TASK 1: Program Inspection:

Program Inspection of Tower of Hanoi Code Fragment:

1. **How many errors are there in the program? Mention the errors you have identified.**
   Errors identified: 3
   - Error 1: Incorrect increment operation (topN++)
     - Category: Computation Error
     - The topN++ increment inside the recursive call is incorrect because ++ performs a post-increment, meaning it will increment topN after using it in the recursive call, which is not intended here. Instead, it should just be topN - 1 like the first recursive call.
   - Error 2: Improper decrement (inter--) and increment (from+1, to+1) on char variables
     - Category: Data Reference Error
     - Decrementing and incrementing characters in this context (i.e., inter--, from+1, to+1) is logically incorrect. These characters represent the tower labels ('A', 'B', 'C'), and arithmetic operations on them would yield unintended results.
   - Error 3: Missing semicolon after the second recursive call
     - Category: Syntax Error
     - In the line doTowers(topN ++, inter--, from+1, to+1), the statement is missing a semicolon at the end, causing a compilation error.

2. **Which category of program inspection would you find more effective?Most effective category: Computation and Data Reference Errors**
    - Reason: These categories helped in identifying the incorrect increment/decrement operations, which are key logical errors in this recursive problem. Without correcting these, the algorithm wouldn't perform the Tower of Hanoi logic correctly.

3. **Which type of error you are not able to identify using the program inspection?**
   Missed error type: Runtime performance issues and deep recursion limits
    - Reason: While this program may run correctly after fixing the identified errors, performance-related issues (such as running out of stack space for deep recursions if the number of disks is large) would not be identifiable through static inspection alone. These would need to be caught through runtime testing.

4. **Is the program inspection technique worth applying?**
   Applicability: Yes, the program inspection technique is worth applying.
    - Reason: Through static inspection, logical and syntactical errors were identified early in the recursive logic, saving time on debugging. This technique ensures that common errors are caught before runtime, though it should be paired with dynamic testing for performance and runtime behavior.

Program Inspection of Stack Implementation:

1. **How many errors are there in the program? Mention the errors you have identified.**
   Errors identified: 3
   - Error 1: Incorrect operation in push method (top-- instead of top++)
     - Category: Logic Error
     - In the push method, the top-- should be top++. Since the stack starts with top initialized to -1, each time a new element is pushed, the top should increment to indicate the next position in the stack.
   - Error 2: Incorrect condition in the display method (wrong loop condition)
     - Category: Logic Error
     - The loop condition for(int i=0;i>top;i++) is incorrect. The loop should run from i = 0 to i <= top, meaning i < top + 1 for a valid display of stack elements.
   - Error 3: Incorrect operation in pop method (top++ instead of top--)
     - Category: Logic Error
     - When pop is called, the value of top should be decremented (top--) to reflect the removal of an element from the stack. The current top++ will increase the position instead of decreasing it.

2. **Which category of program inspection would you find more effective?**
   Most effective category: Logic Errors
   - Reason: The key issues in this program are related to incorrect logic in the push, pop, and display methods. Program inspection

focusing on logical flow, especially with regard to array indices and boundary conditions, is crucial for detecting these errors.

3. **Which type of error you are not able to identify using the program inspection?**
   Missed error type: Memory management or array overflow issues in dynamic stacks
   - Reason: Since this stack has a fixed size, memory issues may not arise in this context. However, if a dynamically sized stack were implemented, errors related to memory management (such as resizing arrays or handling large inputs) would not be detectable through static inspection alone.

4. **Is the program inspection technique worth applying?**
   Applicability: Yes, the program inspection technique is worth applying.
   - Reason: The inspection process identified critical logical errors in the push, pop, and display methods, which would prevent the stack from functioning properly. By addressing these errors early, debugging and runtime issues are minimized, ensuring the stack performs as expected.

Program Inspection of Array Sorting (Ascending Order):

1. **How many errors are there in the program? Mention the errors you have identified.**
   Errors identified: 4
   - Error 1: Incorrect class name (Ascending _Order)
     - Category: Syntax Error
     - The class name Ascending _Order contains a space between Ascending and _Order. This space is not allowed in Java class names. The class name should be AscendingOrder.
   - Error 2: Incorrect loop condition in outer loop (should be i < n, not i >= n)
     - Category: Logic Error
     - The condition in the outer loop for (int i = 0; i >= n; i++) is incorrect. The condition should be i < n, as the loop should iterate until i is less than n to avoid skipping elements.
   - Error 3: Semicolon after the first for loop
     - Category: Logic Error
     - The semicolon after the outer for loop for (int i = 0; i >= n; i++); causes the loop body to be skipped. The semicolon should be removed.
   - Error 4: Incorrect sorting condition (<= should be >)
     - Category: Logic Error
     - The condition if (a[i] <= a[j]) should be if (a[i] > a[j]) to sort the array in ascending order. The current condition swaps elements when a[i] is less than or equal to a[j], which will not sort the array correctly.

2. **Which category of program inspection would you find more effective?**

   Most effective category: Logic Errors

   - Reason: The primary issues in this code are logical, including the loop conditions and the comparison used for sorting. Detecting these errors early through inspection will ensure the sorting logic works as intended.

3. **Which type of error you are not able to identify using the program inspection?**

   Missed error type: Performance or optimization issues for large arrays

   - Reason: Program inspection may not reveal performance-related issues or suggest optimizations for sorting algorithms (e.g., using a more efficient algorithm like quicksort or mergesort). These are typically identified through testing or profiling with large datasets.

4. **Is the program inspection technique worth applying?**

   Applicability: Yes, the program inspection technique is worth applying.

   - Reason: The inspection revealed critical syntax and logic errors that prevent the code from running correctly and sorting the array as intended. Identifying and fixing these errors early improves the reliability of the program.

Program Inspection for the Quadratic Probing Hash Table:

1. **How many errors are there in the program? Mention the errors you have identified.**
   Errors identified: 3
   - Error 1: Syntax error in the insert method (i += (i + h / h--) % maxSize;)
     - Category: Syntax Error
     - The line i += (i + h / h--) % maxSize; has a spacing issue around +=. It should be written as i = (i + h * h++) % maxSize;. Additionally, the use of h / h-- does not make sense. In quadratic probing, the term should be incremented by the square of h. Hence, it should be h * h++ for quadratic probing.
   - Error 2: Improper rehashing in the remove method
     - Category: Logic Error
     - The line currentSize--; appears twice in the remove method, causing the current size to decrease incorrectly by 2 when a key is removed. The second currentSize-- should be removed.
   - Error 3: Inconsistent hashing function
     - Category: Logic Error
     - The hash(String key) function uses key.hashCode() % maxSize, but this can result in negative indices due to how hash codes work in Java. To avoid negative indices, the modulo operation should be adjusted as Math.abs(key.hashCode()) % maxSize.

2. **Which category of program inspection would you find more effective?**
   Most effective category: Logic Errors

- Reason: The primary issues in this code revolve around logic, such as errors in the insert and remove methods related to rehashing and calculating the new positions. Catching logic errors early can help prevent improper key handling, especially when inserting and deleting elements in the hash table.

3. **Which type of error you are not able to identify using the program inspection?**
   Missed error type: Performance optimization and collisions handling in practical scenarios.
   - Reason: Program inspection may not easily reveal performance bottlenecks (e.g., high clustering) or poor handling of frequent collisions in certain key distributions. These issues typically surface during testing with large and specific datasets.

4. **Is the program inspection technique worth applying?**
   Applicability: Yes, program inspection is worth applying.
   - Reason: The inspection helped catch crucial logic and syntax errors that would prevent the hash table from functioning properly. Correcting these early improves the robustness and ensures correct behavior for basic operations like insertions and deletions.

Program Inspection for multiply matrics

1. **How many errors are there in the program? Mention the errors you have identified.**

- Errors Identified: 4

- Uninitialized/Incorrect Variable Access:
  - The program uses first[c-1][c-k] and second[k-1][k-d] in the multiplication logic, which can lead to ArrayIndexOutOfBoundsException when c or k is 0. Array indices in Java are zero-based, so the valid range for c and k should start from 0.

- Logic Error in Matrix Multiplication:
  - The formula for matrix multiplication is incorrectly implemented. The correct access for multiplication should be first[c][k] and second[k][d] instead of first[c-1][c-k] and second[k-1][k-d]. This results in incorrect product values.
- Potential Logic Error with Initialization of sum:
  - The sum variable is initialized to 0 at the beginning of the multiplication loop, which is correct; however, it might cause confusion when the logic fails due to the wrong index access.
- Output Format:
  - The output is printed without a proper heading for the second matrix input and lacks a clear separation between input sections, which could confuse users.

2. **Which category of program inspection would you find more effective?**

- Effective Category:
  - The Control-Flow Errors category was effective in this case. Issues like ensuring loops terminate correctly and checking that array indices are within bounds are crucial for matrix multiplication operations.

3. **Which type of error you are not able to identify using the program inspection?**

- Errors Not Identified:
    - Logic Errors in Mathematical Computation: While program inspections can identify syntax errors and potential runtime exceptions, they may not catch logical errors where the implemented algorithm does not perform the intended mathematical operations correctly.

4. **Is the program inspection technique worth applicable?**

- Applicability of Program Inspection:
    - Yes, the program inspection technique is worth applying. It helps identify a range of potential issues before runtime, improving code quality and reliability. Although it may not catch all logical errors, it provides a framework for systematic error checking, which is essential for complex operations like matrix multiplication.

Program Inspection for MergeSort

1. **How many errors are there in the program? Mention the errors you have identified.**

- Errors Identified: 3
    - Incorrect Array Manipulation:
        - In the mergeSort method, the lines int[] left = leftHalf(array+1); and int[] right = rightHalf(array-1); are incorrect. You cannot add or subtract from an array directly in Java. Instead, the code should pass a subarray of the original array to these methods.
    - Incorrect Merge Method Call:

        - The line merge(array, left++, right--); contains incorrect usage of the ++ and -- operators. These operators cannot be used on array references in this context. Instead, it should just be merge(array, left, right);.
    - Merge Method Logic:
        - The merge function should operate on a temporary array (the result array) to hold the merged values. The merge function is defined correctly, but it's not utilized properly in the mergeSort method.

2. Which category of program inspection would you find more effective?

- Effective Category:
    - Control-Flow Errors: This category is particularly effective here since it deals with the logical flow of the program, especially in how the recursive calls and array handling are managed.

3. Which type of error you are not able to identify using the program inspection?

- Errors Not Identified:
    - Semantic Errors in Array Handling: Program inspections can miss semantic errors like those arising from incorrect manipulation or interpretation of array indices, especially when it comes to passing array sections or creating subarrays.

4. Is the program inspection technique worth applicable?

- Applicability of Program Inspection:
    - Yes, the program inspection technique is worth applying. It aids in identifying structural issues and potential run-time exceptions,

which ultimately contributes to higher code quality. However, it's important to supplement it with other testing methods to catch logical and semantic errors.

Program Inspection for Magic Numbers

1. **How many errors are there in the program? Mention the errors you have identified.**

- Errors Identified: 4
    - Logic Error in Summation:
        - The condition in the inner while loop while(sum==0) is incorrect. It should be while(sum > 0) because you want to extract digits from the number as long as there are digits left.
    - Incorrect Digit Extraction:
        - The line s=s*(sum/10); is incorrect. It should be s += sum % 10; to properly accumulate the digits.
    - Missing Semicolon:
        - There is a missing semicolon at the end of the line sum=sum%10.
    - Overall Logic:
        - The logic used to determine the magic number is flawed. The correct approach involves repeatedly summing the digits until a single-digit number is obtained, and then checking if that number is 1.

2. **Which category of program inspection would you find more effective?**

- Effective Category:

- Logic Errors: This category is particularly effective as it focuses on the flow and correctness of the algorithm being implemented, which is crucial for correctly determining whether a number is a magic number.

3. **Which type of error you are not able to identify using the program inspection?**

- Errors Not Identified:
  - Input Validation Errors: Program inspections may miss input validation issues, such as non-integer inputs or negative numbers, which could cause the program to behave unexpectedly.

4. **Is the program inspection technique worth applicable?**

- Applicability of Program Inspection:
  - Yes, the program inspection technique is worth applying. It helps identify structural and logical errors, leading to more robust code. However, it should be supplemented with testing for various input cases, including edge cases.

Program Inspection for Knapsack

1. **How many errors are there in the program? Mention the errors you have identified.**

- Errors Identified: 4
  - Increment Error: The line int option1 = opt[n++][w]; incorrectly increments n, causing it to exceed the intended range. It should be int option1 = opt[n][w]; to access the correct opt values without modifying n.
  - Profit Indexing Error: The line if (weight[n] > w) option2 = profit[n-2] + opt[n-1][w-weight[n]]; has a logic error in the profit

indexing. It should use profit[n] instead of profit[n-2], as we want to include the current item's profit.
    - Weight Check Logic Error: The condition if (weight[n] > w) should be inverted to if (weight[n] <= w) to correctly check if the item can be included in the knapsack.
    - Array Initialization: The arrays for profit and weight should be initialized starting from index 1 and accessed accordingly, but the initialization should ensure no items are accidentally counted if not generated correctly.

2. **Which category of program inspection would you find more effective?**

- Effective Category:
    - Logic Errors: This category is particularly effective as it deals with the correctness of algorithm flow and ensures that the logic aligns with the problem requirements. In this case, checking conditions and indexing correctly is critical for the expected behavior.

3. **Which type of error you are not able to identify using the program inspection?**

- Errors Not Identified:
    - Performance Issues: While program inspection can highlight logic and syntax errors, it may not effectively identify performance issues, such as inefficiencies in memory usage or runtime complexity, particularly in larger input sizes.

4. **Is the program inspection technique worth applicable?**

- Applicability of Program Inspection:

- Yes, the program inspection technique is worthwhile. It helps identify structural and logical errors, which are essential for ensuring the program functions as intended. However, it should be complemented with testing, especially for edge cases and performance.

Program Inspection for GCD and LCM of two numbers

1. **How many errors are there in the program? Mention the errors you have identified.**

- Errors Identified: 3
  - GCD Logic Error: In the gcd method, the condition in the while loop should be while (a % b != 0) instead of while (a % b == 0). The current condition results in an infinite loop if a is not divisible by b at the start.
  - LCM Logic Error: In the lcm method, the logic to find the LCM is incorrect. The condition if (a % x != 0 && a % y != 0) should be changed to if (a % x == 0 && a % y == 0) to correctly return the LCM when both conditions are satisfied.
  - Unnecessary Variable: The variable r in the gcd method is not needed. You can return b directly once the loop exits.

2. **Which category of program inspection would you find more effective?**

- Effective Category:
  - Logic Errors: This category is highly relevant because it deals with the correctness of the algorithms used to calculate GCD and LCM. Ensuring that mathematical operations are performed correctly is crucial for the expected output.

3. **Which type of error you are not able to identify using the program inspection?**

- Errors Not Identified:

- Performance Issues: While program inspection can identify logical and syntactical errors, it may not highlight performance concerns such as inefficient algorithms, particularly with larger numbers.

4. **Is the program inspection technique worth applicable?**

- Applicability of Program Inspection:
  - Yes, program inspection is a valuable technique as it helps catch logical and syntactical errors before runtime. It ensures that the code behaves as expected. However, it should be combined with testing to verify correctness across various input scenarios.

Program Inspection for Armstrong numbers

1. **How many errors are there in the program? Mention the errors you have identified.**

- Errors Identified: 3
  - Incorrect Calculation of Remainder: In the loop, the calculation of remainder should be done using num % 10 instead of num / 10. The current implementation incorrectly computes the next digit and does not correctly extract the digits of the number.
  - Updating the Number Incorrectly: The line num = num % 10; should be replaced with num = num / 10; to reduce the number correctly for the next iteration of the loop.
  - Inconsistent Output Format: The output statement should use "an" instead of "a" before "Armstrong Number" when referring to "Armstrong Number" because it starts with a vowel.

2. **Which category of program inspection would you find more effective?**

- Effective Category:
    - Logic Errors: This category is essential here since the Armstrong number check involves mathematical calculations and correctly iterating through the digits of a number. Identifying logical flaws is critical to ensure the correct behavior of the program.

3. **Which type of error you are not able to identify using the program inspection?**

- Errors Not Identified:
    - Edge Cases: While program inspection can reveal logical and syntactical errors, it may not fully capture potential issues with edge cases, such as negative numbers or non-integer inputs.

4. **Is the program inspection technique worth applicable?**

- Applicability of Program Inspection:
    - Yes, program inspection is valuable as it helps identify logical and syntactical errors before runtime. However, it should be used alongside unit tests to verify the program's correctness across a variety of input scenarios.

Use of chatGPT for creating text

TASK 2 : II. CODE DEBUGGING: Debugging is the process of localizing, analyzing, and removing suspected errors in the code (Java code given in the .zip file)

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

Identified Errors

1. Incorrect Arithmetic Operations:
    - In the line doTowers(topN ++, inter--, from+1, to+1), the use of post-increment (topN++) and post-decrement (inter--) is not needed and incorrect.
    - Additionally, modifying the character variables (from + 1 and to + 1) will convert them to integers, which is not appropriate for this context.

Corrections Made

1. Recursive Calls:
    - Changed the recursion call from doTowers(topN ++, inter--, from+1, to+1) to simply doTowers(topN - 1, inter, from, to). This keeps the correct values for the characters and uses the right logic for the recursive calls.

Debugging Submission

1. Errors Identified:
    - Total of 1 major issue related to incorrect arithmetic operations in the recursive calls.
2. Breakpoints Needed:

    - Set breakpoints at the beginning of the doTowers method to examine how the topN, from, inter, and to variables change during recursion.
3. Steps Taken to Fix Errors:
    - Removed unnecessary post-increment and post-decrement in the recursive calls.
    - Passed the character variables correctly without modification.
4. Complete Executable Code:
    - The fixed code provided above can be submitted as your executable code.

Fixed Code

```
// 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 {
```

```
        // Recursive call to move (n-1) disks from 'from' to 'inter' via 'to'
        doTowers(topN - 1, from, to, inter);
        // Move the nth disk
        System.out.println("Disk " + topN + " from " + from + " to " + to);
        // Recursive call to move (n-1) disks from 'inter' to 'to' via 'from'
        doTowers(topN - 1, inter, from, to);
    }
  }
}
```

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
```

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

Identified Errors

1. Push Method (Line 18):
    - Error: top-- was incorrectly used; it should be top++ to increment the top index when adding a new element to the stack.
2. Pop Method (Line 26):
    - Error: top++ was incorrectly used; it should be top-- to decrement the top index when removing an element from the stack.
3. Display Method (Line 35):
    - Error: The loop condition i > top was incorrect; it should be i <= top to iterate through the stack elements properly.

Debugging Submission

1. Errors Identified:
    - Total of 3 errors.
    - Errors were in the push, pop, and display methods as described above.
2. Breakpoints Needed:
    - You may need breakpoints at the start of each method (push, pop, display) to examine how values change as the stack is manipulated.
3. Steps Taken to Fix Errors:
    - Modified the logic in the push and pop methods to correctly adjust the top index.
    - Corrected the loop condition in the display method to properly iterate through the stack.

4. Complete Executable Code:
    - You can submit the fixed code provided above as your executable code.

Fixed 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++; // Increment top before adding the value
         stack[top] = value;
      }
   }

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

```
    }

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

    public void display() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
            return;
        }
        for (int i = 0; i <= top; i++) { // Corrected loop to iterate up to top
            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(); // Displays the stack before popping
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.display(); // Displays the stack after popping
    }
```

```
}
```

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

Issues Identified

1. Class Name Formatting:
    - Original Line: public class Ascending _Order
    - Correction: Remove the space to make it public class AscendingOrder.
2. Incorrect Loop Condition:
    - Original Line: for (int i = 0; i >= n; i++);
    - Correction: Change the condition to i < n and remove the unnecessary semicolon to ensure proper iteration.
3. Incorrect Sorting Condition:
    - Original Line: if (a[i] <= a[j])
    - Correction: Change it to if (a[i] > a[j]) to ensure swapping happens only when the current element is greater than the next element.

Fixed Code

```
import java.util.Scanner;
public class AscendingOrder {
    public static void main(String[] args) {
```

```
        int n, temp;
        Scanner s = new Scanner(System.in);
        System.out.print("Enter no. of elements you want in array: ");
        n = s.nextInt();
        int a[] = new int[n];
        System.out.println("Enter all the elements: ");

        // Corrected input loop
        for (int i = 0; i < n; i++) {
            a[i] = s.nextInt();
        }

        // Corrected sorting loop
        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;
                }
            }
        }

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

Input and Output

Input:

Enter no. of elements you want in array: 5

Enter all the elements:
1
12
297
1
27912

Output:
Ascending Order: 1, 1, 12, 297, 27912

Quadratic Probing:

```
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 / 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");
        /** 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');
    }
}
```

Errors Identified

1. Incorrect Probing Logic:
    - Original Line: i += (i + h / h--) % maxSize;
    - Correction: The probing logic should be correctly calculated to i = (i + h * h++) % maxSize;. This change accurately implements quadratic probing.
2. Incomplete Comment Block:
    - Original Comment: /** maxSizeake object of QuadraticProbingHashTable **/
    - Correction: Change the comment to /** make object of QuadraticProbingHashTable **/ for clarity.

Corrections Made

1. Update the probing logic in the insert and get methods.
2. Fix the comment for clarity and accuracy.

Fixed 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 * h++) % maxSize; // Corrected probing logic
    } 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;
    }
    return null;
}

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

    // 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');
      scan.close(); // Close the scanner to avoid resource leaks
    }
}
```

Input and Output

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

Multiply Matrices:

```
//Java program to multiply two matrices
import java.util.Scanner;

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

    Scanner in = new Scanner(System.in);
    System.out.println("Enter the number of rows and columns of 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++ )
```

```
        {
          for ( k = 0 ; k < p ; k++ )
          {
            sum = sum + first[c-1][c-k]*second[k-1][k-d];
          }

          multiply[c][d] = sum;
          sum = 0;
        }
      }

      System.out.println("Product of entered matrices:-");

      for ( c = 0 ; c < m ; c++ )
      {
        for ( d = 0 ; d < q ; d++ )
          System.out.print(multiply[c][d]+"\t");

        System.out.print("\n");
      }
    }
  }
}
```

Input: Enter the number of rows and columns of first matrix
2 2
Enter the elements of first matrix
1 2 3 4
Enter the number of rows and columns of first matrix
2 2
Enter the elements of first matrix
1 0 1 0
Output: Product of entered matrices:

3 0
7 0

Errors Identified

1. Incorrect Array Indexing:
    - Original Line: sum = sum + first[c-1][c-k] * second[k-1][k-d];
    - Correction: Change the indices to use first[c][k] and second[k][d]. This ensures you access the correct elements of the matrices without causing an ArrayIndexOutOfBoundsException.

Breakpoints Needed

You can set breakpoints at the following lines for effective debugging:

- Line 44: To check how matrix multiplication is performed and ensure that the correct indices are being accessed.

Steps to Fix the Errors

1. Change the first index access from first[c-1][c-k] to first[c][k].
2. Change the second index access from second[k-1][k-d] to second[k][d].
3. Ensure the inner loop iterates over k < n instead of k < p, since you want to multiply corresponding elements.

Fixed Code

```
// Java program to multiply two matrices
import java.util.Scanner;
```

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

      System.out.println("Enter the number of rows and columns of 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++) {
                for (k = 0; k < n; k++) { // Fixed index handling
                    sum += first[c][k] * second[k][d]; // Fixed matrix access
                }
                multiply[c][d] = sum;
                sum = 0;
            }
        }

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

    in.close(); // Close the scanner to avoid resource leaks
  }
}
```

Input and Output

Input:

```
Enter the number of rows and columns of first matrix
2 2
Enter the elements of first matrix
1 2
3 4
Enter the number of rows and columns of second matrix
2 2
```

Enter the elements of second matrix
5 6
7 8

Output:
Product of entered matrices:
19 22
43 50

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

Errors Identified

1. Incorrect Method Calls for Splitting the Array:
    - Original Line: int[] left = leftHalf(array + 1);
    - Correction: Change to int[] left = leftHalf(array); (You should pass the entire array without modifying it.)
    - Original Line: int[] right = rightHalf(array - 1);

    - Correction: Change to int[] right = rightHalf(array); (Again, pass the entire array.)
2. Invalid Merge Function Call:
    - Original Line: merge(array, left++, right--);
    - Correction: Change to merge(array, left, right); (Post-increment and post-decrement do not apply to arrays.)

Breakpoints Needed

You can set breakpoints at the following lines for effective debugging:

- Line 15: To check how the left array is created.
- Line 16: To check how the right array is created.
- Line 21: To verify if the merge is done correctly.

Steps to Fix the Errors

1. Change the method calls to leftHalf(array) and rightHalf(array) to avoid modifying the input array.
2. Update the merge function call to merge(array, left, right); instead of using post-increment and post-decrement.

Fixed Code

```
// 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); // Fixed
        int[] right = rightHalf(array); // Fixed
        // recursively sort the two halves
        mergeSort(left);
        mergeSort(right);
        // merge the sorted halves into a sorted whole
        merge(array, left, right); // Fixed
    }
}

// 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.
    // 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 and Output

- Input:
    - Before sorting: {14, 32, 67, 76, 23, 41, 58, 85}
- Output:
    - After sorting: {14, 23, 32, 41, 58, 67, 76, 85}

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.

Errors Identified

1. Incorrect Loop Condition for Summing Digits:
    - Original Line: while(sum == 0)
    - Correction: Change to while (sum > 0) (This ensures the loop processes all digits as long as sum is greater than 0.)
2. Incorrect Digit Sum Calculation:
    - Original Line: s = s * (sum / 10)
    - Correction: Change to s = s + (sum % 10) (This correctly accumulates the sum of the digits.)
3. Incorrect Update of sum to Remove Last Digit:
    - Original Line: sum = sum % 10
    - Correction: Change to sum = sum / 10 (This correctly removes the last digit from sum.)

Breakpoints Needed

You can set breakpoints at the following locations for effective debugging:

- Line 12: To check if the loop that processes digits works correctly.
- Line 14: To verify how s is updated with the sum of digits.
- Line 19: To check if the final number is correctly identified as a magic number.

Steps to Fix the Errors

1. Change the condition in the loop to while (sum > 0).
2. Update the logic to accumulate the sum of digits by changing it to s = s + (sum % 10).
3. Modify the update logic for sum to use integer division: sum = sum / 10.

Fixed Code

```
// Program to check if a number is a Magic number in JAVA
import java.util.Scanner;

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; // Copy the number
      int sum = 0;

      // Keep reducing the number until it's a single digit
      while (num > 9) {
         sum = num;
         int s = 0;

         // Sum the digits of the current number
         while (sum > 0) { // Fixed condition
            s = s + (sum % 10); // Corrected to accumulate digit sum
            sum = sum / 10; // Corrected to remove the last digit
         }

         // Assign sum of digits back to num for the next iteration
         num = s;
      }

      // Check if the resulting number is 1 (Magic Number)
      if (num == 1) {
         System.out.println(n + " is a Magic Number.");
      } else {
         System.out.println(n + " is not a Magic Number.");
      }
```

```
        ob.close();
    }
}
```

Input and Output

- 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.

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

Errors Identified

1. Option1 Calculation:
    - Original Line: int option1 = opt[n++][w];
    - Correction: Change to int option1 = opt[n-1][w]; (This prevents an out-of-bounds error by using the current value of n without incrementing it.)
2. Option2 Calculation:
    - Original Line: option2 = profit[n-2] + opt[n-1][w-weight[n]];
    - Correction: Change to option2 = profit[n] + opt[n-1][w-weight[n]]; (This correctly references the profit of the current item.)
3. Weight Update Logic:
    - Original Logic: if (sol[n][w]) and w = w - weight[n];

    - Correction: Ensure that the condition checks if the item is taken correctly, and the weight update logic should work without causing out-of-bounds errors.

Breakpoints Needed

You can set breakpoints at the following locations for effective debugging:

- Line 20: To check how option1 is assigned.
- Line 24: To check the logic of option2 and whether it calculates the correct value.
- Line 32: To check if the items are being selected correctly.

Steps to Fix the Errors

1. Correct the logic for option1 by changing it to int option1 = opt[n-1][w];.
2. Update the logic for option2 by changing it to option2 = profit[n] + opt[n-1][w - weight[n]];.
3. Ensure correct weight update logic when determining which items to take.

Fixed Code

```
// 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
      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 - 1][w]; // Correct: don't increment

          // Take item n
          int option2 = Integer.MIN_VALUE;
          if (weight[n] <= w) { // Fixed condition: weight[n] should be less or
equal to w
             option2 = profit[n] + opt[n - 1][w - weight[n]]; // Fixed: profit[n],
not profit[n-2]
          }

          // 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]; // Decrease weight
       } 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 and Output

- Input: 6,2000
- Output: The output will vary due to the random generation of profits and weights, but it will print the item number, profit, weight, and whether it was taken or not.

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

Errors Identified

1. GCD Calculation:
    - Original Condition: while(a % b == 0)
    - Correction: Change to while(a % b != 0) (This allows the loop to continue until b is 0, which is when the GCD is found.)
2. LCM Calculation:
    - Original Condition: if(a % x != 0 && a % y != 0)
    - Correction: Change to if(a % x == 0 && a % y == 0) (This checks for the least common multiple correctly.)

Breakpoints Needed

You can set breakpoints at the following locations for effective debugging:

- Line 13: To check the loop logic for GCD.
- Line 24: To check the condition in the if statement for LCM.
- Line 31: To verify the final values of GCD and LCM.

Steps to Fix the Errors

1. Fix GCD Calculation:
    - Change the condition in the while loop to while(a % b != 0).
2. Fix LCM Calculation:
    - Change the condition in the if statement to if(a % x == 0 && a % y == 0).

Fixed Code

```
// Program to calculate the GCD and LCM of two given numbers
import java.util.Scanner;

public class GCD_LCM {
   // Method to calculate GCD using the Euclidean algorithm
   static int gcd(int x, int y) {
      int r = 0, a, b;
      a = (x > y) ? x : y; // a is the greater number
      b = (x < y) ? x : y; // b is the smaller number
      r = b;

      while (a % b != 0) { // Correct condition: loop until remainder is 0
         r = a % b;
         a = b;
         b = r;
      }

      return r; // The last non-zero remainder is the GCD
   }

   // Method to calculate LCM
   static int lcm(int x, int y) {
      int a;
      a = (x > y) ? x : y; // a is the greater number

      while (true) {
         if (a % x == 0 && a % y == 0) // Correct condition: divisible by both x
and y
            return a; // Return the LCM
         ++a; // Increment a if not divisible
      }
```

```
    }

    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 and Output

- Input: 4 5
- Output:

```
The GCD of two numbers is: 1
The LCM of two numbers is: 20
```

Armstrong Number

```
//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.

Errors Identified

1. Incorrect Calculation of Remainder:
    - Original Line: remainder = num / 10;
    - Correction: remainder = num % 10; (This extracts the last digit of the number.)
2. Incorrect Update of num:
    - Original Line: num = num % 10;
    - Correction: num = num / 10; (This removes the last digit from the number.)
3. Missing Closing Bracket:
    - Ensure that the class and main method have matching closing brackets.

Breakpoints Needed

To effectively debug the code, consider setting breakpoints at the following locations:

- After int num = Integer.parseInt(args[0]); to check the initial value of num.
- After remainder = num % 10; to verify the value of remainder.
- After num = num / 10; to see how num changes after extracting the last digit.

Steps to Fix the Errors

1. Change the line for calculating the remainder to remainder = num % 10;.
2. Change the line for updating num to num = num / 10;.
3. Ensure the closing brackets for the class and the main method are properly placed.

Fixed 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; // Extract the last digit
         check = check + (int) Math.pow(remainder, 3); // Sum of cubes of
digits
         num = num / 10; // Remove the last digit
      }

      if (check == n)
         System.out.println(n + " is an Armstrong Number");
      else
         System.out.println(n + " is not an Armstrong Number");
   }
}
```

Input and Output

- Input: 153
- Output: 153 is an Armstrong Number

Use of chatGPT for creating text

TASK 3 : Static Analysis Tools

Choose a static analysis tool (in Java, Python, C, C++) in any programming language of your interest and identify the defects. You can also choose your own code fragment from GitHub (more than 2000 LOC) in any programming language to perform static analysis. Submit your results in the .xls or .jpg format only.

Link of the excel file(.xls):

https://docs.google.com/spreadsheets/d/1F1DSi6vCM-BPmqQvtOgsmOoP5TQYZN8rLLMYtd3lyqg/edit?usp=sharing

	A	B	C	D	E	F
1	**Program Name**	**Issue Type**	**Line/Method**	**Description**	**Severity**	**Recommendation**
2						
3	Tower of Hanoi	Syntax Error	doTowers()	Missing semicolon in recursive call	Critical	Add semicolon after method call
4	Tower of Hanoi	Logic Error	doTowers()	Incorrect increment operation (topN++)	Critical	Replace with (topN - 1)
5	Tower of Hanoi	Security	doTowers()	Potential stack overflow	High	Implement proper base condition
6	Tower of Hanoi	Type Safety	doTowers()	Unsafe character arithmetic	Medium	Remove arithmetic on char parameters
7	Stack Implementation	Logic Error	push()	Incorrect top pointer manipulation (top--)	Critical	Change to top++
8	Stack Implementation	Logic Error	display()	Wrong loop condition (i>top)	High	Change to i<=top
9	Stack Implementation	Security	pop()	Missing boundary check	Medium	Add array bounds validation
10	Stack Implementation	Consistency	All methods	Inconsistent increment operations	Low	Standardize increment/decrement usage
11	Ascending Order Sort	Syntax Error	main()	Extra semicolon after for loop	High	Remove extra semicolon
12	Ascending Order Sort	Logic Error	main()	Wrong comparison operator (<=)	Critical	Change to > for ascending sort
13	Ascending Order Sort	Logic Error	main()	Incorrect loop condition (i >= n)	High	Change to i < n
14	Ascending Order Sort	Performance	main()	Inefficient sorting implementation	Medium	Consider using Arrays.sort()
15	Quadratic Probing	Syntax Error	insert()	Invalid += operator usage	Critical	Correct the increment syntax
16	Quadratic Probing	Security	hash()	Potential integer overflow	High	Implement overflow checking
17	Quadratic Probing	Reliability	multiple	Missing null checks	Medium	Add null pointer validation
18	Quadratic Probing	Performance	remove()	Inefficient rehashing	Medium	Optimize rehashing algorithm
19	Matrix Multiplication	Security	main()	Array index out of bounds	Critical	Correct array indices

20	Matrix Multiplication	Logic Error	main()	Incorrect multiplication algorithm	High	Fix matrix multiplication logic
21	Matrix Multiplication	Performance	main()	Inefficient memory usage	Medium	Optimize memory allocation
22	Matrix Multiplication	Validation	main()	Missing input validation	Low	Add matrix dimension validation
23	Merge Sort	Security	mergeSort()	Array manipulation errors	Critical	Remove array arithmetic
24	Merge Sort	Logic Error	merge()	Incorrect increment in merge call	High	Remove increment operators
25	Merge Sort	Security	multiple	Potential null pointer exceptions	Medium	Add null checks
26	Merge Sort	Performance	multiple	Inefficient array copying	Low	Optimize array operations
27	Magic Number	Logic Error	main()	Wrong while loop condition	Critical	Change condition to sum!=0
28	Magic Number	Logic Error	main()	Incorrect arithmetic operations	High	Fix number processing logic
29	Magic Number	Validation	main()	Missing input validation	Medium	Add input range checking
30	Magic Number	Performance	main()	Inefficient number processing	Low	Optimize calculations
31	Knapsack	Security	main()	Array index errors	Critical	Fix array access operations
32	Knapsack	Logic Error	main()	Incorrect weight comparison	High	Correct comparison logic
33	Knapsack	Performance	main()	Memory inefficiency	Medium	Optimize memory usage
34	Knapsack	Validation	main()	Missing input validation	Low	Add boundary checking
35	GCD_LCM	Logic Error	gcd()	Wrong while loop condition	Critical	Fix loop condition
36	GCD_LCM	Performance	lcm()	Inefficient LCM calculation	High	Optimize LCM algorithm
37	GCD_LCM	Security	lcm()	Potential infinite loop	High	Add loop termination condition
38	GCD_LCM	Validation	main()	Missing input validation	Medium	Add input checking
39	Armstrong Number	Logic Error	main()	Wrong division/modulo usage	Critical	Correct number processing
40	Armstrong Number	Performance	main()	Inefficient power calculation	Medium	Use Math.pow() efficiently
41	Armstrong Number	Validation	main()	Missing input validation	Medium	Add range validation
42	Armstrong Number	Logic Error	main()	Incorrect number processing	High	Fix calculation logic