ID- 202201171
NAME- SAHIL CHAUDHARI

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 nth 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.

<u>Frag-4: Magic number</u>

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.

<u>Frag-5: Merge sort</u>

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.

<u>Frag-8: Sorting array</u>

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

[1] Armstrong Number

- There is one error in the program related to the computation of the remainder, as previously identified.
- To fix this error, one should set a breakpoint at the point where the remainder is computed to ensure it's calculated correctly. Step through the code to observe the values of variables and expressions during execution.
- The corrected executable code is as follows:

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

- There are two errors in the program as mentioned above.
- To fix these errors:
  - For Error 1 in the gcd function, you need one breakpoint at the beginning of the while loop to verify the correct execution of the loop.
  - For Error 2 in the lcm function, you would need to review the logic for calculating LCM, as it's a logical error.

The corrected executable code is as follows:

```
import java.util.Scanner;

public class GCD_LCM {
    static int gcd(int x, int y) {
        int a, b;
        a = (x > y) ? x : y; // a is greater number
        b = (x < y) ? x : y; // b is smaller number
        while (b != 0) { // Fixed the while loop condition
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }
```

```
    static int lcm(int x, int y) {
        return (x * y) / gcd(x, y); // Calculate LCM using GCD
    }

    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();
    }
}
```

[3] Knapsack

- There is one error in the program, as identified above.
- To fix this error, you would need one breakpoint at the line: int option1 = opt[n][w]; to ensure n and w are correctly used without unintended increments.
- The corrected executable code is as follows:

```
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];
        boolean[] take = new boolean[N + 1]; // Array to track items taken

        // 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);
```

```java
        }

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

        for (int n = 1; n <= N; n++) {
            for (int w = 1; w <= W; w++) {
                int option1 = opt[n - 1][w]; // Fixed the increment here
                int option2 = Integer.MIN_VALUE;

                if (weight[n] <= w) {
                    option2 = profit[n] + opt[n - 1][w - weight[n]];
                }

                opt[n][w] = Math.max(option1, option2);
                sol[n][w] = (option2 > option1);
            }
        }

        // Print the items with their profits, weights, and whether they
are taken
        System.out.println("Item" + "\t" + "Profit" + "\t" + "Weight" +
"\t" + "Take");
        for (int n = 1; n <= N; n++) {
            take[n] = sol[n][W]; // Determine if the item is taken
            System.out.println(n + "\t" + profit[n] + "\t" + weight[n] +
"\t" + take[n]);
        }
    }
}
```

[4] Magic Number

- There are two errors in the program, as identified above.
- To fix these errors, you would need one breakpoint at the beginning of the inner while loop to verify the execution of the loop. You can also use breakpoints to check the values of num and s during execution.

- The corrected executable code is as follows:

```
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) { // Fixed the condition here
                s += sum % 10; // Corrected to sum the digits
                sum = sum / 10; // Corrected to divide by 10 to get the
next digit
            }
            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

- There are multiple errors in the program, as identified above.
- To fix these errors, you would need to set breakpoints to examine the values of left, right, and array during execution. You can also use breakpoints to check the values of i1 and i2 inside the merge method.
- The corrected executable code is as follows:

```
import java.util.*;

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

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

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

    public static int[] rightHalf(int[] array) {
        int size1 = array.length / 2;
        int size2 = array.length - size1;
```

```
        int[] right = new int[size2];
        for (int i = 0; i < size2; i++) {
            right[i] = array[i + size1];
        }
        return right;
    }

    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] Multiply Matrices

- There are multiple errors in the program, as identified above.
- To fix these errors, you would need to set breakpoints to examine the values of c, d, k, and sum during execution. You should pay particular attention to the nested loops where the matrix multiplication occurs.
- The corrected executable code is as follows:

```
import java.util.Scanner;

class MatrixMultiplication {
    public static void main(String args[]) {
        int m, n, p, q, sum = 0, c, d, k;
```

```
        Scanner in = new Scanner(System.in);

        System.out.println("Enter the number of rows and columns of the
first matrix");
        m = in.nextInt();
        n = in.nextInt();
        int first[][] = new int[m][n];

        System.out.println("Enter the elements of the first matrix");
        for (c = 0; c < m; c++) {
            for (d = 0; d < n; d++) {
                first[c][d] = in.nextInt();
            }
        }

        System.out.println("Enter the number of rows and columns of the
second matrix");
        p = in.nextInt();
        q = in.nextInt();

        if (n != p) {
            System.out.println("Matrices with entered orders can't be
multiplied with each other.");
        } else {
            int second[][] = new int[p][q];
            int multiply[][] = new int[m][q];

            System.out.println("Enter the elements of the second matrix");
            for (c = 0; c < p; c++) {
                for (d = 0; d < q; d++) {
                    second[c][d] = in.nextInt();
                }
            }

            for (c = 0; c < m; c++) {
                for (d = 0; d < q; d++) {
                    for (k = 0; k < n; k++) {
                        sum = 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");
            }
        }
        in.close(); // Close the scanner
    }
}
```

[7] Quadratic Probing

- There are three errors in the program, as identified above.
- To fix these errors, you would need to set breakpoints and step through the code while examining variables like i, h, tmp1, and tmp2. You should pay attention to the logic of the insert, remove, and get methods.
- The corrected executable code is as follows:

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

    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');

        scan.close(); // Close the scanner
    }
}
```

[8] Sorting Array

- There are two errors in the program as identified above.
- To fix these errors, you need to set breakpoints and step through the code. You should focus on the class name, the loop conditions, and the unnecessary semicolon.
- The corrected executable code is as follows:

```
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 you want in the
array: ");
        n = s.nextInt();
        int a[] = new int[n];

        System.out.println("Enter all the elements:");
        for (int i = 0; i < n; i++) {
            a[i] = s.nextInt();
        }

        // Bubble sort to arrange the elements in ascending order
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) {
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }

        System.out.print("Ascending Order: ");
        for (int i = 0; i < n - 1; i++) {
            System.out.print(a[i] + ", ");
        }
        System.out.print(a[n - 1]); // Print the last element without a
trailing comma
        s.close(); // Close the scanner
    }
}
```

[9] Stack Implementation

- There are three errors in the program, as identified above.

- To fix these errors, you would need to set breakpoints and step through the code, focusing on the push, pop, and display methods. Correct the push and display methods and add the missing pop method to provide a complete stack implementation.
- The corrected executable code is as follows:

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

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

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

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

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

    public void display() {
        for (int i = 0; i <= top; i++) {
```

```
            System.out.print(stack[i] + " ");
        }
        System.out.println();
    }
}
```

[10] Tower of Hanoi

- There is one error in the program, as identified above.
- To fix this error, you need to replace the line:
  doTowers(topN ++, inter--, from+1, to+1);
- with the correct version:
  doTowers(topN - 1, inter, from, to);
- The corrected executable code is as follows:

```
public class MainClass {
    public static void main(String[] args) {
        int nDisks = 3; // Number of disks
        doTowers(nDisks, 'A', 'B', 'C'); // A, B, and C are the names of
the rods
    }

    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); // Move topN-1 disks from
'from' to 'inter'
            System.out.println("Disk " + topN + " from " + from + " to "
+ to); // Move the bottom disk
            doTowers(topN - 1, inter, from, to); // Move the disks from
'inter' to 'to'
        }
    }
}
```

STATIC ANALYSIS

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.

```
Swayam@LAPTOP-6QG4D4A4 MINGW64 ~/WoC_6.0_python_PyQt_FlashCardApp (main)
$ ls
Add_basic_window.py  category_window.py       select_image_window.py
FlashCard.py         finish_cards_window.py   show_answer_window.py
Main_Window.py       image_editing_window.py  show_card_window.py

Swayam@LAPTOP-6QG4D4A4 MINGW64 ~/WoC_6.0_python_PyQt_FlashCardApp (main)
$ pylint FlashCard.py
************* Module FlashCard
FlashCard.py:1:35: C0303: Trailing whitespace (trailing-whitespace)
FlashCard.py:2:0: C0301: Line too long (268/100) (line-too-long)
FlashCard.py:3:0: C0301: Line too long (234/100) (line-too-long)
FlashCard.py:5:0: C0301: Line too long (155/100) (line-too-long)
FlashCard.py:25:77: C0303: Trailing whitespace (trailing-whitespace)
FlashCard.py:26:31: C0303: Trailing whitespace (trailing-whitespace)
FlashCard.py:33:49: C0303: Trailing whitespace (trailing-whitespace)
FlashCard.py:65:0: C0303: Trailing whitespace (trailing-whitespace)
FlashCard.py:93:0: C0301: Line too long (103/100) (line-too-long)
FlashCard.py:106:65: C0303: Trailing whitespace (trailing-whitespace)
FlashCard.py:117:40: C0303: Trailing whitespace (trailing-whitespace)
FlashCard.py:118:0: C0301: Line too long (111/100) (line-too-long)
FlashCard.py:119:0: C0301: Line too long (116/100) (line-too-long)
FlashCard.py:120:0: C0301: Line too long (121/100) (line-too-long)
FlashCard.py:121:0: C0301: Line too long (115/100) (line-too-long)
FlashCard.py:122:0: C0301: Line too long (118/100) (line-too-long)
FlashCard.py:123:0: C0301: Line too long (117/100) (line-too-long)
FlashCard.py:124:0: C0301: Line too long (128/100) (line-too-long)
FlashCard.py:125:0: C0301: Line too long (133/100) (line-too-long)
FlashCard.py:126:0: C0301: Line too long (135/100) (line-too-long)
FlashCard.py:127:0: C0301: Line too long (137/100) (line-too-long)
FlashCard.py:128:0: C0301: Line too long (116/100) (line-too-long)
FlashCard.py:132:0: C0301: Line too long (126/100) (line-too-long)
FlashCard.py:133:0: C0301: Line too long (140/100) (line-too-long)
FlashCard.py:134:0: C0303: Trailing whitespace (trailing-whitespace)
FlashCard.py:135:37: C0303: Trailing whitespace (trailing-whitespace)
FlashCard.py:136:0: C0301: Line too long (110/100) (line-too-long)
FlashCard.py:137:0: C0301: Line too long (115/100) (line-too-long)
FlashCard.py:138:0: C0301: Line too long (120/100) (line-too-long)
FlashCard.py:139:0: C0301: Line too long (114/100) (line-too-long)
FlashCard.py:140:0: C0301: Line too long (117/100) (line-too-long)
FlashCard.py:141:0: C0301: Line too long (116/100) (line-too-long)
FlashCard.py:142:0: C0301: Line too long (127/100) (line-too-long)
FlashCard.py:143:0: C0301: Line too long (132/100) (line-too-long)
FlashCard.py:144:0: C0301: Line too long (134/100) (line-too-long)
FlashCard.py:145:0: C0301: Line too long (136/100) (line-too-long)
FlashCard.py:146:0: C0301: Line too long (114/100) (line-too-long)
FlashCard.py:150:0: C0301: Line too long (125/100) (line-too-long)
FlashCard.py:151:0: C0301: Line too long (139/100) (line-too-long)
FlashCard.py:154:0: C0301: Line too long (125/100) (line-too-long)
FlashCard.py:155:0: C0301: Line too long (127/100) (line-too-long)
FlashCard.py:161:0: C0301: Line too long (102/100) (line-too-long)
FlashCard.py:180:0: C0303: Trailing whitespace (trailing-whitespace)
FlashCard.py:181:0: C0301: Line too long (107/100) (line-too-long)
FlashCard.py:182:0: C0301: Line too long (109/100) (line-too-long)
```

```
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
```