IT 314 Software Engineering
Lab 8 Debugging and Inspection

Student Id : 202201230

Name : Rhythm Panchal

CODE INSPECTION and DEBUGGING:

1. **ArmStrong Number**
   Original 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;
                        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 a Armstrong Number");
        }
```

   Input: 153
   Output: 153 is an armstrong Number.

   → Upon inspection of the Armstrong Number code, the following error was identified: in the while loop, the remainder was calculated incorrectly. The code uses division (/) instead of modulus (%) to calculate the last digit of the number, and it mistakenly uses modulus (%) to update the number instead of division (/). This resulted in incorrect logic for checking if a number is an Armstrong number.

   Errors Identified:

1. Incorrect use of division (/) and modulus (%) for remainder calculation and number reduction.

   Effective Category:

- The most effective category of program inspection for this code is **Category C: Computation Errors**, as the errors pertain to arithmetic operations and computation logic.

Unidentified Errors:

- Program inspection does not identify runtime errors such as handling non-numeric input or other exceptions. It also does not cover style and maintainability issues comprehensively.

Applicability:

- The program inspection technique is valuable for identifying and rectifying issues related to code structure and computation errors, ensuring that the program logic is correct before running the code.

Breakpoints Used:

1. One breakpoint before the while loop to check the initial value of num.
2. Another breakpoint inside the while loop to monitor the values of remainder and check.

Steps Taken to Fix the Error:

- Corrected the operations within the while loop by swapping the use of / and %.

2.**GCD and LCM**

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

      r = b;
      while(a % b == 0) //Error replace it with while(a % b != 0)
      {
         r = a % b;
         a = b;
         b = r;
      }
      return r;
   }

   static int lcm(int x, int y)
   {
      int a;
      a = (x > y) ? x : y; // a is greater number
      while(true)
      {
         if(a % x != 0 && a % y != 0)
            return a;
         ++a;
```

```
    }
  }

  public static void main(String args[])
  {
    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 GCD of two numbers is 20

Upon inspecting the GCD and LCM code, an error was identified in the gcd function. The while condition incorrectly uses a % b == 0 instead of a % b != 0, which causes the loop to terminate prematurely, leading to incorrect GCD calculation. Additionally, the lcm function logic for checking divisibility should return the LCM when both a % x == 0 and a % y == 0 are true, not when both are false.

Errors Identified:

- GCD Calculation:
    - The condition in the while loop should be

while(a % b != 0).

Current condition while(a % b == 0) is incorrect.

- LCM Calculation:
    - The condition in the if statement should be

if(a % x == 0 && a % y == 0).

Current condition if(a % x != 0 && a % y != 0) is incorrect.

- In the LCM calculation, it should break the loop and return a when both x and y are factors of a.

Effective Category:

- The most effective category of program inspection for this code is Category C: Computation Errors, as the errors pertain to logical and arithmetic operations.

Unidentified Errors:

- Program inspection may not identify runtime errors such as invalid input (non-integer values). It also does not address the efficiency and optimization of the code.

Applicability:

- The program inspection technique is valuable for identifying and rectifying logical and computation errors, ensuring the correct functionality of the algorithm**.**

Breakpoints Used:

1. One breakpoint before the while loop in the gcd method to check the initial values of a and b.
2. Another breakpoint inside the while loop to monitor the remainder calculation.
3. A breakpoint inside the lcm loop to track the increments of a and check divisibility.

Steps Taken to Fix the Error:

- Corrected the while condition in the gcd function.
- Fixed the condition in the lcm function to check if both numbers divide a evenly.

3. **KnapSack**

```
//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
    // sol[n][w] = does opt solution to pack items 1..n with weight limit w include item n?
      int[][] opt = new int[N+1][W+1];
      boolean[][] sol = new boolean[N+1][W+1];

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

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

            // take item n
            int option2 = Integer.MIN_VALUE;
            if (weight[n] > w) option2 = profit[n-2] + opt[n-1][w-weight[n]];
```

```
            // select better of two options
            opt[n][w] = Math.max(option1, option2);
            sol[n][w] = (option2 > option1);
          }
        }

        // determine which items to take
        boolean[] take = new boolean[N+1];
        for (int n = N, w = W; n > 0; n--) {
          if (sol[n][w]) { take[n] = true;  w = w - weight[n]; }
          else           { take[n] = false;                    }
        }

        // print results
        System.out.println("item" + "\t" + "profit" + "\t" + "weight" + "\t" + "take");
        for (int n = 1; n <= N; n++) {
          System.out.println(n + "\t" + profit[n] + "\t" + weight[n] + "\t" + take[n]);
        }
      }
}
```

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

Upon inspecting the Knapsack code, several errors were found that impacted the logic of the program. The main issues stem from an incorrect increment operator (n++) and the misuse of array indices in the opt matrix and profit array.

Errors Identified:

- Loop Increment Error:
    - In the option1 calculation, opt[n++][w] should be

opt[n-1][w].

- Index Error:
    - In the option2 calculation, profit[n-2] should be profit[n].

Effective Category:

- The most effective category of program inspection for this code is Category C: Computation Errors, as the errors pertain to array indexing and logical errors in the computation.

Unidentified Errors:

- Program inspection may not identify runtime errors such as invalid input or edge cases where the maximum weight W is zero. It also does not address the efficiency and performance of the algorithm.

Applicability:

- The program inspection technique is valuable for identifying and rectifying logical and computation errors, ensuring the correct functionality of the algorithm**.**

Breakpoints Used:

1. One breakpoint before the for loop to check the initialized values of profit and weight.
2. A breakpoint inside the nested loops to monitor the values of opt[n][w] and sol[n][w] during the calculation of option1 and option2.
3. A breakpoint during the item selection process to ensure the correct items are being chosen.

Steps Taken to Fix the Error:

- Corrected the usage of the increment operator in the line int option1 = opt[n][w]; to avoid skipping indices.
- Adjusted the conditional check if (weight[n] <= w) to ensure the item is selected if it fits within the knapsack.
- Fixed the array index issue in the profit and opt matrix calculations.

4.**Magic Number**

```
// 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 sum=0,num=n;
      while(num>9)
      {
         sum=num;int s=0;
         while(sum==0)
         {
            s=s*(sum/10);
            sum=sum%10
         }
         num=s;
      }
      if(num==1)
      {
         System.out.println(n+" is a Magic Number.");
      }
      else
      {
         System.out.println(n+" is not a Magic Number.");
      }
   }
}
```

Input: Enter the number to be checked 119

Output 119 is a Magic Number.

Input: Enter the number to be checked 199

Output 199 is not a Magic Number.

In the Magic Number program, several errors were identified in the inner while loop where the sum of the digits was calculated. The logic inside the loop for computing the digit sum was incorrect, causing it to malfunction.

Errors Identified:

- **Inner Loop Condition Error:**
    - while(sum==0) should be while(sum!=0).
- **Multiplication Error:**
    - s=s*(sum/10) should be s=s+(sum%10).
- **Semicolon Missing:**
    - sum=sum%10 should be sum=sum/10;.

Effective Category:

- The most effective category of program inspection for this code is **Category C: Computation Errors**, as the errors pertain to logical and arithmetic operations within the loops.

Unidentified Errors:

- Program inspection may not identify runtime errors such as input validation and edge cases where the input number is zero or negative.

Applicability:

- The program inspection technique is valuable for identifying and rectifying logical and computation errors, ensuring the correct functionality of the algorithm.

Breakpoints Used:

1. Before the while(num > 9) loop to check the initial value of num.
2. Inside the second while loop to track the process of summing the digits.

Steps Taken to Fix the Error:

- Changed the loop condition to while(sum != 0).
- Corrected the expression to add digits properly using s = s + (sum % 10).

5.**Merge Sort**

```
// This program implements the merge sort algorithm for
// arrays of integers.

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));
   }

   // Places the elements of the given array into sorted order
   // using the merge sort algorithm.
   // post: array is in sorted (nondecreasing) order
   public static void mergeSort(int[] array) {
      if (array.length > 1) {
         // split array into two halves
         int[] left = leftHalf(array+1);
         int[] right = rightHalf(array-1);

         // recursively sort the two halves
         mergeSort(left);
         mergeSort(right);

         // merge the sorted halves into a sorted whole
         merge(array, left++, right--);
      }
   }
```

```
// Returns the first half of the given array.
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;
}

// Returns the second half of the given array.
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;
}

// Merges the given left and right arrays into the given
// result array.  Second, working version.
// pre : result is empty; left/right are sorted
// post: result contains result of merging sorted lists;
public static void merge(int[] result,
                         int[] left, int[] right) {
   int i1 = 0;   // index into left array
   int i2 = 0;   // index into right array
```

```
      for (int i = 0; i < result.length; i++) {
        if (i2 >= right.length || (i1 < left.length &&
              left[i1] <= right[i2])) {
          result[i] = left[i1];    // take from left
          i1++;
        } else {
          result[i] = right[i2];   // take from right
          i2++;
        }
      }
   }
}
```

Input: before 14 32 67 76 23 41 58 85

after 14 23 32 41 58 67 76 85

Upon inspecting the Merge Sort implementation, several errors were identified in the recursive mergeSort method and its related operations. The issues are mainly due to incorrect array manipulation, such as invalid addition or subtraction when passing arrays and incrementing array pointers.

Errors Identified:

- **Array Modification in mergeSort Function:**
    - int[] left = leftHalf(array+1); should be int[] left = leftHalf(array);.
    - int[] right = rightHalf(array-1); should be int[] right = rightHalf(array);.
- **Index Modification in merge Function:**
    - merge(array, left++, right--); should be merge(array, left, right);.

Effective Category:

- The most effective category of program inspection for this code is **Category C: Computation Errors**, as the errors pertain to incorrect modifications and accesses of array elements.

Unidentified Errors:

- Program inspection may not identify issues related to performance optimizations and edge cases for very large arrays where recursion depth might be a concern.

Applicability:

- The program inspection technique is valuable for identifying and rectifying logical and computation errors, ensuring the correct implementation of the merge sort algorithm.

Breakpoints Used:

1. A breakpoint before the recursive mergeSort() call to monitor the splitting of arrays.
2. A breakpoint inside the merge() function to track how elements from left and right arrays are merged.

Steps Taken to Fix the Errors:

- Removed incorrect arithmetic from leftHalf(array+1) and rightHalf(array-1) and passed the correct array values.
- Fixed the merging step by avoiding unnecessary increments and decrements when calling merge().

6.**Multiply Matrics**

In this Java program for multiplying two matrices, errors were found mainly in the indexing while performing the matrix multiplication in the nested loops.

Errors Identified:

- Indexing Error in Matrix Multiplication Logic:
    - The statements first[c-1][c-k] and second[k-1][k-d] are incorrect; they should be first[c][k] and second[k][d] respectively to access the correct elements for multiplication.

  Resetting the Sum:
    - sum is reset correctly after each multiplication but should be initialized properly at the start of the multiplication loop for clarity.
- Input Prompts Error:
    - The prompt for the second matrix should say "Enter the number of rows and columns of the second matrix" instead of repeating "first matrix".

Effective Category:

- The most effective category of program inspection for this code is Category C: Computation Errors, as the identified issues pertain to incorrect mathematical operations during matrix multiplication.

Unidentified Errors:

- Program inspection may not identify issues related to performance inefficiencies, such as the algorithm's time complexity when dealing with larger matrices**.**

Applicability:

- The program inspection technique is valuable for identifying logical errors in matrix operations and ensuring that the implementation adheres to mathematical principles**.**

Breakpoints Used:

1. A breakpoint at the nested loop inside the multiplication process to track how the values are being accessed from both matrices.

Steps Taken to Fix the Errors:

- Corrected the matrix element access by changing first[c-1][c-k] to first[c][k] and second[k-1][k-d] to second[k][d].

7.**Quadratic Probe**

```
/**

 *  Java Program to implement Quadratic Probing Hash Table

 **/



import java.util.Scanner;



/** Class QuadraticProbingHashTable **/

class QuadraticProbingHashTable

{

    private int currentSize, maxSize;

    private String[] keys;

    private String[] vals;



    /** Constructor **/

    public QuadraticProbingHashTable(int capacity)
```

```
{
    currentSize = 0;
    maxSize = capacity;
    keys = new String[maxSize];
    vals = new String[maxSize];
}

/** Function to clear hash table **/
public void makeEmpty()
{
    currentSize = 0;
    keys = new String[maxSize];
    vals = new String[maxSize];
}

/** Function to get size of hash table **/
```

```
public int getSize()

{

   return currentSize;

}



/** Function to check if hash table is full **/

public boolean isFull()

{

   return currentSize == maxSize;

}



/** Function to check if hash table is empty **/

public boolean isEmpty()

{

   return getSize() == 0;
```

```
}

/** Fucntion to check if hash table contains a key **/
public boolean contains(String key)
{
    return get(key) !=  null;
}

/** Functiont to get hash code of a given key **/
private int hash(String key)
{
    return key.hashCode() % maxSize;
}

/** Function to insert key-value pair **/
public void insert(String key, String val)
```

```
{
    int tmp = hash(key);
    int i = tmp, h = 1;
    do
    {
        if (keys[i] == null)
        {
            keys[i] = key;
            vals[i] = val;
            currentSize++;
            return;
        }
        if (keys[i].equals(key))
        {
            vals[i] = val;
```

```
            return;

        }

        i + = (i + h / h--) % maxSize;

    } while (i != tmp);

}



/** Function to get value for a given key **/

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;

        System.out.println("i "+ i);
```

```
    }

    return null;

}



/** Function to remove key and its value **/

public void remove(String key)

{

    if (!contains(key))

        return;



    /** find position key and delete **/

    int i = hash(key), h = 1;

    while (!key.equals(keys[i]))

        i = (i + h * h++) % maxSize;

    keys[i] = vals[i] = null;
```

```
    /** rehash all keys **/
    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--;
}

/** Function to print HashTable **/
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();
    }
}


/** Class QuadraticProbingHashTableTest **/
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");
```

```
/** maxSizeake object of QuadraticProbingHashTable **/

QuadraticProbingHashTable qpht = new QuadraticProbingHashTable(scan.nextInt() );



char ch;

/**  Perform QuadraticProbingHashTable operations  **/

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;

        }

        /** Display hash table **/

        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');

}
```

```
}
```

Input:

Hash table test

Enter size: 5

Hash Table Operations

1. Insert

2. Remove

3. Get

4. Clear

5. Size

1

Enter key and value

c computer

d desktop

h harddrive

Output:

Hash Table:

c computer

d desktop

h harddrive

The provided Java program implements a hash table using quadratic probing for collision resolution. During the code review, several issues were identified that could lead to runtime errors and logical bugs.

Errors Identified:

- Syntax Error in Incrementing i:

    - The line i + = (i + h / h--) % maxSize; has an extra space which causes a compilation error. It should be i += (i + h * h) % maxSize;.

- Hash Calculation Logic Error in get Method:
    - The expression i = (i + h * h++) % maxSize; incorrectly uses h++. It should be h = h + 1; after using h * h to ensure the correct increment of h for the next iteration.

- Misleading Comments:
    - The comments for insert, get, and remove functions use the wrong spelling and may confuse readers.

- Incorrect Resizing Logic:
    - The makeEmpty function creates new arrays for keys and vals, which may not clear the existing data correctly as references remain. It should reset the existing arrays instead of creating new ones.

Effective Category:

- The most effective category of program inspection for this code is Category B: Data Structure Errors, as the errors primarily relate to the logic of maintaining the hash table and the quadratic probing mechanism.

Unidentified Errors:

- Program inspection may not identify potential performance issues, such as the hash table's behavior under heavy load, which could lead to performance degradation due to increased collision handling.

Applicability:

- The program inspection technique is indeed applicable and beneficial, as it helps identify logical errors and ensures that the hash table operations follow the expected behavior.

Breakpoints Used:

1. A breakpoint at the insertion logic to ensure that keys are being placed correctly in the hash table.
2. A breakpoint in the retrieval logic to check if keys can be fetched properly.

Steps Taken to Fix the Errors:

- Corrected the increment logic in the insertion, retrieval, and removal processes.
- Ensured that hash values are always positive by using Math.abs().
- Revised the comments for better clarity and correctness.

8.**Sorting Array**

```
// sorting the array in ascending order
import java.util.Scanner;
public class Ascending _Order
{
   public static void main(String[] args)
   {
      int n, temp;
      Scanner s = new Scanner(System.in);
      System.out.print("Enter no. of elements you want in array:");
      n = s.nextInt();
      int a[] = new int[n];
      System.out.println("Enter all the elements:");
      for (int i = 0; i < n; i++)
      {
         a[i] = s.nextInt();
      }
      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]);
  }
}
```

Input: Enter no. of elements you want in array: 5

Enter all elements:

1 12 2 9 7

1 2 7 9 12

The provided Java program is meant to sort an array in ascending order. However, it contains multiple errors that prevent it from functioning correctly.

Errors Identified:

1. Syntax Error in Incrementing i:
    - Correction: Change i + = (i + h / h--) % maxSize; to i += (i + h * h) % maxSize;.
2. Hash Calculation Logic Error in get Method:
    - Correction: Replace i = (i + h * h++) % maxSize; with h = h + 1; after using h * h.
3. Misleading Comments:
    - Correction: Revise comments for clarity and correct spelling.
4. Incorrect Resizing Logic in makeEmpty:
    - Correction: Reset existing arrays instead of creating new ones**.**

Effective Category:

- Category B: Data Structure Errors due to issues in hash table logic.

Unidentified Errors:

- Potential Performance Issues: Not addressing behavior under high load could lead to degraded performance.

Applicability:

- Program inspection is beneficial for identifying logical errors and ensuring expected hash table operations.

Breakpoints Used:

1. A breakpoint was placed in the sorting loop to observe how elements are swapped.

2. A breakpoint was placed before printing the sorted array to verify the final array content.

Steps Taken to Fix the Errors:

- Corrected the class name and loop conditions.
- Removed the unnecessary semicolon.
- Fixed the comparison logic in the sorting condition.

9.**Stack Implementation**

```
//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--;
         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();
    }
}
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

The Java program is for a stack data structure but has many errors. It needs fixing to work correctly.

Errors Identified:

1. Logic Error in push Method:
    - Issue: The top index is decremented before storing the value.
    - Correction: Change top-- to top++ to push values correctly.
2. Logic Error in pop Method:
    - Issue: The top index is incremented correctly but does not remove the value from the stack.
    - Correction: You should handle the return value and reset the stack at top.
3. Logic Error in display Method:
    - Issue: The loop condition is incorrect (i > top should be i <= top).
    - Correction: Change for(int i=0;i>top;i++) to for(int i=0; i <= top; i++).

Effective Category:

- Category B: Data Structure Errors
    - The errors relate to the logic of stack operations and index management.

Unidentified Errors:

- Potential array index out-of-bounds errors during push and pop operations under high usage may not be identified.

Applicability:

- The program inspection technique is applicable as it helps identify logical errors in stack operations, ensuring proper functionality.

10. **Tower of Hanoi**

```
//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 ++, inter--, from+1, to+1)
    }
  }
}
```

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

This program solves the Tower of Hanoi puzzle with 3 disks. It shows steps to move disks from one rod to another.

Errors Identified:

1. Logic Error in the Recursive Calls:
    - Issue: The parameters in the recursive call doTowers(topN ++, inter--, from+1, to+1) incorrectly modify topN and inter. This will not work as intended and cause incorrect behavior.
    - Correction: Use topN - 1 and inter directly without modifying them.
2. Output Logic Error:
    - The logic to print "Disk 1 from A to C" is correct for the base case, but subsequent recursive calls are not correctly handling the parameters.
    - Correction: Ensure the parameters are passed correctly to maintain the correct state of the disks.
3. Unnecessary Parameter Modifications:
    - Incrementing and decrementing topN and inter in the recursive call is incorrect.
    - Correction: Use the original values without modifying them in the call.

Effective Category:

- Category A: Logic Errors
    - The errors primarily relate to the logic of recursive function calls and parameter handling.

Unidentified Errors:

- Potential stack overflow errors if the number of disks (nDisks) is too large may not be identified during a basic inspection.

Applicability:

- The program inspection technique is applicable, as it helps identify logical errors in the recursive implementation of the Tower of Hanoi algorithm.