ENGINEERS WITH
SOCIAL RESPONSIBILITY

SOFTWARE ENGINEERING IT-314
LAB 08

NAME:- Jinay Vora
ID:- 202201473

EXERCISE-I
PROGRAM INSPECTION USING ERROR CHECKLIST METHOD

1. Taxi_System.py:

Data Reference Error:

As in the Taxi class, this property current_trip was used without adequate checks for None in many places. For example, in the method complete_trip() of class Taxi: an assumption was made that a current_trip was assigned; AttributeError may occur if taxi completes a trip that was never assigned.

Control flow error:

This find_nearest_taxi() method in the Dispatcher class only searches for a taxi at hand in a specific location. However, if no taxi is found, then there is no search using the taxis in the neighboring locations, which might be suboptimal for a dispatcher.

Calculation Mistake:

The method assign_trip() will change the status to unavailable upon assignment of a trip. Now, if due to a database failure or a communication error, the assignment fails, the status change would persist, perhaps even blocking some other operations from completing.

2. Trip_Management.py:

Data Declaration Error:

In the Passenger class, attribute trip_id is initialized as some random integer with no check for uniqueness. Then this might allow two trips having the same ID; hence when retrieval of trip information takes place, very confusing behaviors are associated.

Arithmetical Error:

In the function calculate_distance(), it uses a fixed distance base on predefined places. If an undefined route is requested, the system reverts to using a random distance. This sometimes could produce plausible behaviors, though highly inconsistent and may result in total wrong fare calculations.

Flow Control Error:

The method start_trip() starts to assume that once the taxi is assigned, the trip would always start fine; it can't roll back to the previous state when the taxi assignment fails in the middle, say due to network errors, leaving behind the incomplete trips going to be recorded.

3. Pricing.py:

Input/Output Error:

This pricing system depends on the use of hard-coded distances between locations, thus is very inflexible in the addition of new locations or in dynamic calculation. Handling edge cases is also not implemented when inputs for pickup or destination are invalid or unknown.

Calculation Mistake:

The function calculate_fare() uses a fixed base fare and cost per mile but does not bring into play any of the other factors, for instance, traffic or time of day to apply surge pricing; thereby at rush hour, for instance, may inflate artificially estimates of fares.

4. Feedback.py:

Data Reference Error:

In the collect_feedback() method, feedback is assigned directly to the taxi without ascertaining whether the taxi really participated in the completed trip. This might therefore lead to wrong feedback being assigned to taxis which did not take part in the completed trip.

Control Flow Error:

There is no mechanism to handle the failure of collecting feedback. Example: In case the feedback system throws an error, no feedback is collected and there is also no retry mechanism or logging for such an event.

5. Passengers.py

Data reference error:

The request_taxi() method creates a new trip for the passenger but does not validate the pickup_location or destination for the passenger. Such attributes might be invalid or empty at times, and on such an event, the system fails without proper error handling.

Interface Error:

The request_taxi() method calls directly into the Trip class, but it neither checks if an attempt to create a trip failed, logs a failure, nor prevents a lost taxi request that neither party knows about.

6. Logging_Module.py:

Interface Failure:

The logging module appropriately logs information, debug messages, and error messages, although it does not lend itself to allowing different logging outputs or to persisting beyond console outputs. For larger systems, persistent logs, e.g., database or file logging, are important for auditing over time and to trace issues over time.

7. Error_Handling.py:

Mistake:

Data Declaration Custom exceptions have been defined but not consistently throughout the whole system. For example, standard Python exceptions are even being applied within the dispatching logic rather than custom exceptions being employed throughout the modules. Control Flow Error Exception handling in this module doesn't provide any suggestion or tasks to resolve the issue.

8. Additional_Functionalities.py:

Data Reference Mistakes:

The execute() method of the AdditionalFeature class does not check for proper execution. With an error, this would lead to the incorrect logging of success result conditions. Such output might lead to confusion in the test process, and the developers might remain confused.

Dispatcher Reference:

At some places in the test_dispatching_system() function, the call to Dispatcher may not be able to complete the initialization process. This might reference a broken dispatcher at runtime. Hence, checks of this type for successful initialization are very much required for their reliability.

Control Flow Bugs:

The function simulated_workload() that iterates 100 times for feature testing does not provide an escape mechanism in case of errors that might cascade without proper error handling within the loop.

User-Interface Errors:

Logging System should log its errors elsewhere, apart from on the console, in files for example, or on remote servers. Furthermore, dynamic adjustment of logging levels depending on the state of the system is important for peak loads.

Dispatcher Module:
The codes importing class Dispatcher from dispatcher_module and not in these code snippets tend to crash the system when initialized. Including fallback schemes for missing modules would be helpful for a more user-friendly performance and easier debugging.

Input/Output Errors:
In the passenger taxi request, the processing of trips is done without validating passenger input data. This may cause some unpredictability for which the system behaves erratically and even crashes. It would be beneficial if integrity checks were incorporated to ensure proper data integrity before passing to the system while making passenger requests.

Simulated Feature Execution:
Passenger requests executed in sequence in the function simulated_workload() does not provide any form of feedback upon success or failure. Consequently, it does not allow the possibility for proper logging and response to the results of executing different features.

Error Handling:
The TaxiNotAvailableException was declared but not used in the program, waiting for occurrences where no taxis are available when they are requested to be sent. No error handling leads to silent crashes that decrease developers' perception of issues.

General Error Handling:
All the main functions like test_dispatching_system() and simulated_workload() do not have any error handling in them. The insertion of try-except blocks will prevent unexpected system crashes and enhance logging and recovery practices

Q1. How many errors are there in the program? Mention the errors you have identified.
A1. Data reference errors: execute() method of class AdditionalFeature does not check if a feature executes with success. The code dispatcher does not initialize which creates problems and causes its runtime

Control flow errors: The simulated_workload() function did not handle errors in its loop, and there could be more than one failure without exiting early. At the end of taxi trip completion process, there is no check for partially completed trips, and errors can come through if something fails.

User-Interface Errors: Only logging prints on the console, with no choices for files or remote servers. There is no error handling on import module-dispatcher, which may crash the whole system if not successful.

Input/Output Errors: There is no validation on passenger input, and bad data causes unpredictable behavior. There is no feedback indicating if the feature executions were successful or not.

Error Handling: There exists an exception class TaxiNotAvailableException, declared but unused, and so there is no feedback information that the taxi is not available.There exists no generic error handling in key functions to avoid crashes.

Q2. Which category of program inspection would you find more effective?
A2. This sort of program fits best in error handling. The system of interaction between the taxi dispatch and the user request becomes so interactive and the errors that arise, as well as the inputs that are unexpected, must be dealt with. Proper error-handling can really strengthen the experience of the users and support the system to run efficiently.

Q3. Which type of error are you not able to identify using the program inspection?
A3. Concurrency Anomalies: There is a possibility of race conditions or deadlocks in the system especially if many passengers order taxis at the same time. The static code analysis would not give evidence of such occurrences.

Performance Bottlenecks: There is no indication of inefficiency that may slow up the system, especially when running it under heavy loads.

Memory Leaks: There will be memory exhaustion due to extended computation. Dynamic testing will be necessary to capture this.

Q4. Is the program inspection technique worth applicable?
A4. I think program inspection is quite useful in software design, especially for a system like this. It does this by:

Detecting Bugs Early: Debugging beforehand before running the code can save a lot of head-aches later on.

Code Quality Improvement: Systematic inspections breed quality code.

Fostering Collaboration: Team members can collaborate while performing inspection and solves together.

Documentation of Learning: Keeping track of issues and their resolutions can act as guidance for future development and orientation of new members.

EXERCISE-II
DEBUGGING JAVA CODES

1. **Armstrong Number:**

- **Error 1:** The calculation of remainder is incorrect. It uses integer division instead of modulus to get the last digit.
- **Error 2:** The update of num after getting the remainder is incorrect. It uses modulus instead of division to remove the last digit.
- There is one breakpoint needed to effectively debug the errors identified. Place a breakpoint at the beginning of the while loop to observe the values of num and check.
- Change **remainder=num/10;** to **remainder=num%10;** to correctly get the last digit of the number.
- Change **num=num%10;** to **num=num/10;** to correctly remove the last digit after processing it.
- Execute the code line by line after making these changes to ensure the logic now correctly identifies Armstrong numbers.

```
// Armstrong Number
    class Armstrong {
        public static void main(String args[]) {
            int num = Integer.parseInt(args[0]);
            int n = num;
            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:**

- **Error 1:** In the GCD function, the condition in the while loop should check for **while(a % b != 0)** instead of **while(a % b == 0)**. This prevents the loop from functioning correctly.
- **Error 2:** In the lcm function, the condition in the if statement should be **if(a % x == 0 && a % y == 0)** to ensure it correctly identifies when a is a common multiple of both numbers.
- There are two breakpoints needed to effectively debug and observe the execution flow in both functions, one at the beginning of GCD function and one at the beginning of LCM function.
- Change the while loop condition from **while(a%b==0)** to **while(a%b!=0)** to ensure it continues until the GCD is found.
- Similarly, change the if statement condition from **if(a%x!=0 && a%y!=0)** to **if(a%x==0 && a%y==0)** to correctly check for the least common multiple.
- After making these changes, run the debugger and step through the code to ensure both functions work as intended and the correct values are calculated.

```
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;
        b = (x < y) ? x : y;
        r = b;
        while (b != 0) {
            r = a % b;
            a = b;
            b = r;
        }
        return a;
    }
    static int lcm(int x, int y) {
        int a = (x > y) ? x : y;
        while (true) {
            if (a % x == 0 && a % y == 0) // Corrected condition
            return a;
          ++a;
        }
    }
    public static void main(String args[]) {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter the two numbers: ");
        int x = input.nextInt();
        int y = input.nextInt();

        System.out.println("The GCD of two numbers is: " +
gcd(x, y));
        System.out.println("The LCM of two numbers is: " +
lcm(x, y));
        input.close();
    }
}
```

3. **Knapsack:**
   - **Error 1:** The increment operator (n++) is incorrectly used in option1. This causes n to skip values, leading to incorrect indexing in the opt array.
   - **Error 2:** In the calculation for option2, the index for profit should be profit[n], not profit[n-2], since we want to include the profit of the current item n.
   - **Error 3:** The condition for option2 when checking if the weight exceeds the limit should be if (weight[n] <= w) to correctly check if the item can be taken.
   - Two breakpoints are needed. One breakpoint inside the nested loop where decisions are made regarding taking or not taking items and another at the beginning of the main function to observe initial values and input.
   - Change int **option1=opt[n++][w];** to int **option1=opt[n][w];**. This prevents the n index from being incremented incorrectly.
   - Change int **option1=opt[n++][w];** to int **option1=opt[n][w];**. This prevents the n index from being incremented incorrectly.
   - Change the condition to **if(weight[n]<=w)** to ensure we are only considering items that can fit

in the knapsack.

```
import java.util.Scanner;
public class Knapsack {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        int W = Integer.parseInt(args[1]);
        int[] profit = new int[N+1];
        int[] weight = new int[N+1];
        for (int n = 1; n <= N; n++) {
            profit[n] = (int) (Math.random() * 1000);
            weight[n] = (int) (Math.random() * W);
        }
        int[][] opt = new int[N+1][W+1];
        boolean[][] sol = new boolean[N+1][W+1];
        for (int n = 1; n <= N; n++) {
            for (int w = 1; w <= W; w++) {
                int option1 = opt[n][w];
                int option2 = Integer.MIN_VALUE;
                if (weight[n] <= w) {
                    option2 = profit[n] + opt[n-1][w-weight[n]]; //
Corrected profit index
                }
                opt[n][w] = Math.max(option1, option2);
                sol[n][w] = (option2 > option1);
            }
        }
        boolean[] take = new boolean[N+1];
        for (int n = N, w = W; n > 0; n--) {
            if (sol[n][w]) {
                take[n] = true;
                w = w - weight[n];
            } else {
                take[n] = false;
            }
        }
        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]);
        }
    }
}
```

Q4. Magic Number:

- **Error 1**: The condition in the inner while loop should check for **while(sum!=0)** instead of **while(sum==0)** to ensure that we process digits as long as the sum is not zero.
- **Error 2**: The line **s=s*(sum/10);** is incorrect. It should be **s+=(sum%10);** to add the last digit of sum to s.
- **Error 3**: The sum should be reset to 0 at the beginning of each outer loop iteration before calculating the sum of digits. It also needs a semicolon at the end of **sum=sum%10;**.
- Two breakpoints are needed. One at the beginning of the outer while loop to observe the state of num and another at the beginning of the inner while loop to watch how sum and s are modified.
- Change **while(sum==0)** to **while(sum != 0)**.
- Change **s=s*(sum/10);** to **s += (sum % 10);** to correctly sum the digits.
- Add **sum = 0;** at the beginning of the outer loop to ensure it's reset for each iteration.
- Ensure that **sum = sum % 10;** has a semicolon at the end.

```
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, num = n;
        while (num > 9) {
            sum = num;
            int s = 0;
            while (sum != 0) { // Corrected condition
                s += (sum % 10); // Corrected logic
                sum = sum / 10; // Fixed semicolon
            }
            num = s; // Set num to the 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
    }
}
```

Q5. Merge Sort:

- **Error 1:** The way the left and right halves are created is incorrect. You should not use **array+1** or **array -1**. Instead, you should directly pass the original array and calculate the indices.
- **Error 2:** The merge method is being called with incorrect parameters. The increment operators **left++** and **right--** should not be used. Instead, just pass left and right arrays.
- **Error 3:** The **leftHalf** and **rightHalf** methods need to handle the case where the array length is odd, so that the left half can correctly take the extra element if needed.
- Three are three breakpoints needed. One in the mergeSort method to observe the state of the array as it is split, another in the merge method to watch how the merging occurs, and the last in either leftHalf or rightHalf to see how the subarrays are formed.
- Change the calls in mergeSort to:
  int[] left=leftHalf(array);
  int[] right=rightHalf(array);
- Change the merge in mergeSort to
  merge(array,left,right);

```
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 mid = array.length / 2;
            int[] left = Arrays.copyOfRange(array, 0, mid);
            int[] right = Arrays.copyOfRange(array, mid,
array.length);
            mergeSort(left);
            mergeSort(right);
            merge(array, left, 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++;

            }

        }

    }

}
```

Q6. Multiply Matrices:

- **Error1:** Incorrect indexing in the multiplication loop: **first[c-1][c-k]** and **second[k-1][k-d]** should be **first[c][k]** and **second[k][d]**.
- **Error2:** The input prompt for the second matrix is mistakenly repeated as "first matrix".
- **Error3:** The variable sum is not reset in the right place; it should be reset at the beginning of the innermost loop.
- Two breakpoints are needed. One in the matrix multiplication loop to observe the indices and sums and another after reading inputs to check the matrices before multiplication.
- Change the indexing in the multiplication logic to sum=sum+first[c][k]*second[k][d];
- Reset sum at the start of the innermost loop.

```
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 first matrix");
      m = in.nextInt();
      n = in.nextInt();
      int first[][] = new int[m][n];
      System.out.println("Enter the elements of first matrix");
      for (c = 0; c < m; c++)
         for (d = 0; d < n; d++)
            first[c][d] = in.nextInt();
      System.out.println("Enter the number of rows and
columns of second matrix");
      p = in.nextInt();
      q = in.nextInt();

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

         int multiply[][] = new int[m][q];

         System.out.println("Enter the elements of second
matrix");

         for (c = 0; c < p; c++)

            for (d = 0; d < q; d++)

              second[c][d] = in.nextInt();

         for (c = 0; c < m; c++) {

            for (d = 0; d < q; d++) {

              sum = 0; // Reset sum here

              for (k = 0; k < n; k++) {

                sum += first[c][k] * second[k][d];

              }


              multiply[c][d] = sum;

            }

         }

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

Q7. Quadratic Probing:

- **Error1:** Syntax error in **i + = (i + h / h--) % maxSize;** it should be **i += (h * h++) % maxSize;**.
- **Error2:** The hash function can return a negative index due to negative hash codes, which should be handled.
- **Error3:** Incorrect logic in the remove and get methods, particularly with the use of h and the increment logic.
- **Error4:** The insert and get methods can lead to infinite loops due to improper indexing.
- Three breakpoints are needed. One in the insert method to monitor the insertion process, another in the get method to observe key lookups and the last in the remove method to watch key removals.
- Correct the insertion logic to **i+= (h * h++) % maxSize;**.
- Ensure the hash code is non-negative.
  return (key.hashCode() % maxSize + maxSize) % maxSize;

```
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 + maxSize) %
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 = (tmp + h * h) % maxSize;
        h++;
    } 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;
        h++;
    }
    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 scanner
    }
}
```

Q8. Sorting Array:

- **Error1:** Class name has a space: Ascending _Order should be AscendingOrder.
- **Error2:** Incorrect loop condition in the outer loop: **for (int i=0; i>=n; i++)** should be **for(int i=0;i<n; i++)**.
- **Error3:** Semicolon at the end of the outer loop: **for(int i = 0; i >= n; i++);** should not have a semicolon.
- **Error4:** The sorting logic is incorrect. It should swap when **a[i] > a[j]** instead of **a[i] <= a[j]**.
- Two breakpoints are needed. One in the outer loop to monitor iterations and the other in the inner loop to check the swapping conditions.

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

Q9. Stack Implement:

- **Error1:** In the push method, top-- should be top++ to correctly increment the index.
- **Error2:** In the pop method, top++ should be top-- to correctly decrement the index.
- **Error3:** In the display method, the loop condition should be i <= top instead of i > top to display all elements correctly.
- Three breakpoints are needed. One in the push method to check value insertion, another in the pop method to verify value removal and the last in the display method to inspect the output.

```
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
            stack[top] = value;
        }
    }
    public void pop() {
        if (!isEmpty()) {
            top--; // Corrected decrement
        } else {
            System.out.println("Can't pop...stack is empty");
        }
    }
    public boolean isEmpty() {
        return top == -1;
    }
    public void display() {
        for (int i = 0; i <= top; i++) { // Corrected loop condition
            System.out.print(stack[i] + " ");
        }
        System.out.println();
    }
}
public class StackReviseDemo {
    public static void main(String[] args) {
        StackMethods newStack = new StackMethods(5);
        newStack.push(10);
        newStack.push(1);
        newStack.push(50);
        newStack.push(20);
        newStack.push(90);

        newStack.display(); // Display stack contents

        newStack.pop();

        newStack.pop();

        newStack.pop();

        newStack.pop();

        newStack.display(); // Display stack contents after
pops

    }

}
```

Q10. Tower of Hanoi:

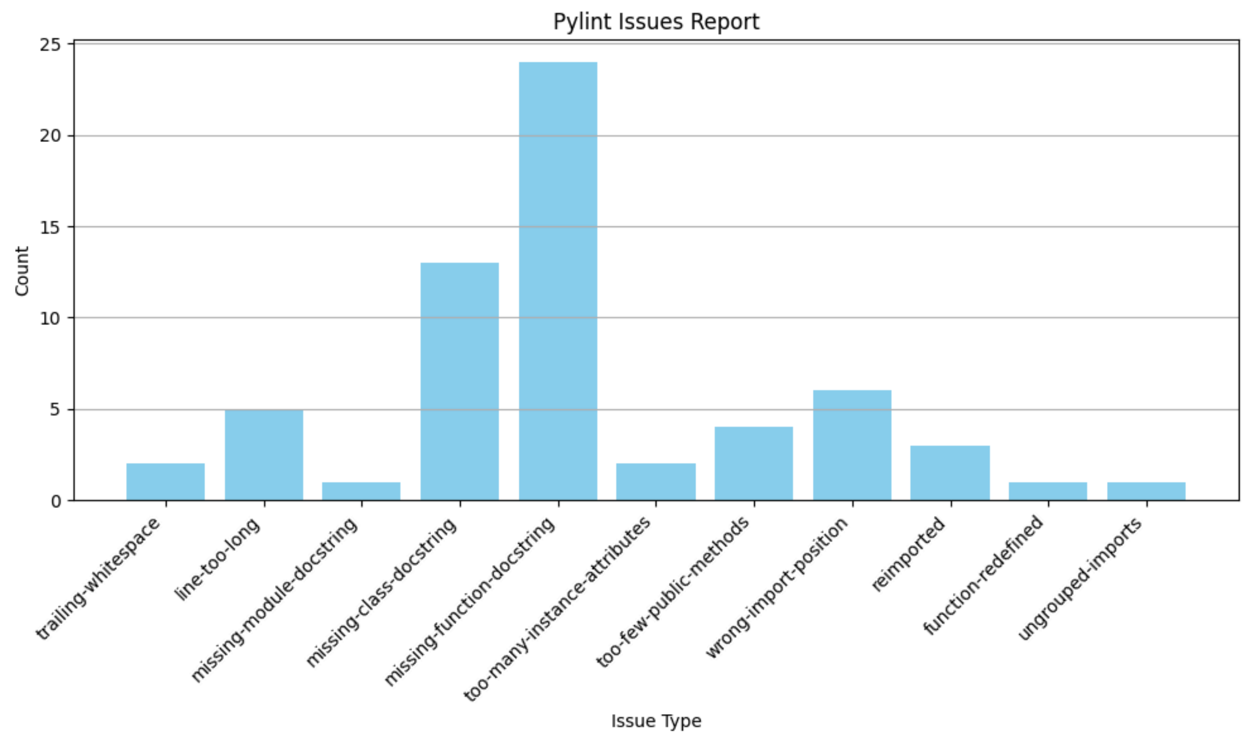
- **Error1:** In the recursive call **doTowers(topN ++, inter--, from+1, to+1)**, the increment and decrement operators are incorrectly used. They should not be there.
- **Error2:** The parameters from and to should remain as characters, not integers, when passing them to the recursive calls.
- Two breakpoints are needed. One in the base case of the doTowers method to check the output when topN is 1 and the other before each recursive call to inspect the values being passed.

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

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

EXERCISE-III
STATIC ANALYSIS

Given below is the Static Analysis of the 2000 LOC that I have used for


Pylint Issues Report
Count
trailing-whitespace
line-too-long
missing-module-docstring
missing-class-docstring
missing-function-docstring
too-many-instance-attributes
too-few-public-methods
wrong-import-position
reimported
function-redefined
ungrouped-imports
Issue Type