IT-314 Software Engineering-Lab 9

Ruchika Amin-202101158

Lab Report : 9_Program Inspection and Debugging.

Lab Group: 2

1. **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");
        }
```

Category A:

1. No, all variables are initialized.
2. There are no arrays, so array bounds are not involved.
3. There are no arrays, so this error is not present.
4. There are no pointer references, so there is no dangling reference problem.
5. There are no alias names, so this error is not there.
6. The code has all integer values but the operation remainder = num/10 has an answer which is float, so it should actually be num%10 to avoid this error.
7. There are no pointer variables in the code, so there are no issues related to referenced memory attributes.
8. The code doesn't reference data structures or use multiple procedures, so there are no issues with inconsistent structure definitions across procedures.
9. String indexing error is not present.

10. String indexing errors are absent.
11. Inheritance concept is not used, so no error.

-> So, in category A, one error is present, that is there are variables other than the type that the compiler is expecting.

Category B:

1. There are no undeclared variables.
2. The defaults in Java are well understood.
3. Variables are initialized correctly.
4. Yes, all variables are assigned correct length and data type.
5. There are no inconsistencies in the memory type.
6. There are no variables with similar names.

-> There are no errors in Category B.

Category C:

1. There is no mixing of data types, however there can be an error due to assigning a float to integer variable. But it would just be a precision error.
2. There are no mixed mode computations with errors.
3. There are no issues with different lengths of data types.
4. There are no issues with the data type of the target variable.
5. there are no apparent possibilities of overflow or underflow
6. The code does not have a division operation where the divisor could be zero.
7. There are only integer operations, so no error with binary operations.
8. The code doesn't contain any checks for values going outside the meaningful range. It assumes that the input number is within the range of integers. So, an error could occur.
9. There are no complex expressions or ambiguities.
10. There are no invalid uses of integer arithmetic, and the expressions used are safe for both even and odd values of the variables involved.

-> there are no errors in this category. However, we need to assume that the input is within the bounds and there are no precision requirements.

Category D:

1. The code does not involve comparisons between variables with different data types, so no error.
2. There are no mixed-mode comparisons or comparisons between variables of different lengths in the code. So, there are no errors there.

3. Comparison operators are used correctly.
4. Boolean comparisons are performed correctly.
5. There is no mixing of comparison and boolean operators.
6. There are no such comparisons involved.
7. The order of evaluation and operator precedence is clear and correct. There is no ambiguity.
8. The code is free from compiler-dependent behavior issues

-> There are no errors in this Category.

Category E:

1. No multiway branch errors are present..
2. There are no infinite loops.
3. Program terminates eventually.
4. There is no loop that is never executed.
5. There are no loop fall-throughs.
6. The code does not contain off-by-one errors
7. There are no issues with mismatched or missing brackets.
8. No, there are no non-exhaustive decisions.

-> There are no errors in this category.

Category F:

1. The code does not involve external modules or function calls, so no error.
2. There are no attributes of parameters or arguments to check in the code.
3. The code does not deal with unit systems.
4. There are no function calls.
5. There are no function calls, so no argument checking involved.
6. The code does not deal with unit systems.
7. No built-in functions involved
8. No subroutine to alter parameter involved
9. No global variables are present

-> There are no errors in this category.

Category G:

1. The code does not involve explicit file handling, so there are no file attributes to check.
2. There are no file operations in the code.
3. There are no file operations or memory-related issues in the code.
4. There are no file operations in the code.

5. There are no file operations in the code.
6. There are no file operations in the code.
7. There are no file operations and I/O handling in the code.
8. There are no file operations in the code.

-> no errors in this category

Category H:

1. There are no unused variables.
2. No explicit attributes to be checked
3. No compiler warnings
4. It does not work on edge cases such as negative numbers.
5. No missing functions

-> The program is not robust, as it does not check for input validity.

Program Inspections:

1. There are 4 errors.
    - there are variables other than the type that the compiler is expecting.
    - Precision is not taken into consideration
    - Input is not validated
    - The program is not robust
2. All errors are useful, but it may depend on the program.
   Here, Data reference, data declaration, computation error, control flow error are more important.
3. It might not identify logic errors.
4. Yes, it is worth applying.

Debugging:

3. Debugged code

```
class Armstrong {
    public static void main(String args[]) {
        int num = Integer.parseInt(args[0]);
        int n = num; // Store the original number
        int check = 0, remainder;
        int numberOfDigits = (int) Math.log10(num) + 1; // Calculate the number of digits

        while (num > 0) {
            remainder = num % 10; // Extract the last digit
            check = check + (int) Math.pow(remainder, numberOfDigits);
            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");
    }
}
```

2. GCD and LCM

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

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

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

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

    public static void main(String args[])
    {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter the two numbers: ");
        int x = input.nextInt();
        int y = input.nextInt();

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

```
Input:4 5
Output: The GCD of two numbers is 1
        The GCD of two numbers is 20
```

Category A:

1. No uninitialized variables as a, b and r are initialized in the starting.
2. No array references in the code
3. No array references in the code
4. No explicit references through pointer or reference variables in the code, hence no issues with the memory location of the referenced.
5. No aliasing issues with the attributes.
6. Uses only basic integer variables and no issues with type or attribute and no use of structure.
7. There are no explicit or implicit addressing problems, as the code works with integers and not bit strings or addresses.

8. No pointer or reference variables used in code, so no issues related to the attributes of referenced memory locations.
9. No reference data structures in multiple procedures or subroutines.
10. No string indexing operations or subscript references to arrays, so no string off-by-one errors.
11. No relation to object-oriented languages, so there are no inheritance requirements to check.

Category B:

1. All variables are explicitly declared before use, avoiding issues with undeclared or shared variables.
2. While variable attributes beyond data type are not explicitly stated, it's understood that the defaults for integers are well-known in Java.
3. Variables in the code are properly initialized before use, ensuring their reliability.
4. Variables are consistently assigned the correct data type (int) and used appropriately.
5. Variable initialization aligns with their data type, with integer variables initialized by integer values.
6. Variable names are distinct, preventing any confusion or ambiguity.

Category C:

1. No, all computations use variables of the same data type (int).
2. No, there are no mixed-mode computations in the code.
3. All computations in the code use variables of the same data type and length (int).
4. The data type of the target variable matches the data type and result of the right-hand expression in all assignments.
5. The code does not involve computations that lead to overflow or underflow.
6. The code does not perform division where the divisor could be zero.
7. The code performs integer arithmetic, so there are no concerns related to base-2 representation inaccuracies.
8. The code does not assign values to variables outside their meaningful range.
9. The code does not contain complex expressions with multiple operators.
10. The code uses integer arithmetic correctly, and there are no invalid uses of integer arithmetic.

Category D:

1. No, there are no comparisons between variables with different data types in the code.
2. There are no mixed-mode comparisons or comparisons between variables of different lengths in the code.

3. The comparison operators used in the code are correct.
4. Each Boolean expression in the code states what it is supposed to state.
5. The operands of Boolean operators in the code are Boolean, and comparison and Boolean operators are used correctly.
6. The code does not contain comparisons between fractional or floating-point numbers represented in base-2.
7. The code does not have expressions containing multiple Boolean operators, so there are no assumptions about the order of evaluation or precedence to consider.
8. The compiler's evaluation of Boolean expressions does not affect the program because the code does not involve complex Boolean expressions with short-circuit evaluation concerns.

Category E:

1. Yes, the index variable `i` in the code can have values 1, 2, or 3 in the `GO TO` statement.
2. Every loop in the code eventually terminates, as there are no infinite loops.
3. The main program and the subroutines in the code eventually terminate.
4. Yes, the while loop in GCD is wrong. It should by while(a%b!=0), only then the program executes correctly.
5. The code does not contain loop fall-through.
6. The code has no off-by-one errors; loops iterate correctly.
7. The code maintains proper block structure with matching brackets.
8. The code doesn't make non-exhaustive decisions.

Category F:

1. The code correctly matches the number and order of parameters and arguments.
2. The attributes of parameters and arguments match.
3. The unit system of parameters and arguments is consistent.
4. The code correctly transmits the number of arguments to other modules.
5. The attributes of arguments match the corresponding parameter attributes.
6. The unit system of arguments matches the units of parameters.
7. Built-in functions are invoked with the correct number and order of arguments.
8. The code does not alter parameters intended as input values.
9. There are no global variables in the code.

Category G:

1. The code does not explicitly declare files.
2. There is no use of file I/O attributes.
3. The code does not read files that would require excessive memory.
4. There are no file open statements in the code.

5. There are no file close statements in the code.
6. The code does not handle end-of-file conditions.
7. The code does not handle I/O error conditions.
8. No, there are no grammatical errors in the text printed.

Category H:

1. The code references all variables appropriately.
2. There are no unexpected default attributes assigned to variables.
3. The code does not produce warning or informational messages.
4. The program does not check input for validity, and hence doesn't take into account the wrong while loop.
5. There are no missing functions in the program.

Program Inspection:

1. Errors in the program:
   - In the `gcd` method, there is an error in the while loop condition. It should be `while (a % b != 0)` to correctly calculate the GCD.
   - In the `lcm` method, there's no error, but the algorithm used for calculating LCM is not efficient. It uses a brute-force approach, which can be improved.

2. The category of program inspection that would be more effective in identifying these errors is a static code review or a code review by peers. Static code reviews involve analyzing the code without executing it, making it effective for identifying syntax and logical errors.

3. Program inspection can identify many types of errors, including syntax errors, logical errors, and code quality issues. However, program inspection alone may not identify runtime errors that occur under specific conditions, which may require dynamic testing.

4. Yes, program inspection techniques are worth applying. They are a crucial part of the software development process, helping to catch errors early and improve code quality. While inspections may not catch all types of errors, they are an essential part of a robust quality assurance process. In addition to inspections, testing and debugging are often used to identify and address runtime errors.

CODE DEBUGGING: Debugging is the process of localizing, analyzing, and removing suspected errors in the code.

1. Errors in the program:
   - In the `gcd` method, there is an error in the while loop condition. It should be `while (a % b != 0)` to correctly calculate the GCD.
   - In the `lcm` method, the algorithm is not efficient, although it's not a syntax error. It uses a brute-force approach for calculating the LCM.

2. To fix the identified errors:
   - You need one breakpoint to fix the error in the `gcd` method by changing the while loop condition to `while (a % b != 0)`.

3. Here is the corrected code fragment:

```
import java.util.Scanner;

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

      while (b != 0) { // Corrected the while loop condition
         int temp = b;
         b = a % b;
         a = temp;
      }
      return a;
   }

   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();

        int gcdResult = gcd(x, y);
        int lcmResult = lcm(x, y);

        System.out.println("The GCD of two numbers is: " + gcdResult);
        System.out.println("The LCM of two numbers is: " + lcmResult);
        input.close();
    }
}
```

Please note that I have corrected the errors in the `gcd` method by changing the while loop condition and used the Euclidean algorithm for GCD calculation, which is more efficient. The corrected code is now executable.

3. Knapsack

```
//Knapsack
public class Knapsack {

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

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

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

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

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

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

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

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

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

        // print results
        System.out.println("item" + "\t" + "profit" + "\t" + "weight" + "\t" + "take");
        for (int n = 1; n <= N; n++) {
            System.out.println(n + "\t" + profit[n] + "\t" + weight[n] + "\t" + take[n]);
        }
    }
}
```
Input: 6, 2000

Output:

Item	Profit	Weight	Take
1	336	784	false
2	674	1583	false
3	763	392	true
4	544	1136	true
5	14	1258	false
6	738	306	true

Category A:

1. No, all the variables are initialized.
2. Yes, all the array references are bounded by the defined dimensions (1 to n OR 1 to w)
3. Yes, all subscripts have an integer value.
4. There are no pointer or reference variables in the provided code. So, the given issue is not applicable.
5. There are no alias names statements in the code. So, the given issue is not applicable.
6. No, there are no mismatches in the variable's value and type.
7. No, there are no explicit or implicit addressing problems as bit-level operations are not involved.
8. The code does not use pointer or reference variables, so this issue is not applicable.
9. The code does not involve any such procedures, so this issue is not applicable.
10. No, this issue is not applicable in this piece of code.
11. The Knapsack code is not written in an object-oriented language. So, this issue is not relevant to the given code.

Category B:

1. Yes, all variables have been explicitly declared.
2. In Java, variables are not declared with implicit default attributes, so there are no issues
3. The code initializes the profit and weight arrays with random values in a loop and they are correctly initialized.
4. Yes, each variable is assigned the correct length and data type.
5. Yes, the initialization of a variable is consistent with its memory type.
6. No, variables don't have similar names.

Category C:

1. There are no computations using variables having inconsistent data types.
2. There are no mixed-mode computations.
3. There are no computations with variables of the same data type but different lengths.
4. There are no issues related to data type size.
5. There are no overflow or underflow issues in this code.

6. The code does not contain any division operations.
7. The code does not involve base-2 representation.
8. The values assigned to profit and weight are generated within reasonable bounds based on random values. So, their value may go outside the range.
9. Yes, they are correct.
10. No, there are no invalid uses of integer arithmetic.

Category D:

1. There are no comparisons between variables of different data types.
2. The code does not contain mixed-mode comparisons.
3. Yes, they are correct.
4. Yes, the boolean expressions are correctly used.
5. Yes, the operands of a Boolean operator Boolean.
6. The code does not involve fractional or floating-point numbers. So, the issues are not relevant to the given code.
7. The code does not contain complex expressions with multiple Boolean operators.
8. The Boolean expressions are straightforward, and there are no conditional evaluations that could lead to issues

Category E:

1. There are no issues related to the index variable exceeding the number of branch possibilities.
2. Yes, every loop will terminate eventually.
3. Yes, the program's execution is finite.
4. No, there are no such loops.
5. There are no issues related to loop fall-through.
6. There are no off-by-one errors in the loops.
7. Yes, code groups are well-formed.
8. The Knapsack code does not assume non-exhaustive decisions.

Category F:

1. There are no parameters or arguments being passed between modules.
2. There are no parameters or arguments to match attributes in this code
3. There are no units to match.
4. There is no module-to-module communication.
5. There are no modules or functions to send or receive arguments and parameters.
6. The code doesn't involve units or unit systems.
7. The Knapsack code does not invoke any built-in functions.
8. There are no subroutines in the code.
9. The code doesn't use global variables or modules.

Category G:

1. The Knapsack code does not involve file I/O.
2. There are no file operations in the code.
3. The Knapsack code doesn't read from files or allocate memory for file operations.
4. There are no file operations in the Knapsack code, so there are no files to open or close.
5. There are no file operations, there are no files to close in the code.
6. There are no end-of-file conditions to detect or handle.
7. There are no I/O error conditions to handle in the code.
8. There are no spelling or grammatical errors in any text.

Category H:

1. There are no variables declared but never used, and each variable appears to be referenced appropriately.
2. The code does not rely on default attributes, and the attributes of variables align with their intended usage.
3. In the provided code, there are no warning or informational messages.
4. The code does not include input validation checks.
5. There are no missing functions.

Program Inspection:

1. There are no errors in the given code.
2. Manual code review allows for the identification of logical errors, while static code analysis tools can assist in identifying potential issues like array index errors or syntax problems.
3. Program inspections, whether manual or automated, may not easily identify runtime errors or issues that require dynamic analysis, such as certain memory management problems or concurrency-related issues.
4. Yes, program inspection techniques are valuable and widely applicable in software development. They help identify a range of issues, including syntax errors, logical errors, code style violations, and potential performance problems.

CODE DEBUGGING:

1. Following are the errors:
    - Incorrect usage of the increment and decrement operators
2. The number of breakpoints needed to fix these errors can vary. In general, you would set breakpoints at key points in the code where you want to pause execution for debugging.
3.

```
// 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];
        int[] weight = new int[N];

        // generate random instance, items 0..N-1
        for (int n = 0; n < N; n++) {
            profit[n] = (int) (Math.random() * 1000);
            weight[n] = (int) (Math.random() * W);
        }

        // opt[n][w] = max profit of packing items 0..n with weight limit w
        // sol[n][w] = does opt solution to pack items 0..n with weight limit w include item n?
        int[][] opt = new int[N][W + 1];
        boolean[][] sol = new boolean[N][W + 1];

        for (int n = 0; 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 the 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];
        for (int n = N - 1, w = W; n >= 0; n--) {
            if (sol[n][w]) {
                take[n] = true;
                w -= weight[n];
            } else {
                take[n] = false;
            }
        }

        // print results
        System.out.println("Item" + "\t" + "Profit" + "\t" + "Weight" + "\t" + "Take");
        for (int n = 0; n < N; n++) {
            System.out.println(n + "\t" + profit[n] + "\t" + weight[n] + "\t" + take[n]);
        }
    }
}
```

4. Magic Number

```
// Program to check if number is Magic number in JAVA
import java.util.*;
public class MagicNumberCheck
{
    public static void main(String args[])
    {
        Scanner ob=new Scanner(System.in);
        System.out.println("Enter the number to be checked.");
        int n=ob.nextInt();
        int sum=0,num=n;
        while(num>9)
        {
            sum=num;int s=0;
            while(sum==0)
            {
                s=s*(sum/10);
                sum=sum%10
            }
            num=s;
        }
        if(num==1)
        {
            System.out.println(n+" is a Magic Number.");
        }
        else
        {
            System.out.println(n+" is not a Magic Number.");
        }
    }
}
```

```
Input: Enter the number to be checked 119
Output 119 is a Magic Number.
Input: Enter the number to be checked 199
Output 199 is not a Magic Number.
```

Category A:

1. There's no issue related to uninitialized variables.
2. There are no array references in the code, so there's no issue with array bounds.
3. The code does not use array subscripts, so there are no issues related to integer subscripts.
4. The code doesn't involve pointer or reference variables, so there are no issues related to dangling references.
5. There are no aliasing issues in the code.
6. The code doesn't have any issues with unexpected type or attribute mismatches.
7. There are no problems with addressing smaller memory units.
8. Since there are no pointer or reference variables used in the code, there are no issues related to referenced memory attributes.
9. The code doesn't involve data structures defined in multiple procedures, so this category is not applicable.
10. The code doesn't have any string indexing issues.
11. The code is not written in an object-oriented language, so there are no inheritance requirements to consider.

Category B:

1. The code explicitly declares the variables n, sum, num, and s, so there are no issues with undeclared variables.

2. In Java, the default attributes for int variables include being initialized to 0, and for Scanner objects, the default attributes are handled by the library. There are no issues with default attribute understanding in the code.
3. The code initializes the variables n, sum, and num appropriately at the beginning. However, there's an issue with the initialization of s. It should be initialized to 1, not 0, before the while loop. If s is initialized to 0, each time the product will result to 0.
4. The code uses int data type for the variables n, sum, num, and s, which is appropriate for the task.
5. The variable initialization is consistent with its memory type.
6. There are no variables with similar names in the code.

Category C:

1. The code only uses variables of the `int` data type, so there are no inconsistent data types used in computations.
2. The code performs integer division and modulus operations with `int` variables. There are no mixed-mode computations with different data types.
3. All variables in the code are of the same data type (`int`), so there are no computations with variables of different lengths.
4. The target variables and right-hand expressions are of the same data type, and there is no issue with the data type size.
5. There are no calculations that can result in overflow or underflow. The code performs basic arithmetic operations on integers, and the range of integers is well within the bounds of the `int` data type in Java.
6. The code does not perform any division operations, so there is no possibility of division by zero.
7. The code doesn't involve floating-point calculations, so there is no inaccuracy related to base-2 representation.
8. The code does not have any variables with meaningful range constraints.
9. The code correctly follows the order of evaluation and operator precedence for the arithmetic operations. There are no issues in this regard.
10. The code uses integer arithmetic, and there are no invalid uses or ambiguous results in integer arithmetic operations.

Category D:

1. The code does not have comparisons between variables of different data types, so there are no errors related to this point.
2. The code only involves comparisons between int variables, so there are no mixed-mode comparison errors.

3. The code uses the comparison operator == correctly to check if num is equal to 1, which is the correct usage for equality comparison. There are no issues with misunderstanding comparison operators.
4. The code uses simple Boolean expressions such as if (num == 1). There are no complex Boolean expressions in the code, and the expressions are clear in their intent.
5. The code doesn't involve complex expressions or mixing comparison and Boolean operators.
6. The code only uses integer values, and there are no comparisons between fractional or floating-point numbers. Therefore, there are no issues related to base-2 representation.
7. The code only uses simple Boolean expressions, and the order of evaluation and precedence of operators are not relevant in this context. There are no complex expressions to evaluate.
8. The code does not contain expressions that would be affected by compiler-specific behavior. The code's logic is straightforward and doesn't involve complex Boolean expressions that might behave differently on different compilers.

Category E:

1. The code does not contain a multiway branch.
2. The code contains a while loop, and it will eventually terminate because it reduces the value of num in each iteration, and the loop condition (num > 9) will become false when num reaches a single-digit value. The inner while loop (sum==0) incorrectly uses == sign, rather it should use != for the termination of the loop.
3. The program contains a main method, which is the entry point of a Java program. The program will eventually terminate when the main method finishes executing.
4. There is no issue with a loop never executing in the provided code. The loop in the code is dependent on the input value n, and as long as n is not less than 1, the loop will execute.
5. The provided code does not have a loop controlled by both iteration and a Boolean condition, so there are no consequences of loop fall-through to consider.
6. The code does not contain off-by-one errors. The loop iterates correctly for values of n greater than or equal to 1.
7. The code uses proper code blocks and brackets. There are matching opening and closing curly braces, so there are no issues with mismatched brackets.
8. The code does not involve non-exhaustive decisions related to input parameters. It simply checks if the input number n is a magic number based on the algorithm.

Category F:

1. The code doesn't involve function calls or interfaces.
2. No function calls or parameters.
3. Not applicable as there are no units involved.
4. No function calls.
5. No function calls.
6. Not relevant as there are no units.
7. No built-in functions used.
8. There are no parameters.
9. The code doesn't use global variables.

Category G:

1. The code doesn't involve file I/O.
2. No file operations in the code.
3. The code doesn't read from files.
4. No file operations in the code.
5. Not applicable as there are no file operations.
6. The code doesn't read from files.
7. There are no file I/O operations.
8. There are no spelling or grammatical errors in any text that is printed or displayed by the program.

Category H:

1. The code doesn't involve complex identifiers.
2. Not applicable, as there are no attributes to check.
3. The code doesn't generate warnings.
4. The code checks if the input number is valid (greater than 0) before proceeding.
5. The code doesn't require additional functions.

Program Inspection:

1. There are two errors in the program:
    - The condition in the inner while loop should be `while (sum != 0)` instead of `while (sum == 0)`.
    - The initialization of the variable `s` should be `s = 1` before the inner while loop, not `s = 0`.
2. For this simple program, a combination of Category B (Data-Declaration Errors) and Category C (Computation Errors) would be more effective in identifying and correcting errors.
3. Program inspection can identify many types of errors, but it may not detect logical errors that are not syntactical. In this specific program, it does not identify the logical error in the while loop condition, which could result in incorrect results.

4. Program inspection is a valuable technique for identifying various errors, especially in larger and more complex programs. However, for very simple programs like the "Magic Number" code, manual inspection may be sufficient, and more sophisticated techniques might be overkill. Automated testing and unit testing could also be used for validation.

Debugging:

1. There are two errors in the program:

   The condition in the inner while loop should be while (sum != 0) instead of while (sum == 0).

   The initialization of the variable s should be s = 1 before the inner while loop, not s = 0.
2. Number of Breakpoints: To fix the two identified errors, we would need at least two breakpoints in your debugging tool. One for each error.
3. Steps to Fix the Errors:

   To fix the first error, modify the inner while loop condition to while (sum != 0):

```
while (sum != 0) {
    s = s * (sum % 10);
    sum = sum / 10;
}
```

   To fix the second error, initialize s to 1 before the inner while loop:

```
while (num > 9) {
    sum = num;
    s = 1;  // Initialize s to 1
    while (sum != 0) {
        s = s * (sum % 10);
        sum = sum / 10;
    }
    num = s;
}
```

   Complete executable code:

```
import java.util.*;

public class MagicNumberCheck {
    public static void main(String args[]) {
        Scanner ob = new Scanner(System.in);
        System.out.println("Enter the number to be checked.");
        int n = ob.nextInt();
        int sum = 0, num = n;
        while (num > 9) {
            sum = num;
            int s = 1;  // Initialize s to 1
            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.");
        }
    }
}
```

5 : Merge Sort

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

import java.util.*;

public class MergeSort {
    public static void main(String[] args) {
        int[] list = {14, 32, 67, 76, 23, 41, 58, 85};
        System.out.println("before: " + Arrays.toString(list));
        mergeSort(list);
        System.out.println("after:  " + Arrays.toString(list));
    }

    // Places the elements of the given array into sorted order
    // using the merge sort algorithm.
    // post: array is in sorted (nondecreasing) order
    public static void mergeSort(int[] array) {
        if (array.length > 1) {
```

```
        // split array into two halves
        int[] left = leftHalf(array+1);
        int[] right = rightHalf(array-1);

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

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

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

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

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

```
        int i1 = 0;   // index into left array
        int i2 = 0;   // index into right array

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

Input: before 14 32 67 76 23 41 58 85
after 14 23 32 41 58 67 76 85

Category A: Data Reference Errors

1. There are no uninitialized variables.
2. In the above code, In the mergeSort method, leftHalf(array+1) and rightHalf(array-1) are incorrect. We can't perform arithmetic operations like +1 or -1 directly on an array in Java. This will result in a compilation error.
3. The program does not contain any issues with subscript values.
4. There are no issues related to dangling references.
5. The program does not use alias names, so there are no aliasing issues.
6. There are no issues related to incorrect types or attributes.
7. The program doesn't contain addressing problems related to memory allocation and addressability units.
8. There are no pointer attribute issues.
9. The data structure (arrays in this case) is defined consistently across procedures.
10. There are no string indexing issues in this code.
11. There are no inheritance requirements to consider.

Category B : Data-Declaration Errors

1. All variables in the code are explicitly declared.
2. The default attributes for Java variables are well understood.
3. All variables that are initialized in declarative statements are properly initialized.
4. The variables in the code are assigned the correct length and data type. There seems to be no issues with incorrect assignments.
5. The initialization of variables is consistent with their memory type i.e. arrays are initialized correctly.
6. There are no variables with similar names in this code.

Category C: Computation Errors

1. There are no computations using variables with inconsistent data types in the code.
2. There are no mixed-mode computations in the code.
3. The code uses variables of the same data type and length in computations.
4. The target variable data type is appropriately sized for the result of the right-hand expression.
5. The code does not contain explicit numerical computations that might lead to overflow or underflow.
6. The code does not have a division operation.
7. The code doesn't contain computations that might lead to base-2 representation inaccuracies.
8. The code does not perform computations that might result in values going outside the meaningful range.
9. The code correctly assumes the order of evaluation and operator precedence for the expressions it contains.
10. There are no invalid uses of integer arithmetic, particularly divisions, in the code.

Category D: Comparison Errors

1. No comparisons between variables with different data types.
2. No mixed-mode comparisons or comparisons between variables of different lengths.
3. Comparison operators are used correctly in the code.
4. Boolean expressions state what they are supposed to.
5. There are no operands mixed between comparison and Boolean operators.
6. No comparisons between fractional or floating-point numbers that are represented in base-2.
7. Assumptions about the order of evaluation and the precedence of operators are correct.
8. Compiler evaluation of Boolean expressions does not affect the program.

Category E: Control-Flow Errors

1. The program does not contain a computed ‘GO TO’ statement.
2. All loops will eventually terminate.
3. The program will terminate.
4. There are no loops that will never execute due to conditions upon entry.
5. There are no loop fall-through issues.
6. There are no off-by-one errors in loops.
7. The program does not contain explicit code blocks.
8. There are no non-exhaustive decisions in the logic.

Category F: Interface Errors

1. The number of parameters received and arguments sent match in each module.
2. The attributes of each parameter match the attributes of each corresponding argument.
3. The units system of each parameter matches the units system of each corresponding argument.
4. The number of arguments transmitted matches the number of parameters expected in other modules.
5. The attributes of each argument transmitted match the attributes of the corresponding parameter in other modules.
6. The units system of each argument transmitted matches the units system of the corresponding parameter in other modules.
7. No built-in functions are invoked.
8. Subroutines do not alter parameters that are intended to be only input values.
9. There are no global variables in the code.

Category G: Input / Output Errors

1. No files are explicitly declared.
2. No files are opened in the code.
3. Memory availability is not relevant for file operations in this code.
4. No files are opened.
5. No files are closed.
6. End-of-file conditions and I/O error conditions are not included in this code.
7. There is no input/output operation in the code.

8. There are no spelling or grammatical errors in any text that is printed or displayed by the program.

Category H: Other Checks

1. All identifiers in the code are referenced.
2. No attribute listings are relevant in this code.
3. No warning or informational messages are produced by the compiler.
4. The code does not include specific input validation checks. It assumes that the input provided will be an array of integers, and it does not handle cases where the input might be invalid.
5. There is no missing function in the program.

Program Inspection:

There are several errors in the program. Here are the identified errors:

1. In the mergeSort method, int[] left = leftHalf(array+1); and int[] right = rightHalf(array-1); are incorrect. You can't perform arithmetic operations like +1 or -1 directly on an array in Java. This will cause a compilation error.

- In mergeSort, merge(array, left++, right--); is incorrect. The ++ and -- operators cannot be used in this context. They should be used separately on left and right arrays before passing them to the merge method.

2. For this program, I would find Category A (Data Reference Errors) more effective. This is because the issues in this code are related to how data references are being handled.

3. Through program inspection, it's difficult to identify runtime errors that might occur due to logical mistakes i.e if there were incorrect sorting logic.

4. Yes, program inspection is a valuable technique for identifying potential issues in code. It helps to catch syntax errors, logic mistakes, and design problems. However, it's not foolproof, and other testing techniques (such as unit testing and integration testing) are also necessary to ensure code correctness and reliability.

CODE DEBUGGING: Debugging is the process of localizing, analyzing, and removing suspected errors in the code.

There are several errors in the program. Here are the identified errors:

1. In the mergeSort method, int[] left = leftHalf(array+1); and int[] right = rightHalf(array-1); are incorrect. You can't perform arithmetic operations like +1 or -1 directly on an array in Java. This will cause a compilation error.

- In mergeSort, merge(array, left++, right--); is incorrect. The ++ and -- operators cannot be used in this context. They should be used separately on left and right arrays before passing them to the merge method.

2. There are three breakpoints that need to be fixed.

a. To fix the error, you need to:
   Replace int[] left = leftHalf(array+1); with int[] left = leftHalf(Arrays.copyOfRange(array, 0, array.length / 2));
   Replace int[] right = rightHalf(array-1); with int[] right = rightHalf(Arrays.copyOfRange(array, array.length / 2, array.length));
   Replace merge(array, left++, right--); with merge(array, left, right); and place left++; right--; in separate lines before this line.

3. 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(Arrays.copyOfRange(array, 0, array.length / 2));
         int[] right = rightHalf(Arrays.copyOfRange(array, array.length / 2, array.length));

         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 / 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++;
            }
        }
    }
}
```

6. Matrix Multiplication

Category A:

1) No uninitialised variables in this code.
2) The code is assuming that the user will provide valid input but this can lead to index out of bounds errors. The code assumes that the user will provide valid input for matrix dimensions.
3) In JAVA the array subscripts must be of type ‘int’ hence this will not be an issue.
4) The code is in JAVA hence dangling reference will not be an issue
5) The code does not use aliasing of memory locations. Each matrix has its own separate array, and there are no EQUIVALENCE or REDEFINES constructs used.
6) In Java, the variables' types and attributes are strictly enforced by the compiler. There is no opportunity for the physical representation of data to differ from its type.
7) Java abstracts memory management and doesn't provide direct access to memory addresses. This mitigates the potential issues related to memory allocation and addressability.
8) Java doesn't have explicit pointer variables, so there are no issues with assigning a pointer to a different data structure.
9) In the code provided, there is only one data structure (2D arrays) used to represent matrices. It's defined consistently within the main method.
10) The code doesn't work with strings, so there are no issues related to string indexing.
11) Not applicable here.

Category B:

1) In the Java code, all variables are explicitly declared and initialized before they are used. There are no undeclared variables or parameters.
2) These defaults are part of the Java language specification and are not typically a source of surprise.
3) In the provided code, variables are properly initialized.
4) In Java, variables must be assigned a specific data type when declared. The data type specifies the range of values the variable can hold. For example, in the code, integer arrays are declared as int, and the variables are correctly assigned the data types based on their intended usage. There are no issues related to incorrect length or data type assignments.
5) In Java, the initialization of a variable is consistent with its memory type. For example, integers are initialized to 0, and objects are initialized to null. The code follows these conventions consistently.

6) The code does not contain variables with similar names that could lead to confusion. Variable names are used appropriately and do not create ambiguity within the program.

Category C:

1) In the Java code, there are no computations that use variables with inconsistent data types. Java enforces strict type checking, and operations between variables of incompatible data types would result in a compilation error.
2) No type casting error.
3) In the Java code, there are no computations using variables of the same data type but different lengths. Java enforces data type consistency and does not allow operations between variables of different lengths.
4) Java enforces type safety, and the data type of the target variable in an assignment must be compatible with the data type or result of the right-hand expression. Type coercion or explicit casting may be required for certain assignments, but the code should ensure data type compatibility.
5) The Java code does not explicitly handle overflow or underflow conditions. In Java, integer overflow results in overflow exceptions, and floating-point numbers have their own representation for such cases. It's important to ensure that computations are within the bounds of the data types used.
6) The code does not explicitly check for division by zero. In Java, dividing by zero will result in an exception being thrown at runtime. Handling division by zero should be considered if it's a possibility in the specific use case.
7) The code doesn't perform floating-point operations that would highlight inaccuracies in binary representation, such as the example you provided (10 x 0.1). These inaccuracies are common in binary representation, but the code does not explicitly demonstrate them.
8) The code does not explicitly check if the value of a variable can go outside the meaningful range. It's important to ensure that assigned values are within the meaningful range based on the application's requirements.
9) In the code, expressions are evaluated according to Java's operator precedence rules. These rules are well-understood and followed in the code.
10) The code does not contain invalid uses of integer arithmetic. Java follows well-defined rules for the order of operations, and expressions like `2*i/2 == i` will produce consistent results, regardless of whether `i` is even or odd.

Category D:

1. The code does not contain comparisons between variables of different data types. In Java, comparing variables of incompatible data types would lead to compilation errors.
2. The code doesn't include mixed-mode comparisons or comparisons between variables of different lengths. Java enforces strict type checking, and explicit type casting would be required to perform such comparisons.

3. The comparison operators in the code appear to be used correctly. For example, `<`, `>`, `==`, and `&&` are used as expected. The code doesn't contain common mistakes related to comparison operators.

4. Boolean expressions in the code appear to state what they are supposed to state. The use of `&&`, `||`, and `!` to form logical expressions seems to be consistent with the intended logic.

5. The code doesn't mix Boolean and comparison operators in ways that would result in common mistakes. For example, it uses `(a==b)&&(b==c)` to compare three numbers for equality, which is correct.

6. The code doesn't include explicit comparisons between fractional or floating-point numbers represented in base-2. Java's floating-point comparisons use IEEE 754 standards, which are designed to handle such cases.

7. The code correctly handles order of evaluation and operator precedence. For example, in expressions like `if((a==2) && (b==2) || (c==3))`, Java's operator precedence ensures that `&&` is evaluated before `||`.

8. The code doesn't contain constructs that are significantly affected by compiler-specific behavior. However, it's essential to be aware of potential differences in compiler behavior, especially in complex or ambiguous expressions. For example, expressions like `if((x==0 && (x/y)>z)` should be used with caution and well-understood in the context of specific compiler behavior.

Category E

1. In the code provided, there are no multiway branches, such as computed `GO TO` statements, so the question of the index variable exceeding the number of branch possibilities does not apply.

2. The loops in the code have well-defined termination conditions. For example, `for` loops have a defined range, and the `while` loop is controlled by a Boolean condition.

3. The `main` method in the code is the program's entry point, and it eventually terminates upon completion. Therefore, the program/module terminates as expected.

4. There are no loops in the code that would never execute, given the provided conditions. The loops are constructed to iterate based on defined conditions.

5. The code doesn't contain a loop with both iteration and a Boolean condition in a way that would result in loop fall-through. The `while` loops have their Boolean conditions checked before entering the loop.

6. The code does not contain off-by-one errors in the loops. The loops iterate within the specified range correctly.

7. The code properly uses statement groups and code blocks, such as `{...}`, and there are no mismatches in the placement of opening and closing brackets.

8. There are no non-exhaustive decisions in the code. The code does not assume that a variable must have a specific value if it is not one of the explicitly checked values.

Category F:

1. In the provided Java code, there are no explicit modules or functions with parameters and arguments. Therefore, questions related to matching the number and order of parameters and arguments do not apply.

2. Java enforces strict type checking, so if you were passing arguments to a method (function) in Java, the data type and size of the parameters must match the data type and size of the arguments. In the code provided, this is not applicable as it doesn't contain such method calls.

3. Java typically does not deal with units systems like degrees and radians explicitly, but rather it relies on the programmer to handle conversions. If units conversion is necessary, it's typically the programmer's responsibility, and it doesn't apply to the code provided.

4. As the code doesn't contain explicit method calls with arguments, the question of matching the number of arguments transmitted to another module doesn't apply.

5. In Java, the attributes of arguments must match the data type and size expected by a method (function). This would be applicable if the code included method calls with arguments, which it does not.

6. In Java, handling units systems like degrees and radians would be the programmer's responsibility. This is not applicable in the code provided.

7. Java provides a standard library of built-in functions and methods, and when invoking them, the number, attributes, and order of the arguments must be correct. In the code provided, it doesn't invoke any built-in functions.

8. In Java, method parameters are typically passed by value. Therefore, unless the parameter is explicitly an object or an array, the method cannot alter the original value of the parameter. However, in the code provided, this is not applicable as it doesn't contain explicit method calls with parameters.

9. Java does not have global variables in the same way that languages like C or C++ do. In Java, variables are typically defined within a class or method scope, and their accessibility is controlled by access modifiers. Therefore, the questions related to global variables are not applicable to the code.

<u>Category G:</u>

1. In the provided Java code, there are no explicitly declared files, so the question of whether their attributes are correct does not apply. Java primarily deals with file I/O using higher-level abstractions provided by classes like `File`, `InputStream`, `OutputStream`, and others.

2. In Java, there is no explicit "OPEN" statement as in some other languages like C. Instead, file I/O is typically performed using classes and methods from the `java.io` or `java.nio` packages. These classes offer methods to open, read, and write to files, and the attributes are set as method parameters rather than in an "OPEN" statement.

3. In Java, the memory required to hold the file is typically managed by the Java Virtual Machine (JVM). The code doesn't need to explicitly manage memory for file reading or writing. However, the code should be efficient in its use of memory.

4. The Java code doesn't explicitly open files using "OPEN" statements; instead, it relies on higher-level abstractions like `FileInputStream` and `FileOutputStream` to read and write files. These classes handle file opening and closing internally.

5. In Java, it's essential to close files after use to release system resources. The code should include proper file-closing statements using the `close()` method for input/output streams. This ensures that the resources are released when they are no longer needed.

6. The code does not explicitly perform file reads where end-of-file conditions need to be handled. However, if it were reading from a file, it should be done using loops that check for the end-of-file condition or use exceptions handling.

7. Java provides exception handling mechanisms to handle I/O error conditions. It's important to include proper error handling using `try-catch` blocks to deal with potential exceptions, such as `IOException`, that can occur during file operations.

8. The code does not include any text printed or displayed, so the question of spelling or grammatical errors in output is not applicable. However, when displaying text in a real application, it's important to ensure correct spelling and grammar.

Category H:

1. The code doesn't contain any variables that are never referenced or referenced only once because it is relatively straightforward and small in scope.

2. Since the code is in Java, which enforces strict type checking and attributes, there are no unexpected default attribute assignments.

3. The code does not generate any warnings or informational messages, as it is a simplified program without complex compiler messages.

4. The code doesn't include extensive input validation, but the provided code assumes that the user enters valid matrix dimensions and values. Improving robustness through input validation would be a good practice in a real-world application.

5. The code does not appear to be missing any functions, as it's a self-contained program to multiply matrices.

Que .1 I found Category A errors the most in the matrix multiplication code.
Que .2 I found Category A, B, D, E to be the most applicable in the current code.

There are several errors in the program. Here's a list of the errors:

1. When entering the number of rows and columns of the first matrix, there's a typo in the input prompt. It should be "Enter the number of rows and columns of first matrix," not "Enter the number of rows and columns of second matrix."

2. When entering the elements of the second matrix, there's another typo in the input prompt. It should be "Enter the elements of the second matrix," not "Enter the elements of second matrix."

3. In the matrix multiplication loop, the indices for accessing the `first` and `second` matrices are incorrect. They should start from 0, not from -1 as in `first[c-1][c-k]` and `second[k-1][k-d]`.

To fix these errors, follow these steps:

1. Update the input prompt for the first matrix:

```
System.out.println("Enter the number of rows and columns of first matrix");
```

2. Update the input prompt for the second matrix:

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

3. Update the matrix multiplication loop to correct the indices:

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

3. Here's the corrected and complete code:

```
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][k] * second[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");
            }
        }
    }
}
```

These changes should fix the errors in the code.

7. Quadratic Probing

Certainly, here are the answers to your questions for each category of checks:

Category A:
1. The goal is to ensure that all referenced variables have been initialized or set to a valid value before they are used in the code.

2. The goal is to check that all array subscripts fall within the defined bounds of the corresponding dimension to prevent out-of-bounds array access.
3. The goal is to ensure that array subscripts are of an integer type to prevent potentially dangerous practices.
4. The goal is to check that memory referenced through pointers or reference variables is currently allocated to avoid "dangling reference" issues.
5. The goal is to verify that alias names for the same memory area have consistent attributes when referenced to prevent data attribute mismatches.
6. The goal is to check that the value of a variable has the expected type or attribute, avoiding potential type mismatches.
7. The goal is to avoid addressing problems when the units of memory allocation are smaller than memory addressability units, which could lead to incorrect memory references.
8. The goal is to ensure that the referenced memory location via pointer or reference variables has the expected attributes, avoiding inconsistencies.
9. The goal is to ensure that data structures are defined identically in all procedures or subroutines that reference them.
10. The goal is to avoid off-by-one errors when indexing into strings or arrays.
11. For object-oriented languages, the goal is to verify that all inheritance requirements are met in implementing classes.

Category B: 1. The goal is to ensure that all variables are explicitly declared to prevent potential issues with undeclared or improperly defined variables.
2. The goal is to verify that default attribute values are well understood in the absence of explicit declarations.
3. The goal is to ensure proper initialization when declaring variables, particularly in complex data types like arrays or strings.
4. The goal is to check that each variable is assigned the correct length and data type.
5. The goal is to verify that variable initialization is consistent with its memory type.
6. The goal is to watch out for variables with similar names that could cause confusion or errors.

Category C:1. The goal is to prevent computations using variables with inconsistent data types that might lead to type-related errors.
2. The goal is to carefully handle mixed-mode computations and understand the language's conversion rules.
3. The goal is to avoid errors when computations involve variables with different data types but similar lengths.
4. The goal is to ensure that the data type of the target variable in an assignment is appropriate for the result.

5. The goal is to prevent overflow or underflow errors by checking the data type limits during computation.
6. The goal is to avoid division by zero.
7. The goal is to handle inaccuracies that may result from base-2 representation in binary machines.
8. The goal is to check that the value of variables remains within the meaningful range.

Category D:1. The goal is to avoid comparisons between variables of different data types to ensure meaningful comparisons.
2. The goal is to carefully handle mixed-mode comparisons by understanding conversion rules.
3. The goal is to use correct comparison operators to express intended relationships between variables.
4. The goal is to ensure that Boolean expressions state what they are supposed to state by using logical operators correctly.
5. The goal is to avoid mixing comparison and Boolean operators improperly.
6. The goal is to handle comparisons between fractional or floating-point numbers correctly, considering potential inaccuracies due to base-2 representation.
7. The goal is to understand the order of evaluation and precedence of operators in complex expressions.
8. The goal is to ensure that the compiler's evaluation of Boolean expressions does not affect the program's functionality negatively.

Category E:
1. The goal is to avoid index variables exceeding the number of branch possibilities in multiway branches.
2. The goal is to ensure that every loop will eventually terminate.
3. The goal is to confirm that the program, module, or subroutine will eventually terminate.
4. The goal is to identify and rectify any situations where loops may never execute due to conditions on entry.
5. The goal is to account for loop fall-through in loops controlled by both iteration and Boolean conditions.
6. The goal is to prevent off-by-one errors in loops, especially in zero-based loops.
7. The goal is to make sure that statement groups and code blocks have proper opening and closing brackets.
8. The goal is to check for non-exhaustive decisions, ensuring that assumptions made regarding input values are valid.

Category F:

1. The goal is to verify that the number and order of parameters and arguments match in module interfaces.
2. The goal is to ensure that attributes of parameters and arguments are consistent in module interfaces.
3. The goal is to confirm that the units system of parameters and arguments matches in module interfaces.
4. The goal is to check that the number of arguments transmitted to a module matches the number of parameters expected by that module.
5. The goal is to ensure that attributes of arguments match the attributes of parameters in called modules.
6. The goal is to verify that the units system of arguments transmitted to other modules matches the units system of the corresponding parameters.
7. The goal is to check that built-in functions are invoked with the correct number, attributes, and order of arguments.
8. The goal is to avoid unintended alteration of parameters in subroutines designed to have them as input values.
9. The goal is to ensure that global variables have consistent definitions and attributes in all modules that reference them.

Category G
1. The goal is to verify that file attributes are correct when explicitly declared in the code.
2. The goal is to confirm that the attributes specified in the file's OPEN statement are correct.
3. The goal is to ensure that the program has sufficient memory to hold the files it reads.
4. The goal is to check that all files are opened before use.
5. The goal is to confirm that all files are closed after use.
6. The goal is to detect and handle end-of-file conditions correctly.
7. The goal is to handle I/O error conditions correctly.
8. The goal is to avoid spelling or grammatical errors in printed or displayed text generated by the program.

Category H: Other Checks
1. The goal is to identify variables that are never referenced or are referenced only once, as reported in a cross-reference listing of identifiers.
2. The goal is to ensure that the attributes of each variable are as expected, checking an attribute listing for potential default attribute issues.
3. The goal is to carefully review any warning or informational messages generated by the compiler to address issues that might affect code quality and optimization.
4. The goal is to check for robustness, including input validation in the program.
5. The goal is to confirm that all required functions are present in the program.

8. Sorting Array

```
// sorting the array in ascending order
import java.util.Scanner;
public class Ascending _Order
{
    public static void main(String[] args)
    {
        int n, temp;
        Scanner s = new Scanner(System.in);
        System.out.print("Enter no. of elements you want in array:");
        n = s.nextInt();
        int a[] = new int[n];
        System.out.println("Enter all the elements:");
        for (int i = 0; i < n; i++)
        {
            a[i] = s.nextInt();
        }
        for (int i = 0; i >= n; i++);
        {
            for (int j = i + 1; j < n; j++)
            {
                if (a[i] <= a[j])
                {
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }
        System.out.print("Ascending Order:");
        for (int i = 0; i < n - 1; i++)
        {
            System.out.print(a[i] + ",");
        }
        System.out.print(a[n - 1]);
    }
}
```

```
Input: Enter no. of elements you want in array: 5
       Enter all elements:
       1 12 2 9 7
       1 2 7 9 12
```

Category A:

1. No uninitialized variables are detected in the code.
2. There is an issue with array bounds. The loop condition in the nested loop should be "i < n," but it's incorrectly written as "i >= n."
3. Subscript values are all integers, so this is not an issue.
4. There are no pointer or reference variables used in this code.
5. There are no aliasing issues in this code.
6. All variables have the correct data types and attributes based on the code.
7. There are no explicit or implicit addressing problems in this code.
8. There are no pointer variable issues in this code.
9. The data structure used (array) is defined consistently in the single procedure.
10. There is an off-by-one error in the loop index used for printing the sorted array.
11. There are no object-oriented concepts used in the provided code, so this question isn't directly applicable.

Category B:

1. All variables are explicitly declared in the code.
2. The default attributes of variables are well understood.
3. Variables are properly initialized in the code.
4. All variables have the correct length and data type.
5. The initialization of variables is consistent with their memory type.
6. There are no variables with similar names in the code.

Category C:

1. There are no computations using variables with inconsistent data types.
2. There's a mixed-mode computation issue in the "z = x/y;" line due to integer division.
3. All variables used in computations have the same data type and length.
4. The data type of the target variable is consistent with the result of the right-hand expression.

5. There is a potential overflow issue in the division operation "z = x/y;" if "y" is 0.
6. There is no explicit check for a divisor being zero, which could lead to a runtime exception in the provided code.
7. The code doesn't contain explicit computations that could reveal inaccuracies due to binary representation.
8. There are no explicit checks in the provided code to ensure that the value of a variable stays within a meaningful range.
9. The code doesn't contain complex expressions that would require attention to the order of evaluation and operator precedence.
10. In the provided code, there is no explicit use of integer arithmetic that might lead to unexpected results.

Category D:

1. Comparisons are made between variables of the same data type.
2. There are no mixed-mode comparisons in the code.
3. The comparison operators are used correctly.
4. Boolean expressions are written correctly.
5. There are no comparisons between fractional or floating-point numbers.
6. The order of evaluation of Boolean expressions is clear.
7. The way the compiler evaluates Boolean expressions doesn't affect the code.
8. There are no instances in the provided code where the compiler's evaluation of Boolean expressions would significantly impact the program's behavior.

Category E:

1. The computed GO TO statement is not present in the code.
2. All loops appear to eventually terminate.
3. The main program will eventually terminate.
4. There's a condition in the code where the loop might not execute if "n" is less than or equal to 0.
5. The loop fall-through condition is not present.
6. There is an off-by-one error in the loop index for printing the sorted array.
7. The code contains proper code blocks.
8. There are no instances in the provided code where the compiler's evaluation of Boolean expressions would significantly impact the program's behavior.

Category F:

1. The number of parameters received by the main method matches the number of arguments sent.
2. The attributes of parameters and arguments match in data type.
3. Units systems do not apply in this code.
4. The code doesn't involve transmitting arguments to other modules or functions.
5. No arguments are transmitted to other modules.
6. No built-in functions are invoked.
7. Subroutines don't alter parameters in this code.
8. There are no global variables used in this code.
9. There are no instances in the provided code where the compiler's evaluation of Boolean expressions would significantly impact the program's behavior.

Category G:

1. The code does not deal with file operations, so these checks do not apply.
2. Since the code doesn't deal with file operations, there is no use of the file's OPEN statement.
3. The code doesn't involve any file reading or writing operations, so memory availability in relation to file reading is not applicable here.
4. As there are no file operations in the code, there are no files being opened.
5. Since there are no file operations, there is no explicit opening or closing of files in the provided code.
6. There is no file handling in the code, so there are no end-of-file conditions to be handled.
7. The code doesn't involve any I/O operations, so I/O error handling is not relevant.
8. The provided code does not contain any text that is printed or displayed, so there are no issues related to spelling or grammatical errors in the context of this code.

Category H:

1. The cross-reference listing of identifiers is not applicable as there are no unused or referenced-only-once variables.
2. The code does not produce an attribute listing.
3. The code does not produce any warning or informational messages.
4. The program checks for the validity of user input, making it somewhat robust.
5. There are no missing functions in the code.

Program Inspection:

1. There are several errors in the code:

    a. There is a space in the class name "Ascending _Order." It should be "Ascending_Order" without spaces.
    b. There is an extra semicolon at the end of the first for loop: `for (int i = 0; i >= n; i++);`. The semicolon should be removed, and the loop should be fixed to `for (int i = 0; i < n; i++)`.
2. The category of program inspection that would be more effective in identifying these errors is "Code Review" or "Manual Code Inspection." These methods involve a human reviewer carefully analyzing the code for errors and improvements.
3. Some types of errors that might be challenging to identify using program inspection alone are logical errors or errors that don't result in compilation or runtime errors but still produce incorrect results. Code inspections are good at finding syntax errors, but they may not always uncover more subtle logic issues.
4. Program inspection is a valuable technique for identifying and correcting errors in code. It helps improve code quality, identify potential issues early in the development process, and ensure that code follows best practices. It is definitely worth applying, especially for complex or critical software projects. However, it should be complemented with other testing techniques (e.g., unit testing, integration testing) to ensure thorough coverage and error detection.

9. Stack Implementation

```
// 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
```

Category A:

1. No uninitialized variables are detected in the code.
2. There is an issue with array bounds. The loop condition in the nested loop should be "i < n," but it's incorrectly written as "i >= n."
3. Subscript values are all integers, so this is not an issue.
4. There are no pointer or reference variables used in this code.
5. There are no aliasing issues in this code.
6. All variables have the correct data types and attributes based on the code.
7. There are no explicit or implicit addressing problems in this code.
8. There are no pointer variable issues in this code.
9. The data structure used (array) is defined consistently in the single procedure.
10. There is an off-by-one error in the loop index used for printing the sorted array.
11. There are no object-oriented concepts used in the provided code, so this question isn't directly applicable.

Category B:

1. All variables are explicitly declared in the code.
2. The default attributes of variables are well understood.
3. Variables are properly initialized in the code.
4. All variables have the correct length and data type.
5. The initialization of variables is consistent with their memory type.
6. There are no variables with similar names in the code.

Category C:

1. There are no computations using variables with inconsistent data types.
2. There's a mixed-mode computation issue in the "z = x/y;" line due to integer division.
3. All variables used in computations have the same data type and length.
4. The data type of the target variable is consistent with the result of the right-hand expression.
5. There is a potential overflow issue in the division operation "z = x/y;" if "y" is 0.
6. There is no explicit check for a divisor being zero, which could lead to a runtime exception in the provided code.
7. The code doesn't contain explicit computations that could reveal inaccuracies due to binary representation.
8. There are no explicit checks in the provided code to ensure that the value of a variable stays within a meaningful range.
9. The code doesn't contain complex expressions that would require attention to the order of evaluation and operator precedence.
10. In the provided code, there is no explicit use of integer arithmetic that might lead to unexpected results.

Category D:

1. Comparisons are made between variables of the same data type.
2. There are no mixed-mode comparisons in the code.
3. The comparison operators are used correctly.
4. Boolean expressions are written correctly.
5. There are no comparisons between fractional or floating-point numbers.
6. The order of evaluation of Boolean expressions is clear.
7. The way the compiler evaluates Boolean expressions doesn't affect the code.
8. There are no instances in the provided code where the compiler's evaluation of Boolean expressions would significantly impact the program's behavior.

Category E:

1. The computed GO TO statement is not present in the code.
2. All loops appear to eventually terminate.
3. The main program will eventually terminate.
4. There's a condition in the code where the loop might not execute if "n" is less than or equal to 0.
5. The loop fall-through condition is not present.
6. There is an off-by-one error in the loop index for printing the sorted array.
7. The code contains proper code blocks.
8. There are no instances in the provided code where the compiler's evaluation of Boolean expressions would significantly impact the program's behavior.

Category F:

1. The number of parameters received by the main method matches the number of arguments sent.
2. The attributes of parameters and arguments match in data type.
3. Units systems do not apply in this code.
4. The code doesn't involve transmitting arguments to other modules or functions.
5. No arguments are transmitted to other modules.
6. No built-in functions are invoked.
7. Subroutines don't alter parameters in this code.
8. There are no global variables used in this code.
9. There are no instances in the provided code where the compiler's evaluation of Boolean expressions would significantly impact the program's behavior.

Category G:

1. The code does not deal with file operations, so these checks do not apply.
2. Since the code doesn't deal with file operations, there is no use of the file's OPEN statement.
3. The code doesn't involve any file reading or writing operations, so memory availability in relation to file reading is not applicable here.
4. As there are no file operations in the code, there are no files being opened.
5. Since there are no file operations, there is no explicit opening or closing of files in the provided code.
6. There is no file handling in the code, so there are no end-of-file conditions to be handled.

7. The code doesn't involve any I/O operations, so I/O error handling is not relevant.
8. The provided code does not contain any text that is printed or displayed, so there are no issues related to spelling or grammatical errors in the context of this code.

Category H:

1. The cross-reference listing of identifiers is not applicable as there are no unused or referenced-only-once variables.
2. The code does not produce an attribute listing.
3. The code does not produce any warning or informational messages.
4. The program checks for the validity of user input, making it somewhat robust.
5. There are no missing functions in the code.

Program Inspection:

1. There are several errors in the code:
   a. There is a space in the class name "Ascending _Order." It should be "Ascending_Order" without spaces.
   b. There is an extra semicolon at the end of the first for loop: `for (int i = 0; i >= n; i++);`. The semicolon should be removed, and the loop should be fixed to `for (int i = 0; i < n; i++)`.
2. The category of program inspection that would be more effective in identifying these errors is "Code Review" or "Manual Code Inspection." These methods involve a human reviewer carefully analyzing the code for errors and improvements.
3. Some types of errors that might be challenging to identify using program inspection alone are logical errors or errors that don't result in compilation or runtime errors but still produce incorrect results. Code inspections are good at finding syntax errors, but they may not always uncover more subtle logic issues.
4. Program inspection is a valuable technique for identifying and correcting errors in code. It helps improve code quality, identify potential issues early in the development process, and ensure that code follows best practices. It is definitely worth applying, especially for complex or critical software projects. However, it should be complemented with other testing techniques (e.g., unit testing, integration testing) to ensure thorough coverage and error detection.

10. Tower of Hanoi

```
// 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
```

Category A:

1. No uninitialized variables are detected in the code.
2. There is an issue with array bounds. The loop condition in the nested loop should be "i < n," but it's incorrectly written as "i >= n."
3. Subscript values are all integers, so this is not an issue.
4. There are no pointer or reference variables used in this code.
5. There are no aliasing issues in this code.
6. All variables have the correct data types and attributes based on the code.
7. There are no explicit or implicit addressing problems in this code.
8. There are no pointer variable issues in this code.
9. The data structure used (array) is defined consistently in the single procedure.
10. There is an off-by-one error in the loop index used for printing the sorted array.
11. There are no object-oriented concepts used in the provided code, so this question isn't directly applicable.

Category B:

1. All variables are explicitly declared in the code.
2. The default attributes of variables are well understood.
3. Variables are properly initialized in the code.
4. All variables have the correct length and data type.
5. The initialization of variables is consistent with their memory type.
6. There are no variables with similar names in the code.

Category C: Computation Errors

1. There are no computations using variables with inconsistent data types.
2. There's a mixed-mode computation issue in the "z = x/y;" line due to integer division.
3. All variables used in computations have the same data type and length.
4. The data type of the target variable is consistent with the result of the right-hand expression.
5. There is a potential overflow issue in the division operation "z = x/y;" if "y" is 0.
6. There is no explicit check for a divisor being zero, which could lead to a runtime exception in the provided code.
7. The code doesn't contain explicit computations that could reveal inaccuracies due to binary representation.
8. There are no explicit checks in the provided code to ensure that the value of a variable stays within a meaningful range.

9. The code doesn't contain complex expressions that would require attention to the order of evaluation and operator precedence.
10. In the provided code, there is no explicit use of integer arithmetic that might lead to unexpected results.

Category D: Comparison Errors

1. Comparisons are made between variables of the same data type.
2. There are no mixed-mode comparisons in the code.
3. The comparison operators are used correctly.
4. Boolean expressions are written correctly.
5. There are no comparisons between fractional or floating-point numbers.
6. The order of evaluation of Boolean expressions is clear.
7. The way the compiler evaluates Boolean expressions doesn't affect the code.
8. There are no instances in the provided code where the compiler's evaluation of Boolean expressions would significantly impact the program's behavior.

Category E: Control-Flow Errors

1. The computed GO TO statement is not present in the code.
2. All loops appear to eventually terminate.
3. The main program will eventually terminate.
4. There's a condition in the code where the loop might not execute if "n" is less than or equal to 0.
5. The loop fall-through condition is not present.
6. There is an off-by-one error in the loop index for printing the sorted array.
7. The code contains proper code blocks.
8. There are no instances in the provided code where the compiler's evaluation of Boolean expressions would significantly impact the program's behavior.

Category F: Interface Errors

1. The number of parameters received by the main method matches the number of arguments sent.
2. The attributes of parameters and arguments match in data type.
3. Units systems do not apply in this code.
4. The code doesn't involve transmitting arguments to other modules or functions.
5. No arguments are transmitted to other modules.
6. No built-in functions are invoked.
7. Subroutines don't alter parameters in this code.

8. There are no global variables used in this code.
9. There are no instances in the provided code where the compiler's evaluation of Boolean expressions would significantly impact the program's behavior.

Category G: Input/Output Errors

1. The code does not deal with file operations, so these checks do not apply.
2. Since the code doesn't deal with file operations, there is no use of the file's OPEN statement.
3. The code doesn't involve any file reading or writing operations, so memory availability in relation to file reading is not applicable here.
4. As there are no file operations in the code, there are no files being opened.
5. Since there are no file operations, there is no explicit opening or closing of files in the provided code.
6. There is no file handling in the code, so there are no end-of-file conditions to be handled.
7. The code doesn't involve any I/O operations, so I/O error handling is not relevant.
8. The provided code does not contain any text that is printed or displayed, so there are no issues related to spelling or grammatical errors in the context of this code.

Category H: Other Checks

1. The cross-reference listing of identifiers is not applicable as there are no unused or referenced-only-once variables.
2. The code does not produce an attribute listing.
3. The code does not produce any warning or informational messages.
4. The program checks for the validity of user input, making it somewhat robust.
5. There are no missing functions in the code.

Program Inspection:

1. There are several errors in the code:
    a. There is a space in the class name "Ascending _Order." It should be "Ascending_Order" without spaces.
    b. There is an extra semicolon at the end of the first for loop: `for (int i = 0; i >= n; i++);`. The semicolon should be removed, and the loop should be fixed to `for (int i = 0; i < n; i++)`.
2. The category of program inspection that would be more effective in identifying these errors is "Code Review" or "Manual Code Inspection." These methods involve a human reviewer carefully analyzing the code for errors and improvements.

3. Some types of errors that might be challenging to identify using program inspection alone are logical errors or errors that don't result in compilation or runtime errors but still produce incorrect results. Code inspections are good at finding syntax errors, but they may not always uncover more subtle logic issues.
4. Program inspection is a valuable technique for identifying and correcting errors in code. It helps improve code quality, identify potential issues early in the development process, and ensure that code follows best practices. It is definitely worth applying, especially for complex or critical software projects. However, it should be complemented with other testing techniques (e.g., unit testing, integration testing) to ensure thorough coverage and error detection.