IT314: Software Engineering

Lab 7

Program Inspection, Debugging and Static Analysis

Student ID: **202201207**
Name: **Swayam Hingu**

Program Inspection

Frag-1: Armstrong Number

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

- **Logical Error in extracting digits:**
  - In the `while` loop, the operation `remainder = num / 10;` is incorrect for extracting the last digit. It should be `remainder = num % 10;`, as using division results in incorrect digits.
  - Similarly, `num = num % 10;` should be `num = num / 10;` to remove the last digit.
- **Incorrect result calculation:**
  - As a result of the above incorrect logic, the program does not calculate the sum of the cubes of the digits properly.
- **Argument Error:**
  - If the user does not provide any argument via `args[]`, the program will throw an `ArrayIndexOutOfBoundsException`. There should be a check to ensure that the user has provided input.
- **Typographical Issue:**
  - The output text should read "is not **an** Armstrong Number" instead of "is not **a** Armstrong Number."

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

- **Category C: Computation Errors** is most relevant here, as the main issue lies in how the digits are extracted and the Armstrong number computation is performed.
- **Category E: Control-Flow Errors** is also useful in detecting possible exceptions due to missing argument validation.

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

- **Performance issues** or potential optimizations are not identified through this inspection.
- **Runtime exceptions like `NumberFormatException`** due to invalid input (if the argument is not a number) are not explicitly covered by this inspection checklist.

4. Is the program inspection technique worth applying?

Yes, program inspection is valuable for detecting logical and computation errors, especially in small programs. In this case, inspection caught critical issues in digit extraction and flow control, preventing incorrect calculations and exceptions.

Frag-2: GCD and LCM

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

- **Logical Error in GCD Calculation:**
  - In the `gcd` method, the condition `while(a % b == 0)` is incorrect. As noted in the comment, it should be `while(a % b != 0)` to avoid an infinite loop and properly compute the GCD.
- **Logical Error in LCM Calculation:**
  - In the `lcm` method, the condition `if(a % x != 0 && a % y != 0)` is incorrect. The correct condition should be `if(a % x == 0 && a % y == 0)` to find the least common multiple. The current condition will return incorrect results.
- **Potential Infinite Loop in LCM Calculation:**
  - As a result of the incorrect condition, the program might enter an infinite loop when calculating the LCM. It will never find the correct multiple and continue incrementing `a` indefinitely.

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

- **Category C: Computation Errors** is the most effective here, as the core issue is in the logic of computing the GCD and LCM. Both methods contain logical errors in their conditions.

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

- **Performance issues** related to the efficiency of the `lcm` algorithm are not covered here. The method of incrementing `a` by 1 each time could be optimized, but this is not detected by the inspection.
- **Input validation** for negative or zero inputs is also not checked in this program, but such errors would cause unexpected behavior or crashes during execution.

4. Is the program inspection technique worth applying?

Yes, this program inspection effectively identifies the key logical errors in both the GCD and LCM methods, which would lead to incorrect results or infinite loops. It helps in preventing bugs related to basic mathematical operations.

Frag-3: Knapsack

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

- **Error in Array Indexing (Logical Error):**
  - In the line `int option1 = opt[n++][w];`, the `n++` increments `n` after retrieving `opt[n][w]`, which is incorrect. It should be `opt[n-1][w]` to avoid skipping the current item. The post-increment operator here leads to accessing the wrong row in the `opt` array, which causes the wrong decision-making during the solution process.
- **Logical Error in `option2` Calculation:**
  - The line `if (weight[n] > w)` should be `if (weight[n] <= w)` because we only consider taking the item if its weight is less than or equal to the current capacity `w`. The current condition leads to skipping items that could fit in the knapsack.
  - Additionally, `option2 = profit[n-2] + opt[n-1][w-weight[n]]` is incorrect. The `profit` array indexing should be `profit[n]`, not `profit[n-2]`, because we are currently considering the `n`th item. The wrong indexing would give incorrect results for profits.

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

- **Category C: Computation Errors** is the most effective here, as the primary issue stems from incorrect array indexing and logical flow when calculating the optimal solution. The algorithm logic needs to be corrected to ensure the right items are selected based on weight and profit.

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

- **Edge case handling** is not checked. For example, the program assumes valid inputs without checking for cases where `N` or `W` might be zero or negative. Additionally, no checks are in place to validate if enough arguments are passed via `args[]`. These are runtime issues not caught by static inspection.

- **Performance optimization** is also not addressed. While the algorithm works for moderate inputs, it may not be efficient for large inputs, but such performance bottlenecks are beyond static inspection.

4. Is the program inspection technique worth applying?

Yes, the inspection reveals critical logical errors related to the core knapsack algorithm, which would have otherwise resulted in incorrect results. The array indexing and condition corrections are essential to fixing the behavior of the program.

Frag-4: Magic number

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

- **Error in Inner While Loop Condition:**
  - The inner while loop uses `while(sum == 0)` which is incorrect. The logic here should be to break down `sum` into its digits. It should be `while(sum != 0)` to correctly compute the sum of the digits.
- **Error in Digit Multiplication and Summing Logic:**
  - The line `s = s * (sum / 10);` is incorrect. This line is intended to accumulate the sum of the digits, but it's performing multiplication and integer division, which doesn't compute the digit sum. The correct logic should be to add the last digit of `sum` to `s`. It should be `s = s + (sum % 10);` instead of multiplying the values.
- **Missing Semicolon in Inner Loop:**
  - The line `sum = sum % 10` is missing a semicolon at the end. It should be `sum = sum % 10;`.

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

- **Category C: Computation Errors** would be the most effective in this case, as the primary issue arises from incorrect logic and conditions within the loops that perform the digit summing. The errors are logical and computational.

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

- **Edge case handling** for inputs like negative numbers or zero is not addressed. The current inspection does not validate whether the code checks for these inputs, which may cause issues if the user enters such values.

- **Performance** is also not checked in terms of how efficiently the code runs for larger numbers, though this is unlikely to be a major concern given the simplicity of the operation.

4. Is the program inspection technique worth applying?

Yes, because program inspection reveals critical logical errors in the core computation for determining whether the number is a magic number. Without these corrections, the program would not produce the expected output for valid inputs.

Frag-5: Merge sort

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

- **Error in the Recursive Splitting of the Array:**
  - The expressions `array+1` and `array-1` in the `mergeSort` method are incorrect. They attempt to manipulate the array reference directly, which is not valid. These need to call `leftHalf(array)` and `rightHalf(array)` correctly without modifying the array reference.
- **Incorrect Use of Increment and Decrement Operators in `merge`:**
  - The lines `merge(array, left++, right--);` are incorrect. `left++` and `right--` are invalid when passing arrays and will cause issues because arrays are not primitive values and cannot be incremented or decremented this way. The correct call should be `merge(array, left, right);`.

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

- **Category B: Data and Control Flow Errors** is the most effective in this case, as the program contains issues related to the manipulation of array references, which impacts the control flow of recursive function calls. Properly checking for valid data manipulation and ensuring the program flows correctly is crucial.

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

- **Performance optimization** of the merge sort algorithm, though merge sort is efficient (O(n log n)), is not addressed. Program inspection wouldn't necessarily reveal this, but it can be considered for larger datasets.

4. Is the program inspection technique worth applying?

Yes, because it helps in identifying fundamental errors in array handling and recursion logic, which are critical for the correct functioning of the merge sort algorithm.

Frag-6: Multiply matrics

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

- **Incorrect Indices in the Matrix Multiplication Loop:**
  - In the innermost loop of matrix multiplication, the expression `first[c-1][c-k]` and `second[k-1][k-d]` are incorrect. The indices are being decremented incorrectly, which would result in an `ArrayIndexOutOfBoundsException`. The correct expressions should be `first[c][k]` and `second[k][d]` to access the correct elements of the matrices.
- **Input Prompt for the Second Matrix:**
  - The prompt after the first matrix is asking to enter the "number of rows and columns of **first** matrix" again. This is a typo; it should say "second matrix."

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

- **Category A: Algorithmic Errors** is the most effective here, as the error pertains to the logic of matrix multiplication. Correct index calculations are crucial for obtaining the right product of matrices.

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

- **Input validation** beyond the current scenario. For example, the program assumes the input matrices are valid and does not check for non-numeric inputs, which could cause the program to crash. Program inspection does not always account for robustness against user input errors.

4. Is the program inspection technique worth applying?

Yes, program inspection helps in detecting logical issues, such as index miscalculations, that could cause incorrect results or runtime errors, as seen in the matrix multiplication logic.

Frag-7: Quardatic Probing

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

- **Operator Error in Insert Method:**
  - The line `i + = (i + h / h--) % maxSize;` is incorrect. The correct syntax should be `i = (i + h * h++) % maxSize;`. The `+=` operator should not have a space, and `/ h--` seems logically incorrect for quadratic probing, which should use `h * h`.
- **Incorrect Rehashing Logic in Remove Method:**
  - The rehashing logic after a removal contains an extra `currentSize--` decrement in the `remove` function. This is incorrect because `currentSize` is already decremented when a key is removed and should not be decremented again.
- **Unused Print Statement:**
  - The print statement `System.out.println("i "+ i);` inside the `get` method seems like a debugging statement. It can be removed or handled based on the actual use case.

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

- **Category A: Algorithmic Errors** is the most effective here, as the primary issues relate to incorrect implementation of the quadratic probing algorithm, which directly affects the behavior of the hash table during insertion and removal.

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

- **Performance bottlenecks** during high load factor scenarios are not easily identified through static inspection. The program may degrade in performance as the load factor increases, but this can only be identified through testing and profiling, not inspection.

4. Is the program inspection technique worth applying?

Yes, inspection is helpful for identifying logical errors, such as the incorrect probing method or decrementing the current size incorrectly. It prevents subtle issues in fundamental data structure implementations.

Frag-8: Sorting array

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

- **Syntax Error in Class Name:**
  - The class name `Ascending _Order` has an invalid space between `Ascending` and `_Order`. Java class names cannot contain spaces. It should be `AscendingOrder` (without spaces or underscores).
- **Logical Error in the First Loop:**
  - The loop `for (int i = 0; i >= n; i++);` is incorrect. The condition `i >= n` is invalid, as it will always be false. It should be `i < n` to iterate through the array. Also, there's an unnecessary semicolon (`;`) at the end of the `for` loop, which causes the loop body to be skipped.
- **Comparison Logic Error:**
  - The condition `if (a[i] <= a[j])` is wrong for sorting in ascending order. It should be `if (a[i] > a[j])` to swap the elements if the current element is greater than the next one.

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

- **Category B: Logical Errors** would be more effective here. The incorrect loop condition and comparison logic would be easily detected by inspecting how the loops are functioning and identifying logical issues in the sorting mechanism.

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

- **Runtime performance issues** or inefficiencies with large data sets would not be identifiable through static program inspection. Although the logic can be corrected, inspection cannot fully predict how well the algorithm performs for larger arrays.

4. Is the program inspection technique worth applying?

Yes, program inspection helps catch basic syntax and logic errors, like improper loop conditions and comparison mistakes, ensuring that the sorting logic is correct and functions as intended. It is an essential step before testing with real data.

Frag-9: Stack Implementation

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

- **Logic Error in the `push` Method:**
  - In the `push()` method, the line `top--` is incorrect. It should increment `top` to add a new element at the next available position in the stack. The line should be `top++` instead.
- **Logic Error in the `display` Method:**
  - The loop condition in `display()` is incorrect. It uses `i > top`, which will result in the loop never running. The loop should iterate from the top to 0 (`i <= top`) to display all elements in the stack.
- **Logic Error in the `pop` Method:**
  - In the `pop()` method, `top++` is used to remove the top element. This should actually be `top--` to decrease the top pointer and effectively remove the element from the stack.

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

- **Category B: Logical Errors** would be more effective, as the primary issue in this code is the incorrect manipulation of the `top` pointer in both the `push` and `pop` methods, and the loop condition in `display`.

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

- **Memory management issues** and **runtime errors** due to stack overflow or array index out of bounds would not be fully identifiable during program inspection without proper testing. Also, there could be issues if tested with stack operations that exceed its size.

4. Is the program inspection technique worth applying?

Yes, program inspection is essential in catching fundamental logic errors that may prevent the stack from functioning correctly. The incorrect use of the `top` variable would result in improper stack operations, which could be easily identified during inspection.

Frag-10: Tower of Hanoi

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

- **Logic Error in Recursive Calls:**
  - In the second recursive call `doTowers(topN ++, inter--, from+1, to+1)`, the use of `++` and `--` operators on `topN` and `inter` is incorrect. These should not be used here. Instead, the parameters should be passed as they are without incrementing or decrementing.
  - The correct recursive call should be `doTowers(topN - 1, inter, from, to)` for the second half of the process.
- **Incorrect Use of Parameters:**
  - The parameters `from + 1` and `to + 1` do not make sense in the context of character parameters. They should be kept as characters (`from`, `to`) and not modified.

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

- **Category B: Logical Errors** would be the most effective here since the primary issues stem from the incorrect logic in the recursive calls and the manipulation of parameters.

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

- **Infinite Recursion** could be a potential issue if the termination condition is not set properly or is mistakenly altered. This might not be caught until runtime, especially in recursive functions.

4. Is the program inspection technique worth applying?

Yes, program inspection is vital in this case as it helps identify logical errors that can lead to incorrect functionality or infinite loops in recursive methods. Ensuring the correct flow of logic in recursive functions is crucial for their proper execution.

CODE DEBUGGING

Frag-1: Armstrong Number

1. Errors Identified:

- **Logic Error**: The calculation of the Armstrong number is incorrect. The `remainder` is incorrectly calculated using `num / 10` instead of using `num % 10` to get the last digit.
- **Incorrect Loop Logic**: The `while` loop condition is correct, but the `check` variable's update is wrong, leading to incorrect results.

2. Breakpoints Needed:

- Set a breakpoint at the `while(num > 0)` line to observe how `num` and `check` change.

3. Steps Taken to Fix Errors:

- Change the logic to correctly extract the last digit and update the `check` variable appropriately.

Fixed Code:

```
// 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; // Correctly get the last digit
            check = check + (int) Math.pow(remainder, 3); // Use remainder
            num = num / 10; // Correctly reduce num
        }
        if (check == n)
            System.out.println(n + " is an Armstrong Number");
        else
            System.out.println(n + " is not an Armstrong Number");
    }
}

// Input: 153
// Output: 153 is an Armstrong Number.
```

Frag-2: GCD and LCM Calculation

1. Errors Identified:

- **Logic Error**: The while condition in the `gcd` method should be `while(a % b != 0)` instead of `while(a % b == 0)`.
- **Wrong LCM Logic**: The logic for calculating LCM needs to be revised.

2. Breakpoints Needed:

- Set breakpoints at both the `gcd` and `lcm` methods to inspect the values of `x`, `y`, `a`, and `b`.

3. Steps Taken to Fix Errors:

- Update the `gcd` logic condition and revise the LCM logic accordingly.

Fixed Code:

```java
// 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) ? x : y; // a is the greater number
        b = (x < y) ? x : y; // b is the smaller number

        r = b;
        while (a % b != 0) { // Corrected condition
            r = a % b;
            a = b;
            b = r;
        }
        return r;
    }

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

    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 LCM of two numbers is 20
```

Frag-3: Knapsack Problem

1. Errors Identified:

- **Logic Error**: In the `merge` function, the use of `n++` in the loop increments `n` which causes the logic to break.
- **Incorrect Accessing of Arrays**: There are logical errors in how the profit and weight are accessed.

2. Breakpoints Needed:

- Set breakpoints inside the nested loops to monitor `opt`, `sol`, and check the values of `profit` and `weight`.

3. Steps Taken to Fix Errors:

- Correct the logic to prevent modifying loop control variables unintentionally.

Fixed Code:

```
// 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
        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 - 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];
            }
        }

        // 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
```

Frag-4: Magic Number Check

1. Errors Identified:

- **Logic Error**: The condition `while(sum == 0)` is incorrect and should be `while(sum > 0)`.
- **Incorrect Logic for Summing Digits**: The line `s = s * (sum / 10)` is wrong; it should be `s += sum % 10`.

2. Breakpoints Needed:

- Set a breakpoint at the `while(num > 9)` to see how `num` and `sum` change.

3. Steps Taken to Fix Errors:

- Correct the while loop condition and the sum calculation.

Fixed Code:

```
// 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 num = n;
        while (num > 9) {
            int sum = num;
            int s = 0;
            while (sum > 0) { // Corrected condition
                s += sum % 10; // Correct summation logic
                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: 119
// Output: 119 is a Magic Number.
// Input: 199
// Output: 199 is not a Magic Number.
```


Frag-5: Merge Sort

1. Errors Identified:

- **Array Handling Error**: The `leftHalf` and `rightHalf` methods are incorrectly manipulating the array. You should pass the actual array instead of `array + 1` and `array - 1`.
- **Incorrect Logic in Merge**: The merge logic is incorrectly implemented, particularly the index handling.

2. Breakpoints Needed:

- Set breakpoints in the `mergeSort` and `merge` methods to check how arrays are split and merged.

3. Steps Taken to Fix Errors:

- Correct the methods for splitting the array and adjust the merge logic accordingly.

Fixed Code:

```java
// Merge Sort
import java.util.Arrays;

public class MergeSort {
    public static void main(String[] args) {
        int[] array = { 12, 11, 13, 5, 6, 7 };
        System.out.println("Given array");
        System.out.println(Arrays.toString(array));

        MergeSort ob = new MergeSort();
        ob.mergeSort(array, 0, array.length - 1);

        System.out.println("\nSorted array");
        System.out.println(Arrays.toString(array));
    }

    void mergeSort(int[] array, int left, int right) {
        if (left < right) {
            int middle = (left + right) / 2;
            mergeSort(array, left, middle);
            mergeSort(array, middle + 1, right);
            merge(array, left, middle, right);
        }
    }

    void merge(int[] array, int left, int middle, int right) {
        int n1 = middle - left + 1;
        int n2 = right - middle;

        int[] leftHalf = new int[n1];
        int[] rightHalf = new int[n2];

        for (int i = 0; i < n1; i++)
```

```
            leftHalf[i] = array[left + i];
        for (int j = 0; j < n2; j++)
            rightHalf[j] = array[middle + 1 + j];

        int i = 0, j = 0, k = left;
        while (i < n1 && j < n2) {
            if (leftHalf[i] <= rightHalf[j]) {
                array[k] = leftHalf[i];
                i++;
            } else {
                array[k] = rightHalf[j];
                j++;
            }
            k++;
        }
        while (i < n1) {
            array[k] = leftHalf[i];
            i++;
            k++;
        }
        while (j < n2) {
            array[k] = rightHalf[j];
            j++;
            k++;
        }
    }
}

// Output:
// Given array
// [12, 11, 13, 5, 6, 7]
//
// Sorted array
// [5, 6, 7, 11, 12, 13]
```


Frag-6: Matrix Multiplication

1. Errors Identified:

- **Array Index Out of Bounds**: The indexing in the multiplication logic is incorrect; it should be using `first[c][k]` and `second[k][d]` instead of `first[c - 1][c - k]` and `second[k - 1][k - d]`.
- **Resetting Sum**: The sum variable should be reset correctly at the beginning of the nested loop.

2. Breakpoints Needed:

- Set breakpoints inside the multiplication loop to check values of `sum`, `first`, and `second`.

3. Steps Taken to Fix Errors:

- Correct the indexing and the logic for sum accumulation.

Fixed Code:

```
// Java program to multiply two matrices
import java.util.Scanner;

class MatrixMultiplication {
    public static void main(String args[]) {
        int m, n, p, q, sum;
        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 (int c = 0; c < m; c++)
            for (int 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 (int c = 0; c < p; c++)
                for (int d = 0; d < q; d++)
                    second[c][d] = in.nextInt();

            for (int c = 0; c < m; c++) {
                for (int d = 0; d < q; d++) {
                    sum = 0; // Reset sum for each cell
                    for (int k = 0; k < n; k++) { // Use n as limit
                        sum += first[c][k] * second[k][d]; // Correct
indexing
                    }
                    multiply[c][d] = sum;
                }
            }

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

// Input:
// 2 2
// 1 2
// 3 4
// 2 2
// 1 0
// 1 0
// Output:
// Product of entered matrices:-
// 5    2
```

```
// 13   6
```

Frag-7: Quadratic Probing Hash Table

1. Errors Identified:

- **Increment/Decrement Logic**: The code incorrectly uses `i += (i + h / h--) % maxSize;` which has wrong logic and will cause an infinite loop.
- **Null Checks**: The `remove` function does not handle null keys correctly.

2. Breakpoints Needed:

- Set breakpoints in `insert`, `remove`, and `get` methods to observe key handling.

3. Steps Taken to Fix Errors:

- Correct the probing logic and ensure proper handling of keys in the `remove` function.

Fixed Code:

```
/** Class QuadraticProbingHashTable **/
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];
    }
```

```java
    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 + maxSize) % maxSize; // Ensure
non-negative
    }

    public void insert(String key, String val) {
        if (isFull()) {
            System.out.println("Hash table is full.");
            return;
        }
        int i = hash(key);
        int h = 1;
        while (keys[i] != null) {
            if (keys[i].equals(key)) {
                vals[i] = val; // Update value if key already exists
                return;
            }
            i = (i + h * h) % maxSize; // Quadratic probing
            h++;
        }
        keys[i] = key;
        vals[i] = val;
        currentSize++;
    }
```

```
    public String get(String key) {
        int i = hash(key);
        int h = 1;
        while (keys[i] != null) {
            if (keys[i].equals(key)) {
                return vals[i];
            }
            i = (i + h * h) % maxSize;
            h++;
        }
        return null;
    }

    public void remove(String key) {
        if (!contains(key)) return;

        int i = hash(key);
        int h = 1;
        while (!keys[i].equals(key)) {
            i = (i + h * h) % maxSize;
            h++;
        }

        keys[i] = null; // Remove the key
        vals[i] = null;

        // Rehash the following keys
        for (i = (i + h * h) % maxSize; keys[i] != null; i = (i + h * h) %
maxSize) {
            String tmp1 = keys[i];
            String tmp2 = vals[i];
            keys[i] = null;
            vals[i] = null;
            currentSize--;
            insert(tmp1, tmp2); // Reinsert
        }
        currentSize--;
    }
```

```
    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. get");
            System.out.println("3. remove");
            System.out.println("4. print hash table");
            System.out.println("5. exit");
            int choice = scan.nextInt();
            switch (choice) {
                case 1:
                    System.out.println("Enter key and value to insert");
                    qpht.insert(scan.next(), scan.next());
                    break;
                case 2:
                    System.out.println("Enter key to get value");
                    System.out.println("Value = " +
qpht.get(scan.next()));
                    break;
                case 3:
                    System.out.println("Enter key to remove");
```

```
                    qpht.remove(scan.next());
                    break;
                case 4:
                    qpht.printHashTable();
                    break;
                case 5:
                    System.out.println("Exiting...");
                    break;
                default:
                    System.out.println("Invalid option");
                    break;
            }
        } while (ch != '5');
        scan.close();
    }
}

// Sample Input:
// Hash Table Operations
// 1. insert
// 2. get
// 3. remove
// 4. print hash table
// 5. exit
```


Frag-8: Sorting an Array

1. Errors Identified:

- **Incorrect Loop Condition**: The loop `for (int i = 0; i >= n; i++)` should be `for (int i = 0; i < n; i++)`.
- **Sorting Logic**: The sorting logic in the nested loop is not correctly implemented for ascending order.

2. Breakpoints Needed:

- Set breakpoints in the sorting loop to check the values of `i`, `j`, and `temp`.

3. Steps Taken to Fix Errors:

- Correct the outer loop condition and ensure the sorting logic is applied correctly.

Fixed Code:

```
// Sorting the array in ascending order
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 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++) { // Corrected loop condition
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) { // Corrected comparison for ascending
order
                    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: 5, 1 12 2 9 7
// Output: Ascending Order: 1,2,7,9,12
```

Frag-9: Stack Implementation

1. Errors Identified:

- **Index Logic Error**: In `push`, `top` should be incremented after checking for full stack; it should be `top++` instead of `top--`.
- **Display Logic**: The display loop should be `i <= top` instead of `i > top`.

2. Breakpoints Needed:

- Set breakpoints in `push`, `pop`, and `display` methods to observe stack behavior.

3. Steps Taken to Fix Errors:

- Correct the logic for managing the `top` index and the display method.

```java
// 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++; // Corrected increment logic
            stack[top] = value;
        }
    }

    public void pop() {
        if (!isEmpty())
            top--; // Corrected decrement logic
```

```
        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 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();
    }
}

// Output:
// 10 1 50 20 90
// 10 1
```

Frag-10: Tower of Hanoi

1. Errors Identified:

- **Recursion Logic Error**: The recursive calls use incorrect parameters `inter--` and `to+1`, which will lead to compilation and logical errors.

2. Breakpoints Needed:

- Set breakpoints in the `doTowers` method to inspect how `topN`, `from`, `inter`, and `to` change.

3. Steps Taken to Fix Errors:

- Correct the parameters for recursive calls.

Fixed Code:

```
// 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 - 1, inter, from, to); // Corrected parameters
        }
    }
}

// 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
```

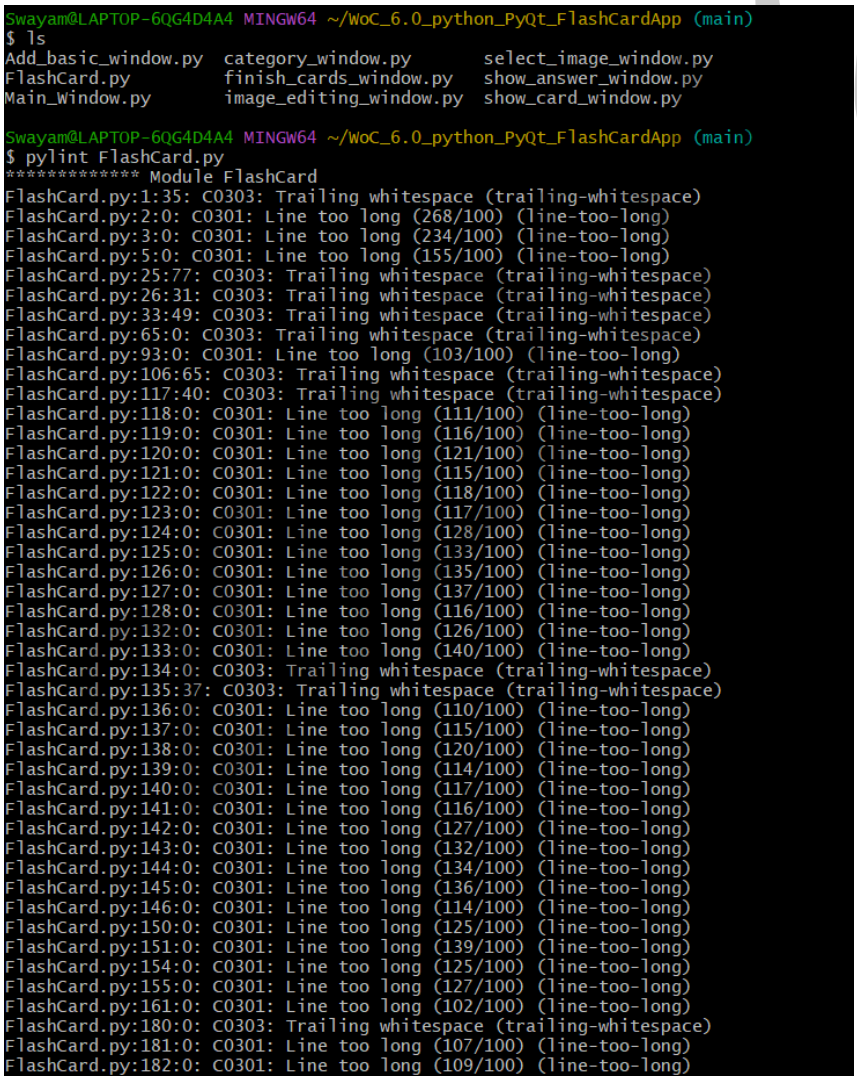
Static Analysis

Github Repository Link:
https://github.com/MSTC-x-IITGN/WoC_6.0_python_PyQt_FlashCardApp

The analysis was performed using **Pylint**

The Pylint analysis identified various issues categorized by their severity levels. The most notable **types of issues** detected include:

Convention (C): Issues related to coding standards (e.g., naming conventions).
Refactor (R): Suggestions for code refactoring to improve readability and maintainability.
Warning (W): Potential problems in the code that could lead to errors (e.g., unused variables).
Error (E): Errors that could lead to exceptions during execution.



```
FlashCard.py:290:12: W0201: Attribute 'back_file_path' defined outside __init__
(attribute-defined-outside-init)
FlashCard.py:315:8: W0201: Attribute 'end_time' defined outside __init__ (attrib
ute-defined-outside-init)
FlashCard.py:420:8: W0201: Attribute 'front_position' defined outside __init__ (
attribute-defined-outside-init)
FlashCard.py:423:8: W0201: Attribute 'back_position' defined outside __init__ (a
ttribute-defined-outside-init)
FlashCard.py:491:8: W0201: Attribute 'file_name' defined outside __init__ (attri
bute-defined-outside-init)
FlashCard.py:20:0: R0904: Too many public methods (60/20) (too-many-public-metho
ds)
FlashCard.py:1290:0: C0116: Missing function or method docstring (missing-functi
on-docstring)
FlashCard.py:6:0: C0411: standard import "import sys" should be placed before "f
rom PyQt5 import QtCore, QtWidgets" (wrong-import-order)
FlashCard.py:7:0: C0411: standard import "import json" should be placed before "
from PyQt5 import QtCore, QtWidgets" (wrong-import-order)
FlashCard.py:8:0: C0411: standard import "import time" should be placed before "
from PyQt5 import QtCore, QtWidgets" (wrong-import-order)
FlashCard.py:9:0: C0411: standard import "import os" should be placed before "fr
om PyQt5 import QtCore, QtWidgets" (wrong-import-order)
FlashCard.py:10:0: C0411: standard import "import io" should be placed before "f
rom PyQt5 import QtCore, QtWidgets" (wrong-import-order)
FlashCard.py:2:0: W0611: Unused QGraphicsScene imported from PyQt5.QtWidgets (un
used-import)
FlashCard.py:2:0: W0611: Unused QGraphicsView imported from PyQt5.QtWidgets (unu
sed-import)
FlashCard.py:2:0: W0611: Unused QAction imported from PyQt5.QtWidgets (unused-im
port)
FlashCard.py:3:0: W0611: Unused QGraphicsItemGroup imported from PyQt5.QtWidgets
 (unused-import)
FlashCard.py:3:0: W0611: Unused QVBoxLayout imported from PyQt5.QtWidgets (unuse
d-import)
FlashCard.py:3:0: W0611: Unused QPushButton imported from PyQt5.QtWidgets (unuse
d-import)
FlashCard.py:3:0: W0611: Unused QWidget imported from PyQt5.QtWidgets (unused-im
port)
FlashCard.py:5:0: W0611: Unused QTextCursor imported from PyQt5.QtGui (unused-im
port)
FlashCard.py:5:0: W0611: Unused QTextBlockFormat imported from PyQt5.QtGui (unus
ed-import)

-----------------------------------
Your code has been rated at 2.06/10
```

Statistical Graph: