IT314 : Software Engineering

LAB - 07 : Program Inspection, Debugging and Static Analysis

Name : Tushar S. Mori

ID : 202201502

I. PROGRAM INSPECTION:

(1) File Name : Armstrong.txt

A. Data Reference Errors

1. **No unset/uninitialized variables:** All variables are initialized before use.
2. **Array references:** Not applicable (no arrays used).
3. **Integer subscripts:** Not applicable.
4. **Dangling references:** Not applicable.
5. **Alias names:** Not applicable.
6. **Variable value types:** Correct types used.
7. **Addressing problems:** Not applicable.
8. **Pointer/reference attributes:** Not applicable.
9. **Data structure consistency:** Not applicable.
10. **Off-by-one errors in indexing:** Not applicable.
11. **Inheritance requirements:** Not applicable.

B. Data-Declaration Errors

1. **Explicitly declared variables:** All variables are declared properly.
2. **Default attributes understood:** Not applicable.
3. **Proper initialization:** All variables are initialized correctly.
4. **Correct length and data type:** All variables have appropriate types.

5. **Memory type initialization:** Not applicable.
6. **Similar variable names:** No confusing names found.

C. Computation Errors

1. **Inconsistent data types:** No inconsistencies found.
2. **Mixed-mode computations:** Not applicable.
3. **Different lengths of variables:** Not applicable.
4. **Data type of target variable:** No issues.
5. **Overflow/underflow expressions:** Not applicable (int type is sufficient for input).
6. **Divisor being zero:** Not applicable.
7. **Base-2 representation issues:** Not applicable.
8. **Value outside meaningful range:** Not applicable.
9. **Order of evaluation/precedence:** No issues.
10. **Invalid integer arithmetic:** Incorrect calculation logic when checking for Armstrong condition.

D. Comparison Errors

1. **Comparisons of different data types:** No issues found.
2. **Mixed-mode comparisons:** Not applicable.
3. **Comparison operators:** Correctly used.
4. **Boolean expressions:** Not applicable.
5. **Boolean operator operands:** Not applicable.
6. **Floating-point comparisons:** Not applicable.
7. **Order of evaluation with Boolean operators:** Not applicable.
8. **Compiler evaluation affecting the program:** Not applicable.

E. Control-Flow Errors

1. **Multiway branch:** Not applicable.
2. **Loop termination:** The while loop condition is valid.
3. **Module/subroutine termination:** All modules will eventually terminate.
4. **Loop execution:** Correctly implemented.
5. **Loop fall-through consequences:** Not applicable.
6. **Off-by-one errors:** Not applicable.
7. **Mismatched brackets:** No issues found.
8. **Non-exhaustive decisions:** Not applicable.

F. Interface Errors

1. **Parameter and argument count match:** Correct.
2. **Parameter attributes match arguments:** Correct.
3. **Units system match:** Not applicable.
4. **Arguments transmitted to another module:** Not applicable.
5. **Attributes of transmitted arguments match:** Not applicable.
6. **Units system match for transmitted arguments:** Not applicable.
7. **Built-in function arguments:** Not applicable.
8. **Subroutine alters input parameters:** Not applicable.
9. **Global variable definitions:** Not applicable.

G. Input/Output Errors

1. **File attributes:** No files declared.
2. **OPEN statement attributes:** Not applicable.
3. **Memory for file read:** Not applicable.
4. **Files opened before use:** Not applicable.
5. **Files closed after use:** Not applicable.
6. **End-of-file conditions:** Not applicable.
7. **I/O error conditions:** Handled appropriately.
8. **Spelling/grammatical errors:** "Armstrong" is incorrectly spelled in the output.

H. Other Checks

1. **Cross-reference listing of identifiers:** Not applicable.
2. **Attribute listing check:** Not applicable.
3. **Compiler warning messages:** Not applicable.
4. **Program robustness:** Validity checks could be added for user input.

Answers to Questions

1. **How many errors are there in the program? Mention the errors you have identified.**
    - **Errors Identified:** There is an error in calculating the Armstrong number where the wrong operation is performed to find the remainder (`num / 10` instead of `num % 10`), and the result of `num` is incorrectly used for checking the Armstrong condition.
2. **Which category of program inspection would you find more effective?**
    - **Most Effective Category:** Computation Errors, as they help identify logical errors in calculations that are crucial for determining whether the number is an Armstrong number.
3. **Which type of error are you not able to identify using the program inspection?**

    - **Type of Error Not Identified:** Edge cases for single-digit numbers and performance inefficiencies for larger numbers could be missed.
4. **Is the program inspection technique worth applicable?**
    - **Applicability of Technique:** Yes, program inspection techniques are valuable as they help identify a variety of potential errors and improve code quality through systematic review.

(2) File Name : GCD and LCM.txt

A. Data Reference Errors

1. **No unset/uninitialized variables**: All variables are initialized before use.
2. **Array references**: Not applicable (no arrays used).
3. **Integer subscripts**: Not applicable.
4. **Dangling references**: Not applicable.
5. **Alias names**: Not applicable.
6. **Variable value types**: Correct types used.
7. **Addressing problems**: Not applicable.
8. **Pointer/reference attributes**: Not applicable.
9. **Data structure consistency**: Not applicable.
10. **Off-by-one errors in indexing**: Not applicable.
11. **Inheritance requirements**: Not applicable.

B. Data-Declaration Errors

1. **Explicitly declared variables**: All variables are declared properly.
2. **Default attributes understood**: Not applicable.
3. **Proper initialization**: All variables are initialized correctly.
4. **Correct length and data type**: All variables have appropriate types.
5. **Memory type initialization**: Not applicable.
6. **Similar variable names**: No confusing names found.

C. Computation Errors

1. **Inconsistent data types**: No inconsistencies found.
2. **Mixed-mode computations**: No mixed-mode computations present.
3. **Different lengths of variables**: Not applicable.
4. **Data type of target variable**: No issues.
5. **Overflow/underflow expressions**: Not applicable (int type is sufficient for input).
6. **Divisor being zero**: Checked properly in GCD function.
7. **Base-2 representation issues**: Not applicable.
8. **Value outside meaningful range**: Not applicable.

9. **Order of evaluation/precedence**: No issues.
10. **Invalid integer arithmetic**: Not applicable.

D. Comparison Errors

1. **Comparisons of different data types**: No issues found.
2. **Mixed-mode comparisons**: Not applicable.
3. **Comparison operators**: Correctly used.
4. **Boolean expressions**: Not applicable.
5. **Boolean operator operands**: Not applicable.
6. **Floating-point comparisons**: Not applicable.
7. **Order of evaluation with Boolean operators**: Not applicable.
8. **Compiler evaluation affecting the program**: Not applicable.

E. Control-Flow Errors

1. **Multiway branch**: Not applicable.
2. **Loop termination**: The while loop in the GCD function has a potential logical error.
3. **Module/subroutine termination**: All modules will eventually terminate.
4. **Loop execution**: No oversight found.
5. **Loop fall-through consequences**: Not applicable.
6. **Off-by-one errors**: No issues found.
7. **Mismatched brackets**: No issues found.
8. **Non-exhaustive decisions**: Not applicable.

F. Interface Errors

1. **Parameter and argument count match**: Correct.
2. **Parameter attributes match arguments**: Correct.
3. **Units system match**: Not applicable.
4. **Arguments transmitted to another module**: Not applicable.
5. **Attributes of transmitted arguments match**: Not applicable.
6. **Units system match for transmitted arguments**: Not applicable.
7. **Built-in function arguments**: Not applicable.
8. **Subroutine alters input parameters**: Not applicable.
9. **Global variable definitions**: Not applicable.

G. Input / Output Errors

1. **File attributes**: No files declared.
2. **OPEN statement attributes**: Not applicable.
3. **Memory for file read**: Not applicable.
4. **Files opened before use**: Not applicable.

5. **Files closed after use**: Not applicable.
6. **End-of-file conditions**: Not applicable.
7. **I/O error conditions**: Handled appropriately.
8. **Spelling/grammatical errors**: No errors found in output.

H. Other Checks

1. **Cross-reference listing of identifiers**: Not applicable.
2. **Attribute listing check**: Not applicable.
3. **Compiler warning messages**: Not applicable.
4. **Program robustness**: Validity checks could be added for user input.

Answers to Questions

1. **How many errors are there in the program? Mention the errors you have identified.**
   - **Errors Identified**: There is a logical error in the GCD function where the while condition should be `while(a % b != 0)` instead of `while(a % b == 0)`. Additionally, the LCM function may not be efficient as it increments `a` unnecessarily; it could use the relationship between GCD and LCM.
2. **Which category of program inspection would you find more effective?**
   - **Most Effective Category**: **Control-Flow Errors**, as they help identify logical errors in loops and branching, which are critical for correct program execution.
3. **Which type of error are you not able to identify using the program inspection?**
   - **Type of Error Not Identified**: **Performance inefficiencies**, such as the inefficiency in the LCM calculation which could be optimized using the relationship with GCD.
4. **Is the program inspection technique worth applicable?**
   - **Applicability of Technique**: Yes, program inspection techniques are valuable as they help identify a variety of potential errors and improve code quality through systematic review.

(3) File Name : Knapsack.txt

A. Data Reference Errors

1. **Unset/uninitialized variables**: All variables are properly initialized before use.
2. **Array references**: The use of `profit` and `weight` arrays is appropriate.
3. **Integer subscripts**: Indexing of arrays is correctly handled.
4. **Dangling references**: No dangling references found.

5. **Alias names**: Not applicable.
6. **Variable value types**: Correct types are used.
7. **Addressing problems**: Not applicable.
8. **Pointer/reference attributes**: Not applicable.
9. **Data structure consistency**: No issues found.
10. **Off-by-one errors in indexing**: Proper handling of array lengths.
11. **Inheritance requirements**: Not applicable.

B. Data-Declaration Errors

1. **Explicitly declared variables**: All variables are declared correctly.
2. **Default attributes understood**: Not applicable.
3. **Proper initialization**: All variables are initialized.
4. **Correct length and data type**: Proper types are used for all variables.
5. **Memory type initialization**: Not applicable.
6. **Similar variable names**: No confusing variable names.

C. Computation Errors

1. **Inconsistent data types**: No inconsistencies found.
2. **Mixed-mode computations**: Not applicable.
3. **Different lengths of variables**: Not applicable.
4. **Data type of target variable**: Correct data types used.
5. **Overflow/underflow expressions**: Possible risk of overflow in profit calculation; could be mitigated by checking ranges.
6. **Divisor being zero**: Not applicable.
7. **Base-2 representation issues**: Not applicable.
8. **Value outside meaningful range**: Randomly generated weights could result in zero.
9. **Order of evaluation/precedence**: Correct usage.
10. **Invalid integer arithmetic**: Not applicable.

D. Comparison Errors

1. **Comparisons of different data types**: No issues found.
2. **Mixed-mode comparisons**: Not applicable.
3. **Comparison operators**: Correctly used.
4. **Boolean expressions**: No issues found.
5. **Boolean operator operands**: No issues found.
6. **Floating-point comparisons**: Not applicable.
7. **Order of evaluation with Boolean operators**: Not applicable.
8. **Compiler evaluation affecting the program**: Not applicable.

E. Control-Flow Errors

1. **Multiway branch**: Not applicable.
2. **Loop termination**: The loop handling in the profit and weight calculations is correct but `n++` should be `n` to ensure proper indexing.
3. **Module/subroutine termination**: All modules terminate correctly.
4. **Loop execution**: Correct execution flow.
5. **Loop fall-through consequences**: Not applicable.
6. **Off-by-one errors**: Not applicable.
7. **Mismatched brackets**: No issues found.
8. **Non-exhaustive decisions**: Not applicable.

F. Interface Errors

1. **Parameter and argument count match**: Correct.
2. **Parameter attributes match arguments**: Correct.
3. **Units system match**: Not applicable.
4. **Arguments transmitted to another module**: Not applicable.
5. **Attributes of transmitted arguments match**: Not applicable.
6. **Units system match for transmitted arguments**: Not applicable.
7. **Built-in function arguments**: Not applicable.
8. **Subroutine alters input parameters**: Not applicable.
9. **Global variable definitions**: Not applicable.

G. Input / Output Errors

1. **File attributes**: Not applicable (no file operations).
2. **OPEN statement attributes**: Not applicable.
3. **Memory for file read**: Not applicable.
4. **Files opened before use**: Not applicable.
5. **Files closed after use**: Not applicable.
6. **End-of-file conditions**: Not applicable.
7. **I/O error conditions**: Handled appropriately.
8. **Spelling/grammatical errors**: No errors in the output.

H. Other Checks

1. **Cross-reference listing of identifiers**: Not applicable.
2. **Attribute listing check**: Not applicable.
3. **Compiler warning messages**: No warnings noted.
4. **Program robustness**: Random weight generation could lead to invalid scenarios (e.g., zero weight).

Answers to Questions

1. **How many errors are there in the program? Mention the errors you have identified.**
    - **Errors Identified**: The loop in the knapsack algorithm should use `n` instead of `n++` when referring to the `opt` array, as incrementing `n` alters the logic. Additionally, `option2` calculations use `profit[n-2]`, which may lead to index errors.
2. **Which category of program inspection would you find more effective?**
    - **Most Effective Category**: **Control-Flow Errors**, as they can help identify logical issues in loops and branching which are crucial for correct execution.
3. **Which type of error are you not able to identify using the program inspection?**
    - **Type of Error Not Identified**: **Performance issues**, such as potential inefficiencies in handling weights and profits due to the random generation of values, which may not lead to meaningful results.
4. **Is the program inspection technique worth applicable?**
    - **Applicability of Technique**: Yes, program inspection techniques are beneficial for identifying logical, structural, and potential performance issues, enhancing the overall quality of the code.

(4) File Name : Magic Number.txt

A. Data Reference Errors

1. **Unset/uninitialized variables**: All variables (`sum`, `num`, `s`) are properly initialized before use.
2. **Array references**: No arrays are used in this code.
3. **Integer subscripts**: Not applicable as no subscripts are involved.
4. **Dangling references**: No dangling references found.
5. **Alias names**: Not applicable.
6. **Variable value types**: All variables use appropriate types.
7. **Addressing problems**: Not applicable.
8. **Pointer/reference attributes**: Not applicable.
9. **Data structure consistency**: No issues found.
10. **Off-by-one errors in indexing**: Not applicable.
11. **Inheritance requirements**: Not applicable.

B. Data-Declaration Errors

1. **Explicitly declared variables**: All variables are declared correctly.
2. **Default attributes understood**: Not applicable.
3. **Proper initialization**: All variables are initialized correctly.
4. **Correct length and data type**: Proper data types used for all variables.

5. **Memory type initialization**: Not applicable.
6. **Similar variable names**: No confusing variable names.

C. Computation Errors

1. **Inconsistent data types**: No inconsistencies found.
2. **Mixed-mode computations**: Not applicable.
3. **Different lengths of variables**: Not applicable.
4. **Data type of target variable**: Correct data types used.
5. **Overflow/underflow expressions**: Not applicable; integers are used.
6. **Divisor being zero**: Not applicable.
7. **Base-2 representation issues**: Not applicable.
8. **Value outside meaningful range**: Not applicable.
9. **Order of evaluation/precedence**: Incorrect handling in nested while loops.
10. **Invalid integer arithmetic**: Found an issue in the nested loop.

D. Comparison Errors

1. **Comparisons of different data types**: No issues found.
2. **Mixed-mode comparisons**: Not applicable.
3. **Comparison operators**: Correctly used.
4. **Boolean expressions**: No issues found.
5. **Boolean operator operands**: No issues found.
6. **Floating-point comparisons**: Not applicable.
7. **Order of evaluation with Boolean operators**: Not applicable.
8. **Compiler evaluation affecting the program**: Not applicable.

E. Control-Flow Errors

1. **Multiway branch**: Not applicable.
2. **Loop termination**: The first while loop correctly checks for `num > 9`, but the inner while loop has a condition that never executes correctly.
3. **Module/subroutine termination**: All modules terminate correctly.
4. **Loop execution**: Incorrect logic in the inner loop.
5. **Loop fall-through consequences**: Not applicable.
6. **Off-by-one errors**: Not applicable.
7. **Mismatched brackets**: Missing semicolon in `sum=sum%10`.
8. **Non-exhaustive decisions**: Not applicable.

F. Interface Errors

1. **Parameter and argument count match**: Correct.
2. **Parameter attributes match arguments**: Correct.
3. **Units system match**: Not applicable.

4. **Arguments transmitted to another module**: Not applicable.
5. **Attributes of transmitted arguments match**: Not applicable.
6. **Units system match for transmitted arguments**: Not applicable.
7. **Built-in function arguments**: Not applicable.
8. **Subroutine alters input parameters**: Not applicable.
9. **Global variable definitions**: Not applicable.

G. Input / Output Errors

1. **File attributes**: Not applicable (no file operations).
2. **OPEN statement attributes**: Not applicable.
3. **Memory for file read**: Not applicable.
4. **Files opened before use**: Not applicable.
5. **Files closed after use**: Not applicable.
6. **End-of-file conditions**: Not applicable.
7. **I/O error conditions**: Handled appropriately.
8. **Spelling/grammatical errors**: No errors in output.

H. Other Checks

1. **Cross-reference listing of identifiers**: Not applicable.
2. **Attribute listing check**: Not applicable.
3. **Compiler warning messages**: No warnings noted.
4. **Program robustness**: The program logic for magic number checks needs improvement to ensure correct calculations.

Answers to Questions

1. **How many errors are there in the program? Mention the errors you have identified.**
    - **Errors Identified**: The inner while loop uses `while(sum==0)` instead of the correct condition to iterate through digits. Additionally, the expression `s=s*(sum/10)` should be `s=s+(sum%10)` to accumulate the sum of digits.
2. **Which category of program inspection would you find more effective?**
    - **Most Effective Category**: **Control-Flow Errors**, as they are crucial for ensuring the logical correctness of loops and conditional statements.
3. **Which type of error are you not able to identify using the program inspection?**
    - **Type of Error Not Identified**: **Logical errors** resulting from incorrect conditions in loops, which may not be caught through static inspection alone.
4. **Is the program inspection technique worth applicable?**

- **Applicability of Technique**: Yes, program inspection techniques are valuable for identifying logical, structural, and potential performance issues, enhancing the overall quality of the code.

(5) File Name : Merge Sort.txt

A. Data Reference Errors

1. **Unset/uninitialized variables**: No issues found; all variables are properly initialized.
2. **Array references**: The method leftHalf(array + 1) and rightHalf(array - 1) are incorrect.
3. **Integer subscripts**: Correct usage throughout.
4. **Dangling references**: Not applicable.
5. **Alias names**: No issues found.
6. **Variable value types**: All variable types are correctly used.
7. **Addressing problems**: Not applicable.
8. **Pointer/reference attributes**: Not applicable.
9. **Data structure consistency**: No issues found.
10. **Off-by-one errors in indexing**: Possible in leftHalf() and rightHalf().
11. **Inheritance requirements**: Not applicable.

B. Data-Declaration Errors

1. **Explicitly declared variables**: All variables declared correctly.
2. **Default attributes understood**: Correct.
3. **Proper initialization**: All variables are initialized properly.
4. **Correct length and data type**: Data types are correctly assigned.
5. **Memory type initialization**: No issues found.
6. **Similar variable names**: No confusing variable names.

C. Computation Errors

1. **Inconsistent data types**: No inconsistencies found.
2. **Mixed-mode computations**: Not applicable.
3. **Different lengths of variables**: Not applicable.
4. **Data type of target variable**: Proper data types used.
5. **Overflow/underflow expressions**: Not applicable.
6. **Divisor being zero**: Not applicable.
7. **Base-2 representation issues**: Not applicable.
8. **Value outside meaningful range**: Not applicable.
9. **Order of evaluation/precedence**: Not applicable.
10. **Invalid integer arithmetic**: Issues found in how the arrays are handled.

D. Comparison Errors

1. **Comparisons of different data types**: No issues found.
2. **Mixed-mode comparisons**: Not applicable.
3. **Comparison operators**: Correctly used.
4. **Boolean expressions**: No issues found.
5. **Boolean operator operands**: No issues found.
6. **Floating-point comparisons**: Not applicable.
7. **Order of evaluation with Boolean operators**: Not applicable.
8. **Compiler evaluation affecting the program**: Not applicable.

E. Control-Flow Errors

1. **Multiway branch**: Not applicable.
2. **Loop termination**: Proper termination found.
3. **Module/subroutine termination**: Correct.
4. **Loop execution**: No issues found.
5. **Loop fall-through consequences**: Not applicable.
6. **Off-by-one errors**: Possible in array handling.
7. **Mismatched brackets**: Not applicable.
8. **Non-exhaustive decisions**: Not applicable.

F. Interface Errors

1. **Parameter and argument count match**: Correct.
2. **Parameter attributes match arguments**: Correct.
3. **Units system match**: Not applicable.
4. **Arguments transmitted to another module**: Not applicable.
5. **Attributes of transmitted arguments match**: Not applicable.
6. **Units system match for transmitted arguments**: Not applicable.
7. **Built-in function arguments**: Correct.
8. **Subroutine alters input parameters**: Not applicable.
9. **Global variable definitions**: Not applicable.

G. Input / Output Errors

1. **File attributes**: Not applicable (no file operations).
2. **OPEN statement attributes**: Not applicable.
3. **Memory for file read**: Not applicable.
4. **Files opened before use**: Not applicable.
5. **Files closed after use**: Not applicable.
6. **End-of-file conditions**: Not applicable.
7. **I/O error conditions**: Handled appropriately.
8. **Spelling/grammatical errors**: No issues found.

H. Other Checks

1. **Cross-reference listing of identifiers**: Not applicable.
2. **Attribute listing check**: Not applicable.
3. **Compiler warning messages**: Possible warnings regarding incorrect array handling.
4. **Program robustness**: The merge sort implementation has logical flaws, particularly in array manipulations.

Answers to Questions

1. **How many errors are there in the program? Mention the errors you have identified.**
    - **Errors Identified**:
        - Incorrect array handling in `mergeSort()` where `array + 1` and `array - 1` should be using `Arrays.copyOfRange()`.
        - Incorrect usage of `left++` and `right--` in the `merge()` call, which should be just `left` and `right`.
2. **Which category of program inspection would you find more effective?**
    - **Most Effective Category**: **Computation Errors**, as they address logical errors in arithmetic and data handling.
3. **Which type of error are you not able to identify using the program inspection?**
    - **Type of Error Not Identified**: **Logical errors** related to incorrect array handling and boundaries, which may not be immediately visible through inspection.
4. **Is the program inspection technique worth applicable?**
    - **Applicability of Technique**: Yes, program inspection techniques are valuable for identifying structural, logical, and potential performance issues, contributing to code quality improvement.

(6) File Name : Multiply Matrics.txt

A. Data Reference Errors

1. **Unset/uninitialized variables**: `sum`, `c`, `d`, and `k` are initialized correctly.
2. **Array references**: Incorrectly indexed in `sum = sum + first[c-1][c-k]*second[k-1][k-d];`.
3. **Integer subscripts**: Properly declared, but accessed incorrectly.
4. **Dangling references**: Not applicable.
5. **Alias names**: No issues found.
6. **Variable value types**: All types are appropriate for their usage.
7. **Addressing problems**: Misaddressing due to incorrect array indices.

8. **Pointer/reference attributes**: Not applicable.
9. **Data structure consistency**: Consistent across the program.
10. **Off-by-one errors in indexing**: Multiple off-by-one errors in array indexing.
11. **Inheritance requirements**: Not applicable.

B. Data-Declaration Errors

1. **Explicitly declared variables**: All variables are properly declared.
2. **Default attributes understood**: Correctly used.
3. **Proper initialization**: Variables are initialized correctly.
4. **Correct length and data type**: Data types are correct.
5. **Memory type initialization**: Not applicable.
6. **Similar variable names**: No confusing variable names.

C. Computation Errors

1. **Inconsistent data types**: All types are consistent.
2. **Mixed-mode computations**: Not applicable.
3. **Different lengths of variables**: Not applicable.
4. **Data type of target variable**: Correctly used.
5. **Overflow/underflow expressions**: Not applicable.
6. **Divisor being zero**: Not applicable.
7. **Base-2 representation issues**: Not applicable.
8. **Value outside meaningful range**: Not applicable.
9. **Order of evaluation/precedence**: Not applicable.
10. **Invalid integer arithmetic**: Found issues in how multiplication and indexing are performed.

D. Comparison Errors

1. **Comparisons of different data types**: Not applicable.
2. **Mixed-mode comparisons**: Not applicable.
3. **Comparison operators**: Properly used.
4. **Boolean expressions**: No issues found.
5. **Boolean operator operands**: No issues found.
6. **Floating-point comparisons**: Not applicable.
7. **Order of evaluation with Boolean operators**: Not applicable.
8. **Compiler evaluation affecting the program**: Not applicable.

E. Control-Flow Errors

1. **Multiway branch**: Not applicable.
2. **Loop termination**: Properly defined.
3. **Module/subroutine termination**: Correct.

4. **Loop execution**: No issues found.
5. **Loop fall-through consequences**: Not applicable.
6. **Off-by-one errors**: Found in matrix indexing.
7. **Mismatched brackets**: Not applicable.

F. Interface Errors

1. **Parameter and argument count match**: Correct.
2. **Parameter attributes match arguments**: Correct.
3. **Units system match**: Not applicable.
4. **Arguments transmitted to another module**: Not applicable.
5. **Attributes of transmitted arguments match**: Not applicable.
6. **Units system match for transmitted arguments**: Not applicable.
7. **Built-in function arguments**: Not applicable.
8. **Subroutine alters input parameters**: Not applicable.
9. **Global variable definitions**: Not applicable.

G. Input / Output Errors

1. **File attributes**: Not applicable (no file operations).
2. **OPEN statement attributes**: Not applicable.
3. **Memory for file read**: Not applicable.
4. **Files opened before use**: Not applicable.
5. **Files closed after use**: Not applicable.
6. **End-of-file conditions**: Not applicable.
7. **I/O error conditions**: Handled correctly.
8. **Spelling/grammatical errors**: No issues found.

H. Other Checks

1. **Cross-reference listing of identifiers**: Not applicable.
2. **Attribute listing check**: Not applicable.
3. **Compiler warning messages**: May receive warnings regarding array access.
4. **Program robustness**: The program has logical flaws, especially in matrix indexing, that could lead to runtime errors or incorrect results.

Answers to Questions

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

        - Incorrectly accessing arrays in `sum = sum + first[c-1][c-k]*second[k-1][k-d];`. This should use `first[c][k]` and `second[k][d]` for proper multiplication.
        - Off-by-one errors in the loop conditions.
2. **Which category of program inspection would you find more effective?**
    - **Most Effective Category**: **Computation Errors**, as they directly relate to the core functionality of the matrix multiplication algorithm.
3. **Which type of error are you not able to identify using the program inspection?**
    - **Type of Error Not Identified**: **Logical errors** related to incorrect indexing, which may not be evident through inspection alone.
4. **Is the program inspection technique worth applicable?**
    - **Applicability of Technique**: Yes, program inspection techniques are valuable for catching structural issues and potential logical flaws, leading to improvements in code quality and reliability.

(7) File Name : Quadratic Probing.txt

A. Data Reference Errors

1. **Unset/uninitialized variables**: All variables are properly initialized.
2. **Array references**: The array access is generally correct but may have issues with collision handling.
3. **Integer subscripts**: Used appropriately but need careful checks in the hash function.
4. **Dangling references**: Not applicable here.
5. **Alias names**: No issues found.
6. **Variable value types**: Types are appropriate for their usage.
7. **Addressing problems**: Correctly indexed except for errors in the insertion and retrieval logic.
8. **Pointer/reference attributes**: Not applicable.
9. **Data structure consistency**: Consistent across the program.
10. **Off-by-one errors in indexing**: May occur if the `hash()` method results in a negative index.
11. **Inheritance requirements**: Not applicable.

B. Data-Declaration Errors

1. **Explicitly declared variables**: All variables are properly declared.
2. **Default attributes understood**: Correctly used.
3. **Proper initialization**: All necessary variables are initialized correctly.
4. **Correct length and data type**: Data types are correct.
5. **Memory type initialization**: Not applicable.

6. **Similar variable names**: No confusing variable names.

C. Computation Errors

1. **Inconsistent data types**: All types are consistent.
2. **Mixed-mode computations**: Not applicable.
3. **Different lengths of variables**: Not applicable.
4. **Data type of target variable**: Correctly used.
5. **Overflow/underflow expressions**: Potential for overflow in the hash method if maxSize is small.
6. **Divisor being zero**: Not applicable (no division by zero).
7. **Base-2 representation issues**: Not applicable.
8. **Value outside meaningful range**: The hash() function may return a negative index.
9. **Order of evaluation/precedence**: Potential issues in the expression for inserting elements.
10. **Invalid integer arithmetic**: Errors in the insertion and retrieval algorithms.

D. Comparison Errors

1. **Comparisons of different data types**: Not applicable.
2. **Mixed-mode comparisons**: Not applicable.
3. **Comparison operators**: Properly used.
4. **Boolean expressions**: No issues found.
5. **Boolean operator operands**: No issues found.
6. **Floating-point comparisons**: Not applicable.
7. **Order of evaluation with Boolean operators**: Not applicable.
8. **Compiler evaluation affecting the program**: Not applicable.

E. Control-Flow Errors

1. **Multiway branch**: Correctly implemented with switch-case.
2. **Loop termination**: Properly defined.
3. **Module/subroutine termination**: Correct.
4. **Loop execution**: No issues found.
5. **Loop fall-through consequences**: Not applicable.
6. **Off-by-one errors**: Possible errors when incrementing the h value and in the hash() function.
7. **Mismatched brackets**: Not applicable.

F. Interface Errors

1. **Parameter and argument count match**: Correct.
2. **Parameter attributes match arguments**: Correct.

3. **Units system match**: Not applicable.
4. **Arguments transmitted to another module**: Not applicable.
5. **Attributes of transmitted arguments match**: Not applicable.
6. **Units system match for transmitted arguments**: Not applicable.
7. **Built-in function arguments**: Not applicable.
8. **Subroutine alters input parameters**: Not applicable.
9. **Global variable definitions**: Not applicable.

G. Input / Output Errors

1. **File attributes**: Not applicable (no file operations).
2. **OPEN statement attributes**: Not applicable.
3. **Memory for file read**: Not applicable.
4. **Files opened before use**: Not applicable.
5. **Files closed after use**: Not applicable.
6. **End-of-file conditions**: Not applicable.
7. **I/O error conditions**: Handled correctly.
8. **Spelling/grammatical errors**: Minor issues in method names.

H. Other Checks

1. **Cross-reference listing of identifiers**: Not applicable.
2. **Attribute listing check**: Not applicable.
3. **Compiler warning messages**: Likely to have warnings related to improper array access.
4. **Program robustness**: Logical flaws and potential runtime errors could lead to exceptions or incorrect behavior.

Answers to Questions

1. **How many errors are there in the program? Mention the errors you have identified.**
    - **Errors Identified**:
        - In the insertion method, the statement `i + = (i + h / h--) % maxSize;` is incorrect due to a space; it should be `i += (i + h * h) % maxSize;` (correct increment logic).
        - In the retrieval method, the same logic error occurs.
        - The `hash()` method can produce negative indices; should ensure it always returns a non-negative value using `Math.abs()`.
        - Incorrectly using `h++` which modifies `h` before evaluating, leading to unintended increments.
2. **Which category of program inspection would you find more effective?**

    - **Most Effective Category**: **Computation Errors**, as they directly affect the algorithm's functionality and can lead to incorrect behavior.
3. **Which type of error are you not able to identify using the program inspection?**
    - **Type of Error Not Identified**: **Logical errors** related to handling collisions and proper key management, which may not be evident without testing the program.
4. **Is the program inspection technique worth applicable?**
    - **Applicability of Technique**: Yes, program inspection techniques are valuable for identifying structural issues and potential logical flaws, leading to improvements in code quality and reliability.

(8) File Name : Sorting Array.txt

A. Data Reference Errors

- **No unset/uninitialized variables:** All variables are initialized before use.
- **Array references:** The array is declared and initialized correctly.
- **Integer subscripts:** Subscript issues can arise in loops.
- **Dangling references:** Not applicable.
- **Alias names:** Not applicable.
- **Variable value types:** Correct types used.
- **Addressing problems:** The addressing in the nested loops contains an error.
- **Pointer/reference attributes:** Not applicable.
- **Data structure consistency:** The array's data structure is consistent.
- **Off-by-one errors in indexing:** Potential off-by-one error in the sorting loop condition.
- **Inheritance requirements:** Not applicable.

B. Data-Declaration Errors

- **Explicitly declared variables:** All variables are declared properly.
- **Default attributes understood:** Not applicable.
- **Proper initialization:** All variables are initialized correctly.
- **Correct length and data type:** All variables have appropriate types.
- **Memory type initialization:** Not applicable.
- **Similar variable names:** No confusing names found.

C. Computation Errors

- **Inconsistent data types:** No inconsistencies found.
- **Mixed-mode computations:** Not applicable.

- **Different lengths of variables:** Not applicable.
- **Data type of target variable:** No issues.
- **Overflow/underflow expressions:** Not applicable (int type is sufficient).
- **Divisor being zero:** Not applicable.
- **Base-2 representation issues:** Not applicable.
- **Value outside meaningful range:** Not applicable.
- **Order of evaluation/precedence:** Potential issue in sorting logic.
- **Invalid integer arithmetic:** The sorting condition is incorrectly set up.

D. Comparison Errors

- **Comparisons of different data types:** No issues found.
- **Mixed-mode comparisons:** Not applicable.
- **Comparison operators:** Used incorrectly in the sorting logic.
- **Boolean expressions:** Not applicable.
- **Boolean operator operands:** Not applicable.
- **Floating-point comparisons:** Not applicable.
- **Order of evaluation with Boolean operators:** Not applicable.
- **Compiler evaluation affecting the program:** Not applicable.

E. Control-Flow Errors

- **Multiway branch:** Not applicable.
- **Loop termination:** The condition for the first loop is incorrect (`i >= n` should be `i < n`).
- **Module/subroutine termination:** All modules will eventually terminate.
- **Loop execution:** The outer loop for sorting does not execute as intended due to the wrong condition.
- **Loop fall-through consequences:** Not applicable.
- **Off-by-one errors:** The sorting loop may result in unintentional behavior.
- **Mismatched brackets:** No issues found.
- **Non-exhaustive decisions:** Not applicable.

F. Interface Errors

- **Parameter and argument count match:** Correct.
- **Parameter attributes match arguments:** Correct.
- **Units system match:** Not applicable.
- **Arguments transmitted to another module:** Not applicable.
- **Attributes of transmitted arguments match:** Not applicable.
- **Units system match for transmitted arguments:** Not applicable.
- **Built-in function arguments:** Not applicable.

- **Subroutine alters input parameters:** Not applicable.
- **Global variable definitions:** Not applicable.

G. Input/Output Errors

- **File attributes:** No files declared.
- **OPEN statement attributes:** Not applicable.
- **Memory for file read:** Not applicable.
- **Files opened before use:** Not applicable.
- **Files closed after use:** Not applicable.
- **End-of-file conditions:** Not applicable.
- **I/O error conditions:** Handled appropriately.
- **Spelling/grammatical errors:** Not applicable.

H. Other Checks

- **Cross-reference listing of identifiers:** Not applicable.
- **Attribute listing check:** Not applicable.
- **Compiler warning messages:** No warnings found.
- **Program robustness:** Validity checks could be added for user input.

Answers to Questions

1. **How many errors are there in the program? Mention the errors you have identified.**
    - **Errors Identified:** There are two main errors:
        - The outer loop's condition is incorrect (`for (int i = 0; i >= n; i++);` should be `for (int i = 0; i < n; i++)`).
        - The sorting logic compares `a[i] <= a[j]` but should be `a[i] > a[j]` to sort in ascending order.
2. **Which category of program inspection would you find more effective?**
    - **Most Effective Category:** Control-Flow Errors, as they help identify issues in loop execution and conditions, which are critical for the sorting algorithm to function correctly.
3. **Which type of error are you not able to identify using the program inspection?**
    - **Type of Error Not Identified:** Logical errors in sorting conditions that may not be evident until runtime.
4. **Is the program inspection technique worth applicable?**
    - **Applicability of Technique:** Yes, program inspection techniques are valuable as they help identify a variety of potential errors and improve code quality through systematic review.

(9) File Name : Stack Implementation.txt

A. Data Reference Errors

- **No unset/uninitialized variables:** All variables are initialized before use.
- **Array references:** Arrays are used (stack array).
- **Integer subscripts:** Not applicable.
- **Dangling references:** Not applicable.
- **Alias names:** Not applicable.
- **Variable value types:** Correct types used.
- **Addressing problems:** Not applicable.
- **Pointer/reference attributes:** Not applicable.
- **Data structure consistency:** Stack implementation is consistent.
- **Off-by-one errors in indexing:** Issues present in the push, pop, and display methods.
- **Inheritance requirements:** Not applicable.

B. Data-Declaration Errors

- **Explicitly declared variables:** All variables are declared properly.
- **Default attributes understood:** Not applicable.
- **Proper initialization:** All variables are initialized correctly.
- **Correct length and data type:** All variables have appropriate types.
- **Memory type initialization:** Not applicable.
- **Similar variable names:** No confusing names found.

C. Computation Errors

- **Inconsistent data types:** No inconsistencies found.
- **Mixed-mode computations:** Not applicable.
- **Different lengths of variables:** Not applicable.
- **Data type of target variable:** No issues.
- **Overflow/underflow expressions:** Not applicable (int type is sufficient for input).
- **Divisor being zero:** Checked properly in pop method.
- **Base-2 representation issues:** Not applicable.
- **Value outside meaningful range:** Not applicable.
- **Order of evaluation/precedence:** No issues.
- **Invalid integer arithmetic:** Not applicable.

D. Comparison Errors

- **Comparisons of different data types:** No issues found.
- **Mixed-mode comparisons:** Not applicable.
- **Comparison operators:** Correctly used.
- **Boolean expressions:** Not applicable.
- **Boolean operator operands:** Not applicable.
- **Floating-point comparisons:** Not applicable.
- **Order of evaluation with Boolean operators:** Not applicable.
- **Compiler evaluation affecting the program:** Not applicable.

E. Control-Flow Errors

- **Multiway branch:** Not applicable.
- **Loop termination:** The display method has a potential logical error.
- **Module/subroutine termination:** All modules will eventually terminate.
- **Loop execution:** Oversight found in the loop conditions.
- **Loop fall-through consequences:** Not applicable.
- **Off-by-one errors:** Present in display method.
- **Mismatched brackets:** No issues found.
- **Non-exhaustive decisions:** Not applicable.

F. Interface Errors

- **Parameter and argument count match:** Correct.
- **Parameter attributes match arguments:** Correct.
- **Units system match:** Not applicable.
- **Arguments transmitted to another module:** Not applicable.
- **Attributes of transmitted arguments match:** Not applicable.
- **Units system match for transmitted arguments:** Not applicable.
- **Built-in function arguments:** Not applicable.
- **Subroutine alters input parameters:** Not applicable.
- **Global variable definitions:** Not applicable.

G. Input/Output Errors

- **File attributes:** No files declared.
- **OPEN statement attributes:** Not applicable.
- **Memory for file read:** Not applicable.
- **Files opened before use:** Not applicable.
- **Files closed after use:** Not applicable.
- **End-of-file conditions:** Not applicable.
- **I/O error conditions:** Handled appropriately in push and pop.
- **Spelling/grammatical errors:** No errors found in output.

H. Other Checks

- **Cross-reference listing of identifiers:** Not applicable.
- **Attribute listing check:** Not applicable.
- **Compiler warning messages:** Not applicable.
- **Program robustness:** Validity checks could be added for stack operations.

Answers to Questions

1. **How many errors are there in the program? Mention the errors you have identified.**
    - **Errors Identified:** There are multiple logical errors in the push, pop, and display methods, specifically in how the top index is manipulated and in the display loop condition.
2. **Which category of program inspection would you find more effective?**
    - **Most Effective Category:** Control-Flow Errors, as they help identify logical errors in the stack operations, which are critical for correct functionality.
3. **Which type of error are you not able to identify using the program inspection?**
    - **Type of Error Not Identified:** Performance inefficiencies, such as the potential issues with stack size and memory management in larger applications.
4. **Is the program inspection technique worth applicable?**
    - **Applicability of Technique:** Yes, program inspection techniques are valuable as they help identify a variety of potential errors and improve code quality through systematic review.

(10) File Name : Tower of Hanoi.txt

A. Data Reference Errors

- **No unset/uninitialized variables:** All variables are initialized before use.
- **Array references:** Not applicable (no arrays used).
- **Integer subscripts:** Not applicable.
- **Dangling references:** Not applicable.
- **Alias names:** Not applicable.
- **Variable value types:** Correct types used.
- **Addressing problems:** Not applicable.
- **Pointer/reference attributes:** Not applicable.
- **Data structure consistency:** Not applicable.
- **Off-by-one errors in indexing:** Not applicable.

- **Inheritance requirements:** Not applicable.

B. Data-Declaration Errors

- **Explicitly declared variables:** All variables are declared properly.
- **Default attributes understood:** Not applicable.
- **Proper initialization:** All variables are initialized correctly.
- **Correct length and data type:** All variables have appropriate types.
- **Memory type initialization:** Not applicable.
- **Similar variable names:** No confusing names found.

C. Computation Errors

- **Inconsistent data types:** No inconsistencies found.
- **Mixed-mode computations:** Not applicable.
- **Different lengths of variables:** Not applicable.
- **Data type of target variable:** No issues.
- **Overflow/underflow expressions:** Not applicable (int type is sufficient for input).
- **Divisor being zero:** Not applicable.
- **Base-2 representation issues:** Not applicable.
- **Value outside meaningful range:** Not applicable.
- **Order of evaluation/precedence:** No issues.
- **Invalid integer arithmetic:** Issues in the recursive calls.

D. Comparison Errors

- **Comparisons of different data types:** No issues found.
- **Mixed-mode comparisons:** Not applicable.
- **Comparison operators:** Correctly used.
- **Boolean expressions:** Not applicable.
- **Boolean operator operands:** Not applicable.
- **Floating-point comparisons:** Not applicable.
- **Order of evaluation with Boolean operators:** Not applicable.
- **Compiler evaluation affecting the program:** Not applicable.

E. Control-Flow Errors

- **Multiway branch:** Not applicable.
- **Loop termination:** Not applicable.
- **Module/subroutine termination:** All modules will eventually terminate.
- **Loop execution:** Not applicable.

- **Loop fall-through consequences:** Not applicable.
- **Off-by-one errors:** Not applicable.
- **Mismatched brackets:** No issues found.
- **Non-exhaustive decisions:** The base case for recursion is defined.

F. Interface Errors

- **Parameter and argument count match:** Correct.
- **Parameter attributes match arguments:** Correct.
- **Units system match:** Not applicable.
- **Arguments transmitted to another module:** Not applicable.
- **Attributes of transmitted arguments match:** Not applicable.
- **Units system match for transmitted arguments:** Not applicable.
- **Built-in function arguments:** Not applicable.
- **Subroutine alters input parameters:** Not applicable.
- **Global variable definitions:** Not applicable.

G. Input/Output Errors

- **File attributes:** No files declared.
- **OPEN statement attributes:** Not applicable.
- **Memory for file read:** Not applicable.
- **Files opened before use:** Not applicable.
- **Files closed after use:** Not applicable.
- **End-of-file conditions:** Not applicable.
- **I/O error conditions:** Handled appropriately.
- **Spelling/grammatical errors:** No errors found in output.

H. Other Checks

- **Cross-reference listing of identifiers:** Not applicable.
- **Attribute listing check:** Not applicable.
- **Compiler warning messages:** Not applicable.
- **Program robustness:** Validity checks could be added for user input.

Answers to Questions

1. **How many errors are there in the program? Mention the errors you have identified.**
    - **Errors Identified:** There are logical errors in the recursive calls of the doTowers method where the parameters topN, inter, and to are incorrectly modified using ++ and -- operators.

2. **Which category of program inspection would you find more effective?**
    - **Most Effective Category:** Computation Errors, as they help identify logical errors in calculations and recursive calls that are crucial for correct program execution.
3. **Which type of error are you not able to identify using the program inspection?**
    - **Type of Error Not Identified:** Potential infinite recursion or stack overflow due to incorrect handling of recursive parameters.
4. **Is the program inspection technique worth applicable?**
    - **Applicability of Technique:** Yes, program inspection techniques are valuable as they help identify a variety of potential errors and improve code quality through systematic review.

II. CODE DEBUGGING

(1) File Name : Armstrong.txt

1. **How many errors are there in the program?**
    - There were two errors: the incorrect calculation of the remainder and incorrect updating of the number.
2. **How many breakpoints did you need to fix those errors?**
    - Two breakpoints were set: one at the start of the while loop and one before the if-else condition.

a. What are the steps you have taken to fix the error?

➢ The remainder calculation was fixed by using num % 10 instead of num / 10.

➢ The update of the number was fixed by using num / 10 instead of num % 10.

3. **Submit your complete executable code**:

```
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; // Corrected to get the last digit
         check = check + (int) Math.pow(remainder, 3); // Cube of the digit
         num = num / 10; // Corrected to 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) File Name : GCD and LCM.txt

1. **How many errors are there in the program? Mention the errors you have identified.**
   - In the gcd() function, the original while condition (while (a % b == 0)) should be replaced with while (b != 0).
   - In the lcm() function, the condition if (a % x != 0 && a % y != 0) was incorrect; the LCM calculation should be based on the formula LCM(x, y) = (x * y) / GCD(x, y).
2. **How many breakpoints did you need to fix those errors?**
   - **Breakpoint 1:** Set at the while(a % b == 0) line in the gcd() function. This helped identify the logical error in the while condition.
   - **Breakpoint 2:** Set at the if(a % x != 0 && a % y != 0) line in the lcm() function to verify the incorrect logic for calculating the LCM.

   a. **What are the steps you have taken to fix the error you identified in the code fragment?**

   ➢ **Fix 1:** Corrected the gcd() function's while condition to while(b != 0) since the GCD algorithm iterates until the remainder is zero.

➢ **Fix 2:** Modified the lcm() function to use the correct formula LCM(x, y) = (x * y) / GCD(x, y) instead of incrementing a in a loop.

3. **Submit your complete executable code**:

```
// 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) ? x : y; // a is greater number
      b = (x < y) ? x : y; // b is smaller number
      while (b != 0) { // Error fixed: a % b != 0 changed to b != 0
         r = a % b;
         a = b;
         b = r;
      }
      return a;
   }
   static int lcm(int x, int y) {
      return (x * y) / gcd(x, y); // Corrected LCM formula
   }
   public static void main(String args[]) {
      Scanner input = new Scanner(System.in);
      System.out.println("Enter the two numbers: ");
      int x = input.nextInt();
      int y = input.nextInt();
      System.out.println("The GCD of two numbers is: " + gcd(x, y));
      System.out.println("The LCM of two numbers is: " + lcm(x, y));
      input.close();
   }
}
```

(3) File Name : Knapsack.txt

1. **How many errors are there in the program? Mention the errors you have identified.**
   - **Error 1:** In the opt[n++][w] statement inside the loop, n++ should be replaced with n - 1 to properly reference the correct item in the opt array.
   - **Error 2:** In the if(weight[n] > w) condition, it should be if(weight[n] <= w) to allow the item to be taken if its weight is within the limit.
   - **Error 3:** The calculation for option2 used profit[n-2], which was incorrect. It should use profit[n].
   - **Error 4:** In the option2 calculation, the array access for opt[n-1][w-weight[n]] was incorrect due to the faulty weight comparison logic.
2. **How many breakpoints did you need to fix those errors?**
   - **Breakpoint 1:** Set at int option1 = opt[n++][w]; line to check for the error in incrementing n inside the loop.

- **Breakpoint 2:** Set at the if(weight[n] > w) condition to validate the weight comparison logic and catch the incorrect condition.
- **Breakpoint 3:** Set at the option2 = profit[n-2] + opt[n-1][w-weight[n]]; line to ensure proper indexing of profit and opt arrays.

a. What are the steps you have taken to fix the error you identified in the code fragment?

➢ **Fix 1:** Replaced opt[n++][w] with opt[n-1][w] to properly reference the previous item.

➢ **Fix 2:** Changed if(weight[n] > w) to if(weight[n] <= w) to correctly allow items with weight within the knapsack's capacity.

➢ **Fix 3:** Corrected profit[n-2] to profit[n] so that the correct item's profit is used.

➢ **Fix 4:** Verified and corrected the index calculation for opt[n - 1][w - weight[n]] inside the LCM condition.

3. Submit your complete executable code:

```
// Knapsack Program
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 - 1][w];  // Corrected n++ to n-1
            // take item n
            int option2 = Integer.MIN_VALUE;
            if (weight[n] <= w) {  // Corrected comparison to `weight[n] <= w`
               option2 = profit[n] + opt[n - 1][w - weight[n]]; // Corrected profit and opt index
            }
            // select better of two options
            opt[n][w] = Math.max(option1, option2);
            sol[n][w] = (option2 > option1);
         }
      }
      // determine which items to take
      boolean[] take = new boolean[N + 1];
      for (int n = N, w = W; n > 0; n--) {
```

```
            if (sol[n][w]) {
                take[n] = true;
                w = w - weight[n];
            } else {
                take[n] = false;
            }
        }
        // print results
        System.out.println("Item" + "\t" + "Profit" + "\t" + "Weight" + "\t" + "Take");
        for (int n = 1; n <= N; n++) {
            System.out.println(n + "\t" + profit[n] + "\t" + weight[n] + "\t" + take[n]);
        }
    }
}
```

(4) File Name : Magic Number.txt

1. **How many errors are there in the program? Mention the errors you have identified.**
    - **Error 1:** Inside the inner `while(sum == 0)` loop, the condition should have been `while(sum != 0)`, as we need to loop while the sum is greater than zero to break the number into digits.
    - **Error 2:** In the line `s = s * (sum / 10);`, the operation should have been summing the digits of the number, not multiplying. It should have been changed to `s = s + (sum % 10);`.
    - **Error 3:** There was a missing semicolon (;) after `sum = sum % 10`.
2. **How many breakpoints did you need to fix those errors?**
    - **Breakpoint 1:** Set at `while(sum == 0)` to check the logic of the condition and identify the error in checking whether `sum` is zero instead of looping until `sum` becomes zero.
    - **Breakpoint 2:** Set at `s = s * (sum / 10);` to observe the incorrect multiplication and replace it with summing the digits.
    - **Breakpoint 3:** Set at `sum = sum % 10` to check for missing semicolon and step through the code to verify if the division and sum operations were being handled properly.

    a. **What are the steps you have taken to fix the error you identified in the code fragment?**

    ➢ **Fix 1:** Changed the `while(sum == 0)` condition to `while(sum != 0)` to properly sum the digits of the number.

    ➢ **Fix 2:** Replaced `s = s * (sum / 10);` with `s = s + (sum % 10);` to correctly sum the digits of the number instead of multiplying them.

    ➢ **Fix 3:** Added the missing semicolon after `sum = sum / 10` to ensure proper execution.

3. **Submit your 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;
        // Loop until num is reduced to a single digit
        while (num > 9) {
            sum = num;
            int s = 0;

            // Sum the digits of the current number
            while (sum != 0) {  // Error: Changed from sum == 0 to sum != 0
                s = s + (sum % 10);  // Error: Changed s = s * (sum / 10) to s = s + (sum % 10)
                sum = sum / 10;  // Get the next digit by dividing sum by 10
            }
            num = s;  // Assign the sum of digits back to num
        }
        // Check if the final value of num is 1 (magic number condition)
        if (num == 1) {
            System.out.println(n + " is a Magic Number.");
        } else {
            System.out.println(n + " is not a Magic Number.");
        }
        ob.close();
    }
}
```

(5) File Name : Merge Sort.txt

Step 6: Debugging Questions and Answers

1. **How many errors are there in the program? Mention the errors you have identified.**
    - **Error 1:** In the `mergeSort` method, the statement `int[] left = leftHalf(array+1);` was incorrect. We cannot add an integer to an array. It should be `int[] left = leftHalf(array);`.
    - **Error 2:** Similarly, the statement `int[] right = rightHalf(array-1);` was incorrect and should be `int[] right = rightHalf(array);`.
    - **Error 3:** In the `merge` method call, there were incorrect increment and decrement operators (`left++` and `right--`), which would lead to errors. These have been removed in the corrected version.
2. **How many breakpoints did you need to fix those errors?**

- **Breakpoint 1:** Set on int[] left = leftHalf(array+1); to check the error in attempting to add an integer to an array.
- **Breakpoint 2:** Set on merge(array, left++, right--); to identify the incorrect increment/decrement operators.

a. What are the steps you have taken to fix the error you identified in the code fragment?

➢ **Fix 1:** Corrected leftHalf(array+1) to leftHalf(array) to pass the entire array as a parameter.

➢ **Fix 2:** Corrected rightHalf(array-1) to rightHalf(array) for the same reason.

➢ **Fix 3:** Removed the unnecessary increment (left++) and decrement (right--) operators in the merge call as they were causing incorrect behavior.

3. Submit your complete executable code:

```
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.
   public static void mergeSort(int[] array) {
      if (array.length > 1) {
         // split array into two halves
         int[] left = leftHalf(array); // Error: Corrected from array+1 to array
         int[] right = rightHalf(array); // Error: Corrected from array-1 to array

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

         // merge the sorted halves into a sorted whole
         merge(array, left, right); // Error: Removed incorrect increment/decrement operators
      }
   }

   // 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.
    public static void merge(int[] result, int[] left, int[] right) {
        int i1 = 0;   // index into left array
        int i2 = 0;   // index into right array

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

(6) File Name : Multiply Matrices.txt

1. **How many errors are there in the program? Mention the errors you have identified.**
    - **Error 1:** In the innermost loop, first[c-1][c-k] and second[k-1][k-d] were incorrect. The correct indices are first[c][k] and second[k][d].
    - **Error 2:** The loop control variable k should iterate from 0 to n, not p, because the number of columns in the first matrix should match the number of rows in the second matrix.
2. **How many breakpoints did you need to fix those errors?**
    - **Breakpoint 1:** Set at the innermost loop (sum = sum + first[c-1][c-k]*second[k-1][k-d];) to check the incorrect indexing.
    - **Breakpoint 2:** Set in the loop where k is used, as the loop condition incorrectly uses p instead of n.

    a. What are the steps you have taken to fix the error you identified in the code fragment?

    ➢ **Fix 1:** Changed first[c-1][c-k] to first[c][k] and second[k-1][k-d] to second[k][d] for correct matrix access.

- **Fix 2:** Corrected the loop control variable from `k = 0; k < p` to `k = 0; k < n` so that the matrix multiplication works correctly.

3. Submit your complete executable 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 < n; k++) { // Corrected p to n
                  sum = sum + first[c][k] * second[k][d]; // Corrected index access
               }
               multiply[c][d] = sum;
               sum = 0;
            }
         }
         System.out.println("Product of entered matrices:");
         for (c = 0; c < m; c++) {
            for (d = 0; d < q; d++)
               System.out.print(multiply[c][d] + "\t");
            System.out.print("\n");
         }
      }
   }
}
```

(7) File Name : Quadratic Probing.txt

1. **How many errors are there in the program?**

- There was **one main error** in the program: an incorrect calculation in the insert function where i += (i + h / h--) % maxSize; should be corrected to i = (i + h * h) % maxSize;. This error affected the quadratic probing process.

2. **How many breakpoints did you need to fix those errors?**
   - **One breakpoint** was sufficient. The breakpoint was set at the point where quadratic probing happens (inside the do-while loop in the insert function). This allowed us to check the update of the index i.

   a. **What steps did you take to fix the error?**

   ➢ The step taken was to correct the faulty index update logic in the insert method. The term i += (i + h / h--) % maxSize; was replaced with i = (i + h * h) % maxSize;. This ensures proper quadratic probing where h is incremented after each iteration, and i is updated correctly based on the quadratic step (h * h).

3. Submit your complete executable code:

```
/**
 * Java Program to implement Quadratic Probing Hash Table
 */
import java.util.Scanner;
/** Class QuadraticProbingHashTable **/
class QuadraticProbingHashTable {
    private int currentSize, maxSize;
    private String[] keys;
    private String[] vals;
    /** Constructor **/
    public QuadraticProbingHashTable(int capacity) {
        currentSize = 0;
        maxSize = capacity;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }
    /** Function to clear hash table **/
    public void makeEmpty() {
        currentSize = 0;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }
    /** Function to get size of hash table **/
    public int getSize() {
        return currentSize;
    }
    /** Function to check if hash table is full **/
    public boolean isFull() {
        return currentSize == maxSize;
    }
    /** Function to check if hash table is empty **/
    public boolean isEmpty() {
        return getSize() == 0;
```

```
}
/** Function to check if hash table contains a key **/
public boolean contains(String key) {
    return get(key) != null;
}
/** Function to get hash code of a given key **/
private int hash(String key) {
    return key.hashCode() % maxSize;
}
/** Function to insert key-value pair **/
public void insert(String key, String val) {
    int tmp = hash(key);
    int i = tmp, h = 1;
    do {
        if (keys[i] == null) {
            keys[i] = key;
            vals[i] = val;
            currentSize++;
            return;
        }
        if (keys[i].equals(key)) {
            vals[i] = val;
            return;
        }
        i = (i + h * h) % maxSize; // Corrected line
        h++;
    } while (i != tmp);
}
/** Function to get value for a given key **/
public String get(String key) {
    int i = hash(key), h = 1;
    while (keys[i] != null) {
        if (keys[i].equals(key))
            return vals[i];
        i = (i + h * h) % maxSize;
        h++;
    }
    return null;
}
/** Function to remove key and its value **/
public void remove(String key) {
    if (!contains(key))
        return;
    /** find position of key and delete **/
    int i = hash(key), h = 1;
    while (!key.equals(keys[i]))
        i = (i + h * h) % maxSize;
    keys[i] = vals[i] = null;
    /** rehash all keys **/
    for (i = (i + h * h++) % maxSize; keys[i] != null; i = (i + h * h++) % maxSize) {
        String tmp1 = keys[i], tmp2 = vals[i];
        keys[i] = vals[i] = null;
        currentSize--;
        insert(tmp1, tmp2);
    }
```

```
        currentSize--;
    }
    /** Function to print HashTable **/
    public void printHashTable() {
        System.out.println("\nHash Table:");
        for (int i = 0; i < maxSize; i++)
            if (keys[i] != null)
                System.out.println(keys[i] + " " + vals[i]);
        System.out.println();
    }
}
/** Class QuadraticProbingHashTableTest **/
public class QuadraticProbingHashTableTest {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Hash Table Test\n\n");
        System.out.println("Enter size");

        /** Create object of QuadraticProbingHashTable **/
        QuadraticProbingHashTable qpht = new QuadraticProbingHashTable(scan.nextInt());
        char ch;
        /** Perform QuadraticProbingHashTable operations **/
        do {
            System.out.println("\nHash Table Operations\n");
            System.out.println("1. insert ");
            System.out.println("2. remove");
            System.out.println("3. get");
            System.out.println("4. clear");
            System.out.println("5. size");
            int choice = scan.nextInt();
            switch (choice) {
                case 1:
                    System.out.println("Enter key and value");
                    qpht.insert(scan.next(), scan.next());
                    break;
                case 2:
                    System.out.println("Enter key");
                    qpht.remove(scan.next());
                    break;
                case 3:
                    System.out.println("Enter key");
                    System.out.println("Value = " + qpht.get(scan.next()));
                    break;
                case 4:
                    qpht.makeEmpty();
                    System.out.println("Hash Table Cleared\n");
                    break;
                case 5:
                    System.out.println("Size = " + qpht.getSize());
                    break;
                default:
                    System.out.println("Wrong Entry\n");
                    break;
            }
            /** Display hash table **/
```

```
            qpht.printHashTable();
            System.out.println("\nDo you want to continue (Type y or n)\n");
            ch = scan.next().charAt(0);
        } while (ch == 'Y' || ch == 'y');
    }
}
```

(8) File Name : Quadratic Probing.txt

1. **How many errors are there in the program?**
    - There were **two main errors** in the program:
    - Incorrect loop condition in the outer for loop (for (int i = 0; i >= n; i++)), which should be for (int i = 0; i < n - 1; i++).
    - The incorrect comparison operator in the sorting logic (if (a[i] <= a[j])), which should be if (a[i] > a[j]) to sort in ascending order.
2. **How many breakpoints did you need to fix those errors?**
    - **Two breakpoints** were set:
    - At the beginning of the outer for loop to check its iteration.
    - Inside the inner for loop at the if condition to verify the comparison logic for sorting.

    a. What steps did you take to fix the errors?

    ➢ The outer loop condition was corrected from i >= n to i < n - 1 to ensure it iterates correctly.

    ➢ The comparison inside the if statement was changed from a[i] <= a[j] to a[i] > a[j] to sort the array in ascending order.

3. Submit your complete executable code:

```
import java.util.Scanner;
public class Ascending_Order {
    public static void main(String[] args) {
        int n, temp;
        Scanner s = new Scanner(System.in);

        // Input number of elements
        System.out.print("Enter number of elements you want in array: ");
        n = s.nextInt();
        int[] a = new int[n];

        // Input elements of the array
        System.out.println("Enter all the elements:");
        for (int i = 0; i < n; i++) {
            a[i] = s.nextInt();
        }
        // Sorting logic corrected
        for (int i = 0; i < n - 1; i++) { // Corrected loop condition
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) { // Corrected comparison for ascending order
                    temp = a[i];
```

```
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    // Display sorted array
    System.out.print("Ascending Order: ");
    for (int i = 0; i < n - 1; i++) {
        System.out.print(a[i] + ", ");
    }
    System.out.print(a[n - 1]);
  }
}
```

(9) File Name : Quadratic Probing.txt

1. **How many errors are there in the program?**
    - Incorrect logic in the push() method where top-- should be top++.
    - Incorrect handling of the pop() method, where top needed to be decremented instead of incremented.
    - The loop condition in the display() method needed to be changed to i <= top to print all elements correctly.
2. **How many breakpoints did you need to fix those errors?**
    - At the push() method to observe how top was being updated.
    - In the pop() method to verify that top was correctly removing elements.
    - In the display() method to ensure all elements were being printed correctly.

    a. **What steps did you take to fix the errors?**

    ➢ Corrected the top update in the push() method to increment it before inserting.

    ➢ Adjusted the top decrement in the pop() method to remove the correct element.

    ➢ Modified the display loop to print all elements up to top.

3. **Submit your complete executable code**:

```
import java.util.Arrays;
class StackMethods {
    private int top;
    int size;
    int[] stack;
    public StackMethods(int arraySize) {
        size = arraySize;
        stack = new int[size];
        top = -1;
    }
    // Corrected push method
    public void push(int value) {
```

```
        if (top == size - 1) {
            System.out.println("Stack is full, can't push a value");
        } else {
            top++;  // Increment top before inserting
            stack[top] = value;
        }
    }
    // Corrected pop method
    public void pop() {
        if (!isEmpty()) {
            top--;  // Decrement top to "remove" the top element
        } else {
            System.out.println("Can't pop...stack is empty");
        }
    }
    public boolean isEmpty() {
        return top == -1;
    }
    // Corrected display method
    public void display() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
        } else {
            for (int i = 0; i <= top; i++) {
                System.out.print(stack[i] + " ");
            }
            System.out.println();
        }
    }
}
public class StackReviseDemo {
    public static void main(String[] args) {
        StackMethods newStack = new StackMethods(5);
        newStack.push(10);
        newStack.push(1);
        newStack.push(50);
        newStack.push(20);
        newStack.push(90);
        newStack.display();
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.display();
    }
}
```

(10) File Name : Tower Of Hanoi.txt

1. **How many errors are there in the program?**
    - There was **one main error**: the incorrect increment (topN++) and decrement (inter--, from+1, to+1) in the recursive call.
2. **How many breakpoints did you need to fix those errors?**

- **One breakpoint** was enough, set at the recursive call to verify the correct movement of disks.

a. What steps did you take to fix the errors?

➢ The unnecessary increment and decrement operations were removed, and the recursive logic was corrected to properly pass the parameters without altering them.

3. Submit your complete executable code:

```
public class MainClass {
    public static void main(String[] args) {
        int nDisks = 3;  // Number of disks
        doTowers(nDisks, 'A', 'B', 'C');  // Call the method to solve Tower of Hanoi
    }
    public static void doTowers(int topN, char from, char inter, char to) {
        if (topN == 1) {
            // Base case: move the top disk directly
            System.out.println("Disk 1 from " + from + " to " + to);
        } else {
            // Recursive case
            doTowers(topN - 1, from, to, inter);  // Move topN-1 disks from "from" to "inter"
            System.out.println("Disk " + topN + " from " + from + " to " + to);  // Move the nth disk from
"from" to "to"
            doTowers(topN - 1, inter, from, to);  // Move topN-1 disks from "inter" to "to"
        }
    }
}
```