out.print("Enter maximum weight (W): ");
        int W = scanner.nextInt();

        int[] profit = new int[N + 1];
        int[] weight = new int[N + 1];
        int[][] opt = new int[N + 1][W + 1];
        boolean[][] sol = new boolean[N + 1][W + 1];

        // Input profits and weights
        for (int i = 1; i <= N; i++) {
            System.out.print("Enter profit for item " + i +
": ");
            profit[i] = scanner.nextInt();
            System.out.print("Enter weight for item " + i +
": ");
            weight[i] = scanner.nextInt();
        }
```

```
        // Knapsack calculation
        for (int n = 1; n <= N; n++) {
            for (int w = 1; w <= W; w++) {
                int option1 = opt[n - 1][w];
                int option2 = (weight[n] <= w) ? profit[n] +
opt[n - 1][w - weight[n]] : Integer.MIN_VALUE;
                opt[n][w] = Math.max(option1, option2);
                sol[n][w] = (option2 > option1);
            }
        }

        // Output the maximum profit
        System.out.println("Maximum profit: " + opt[N][W]);
        scanner.close();
    }
}
```

d) Magic Number

Two significant errors were identified. The first error involved an incorrect condition in the inner while loop, which could lead to improper calculations. The second error was the absence of a semicolon in one of the expressions, which would cause a compilation issue.

To assist in debugging, it was suggested to set breakpoints at the beginning of the inner while loop. This would allow for monitoring the values of relevant variables and understanding how they change throughout the execution of the loop.

To correct the identified issues, the code needed to be updated. The condition in the inner loop was fixed to ensure that it properly iterates through the digits of the number. Additionally, the missing semicolon was added to the line s = s * (sum / 10); to prevent compilation errors.

Corrected Code:

```
import java.util.Scanner;
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 num = n;
        while (num > 9) {
            int sum = 0;
            while (num > 0) {
                sum += num % 10; // Correctly accumulate the
last digit
                num /= 10; // Move to the next digit
            }
            num = sum; // Update num to the new sum of
digits
        }
        if (num == 1) {
            System.out.println(n + " is a Magic Number.");
        } else {
            System.out.println(n + " is not a Magic
Number.");
        }
        ob.close(); // Close the scanner
    }
}
```

e) Merge Sort

The program contains five identified errors related to array handling and method calls. The first error involves

incorrectly trying to get the left and right halves of the array using expressions like array + 1 and array - 1, which are invalid. Instead, the entire array should be passed to the leftHalf and rightHalf methods directly.

The second error pertains to the leftHalf method, which should correctly specify the starting index as 0 in the mergeSort method when retrieving the left half. The third error is in the merge method call, which incorrectly uses merge(array, left++, right--); it should simply be merge(array, left, right) without the increment and decrement operators.

The fourth error is that the mergeSort method does not initialize the result array to the same size as the input array before merging, which is necessary for storing merged results. The fifth error concerns the loop condition in the merge method, which should ensure the correct lengths for the left and right arrays to avoid index out-of-bounds errors.

To address these errors, five breakpoints are suggested. The first breakpoint is at the line int[] left = leftHalf(array + 1); to verify the correct array reference is being passed. The second is placed at the line where the merge function is called to check the parameters being passed. The third breakpoint is in the merge function to observe the merging process and confirm proper index handling. The fourth is at the start of the mergeSort method to ensure the result array is initialized correctly. Finally, the fifth breakpoint is at the end of the merge method to verify that the merged result is correct.

Corrected Code:

```
import java.util.Arrays; // Import the Arrays class to use
Arrays.toString()
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];
        System.arraycopy(array, 0, left, 0, size1);
        return left;
    }

    public static int[] rightHalf(int[] array) {
        int size1 = array.length / 2;
        int size2 = array.length - size1;
        int[] right = new int[size2];
        System.arraycopy(array, size1, right, 0, size2);
        return right;
    }
```

```
    public static void merge(int[] result, int[] left, int[]
right) {
        int i1 = 0, 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++;
            }
        }
    }
}
```

f) Multiply Matrices

The program contains four identified errors. The first error pertains to the incorrect indexing of the first matrix, where the variable first[c-1][c-k] should be corrected to first[c][k] to properly access the current row and the k-th column. The second error involves the incorrect access of the second matrix, where second[k-1][k-d] should be changed to second[k][d] for accurate indexing. The third error lies in the input prompt for the second matrix, which mistakenly asks for "the number of rows and columns of the first matrix" instead of the correct phrase for the second matrix. Lastly, the fourth error is related to the logic of the multiplication process; the sum variable needs to be reset at the beginning of the outer loop that computes the dot product for each element in the resulting matrix.

To address these errors, four breakpoints are recommended. The first breakpoint is placed at the line where sum = sum + first[c-1][c-k] * second[k-1][k-d]; to verify the indices used for the first matrix. The second breakpoint is also at the same line to check the indices for the second matrix. The third breakpoint should be set at the prompt asking for the second matrix dimensions to ensure the input message is correct. Finally, the fourth breakpoint should be positioned at the start of the innermost loop where the sum is computed, to confirm that the sum variable is reset properly for each element of the resulting matrix.

Corrected Code:

```
import java.util.Scanner;

class MatrixMultiplication {
    public static void main(String args[]) {
        int m, n, p, q, sum, 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.");
        } 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; // Reset sum for each element
in the result matrix
                    for (k = 0; k < n; k++) { // Use n for
the inner loop
                        sum += first[c][k] * second[k][d];
                    }
                    multiply[c][d] = sum; // Store the
computed sum in the result matrix
                }
            }
```

```
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");
            }
        }

        in.close(); // Close the scanner to prevent resource
leaks
    }
}
```

g) Quadratic Program

First, in the insert method, there's a syntax error with the expression I + = (I + h / h--) % maxSize, which should be corrected to I += (h * h) % maxSize for proper quadratic probing. In the get method, the line I = (I + h * h++) % maxSize increments I incorrectly because of the post-increment of h. It should instead be I = (I + h * h) % maxSize, followed by an increment of h. The same issue occurs in the remove method, which also needs this adjustment. Additionally, in the remove method, the statement keys[i] = vals[i] = null should be split into two separate assignments for clarity, changing it to keys[i] = null; vals[i] = null;. Lastly, a typo in the comment /** maxSizeake object of QuadraticProbingHashTable **/ should be corrected to /** Make object of QuadraticProbingHashTable **/.

To address these errors, five breakpoints are needed. The first breakpoint will be at the syntax error in the insert method to correct it. The second and third breakpoints will be in the get and remove methods, respectively, to fix the increment logic for I. The fourth breakpoint will be placed

at the line in the remove method where the assignments occur, to separate them for clarity. Finally, the fifth breakpoint will help correct the comment typo.

To fix the identified errors, the following steps will be taken:

1. Change the incorrect syntax in the insert method.
2. Update the index calculation in both the get and remove methods to correct the increment of h.
3. Replace the combined assignment in the remove method with separate statements for clarity.
4. Correct the typo in the comment to properly describe the purpose.

Corrected Code:

```
import java.util.Scanner;

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

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

    public void insert(String key, String val) {
        int tmp = hash(key), 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;  // Corrected
increment logic
            h++;  // Increment h after using it
        } 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;  // Corrected
increment logic
            h++;  // Increment h after using it
        }
        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;  // Corrected
increment logic
```

```
            h++;  // Increment h after using it
        }
        keys[i] = null;  // Separate assignments for clarity
        vals[i] = null;
    }

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

    private int hash(String key) {
        return Math.abs(key.hashCode()) % maxSize;  //
Ensured non-negative index
    }
}

public class HashTableTest {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter the size of the hash table:
");
        QuadraticProbingHashTable hashTable = new
QuadraticProbingHashTable(scan.nextInt());
        hashTable.insert("key1", "value1");
        System.out.println("Value: " +
hashTable.get("key1"));
        scan.close();  // Close the scanner
    }
}
```

h) Sorting Array

In the program, two errors were identified. The first error is in the outer loop condition, which incorrectly uses `for (int i = 0; i >= n; i++)`. This should be corrected to `for (int i = 0; i < n; i++)` to ensure that the loop properly iterates over the array elements. The second error lies in the inner loop's comparison logic; it currently uses `if (a[i] <= a[j])`, which sorts the array in descending order. To sort in ascending order, this condition should be changed to `if (a[i] > a[j])`.

To address these errors, two breakpoints are needed. The first breakpoint should be placed at the line checking the outer loop condition to confirm that it iterates through all elements correctly. The second breakpoint should be at the comparison line to ensure that the sorting logic correctly compares elements for ascending order.

Corrected Code:

```
import java.util.Arrays; // Import the Arrays class
import java.util.Scanner;

public class AscendingOrder {
    public static void main(String[] args) {
        int n, temp;
        Scanner s = new Scanner(System.in);
        System.out.print("Enter the number of elements: ");
        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.println("Sorted Array: " +
Arrays.toString(a));
    }
}
```

i) Stack Implementation

The program has four main errors that affect its functionality.

First, the push method incorrectly decrements the top index with top--;, which should instead be top++; since the index should increase when a value is pushed onto the stack. Second, the pop method incorrectly increments the top index with top++; when it should decrement it using top--; as the index should decrease when a value is popped from the stack. Third, the loop condition in the display method is incorrectly set to for(int i=0;i>top;i++), which should be changed to for(int i=0;i<=top;i++) to ensure it iterates over all stack elements from index 0 to top. Lastly, there is a need to ensure that the stack is properly initialized to store values upon pushing.

To fix these errors, four breakpoints are necessary: one at the line stack[top] = value; in the push method to verify that top is within bounds; one at top--; in the pop method to confirm the correct decrement of top; one at the loop condition in the display method to ensure it iterates

through the correct indices; and one at the beginning of the push method to verify that top is initialized correctly before adding elements.

The steps taken to correct these errors include updating the push method to increment top, modifying the pop method to decrement top, correcting the loop condition in the display method, and ensuring that stack operations manage the top variable effectively to avoid exceeding the bounds of the stack array.

Corrected Code:

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

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

public void push(int value) {
if (top == stack.length - 1) {
System.out.println("Stack full");
} else {
stack[++top] = value;
}
}

public void pop() {
if (top == -1) {
System.out.println("Stack empty");
} else {
```

```
top--;
}
}

public void display() {
for (int i = 0; i <= top; i++) {
System.out.print(stack[i] + " ");
}
System.out.println();
}
}
```

j) Tower of Hanoi

The program has three errors related to its recursive calls in the Tower of Hanoi implementation. The first error involves incorrect expressions in the second call to doTowers, where topN++ and inter-- modify the values instead of passing the originals. This should be corrected to use topN - 1, while keeping inter unchanged. The second error arises from incorrectly passing parameters to the recursive call, as characters cannot be modified like integers; the same characters should be passed. The third error is in the first call to doTowers in the else block, which incorrectly modifies variables; it should pass the from, inter, and to arguments properly.

To address these errors, two breakpoints are needed: the first at the first recursive call to doTowers to ensure the correct parameters are passed, and the second at the second recursive call to confirm that the parameters are not modified incorrectly.

The steps taken to fix the errors include changing the second call to doTowers to pass topN - 1, ensuring that the correct character parameters are maintained for the source,

intermediate, and destination rods, and verifying the program's flow to ensure that it produces the expected Tower of Hanoi solution.

Corrected Code:

```
public class TowerOfHanoi {
    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);
        }
    }
}
```

3. Static Analysis Tools

```
C:\Users\Admin>pip install pylint
Collecting pylint
  Downloading pylint-3.3.1-py3-none-any.whl (521 kB)
     ---------------------------------------- 521.8/521.8 kB 2.5 MB/s eta 0:00:00
Collecting platformdirs>=2.2.0
  Downloading platformdirs-4.3.6-py3-none-any.whl (18 kB)
Collecting astroid<=3.4.0-dev0,>=3.3.4
  Downloading astroid-3.3.5-py3-none-any.whl (274 kB)
     ---------------------------------------- 274.6/274.6 kB 2.4 MB/s eta 0:00:00
Collecting isort!=5.13.0,<6,>=4.2.5
  Downloading isort-5.13.2-py3-none-any.whl (92 kB)
     ---------------------------------------- 92.3/92.3 kB 2.6 MB/s eta 0:00:00
Collecting mccabe<0.8,>=0.6
  Downloading mccabe-0.7.0-py2.py3-none-any.whl (7.3 kB)
Collecting tomlkit>=0.10.1
  Downloading tomlkit-0.13.2-py3-none-any.whl (37 kB)
Collecting dill>=0.3.6
  Downloading dill-0.3.9-py3-none-any.whl (119 kB)
     ---------------------------------------- 119.4/119.4 kB 3.5 MB/s eta 0:00:00
Collecting colorama>=0.4.5
  Downloading colorama-0.4.6-py2.py3-none-any.whl (25 kB)
Installing collected packages: tomlkit, platformdirs, mccabe, isort, dill, colorama, astroid, pylint
Successfully installed astroid-3.3.5 colorama-0.4.6 dill-0.3.9 isort-5.13.2 mccabe-0.7.0 platformdirs-4.3.6 pylint-3.3.1 tomlkit-0.13.2

[notice] A new release of pip available: 22.3.1 -> 24.2
[notice] To update, run: python.exe -m pip install --upgrade pip
```

```
C:\Users\Admin>pylint C:\Users\Admin\Documents\Program.py --errors-only
************ Module Program
Documents\Program.py:37:0: E0401: Unable to import 'PIL' (import-error)
Documents\Program.py:38:0: E0401: Unable to import 'PIL' (import-error)
Documents\Program.py:39:0: E0401: Unable to import 'PIL' (import-error)
Documents\Program.py:1077:0: E0102: function already defined line 749 (function-redefined)
Documents\Program.py:1244:0: E0401: Unable to import 'PIL' (import-error)
Documents\Program.py:1375:17: E0602: Undefined variable 'img' (undefined-variable)
Documents\Program.py:1381:21: E0602: Undefined variable 'colors' (undefined-variable)
Documents\Program.py:1547:4: E0602: Undefined variable 'data' (undefined-variable)
Documents\Program.py:1561:4: E0602: Undefined variable 'data' (undefined-variable)
Documents\Program.py:1566:0: E0102: function already defined line 1241 (function-redefined)
Documents\Program.py:1610:63: E0602: Undefined variable 'Imag' (undefined-variable)
```