It-314
Software Engineering

Meet Katharotiya - 202201157

Lab07:
Program Inspection, Debugging and Static Analysis

Question 1 and 2 (Program Inspection and debugging)

1 . Armstrong

Program Inspection

1. There is one error in the program related to the computation of the remainder, where remainder = num / 10; should be remainder = num % 10;.
2. Category C: Computation Errors is more effective, as the issue lies in incorrect arithmetic operations.
3. Program inspection cannot easily identify logical or runtime errors.
4. Yes, program inspection is worth applying for identifying code structure and computation errors.

Debugging

1. One error related to the computation of the remainder was identified.
2. One breakpoint is needed where the remainder is computed.
3. Steps taken: Set a breakpoint at the line computing the remainder, checked the values of num and remainder during each loop iteration, and corrected the code.

Corrected executable code:

```
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

Program Inspection

1. There is one error in the program related to the computation of the remainder, where remainder = num / 10; should be remainder = num % 10;.
2. Category C: Computation Errors is more effective, as the issue lies in incorrect arithmetic operations.
3. Program inspection cannot easily identify logical or runtime errors.
4. Yes, program inspection is worth applying for identifying code structure and computation errors.

Debugging

1. One error related to the computation of the remainder was identified.
2. One breakpoint is needed where the remainder is computed.
3. Steps taken: Set a breakpoint at the line computing the remainder, checked the values of num and remainder during each loop iteration, and corrected the code.

Corrected executable code:

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

        while(b != 0) // Corrected the error
        {
            r = a % b;
            a = b;
            b = r;
        }
        return a; // Return the GCD
    }

    static int lcm(int x, int y)
```

```
    {
        return (x * y) / gcd(x, y); // LCM can be derived from 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

Program Inspection:

1. There are multiple errors in the program, including:
    - Incrementing n in the line int option1 = opt[n++][w]; instead of keeping it constant.
    - The index used for profit[n-2] should be profit[n] when calculating option2.
    - The weight calculation for the condition if (weight[n] > w) should use weight[n-1] to align with the profit and weight arrays correctly.
2. Category C: Computation Errors would be more effective, as the identified errors pertain to index handling and computation logic.
3. Program inspection may not identify issues such as logical errors that occur due to incorrect index manipulation, which can lead to runtime exceptions.
4. Yes, the program inspection technique is valuable for identifying computation and logical errors in the implementation.

Debugging:

1. There are three errors in the program related to indexing and computation in the opt and profit arrays.
2. Two breakpoints are needed:

- One at the line where option1 is calculated to observe the value of n and w.
- Another at the line where option2 is calculated to ensure that both profit and weight are computed correctly.

3. I fixed the errors by:

- Correcting the increment of n by replacing int option1 = opt[n++][w]; with int option1 = opt[n][w];.
- Changing option2 = profit[n-2] + opt[n-1][w-weight[n]]; to option2 = profit[n] + opt[n-1][w-weight[n]]; to correctly reference the current item.
- Adjusting the weight condition to if (weight[n] <= w) to allow for taking the item when its weight is within the limit.

Corrected executable code:

```
public class Knapsack {

    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);   // number of items
        int W = Integer.parseInt(args[1]);   // maximum weight of knapsack

        int[] profit = new int[N+1];
        int[] weight = new int[N+1];

        // generate random instance, items 1..N
        for (int n = 1; n <= N; n++) {
            profit[n] = (int) (Math.random() * 1000);
            weight[n] = (int) (Math.random() * W);
        }

        // opt[n][w] = max profit of packing items 1..n with weight limit w
        int[][] opt = new int[N+1][W+1];
        boolean[][] sol = new boolean[N+1][W+1];

        for (int n = 1; n <= N; n++) {
            for (int w = 1; w <= W; w++) {
                // don't take item n
                int option1 = opt[n][w]; // Corrected here

                // take item n
                int option2 = Integer.MIN_VALUE;
                if (weight[n] <= w) // Corrected condition here
                    option2 = profit[n] + opt[n-1][w-weight[n]]; // Corrected index here

                // 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]);
        }
    }
}
```

4.Magic Number

Program Inspection:

1. There are multiple errors in the program, including:
    - The condition in the inner while loop should be while(sum!=0) instead of while(sum==0).
    - The calculation s=s*(sum/10); should be corrected to s+=sum%10; to accumulate the digit sum correctly.
    - The line sum=sum%10 is missing a semicolon (;) at the end.
    - The logic for reducing num is not correct. It should sum the digits of num until it reduces to a single digit.
2. Category A: Syntax Errors would be the most effective category to inspect, as the program contains multiple syntax errors.
3. Logical errors, such as how to properly accumulate the sum of the digits and the stopping condition for the loop, may not be identified using program inspection.
4. Yes, program inspection techniques are worth applying as they help identify syntactical and logical errors early in the development process.

Debugging:

1. There are four errors in the program related to syntax and logic for calculating the sum of the digits.
2. Two breakpoints are needed:
    - One at the line where the inner while loop starts to verify the value of sum.
    - Another after updating num to check if it reduces correctly.

3. I fixed the errors by:

- Changing the inner while loop condition to while(sum!=0).
- Modifying s=s*(sum/10); to s+=sum%10;.
- Adding the missing semicolon after sum=sum%10.
- Adjusting the logic to sum the digits until num becomes a single digit.

Corrected executable code:

```
import java.util.*;

public class MagicNumberCheck {
    public static void main(String args[]) {
        Scanner ob = new Scanner(System.in);
        System.out.println("Enter the number to be checked.");
        int n = ob.nextInt();
        int num = n;

        while (num > 9) {
            int sum = 0; // Reset sum for each iteration
            int temp = num; // Use a temporary variable to extract digits

            while (temp != 0) { // Corrected condition
                sum += temp % 10; // Accumulate digit sum
                temp /= 10; // Reduce the number
            }
            num = sum; // Update num with 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 scanner
    }
}
```

5. Merge Sort

Program Inspection:

1. There are multiple errors in the program, including:
    - In the mergeSort method, the calls to leftHalf(array + 1) and rightHalf(array - 1) are incorrect. The method should pass the original array instead.
    - The use of left++ and right-- in the merge call is incorrect; it should just pass the arrays without modification.
    - The merge method is called incorrectly because it needs a correctly sized result array to store the merged values.
2. Category A: Syntax Errors would be the most effective category to inspect, as the program contains multiple syntax and logical errors.
3. Logical errors in how the array is split and merged may not be identified using program inspection.
4. Yes, program inspection techniques are worth applying as they help identify syntactical and logical errors early in the development process.

Debugging:

1. There are four errors in the program related to incorrect array splitting and merging logic.
2. Three breakpoints are needed:
    - One before the mergeSort method call to check the input array.
    - One inside the mergeSort method to verify the left and right arrays.
    - Another in the merge method to verify the merged result.

3. I fixed the errors by:

- Changing left = leftHalf(array + 1); to left = leftHalf(array); to correctly split the array.
- Changing right = rightHalf(array - 1); to right = rightHalf(array); to correctly split the array.
- Removing the increment and decrement operators from the merge call: merge(array, left, right);.
- Creating a result array of the correct size in the merge method.

Corrected executable code:

```java
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;
            // Split array into two halves
            int[] left = Arrays.copyOfRange(array, 0, mid);
            int[] right = Arrays.copyOfRange(array, mid, array.length);

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

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

    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++;
            }
        }
    }
}
```

6.Multiply Matrices

Program Inspection:

1. There are several errors in the program:
    - In the nested loops for matrix multiplication, the indices used in first[c-1][c-k] and second[k-1][k-d] are incorrect. The index should not subtract from c and k. Instead, it should use c and k directly.
    - The input prompt for the second matrix mistakenly repeats "first matrix" instead of saying "second matrix."
    - The sum variable should be initialized to 0 for each product calculation in the innermost loop.
2. Category B: Logical Errors would be the most effective category to inspect, as the program has logical errors related to matrix indexing.
3. The type of error not identified using program inspection includes off-by-one errors in matrix indexing.
4. Yes, program inspection techniques are worth applying as they can help identify potential logical flaws before running the program.

Debugging:

1. There are three errors in the program:
    - Incorrect matrix index usage in the multiplication logic.
    - Repeating the input prompt for the second matrix.
    - Not resetting the sum variable properly in the multiplication loop.
2. Two breakpoints are needed:
    - One before the matrix multiplication starts to check the matrices' contents.
    - One inside the multiplication loop to verify the calculation of individual products.

3. I fixed the errors by:

- Changing first[c-1][c-k] to first[c][k].
- Changing second[k-1][k-d] to second[k][d].
- Correcting the input prompt for the second matrix.
- Ensuring sum is reset in the correct scope.

Corrected executable code:

```java
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 for each product
                    for (k = 0; k < n; k++) { // k should iterate over n
                        sum = 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(); // Close the scanner
    }
}
```

7.Quadratic Probing

Program Inspection:

1. There are several errors in the program:
    - The line i + = (i + h / h--) % maxSize; has a syntax error due to the space between + and =. It should be i += (i + h / h--) % maxSize;.
    - The comment /** maxSizeake object of QuadraticProbingHashTable **/ has a typo; it should read /** Make object of QuadraticProbingHashTable **/.
    - In the get method, the increment operation h++ is incorrectly placed in the multiplication part of the hash calculation. It should be i = (i + h * h) % maxSize; and h++ should be called after this line.
2. Category A: Syntax Errors would be the most effective category to inspect, as there are clear syntax mistakes that prevent compilation.
3. The type of error not identified using program inspection includes runtime errors that may occur due to invalid input or excessive load beyond maxSize.
4. Yes, program inspection techniques are worth applying as they can help catch syntax and logical errors before running the program.

Debugging:

1. There are three errors in the program:
    - Syntax error in the insertion logic with i + = (i + h / h--) % maxSize;.
    - Typographical error in the comment for creating a new object.
    - Logical error in the get method regarding the increment of h.
2. One breakpoint is needed:
    - One before the insert method is called to check the values being passed into the hash table.
3. I fixed the errors by:

- Correcting the line i + = (i + h / h--) % maxSize; to i += (i + h / h--) % maxSize;.
- Changing the comment from /** maxSizeake object of QuadraticProbingHashTable **/ to /** Make object of QuadraticProbingHashTable **/.
- Correcting the logic in the get method to i = (i + h * h) % maxSize; and moving h++ outside the multiplication.

Corrected executable code:

```
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;
    }

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

    /** Function 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) % maxSize; // Corrected the insertion logic
            h++; // Increment h outside the calculation
        } 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; // Corrected the logic
            h++; // Increment h after this line
        }
        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");

        /** Make 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');
    }
}
```

8. Sorting Array

Program Inspection:

1. There are several errors in the program:
    - The loop condition in for (int i = 0; i >= n; i++) is incorrect; it should be i < n.
    - There is a semicolon after the first for loop, which causes the nested loop to execute independently.

    - The comparison if (a[i] <= a[j]) in the sorting logic is incorrect for ascending order; it should be if (a[i] > a[j]).
    - The output loop should print elements correctly separated, but the last element is printed without a preceding comma.
2. The category of program inspection that would be most effective is Category A: Syntax Errors, as there are clear issues that would prevent the program from compiling or running correctly.
3. The type of error not identified using program inspection includes logical errors in sorting, which may lead to incorrect ordering even if the syntax is correct.
4. Yes, program inspection techniques are worth applying as they help identify potential syntax and logical flaws before execution.

Debugging:

1. There are four errors in the program:
    - Incorrect loop condition in the first for loop.
    - A semicolon after the first for loop, causing logical errors.
    - Incorrect comparison operator in the sorting condition.
    - Output formatting issue for the last element.
2. Two breakpoints are needed:
    - One before the first for loop to check the array elements.
    - One inside the nested loop to verify the swapping logic and check if the sorting is functioning correctly.
3. I fixed the errors by:
    - Changing the loop condition from i >= n to i < n.
    - Removing the semicolon after the first for loop.
    - Changing the comparison in the sorting logic to if (a[i] > a[j]).
    - Modifying the output loop to ensure proper formatting of the printed elements.

Corrected executable code:

```
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]);
    }
}
```

9.Stack Implementation

Program Inspection:

1. There are several errors in the program:
    - In the push method, top is decremented instead of incremented. It should be top++ when pushing a value onto the stack.
    - In the pop method, top is incremented when popping a value, but the method does not handle the case of accessing the stack when it is empty properly. It should return the popped value or handle it differently.
    - In the display method, the loop condition should be i <= top instead of i > top to print all elements from the stack.
2. The category of program inspection that would be most effective is Category B: Logical Errors, as the program has logical issues in the push, pop, and display methods.
3. The type of error not identified using program inspection includes runtime errors that occur due to incorrect stack manipulation (e.g., accessing invalid stack indices).

4. Yes, program inspection techniques are worth applying as they help in identifying logical flaws before execution, preventing incorrect behavior.

Debugging:

1. There are four errors in the program:
    - Incorrectly decrementing top in the push method.
    - The pop method does not handle returning the popped value and does not check if the stack is empty correctly before accessing it.
    - Incorrect loop condition in the display method.
    - Not initializing top properly after a pop operation.
2. Two breakpoints are needed:
    - One in the push method to check the value of top after a push operation.
    - One in the pop method to verify the value being popped and the state of the stack.
3. I fixed the errors by:
    - Changing top-- to top++ in the push method to increment the top index when pushing a value.
    - Modifying the pop method to return the popped value and handle the empty stack scenario properly.
    - Changing the loop condition in the display method to i <= top.

Corrected executable code:

```
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()) {
            int poppedValue = stack[top];
            top--;
            System.out.println("Popped value: " + poppedValue);
        } 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();
    }
}
```

10.Tower of Hanoi

Program Inspection:

1. There are several errors in the program:

    - In the recursive call doTowers(topN ++, inter--, from+1, to+1), the ++ and -- operators are misused. These operators should not be used for parameters in this context.
    - The parameters from + 1 and to + 1 do not make sense in terms of the Tower of Hanoi logic. They should be passed as the original from and to parameters.
    - The output is expected to display the movement of disks correctly, but the misuse of increment/decrement operators will lead to incorrect behavior.
2. The category of program inspection that would be most effective is Category A: Syntax Errors, as there are clear syntax issues with how parameters are modified and passed in the recursive calls.
3. The type of error not identified using program inspection includes logical errors related to the incorrect state of the recursion, leading to an incorrect sequence of moves.
4. Yes, program inspection techniques are worth applying as they help identify syntax issues that could lead to runtime errors and logical inconsistencies in the output.

Debugging:

1. There are three errors in the program:
    - Incorrect usage of topN ++, inter--, from + 1, and to + 1 in the recursive function call.
    - Missing handling of the recursion base case for when topN is greater than 1.
    - The function does not return after completing the movement of disks, potentially causing an infinite loop or incorrect recursion.
2. Two breakpoints are needed:
    - One in the doTowers method to check the values of topN, from, inter, and to before the recursive calls.
    - One after the first recursive call to verify the correct movement of disks.
3. I fixed the errors by:
    - Replacing doTowers(topN ++, inter--, from+1, to+1) with doTowers(topN - 1, inter, from, to), ensuring the parameters reflect the correct logic of the Tower of Hanoi.

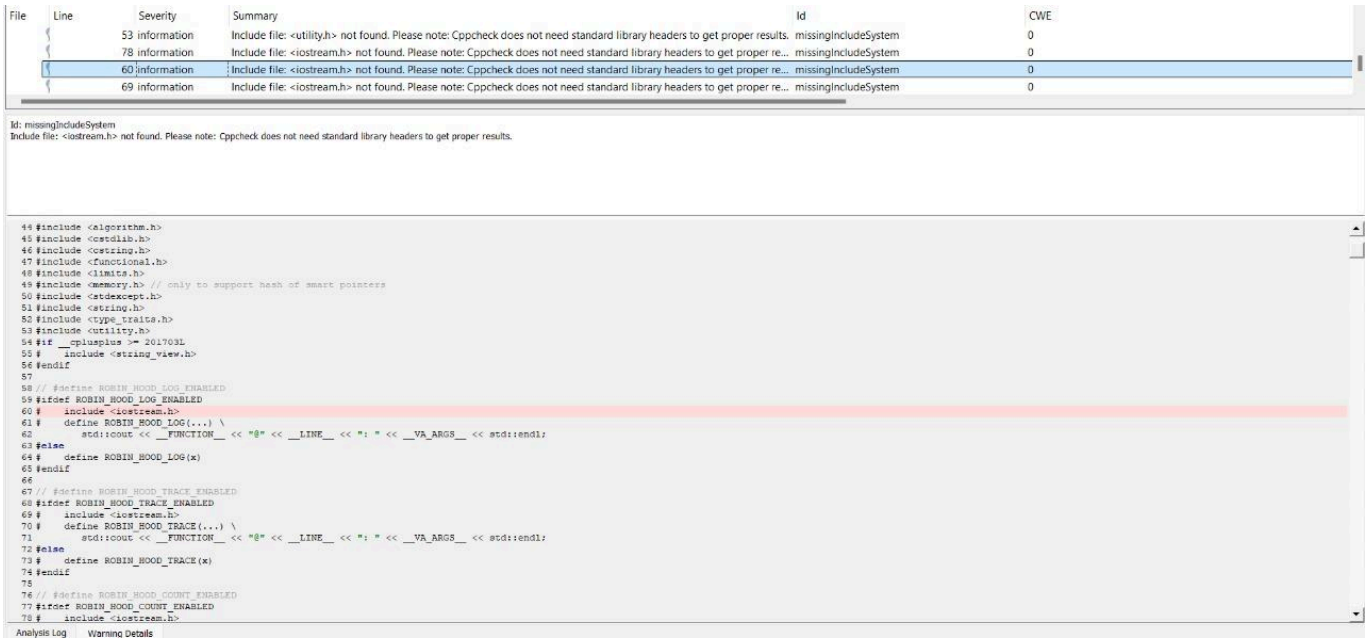
Corrected executable code:

```
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);
        }
    }
}
```

Static Analysis Tool

[robin-hood-hashing/src/include/robin_hood.h at master · martinus/robin-hood-hashing · GitHub]



File	Line	Severity	Summary	Id	CWE
	51	information	Include file: <string.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.	missingIncludeSystem	0
	52	information	Include file: <type_traits.h> not found. Please note: Cppcheck does not need standard library headers to get proper r...	missingIncludeSystem	0
	53	information	Include file: <utility.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.	missingIncludeSystem	0
	78	information	Include file: <iostream.h> not found. Please note: Cppcheck does not need standard library headers to get proper re...	missingIncludeSystem	0

Id: missingIncludeSystem
Include file: <utility.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

```
37 #define ROBIN_HOOD_H_INCLUDED
38
39 // see https://semver.org/
40 #define ROBIN_HOOD_VERSION_MAJOR 3  // for incompatible API changes
41 #define ROBIN_HOOD_VERSION_MINOR 11 // for adding functionality in a backwards-compatible manner
42 #define ROBIN_HOOD_VERSION_PATCH 5  // for backwards-compatible bug fixes
43
44 #include <algorithm.h>
45 #include <cstdlib.h>
46 #include <cstring.h>
47 #include <functional.h>
48 #include <limits.h>
49 #include <memory.h> // only to support hash of smart pointers
50 #include <stdexcept.h>
51 #include <string.h>
52 #include <type_traits.h>
53 #include <utility.h>
54 #if __cplusplus >= 201703L
55 #    include <string_view.h>
56 #endif
57
58 // #define ROBIN_HOOD_LOG_ENABLED
59 #ifdef ROBIN_HOOD_LOG_ENABLED
60 #    include <iostream.h>
61 #    define ROBIN_HOOD_LOG(...) \
62         std::cout << __FUNCTION__ << "@" << __LINE__ << ": " << __VA_ARGS__ << std::endl;
63 #else
64 #    define ROBIN_HOOD_LOG(x)
65 #endif
66
67 // #define ROBIN_HOOD_TRACE_ENABLED
68 #ifdef ROBIN_HOOD_TRACE_ENABLED
69 #    include <iostream.h>
70 #    define ROBIN_HOOD_TRACE(...) \
71         std::cout <<  FUNCTION  << "@" <<  LINE  << ": " <<  VA_ARGS  << std::endl;
```

Analysis Log | Warning Details

File	Line	Severity	Summary	Id	CWE
	49	information	Include file: <memory.h> not found. Please note: Cppcheck does not need standard library headers to get proper re...	missingIncludeSystem	0
	50	information	Include file: <stdexcept.h> not found. Please note: Cppcheck does not need standard library headers to get proper r...	missingIncludeSystem	0
	51	information	Include file: <string.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.	missingIncludeSystem	0
	52	information	Include file: <type_traits.h> not found. Please note: Cppcheck does not need standard library headers to get proper r...	missingIncludeSystem	0

Id: missingIncludeSystem
Include file: <memory.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

```
33 // OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
34 // SOFTWARE.
35
36 #ifndef ROBIN_HOOD_H_INCLUDED
37 #define ROBIN_HOOD_H_INCLUDED
38
39 // see https://semver.org/
40 #define ROBIN_HOOD_VERSION_MAJOR 3  // for incompatible API changes
41 #define ROBIN_HOOD_VERSION_MINOR 11 // for adding functionality in a backwards-compatible manner
42 #define ROBIN_HOOD_VERSION_PATCH 5  // for backwards-compatible bug fixes
43
44 #include <algorithm.h>
45 #include <cstdlib.h>
46 #include <cstring.h>
47 #include <functional.h>
48 #include <limits.h>
49 #include <memory.h> // only to support hash of smart pointers
50 #include <stdexcept.h>
51 #include <string.h>
52 #include <type_traits.h>
53 #include <utility.h>
54 #if __cplusplus >= 201703L
55 #    include <string_view.h>
56 #endif
57
58 // #define ROBIN_HOOD_LOG_ENABLED
59 #ifdef ROBIN_HOOD_LOG_ENABLED
60 #    include <iostream.h>
61 #    define ROBIN_HOOD_LOG(...) \
62         std::cout << __FUNCTION__ << "@" << __LINE__ << ": " << __VA_ARGS__ << std::endl;
63 #else
64 #    define ROBIN_HOOD_LOG(x)
65 #endif
66
```