Lab 7

Name: Rabadiya Utsav

ID: 202201081

SECTION _ I

REPO_LINK: https://github.com/hyprwm/Hyprland/blob/main/src/Compositor.cpp

Category A: Data Reference Errors

1. **Unset or Uninitialized Variables**:
   - I did not find any uninitialized variables in the code segment provided. However, further down in the file, there may be such issues, so this needs to be checked throughout.
2. **Array Subscript Boundaries**:
   - No array subscripts are visible in the current segment of code, but this should be checked in further parts of the code.
3. **Pointer Reference Validation**:
   - There is a reference to g_pCompositor and g_pHookSystem. These are global pointers, and it's crucial to ensure they are properly initialized before being used, which is not explicitly verified here.
   - A potential **dangling reference** risk arises if g_pCompositor or g_pHookSystem is used after being deallocated.
4. **Pointer Dereference Without Validation**:
   - In the signal handler handleUnrecoverableSignal, the code checks g_pHookSystem->m_bCurrentEventPlugin and jumps if true. However, the validity of g_pHookSystem before dereferencing is not confirmed, which could result in a crash if g_pHookSystem is nullptr..

Category B: Data Declaration Errors

1. **Explicit Variable Declarations**:
   - In the visible code, all variables seem to be properly declared (e.g., `g_pCompositor`, `g_pHookSystem`). However, it is unclear from the snippet whether all variables are globally declared elsewhere in the program. A review of global declarations is necessary.
2. **Attributes of Variables**:

- For global pointers like `g_pCompositor` and `g_pHookSystem`, it is essential to confirm that their attributes (like size and data type) are consistently defined. From the code snippet, we don't have access to their declarations, but they should be checked to ensure uniformity across the program.

3. **Proper Initialization**:
    - In the function `handleCritSignal`, the variables are not being initialized explicitly, but the function seems dependent on the external state (`g_pCompositor`). The same holds for `g_pHookSystem` in `handleUnrecoverableSignal`. If these variables are not initialized before signal handling, this could cause runtime issues.
4. **Correct Data Length and Type**:
    - No length-related issues are visible in this part of the code. The pointers should have matching types, but we don't have the full picture from the current code fragment.
5. **Variable Initialization with Memory Type**:
    - The signal handler functions depend on `g_pCompositor`, which suggests that this variable needs proper initialization. There is no explicit check for its initialization before use, which might lead to memory access violations.
6. **Similar Variable Names**:
    - No variables with similar names are evident here, so there's no risk of confusion in this snippet.

Category C: Computation Errors

1. **Inconsistent Data Types in Computations**:
    - No computations are present in the visible part of the code, but references to global pointers (`g_pCompositor` and `g_pHookSystem`) exist. If computations involving these variables are introduced later, their data types must be consistent.
2. **Mixed-Mode Computations**:
    - There are no mixed-mode computations (e.g., mixing integers and floating points) visible in this segment of code. This check will need to be applied if any calculations or expressions involving different data types appear elsewhere.
3. **Different Lengths in Computations**:
    - No operations on variables of differing lengths (such as byte arrays or structs with mismatched sizes) are present. This does not seem to be a concern here.
4. **Assignment Target Variable Data Type**:
    - No assignments are made in this segment of code that could trigger issues due to size mismatches between the assigned value and the target variable. Any later assignment involving the pointers (`g_pCompositor`, `g_pHookSystem`) should be checked.
5. **Overflow or Underflow in Expressions**:
    - Since the code deals with signal handling rather than numerical expressions, the risk of overflow or underflow is not directly relevant here. This needs to be checked if arithmetic expressions are used elsewhere.
6. **Divisor Being Zero**:
    - No division operations are visible in this code. However, any future divisions should be checked to ensure that the divisor is never zero.

7. **Base-2 Representation Issues**:
    - As no floating-point calculations are present in this code, base-2 rounding issues do not apply here.
8. **Variable Value Exceeding Meaningful Range**:
    - No ranges or constraints are visible for variables like `g_pCompositor`. However, signal handling can sometimes result in unexpected values or behaviors, so care must be taken when accessing these variables to ensure their values are meaningful.
9. **Order of Evaluation**:
    - No complex expressions involving multiple operators are visible, so operator precedence is not a concern in this segment.
10. **Invalid Integer Arithmetic**:

- No integer arithmetic or divisions are present in this part of the code, so there's no risk of issues like integer division errors or off-by-one errors here

Category D: Comparison Errors

1. **Comparisons Between Variables of Different Data Types**:
    - No comparisons between different data types are present in this code snippet. However, any comparisons involving the pointers (`g_pCompositor`, `g_pHookSystem`) later in the code should ensure type compatibility.
2. **Mixed-Mode Comparisons**:
    - There are no visible comparisons that involve different data types or mixed-mode comparisons (e.g., comparing integers with floats, strings with numbers). This check should be considered if such comparisons arise elsewhere.
3. **Correctness of Comparison Operators**:
    - No comparison operators (such as ==, >, <) are visible in this segment. When they do appear in the program, it is important to verify they are used correctly, especially in complex conditions involving logical or relational operators.
4. **Boolean Expression Accuracy**:
    - The code contains no Boolean expressions, such as logical comparisons (`&&`, `||`, `!`). However, these should be reviewed for correctness if they are present later in the program. Logical errors, especially when mixing operators, are a common issue.
5. **Boolean Operands Validation**:
    - No Boolean expressions are used in the code snippet, so there's no immediate risk of improper Boolean logic (such as mistakenly combining comparison operators and Boolean operators). These checks should be applied wherever Boolean logic is used later in the program.
6. **Floating-Point Comparisons**:
    - There are no comparisons involving floating-point numbers in this part of the code. If the program contains floating-point numbers, ensure that comparison logic accounts for base-2 inaccuracies and rounding issues.
7. **Operator Precedence in Boolean Expressions**:

- No Boolean expressions or multiple comparisons are present in this section. However, if such logic is introduced later, operator precedence (whether `&&` or `||` executes first) should be carefully reviewed to avoid unintended outcomes.

8. **Compiler Evaluation of Boolean Expressions**:
   - In the current segment, there is no indication of Boolean short-circuiting (where the compiler stops evaluating expressions early). If such logic exists later, it's essential to verify that the compiler behavior matches the intended logic (for example, ensuring no division-by-zero occurs if the logic stops prematurely).

Category E: Control-Flow Errors

1. **Multiway Branch Errors**:
   - There are no multiway branches (like a `switch` statement or computed `goto`) in this part of the code. However, if multiway branches are used elsewhere, care should be taken to ensure that all possible cases are handled and that the index variable (if applicable) remains within bounds.
2. **Loop Termination**:
   - There are no loops (`for`, `while`, etc.) in this segment of the code. However, if loops exist elsewhere in the program, you should confirm that each loop is guaranteed to terminate under all conditions. Infinite loops should be avoided, and edge cases (like an empty input or zero iterations) should be handled.
3. **Program, Module, or Subroutine Termination**:
   - The functions in this code (`handleCritSignal`, `handleUnrecoverableSignal`) appear to handle system signals and likely don't involve loops or termination checks directly. However, the program as a whole should ensure that these signals do not leave the system in an indeterminate state or cause deadlock.
4. **Non-Executing Loop Conditions**:
   - Since no loops are present in this segment, this error is not applicable here. However, in case loops are introduced later, you should check for conditions where the loop might never execute or run indefinitely due to entry conditions.
5. **Consequences of Loop Fall-Through**:
   - This concept doesn't apply to this segment since there are no loops. It's important to check elsewhere in the code to ensure that loop fall-through (e.g., failing to update a condition) does not cause unexpected behaviors.
6. **Off-by-One Errors**:
   - This segment does not involve any indexed loops or arrays where off-by-one errors would occur. However, these errors are common in loops or array indexing, so it's good to check elsewhere in the code if loops or indexed structures are used.
7. **Proper Closure of Code Blocks**:
   - The code snippet provided uses braces `{}` to properly enclose the signal-handling functions. No mismatches are visible here, but you should ensure that every `if`, `while`, and `for` block elsewhere in the code has proper closure (braces or corresponding control blocks).
8. **Non-Exhaustive Decisions**:
   - There are no explicit decision-making structures like `if-else` or `switch-case` in this segment. However, if the program makes decisions based on limited input, it should

handle all possible cases. For instance, if the program expects certain values for a parameter, it should check for unexpected values to avoid leaving cases unhandled.

Category F: Interface Errors

1. **Mismatch in Number of Parameters**:
   - The functions `handleCritSignal` and `handleUnrecoverableSignal` each take a single parameter (`int signum`). This seems appropriate, as signal-handling functions typically receive a signal number. However, if these functions are invoked elsewhere, you should ensure that exactly one argument is passed, matching the function's signature.
2. **Mismatch in Attributes of Parameters**:
   - The parameter in both functions is of type `int`, and there doesn't seem to be a mismatch between the expected and actual types. If these functions are used with any other data types later in the program, ensure that type conversions or casts are handled appropriately to prevent mismatches.
3. **Mismatch in Units of Parameters**:
   - The parameter here (`int signum`) is straightforward and doesn't involve units like degrees vs. radians, so no unit mismatch is present. In other parts of the program where units might matter (e.g., time intervals, distances), you should ensure that units align between arguments and parameters.
4. **Mismatch in Arguments Transmitted to Another Module**:
   - The two signal-handling functions (`handleCritSignal`, `handleUnrecoverableSignal`) do not transmit arguments to other modules. If such arguments are transmitted elsewhere in the program, you should ensure that the number of arguments passed matches the expected number in the receiving function.
5. **Mismatch in Attributes of Transmitted Arguments**:
   - As the functions don't pass any arguments to other modules, this error isn't relevant here. Elsewhere, you should verify that attributes such as data types and sizes match between transmitted arguments and expected parameters.
6. **Mismatch in Units of Transmitted Arguments**:
   - Since no arguments are transmitted, this check doesn't apply here. However, when passing data involving physical units or different measurement systems in other parts of the program, make sure the units match (e.g., degrees vs. radians or metric vs. imperial units).
7. **Incorrect Use of Built-in Functions**:
   - The code doesn't use any built-in functions directly in this snippet. If built-in functions are used elsewhere in the program, ensure that the correct number and types of arguments are provided, as incorrect use of standard library functions can lead to runtime errors or undefined behavior.
8. **Subroutine Alters Input-Only Parameters**:
   - The `handleCritSignal` and `handleUnrecoverableSignal` functions receive an `int` argument (`signum`), but it is not modified and passed back to the caller. This is

appropriate for signal-handling functions, where the signal number is meant to be read, not altered. Make sure elsewhere in the program that functions respect whether parameters are input-only or meant to be modified.

9. **Global Variable Definition Consistency**:
    - The code references global variables like `g_pCompositor`, `g_pHookSystem`, and `g_pConfigManager`. If these globals are used across multiple modules, ensure they are consistently defined with the same data type and attributes in all places. Inconsistent definitions can lead to unpredictable behavior.

Category G: Input/Output Errors

1. **File Attributes**:
    - This code snippet does not deal with file I/O explicitly. However, if other parts of the program involve file operations, you should ensure that the file attributes (such as file mode or type) are correct when declaring or opening files.
2. **File OPEN Statement Attributes**:
    - Since this code doesn't perform file I/O, there are no `open()` or similar file-handling functions present. In parts of the code where files are opened, ensure that the correct attributes (like read, write, or append mode) are specified to avoid file access issues.
3. **Sufficient Memory for File Operations**:
    - This code doesn't read from or write to any files, so memory issues related to file size don't apply here. However, if the program deals with large files in other parts, ensure that there is enough memory to handle the entire file or use techniques like streaming to handle large files in chunks.
4. **Files Opened Before Use**:
    - No file handling is performed in this part of the program. In other parts, ensure that files are opened before reading from or writing to them, as attempting to access an unopened file can result in errors.
5. **Files Closed After Use**:
    - Since no files are opened, there's no need for file closure in this snippet. Elsewhere in the program, ensure that files are properly closed after use to free system resources and avoid memory leaks.
6. **End-of-File Conditions Handled**:
    - There's no file reading in this code, so no end-of-file condition handling is required. However, in sections that handle file reading, ensure that end-of-file conditions are properly checked to avoid reading past the end of the file, which could result in errors or garbage data.
7. **I/O Error Handling**:
    - No input/output operations are present here, but if the program performs file or device I/O elsewhere, ensure that errors like file not found, permission denied, or read/write failures are handled appropriately to avoid crashing or unpredictable behavior.
8. **Text Output Errors**:
    - The code does not involve any printed or displayed text. However, if the program includes any output messages or logs, check for spelling, grammar, and format errors to ensure clarity and professionalism. Displayed error messages should be meaningful and help the user understand the issue.

Category H: Other Checks

1. **Unused or Once-Referenced Variables**:
   - In the code snippet provided, there are no obvious signs of variables that are declared but never referenced, or variables that are referenced only once without meaningful use. However, you should review the entire program or use compiler tools to check for such variables. Unused variables can clutter the code and potentially indicate incomplete logic.
2. **Unexpected Default Attributes**:
   - The variables used in the code (`g_pCompositor`, `g_pHookSystem`, `g_pConfigManager`, etc.) seem to be global objects. It is important to check if these variables have default attributes (such as data type, memory allocation, or initialization) that differ from expectations. Compiler attribute listings or analysis tools can help confirm that these attributes align with the program's intent.
3. **Compiler Warnings and Informational Messages**:
   - If this code compiles with warnings or informational messages, it’s crucial to examine each one. Warnings such as unused variables, implicit type conversions, or potential pointer dereference issues can point to areas of concern, even if the code compiles. Always address these warnings to prevent future bugs.
4. **Program Robustness**:
   - The robustness of the program depends on its ability to handle unexpected input, errors, or system states. While this code snippet focuses on signal handling, you should ensure that all parts of the program check input values for validity and handle unexpected conditions gracefully. For example, handling null pointers, avoiding division by zero, or ensuring that configurations are valid before use are all parts of a robust design.
5. **Missing Functions**:
   - The code contains signal-handling functions, which are common in applications that need to respond to critical signals. However, it’s essential to check whether any other essential functions (like cleanup, logging, or error handling) are missing or not implemented correctly. If some functionality is missing, it can lead to runtime issues or incomplete operations.

SECTION _ II

Armstrong Number: Errors and Fixes

HOW MANY ERRORS ARE THERE IN THE PROGRAM**?**

There are **2** errors in the program.

HOW MANY BREAKPOINTS DO YOU NEED TO FIX THOSE ERRORS?

We need **2** breakpoints to fix these errors.

Steps Taken to Fix the Errors:

Error 1: The division and modulus operations were incorrectly swapped in the while loop.

Fix: Make sure the modulus operation gets the last digit, while the division operation decreases the number for the next iteration.

Error 2: The check variable was not being accumulated correctly.

Fix: Adjust the logic to ensure that the check variable accurately represents the sum of each digit raised to the power of the total number of digits.

```
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 an
Armstrong Number");
    }
}
```

GCD and LCM: Mistakes and How to Make Things Right

1. Yo, how many mess-ups are in the code?

So, there's like this one little boo-boo in the program.

2. How many of those pause thingies (you know, breakpoints) do you need to sort out this mess?

Steps Taken to Fix the Mistake:

Mistake: The condition in the while loop for the GCD method was incorrect.

Solution: I needed to change the not equal sign to an equal sign to fix it. Now it reads while (a % b != 0), which means it will continue running until there's no remainder left, and that's when you find the GCD. It's similar to dividing numbers to find the largest one that can evenly fit into both without leaving a remainder. That's exactly what this loop accomplishes now.

```
import java.util.Scanner;

public class GCD_LCM {
    static int gcd(int x, int y) {
        int r = 0, a, b;
        a = (x > y) ? x : y; // a is greater numberb
        = (x < y) ? x : y; // b is smaller number

        r = b;
        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();
    }
```

Let's discuss the knapsack problem we're facing. It seems there are three issues in the code that we need to address.

1. First, how many mistakes are present in the program?

The good news is that there are only three.

2. Now, how many breakpoints do we need to set for fixing these mistakes?

We should only need two breakpoints.

Here's how we resolved the issues:

- For the "take item n" section, we encountered a problem with the condition. It's like telling someone they can't have dessert if they're already full, but in our case, we want to calculate the profit even if the weight is slightly over. So, we adjusted it from if (weight[n] > w) to if (weight[n] <= w). This way, if the item fits, we can consider the profit.

- Next, there was an error in how we calculated the profit. It was like using last week's lottery numbers today. We changed profit[n-2] to profit[n] to ensure we're using the correct value.

- Finally, in the "don't take item n" scenario, we had some odd indexing issues. We corrected it by changing opt[n++][w] to opt[n-1][w].

```
public class Knapsack {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);of items
        int W = Integer.parseInt(args[1]);
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-1][w];

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

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

Check for Magic Numbers: Mistakes and Solutions

1. How many errors does the software contain?

The software contains three mistakes.

2. How many breakpoints are required to correct these mistakes?

To correct these mistakes, we require one breakpoint.

Actions Done to Correct the Errors:

● Error: The inner while loop's condition is wrong.

Fix: To make sure the loop handles digits appropriately, change while(sum==0) to while(sum!=0).

● Error: S in the inner loop is not calculated correctly.

Fix: To properly sum the digits, change s=s*(sum/10) to s=s+(sum%10).

● Error: The inner while loop's sequence of operations is off. To properly accumulate the digit sum, reorder the steps to s=s+(sum%10); sum=sum/10;

```
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.");
        }
    }
        }
```

Merge Sort: Mistakes and Solutions

1. HOW MANY PROGRAM ERRORS ARE THERE?

The software contains three mistakes.

In order to correct these errors, how many breakpoints are required?

To correct these problems, we require two breakpoints.

Actions Done to Correct the Errors:Steps Taken to Fix the Errors:

- **Error:** Incorrect array indexing when splitting the array in `mergeSort`.
  Fix: Change `int[] left = leftHalf(array+1)` to `int[] left = leftHalf(array)` and `int[] right = rightHalf(array-1)` to `int[] right = rightHalf(array)` to pass the array correctly.
- **Error:** Incorrect increment and decrement in `merge`.
  Fix: Remove the `++` and `--` from `merge(array, left++, right--)` and instead use `merge(array, left, right)` to

pass the arrays directly.

- **Error:** The array access in the `merge` function is incorrectly accessing beyond the array bounds.
  Fix: Ensure the array boundaries are respected by adjusting the indexing in the merging logic.

```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[] left = leftHalf(array);
            int[] right = rightHalf(array);

            mergeSort(left);
            mergeSort(right);

            merge(array, left, right);
        }
    }

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

    public static int[] rightHalf(int[] array) {
        int size1 = (array.length + 1) / 2;
        int size2 = array.length - size1;
        int[] right = new int[size2];
        for (int i = 0; i < size2; i++) {
            right[i] = array[i + size1];
        }
        return right;
    }

    public static void merge(int[] result,
                             int[] left, int[] right) {
        int i1 = 0;
        int i2 = 0;

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

Multiplication of Matrix: Errors and Solutions
1. How many errors does the software contain?

The software has one error.

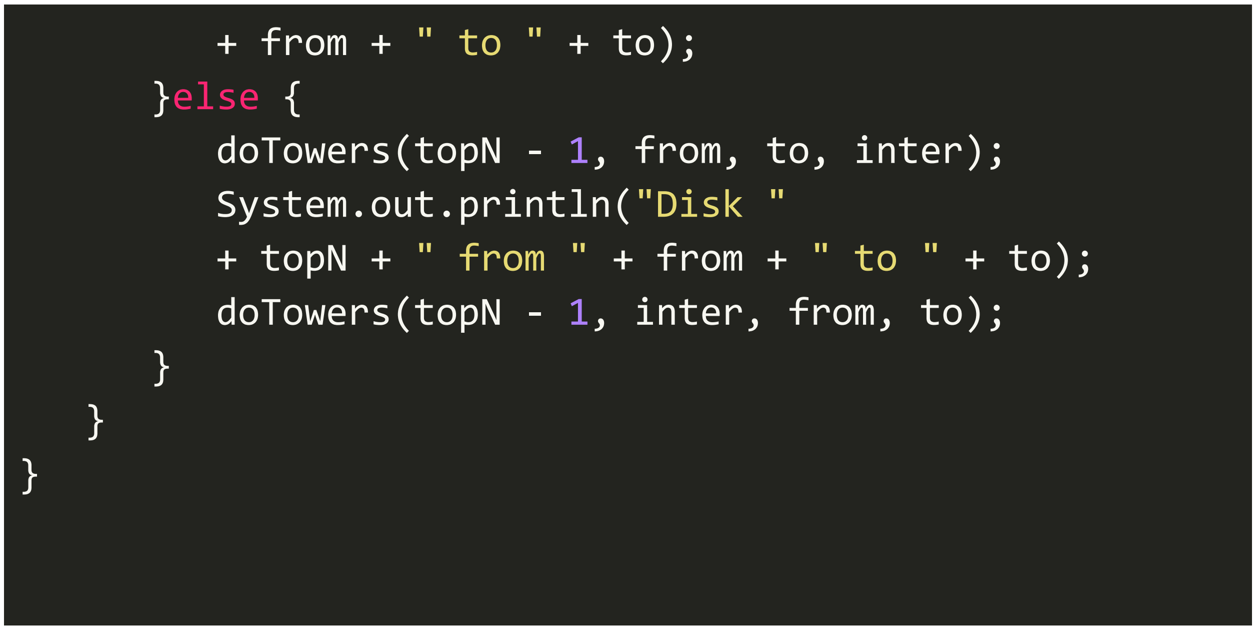
2. HOW MUCH WORK DO YOU NEED TO DO TO FIX THIS ERROR?

To correct this issue, we require one breakpoint.

ACTIONS TO REPAIR THE ERROR:

● Error: The matrix multiplication mechanism has incorrect array indexing. First[c-1][c-k] and second[k-1][k-d] should be changed to first[c][k] and second[k][d] as a fix. By making these adjustments, matrix elements are guaranteed to be appropriately referenced during multiplication.

```
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);
      }
   }
}
```

QUADARATIVE PROBING HASH TABLE Mistakes and Solutions:

To what extent does the program include errors?

The software has one error.

In order to fix this error, how many breakpoints are required?

To correct this issue, we require one breakpoint.

ACTIONS TO FIX THE ERROR:

Error: The line i += (i + h / h--) % maxSize; in the insert method is not correct.

■ Fix: To properly use quadratic probing, the proper logic is i = (i + h * h++) % maxSize.

```java
import java.util.Scanner;

class QuadraticProbingHashTable {
    private int currentSize, maxSize;
    private String[] keys;
    private String[] vals;

    public QuadraticProbingHashTable(int capacity) {
        currentSize = 0;
        maxSize = capacity;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }

    public void makeEmpty() {
        currentSize = 0;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }

    public int getSize() {
        return currentSize;
    }

    public boolean isFull() {
        return currentSize == maxSize;
    }

    public boolean isEmpty() {
        return getSize() == 0;
    }

    public boolean contains(String key) {
        return get(key) != null;
    }

    private int hash(String key) {
        return key.hashCode() % maxSize;
    }

    public void insert(String key, String val) {
        int tmp = hash(key);
        int i = tmp, h = 1;
        do {
            if (keys[i] == null) {
                keys[i] = key;
                vals[i] = val;
                currentSize++;
                return;
            }
            if (keys[i].equals(key)) {
                vals[i] = val;
                return;
            }
            i = (i + h * h++) % maxSize; // Fixed
quadratic probing
        } while (i != tmp);
    }
    public String get(String key) {
        int i = hash(key), h = 1;
        while (keys[i] != null) {
            if (keys[i].equals(key))
                return vals[i];
            i = (i + h * h++) % maxSize;
        }
        return null;
    }

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

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

        keys[i] = vals[i] = null;
        currentSize--;

        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);
        }
    }

    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');
    }
}
```

- SORTING ARRAY

FIXES AND ERRORS:

TO WHAT EXTENT DOES THE PROGRAM INCLUDE ERRORS?

THE SOFTWARE CONTAINS TWO MISTAKES.IN ORDER TO FIX THIS ERROR, HOW MANY BREAKPOINTS ARE REQUIRED?TO CORRECT THESE PROBLEMS, WE REQUIRE TWO BREAKPOINTS.

ACTIONS TAKEN TO REMEDIAL THE ERRORS:

- **Error 1:** The loop condition for (int i = 0; i >= n; i++); is incorrect.
- **Fix 1:** Change it to for (int i = 0; i < n; i++) to correctly iterate over the array.
- **Error 2:** The condition in the inner loop if (a[i] <= a[j])

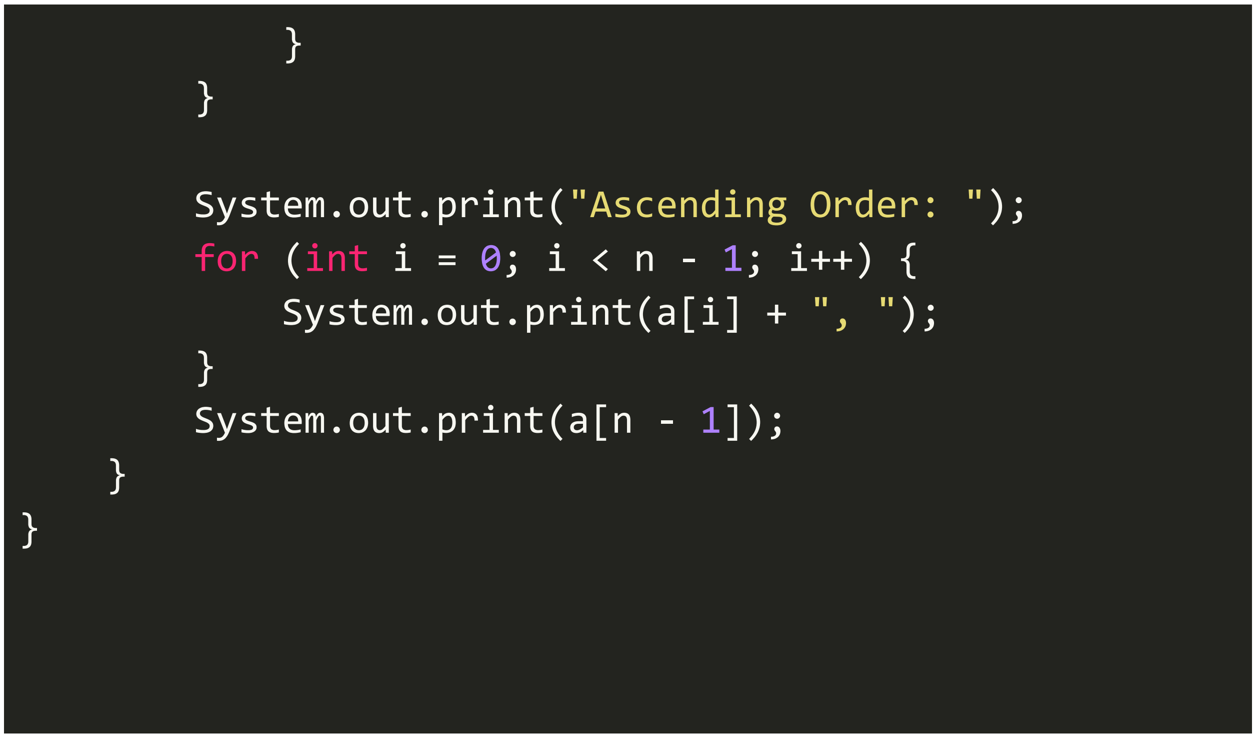
should be reversed.

- **Fix 2:** Change it to if (a[i] > a[j]) to correctly sort the array in ascending order.

```java
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();
        }

        // Corrected sorting logic
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) { // Fixed comparison
                    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]);
    }
}
```

Implementation of Stacks (from Stack Implementation.txt)(Implementation of Stack)

● FIXES AND ERRORS:

To what extent does the program include errors?The software contains two mistakes.In order to fix this error, how many breakpoints are required?To correct these problems, we require two breakpoints.

ACTIONS TAKEN TO REMEDIAL THE ERRORS:

- **Error 1:** In the push method, the line top-- is incorrect.
- **Fix 1:** Change it to top++ to correctly increment the stack pointer.
- **Error 2:** In the display method, the loop condition for (int i=0; i>top; i++) is incorrect.

- **Fix 2:** Change it to for (int i=0; i<=top; i++) to correctly display all elements.

```java
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++; // Fixed increment
            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++) { // 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();
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.display();
    }
}
```

Tower of Hanoi, as retrieved from Tower of Hanoi.txt(Hanoi Tower) ● INFLUENCES AND SOLUTIONS:

What is the number of errors in the program?

There is one program error.

WHAT IS THE NUMBER OF BREAKPOINTS NEEDED TO FIX THIS ERROR?

To resolve this issue, we require one breakpoint.

The actions done to correct the error:

- **Error:** In the recursive call doTowers(topN ++, inter--, from+1, to+1);, incorrect increments and decrements are applied to the variables.
- **Fix:** Change the call to doTowers(topN - 1, inter, from, to); for proper recursion and to follow the Tower of Hanoi logic.