IT314: Software Engineering

Lab 7

Program Inspection, Debugging and Static Analysis

Name: Dev Davda

ID: 202201242

Category A: Data Reference Errors

1. **Armstrong Number**:
   - **Potential Error**: remainder = num / 10; should be remainder = num % 10; because the remainder is obtained using the modulus operator.
   - **Array Indexing**: There is no array usage, so no index out-of-bounds errors are possible.
   - **Type mismatch**: The program uses integer division where floating-point division might be more appropriate for accurate calculations.

2. **GCD and LCM**:
   - **Potential Error**: In the gcd function, the line while(a % b == 0) is incorrect and should be while(a % b != 0) to continue looping while the remainder is not zero.
   - **Array Indexing**: No array usage, so no out-of-bounds concerns.

3. **Knapsack**:
   - **Array Reference**: The array indexing opt[n++][w] and opt[n-1][w-weight[n]] can cause an out-of-bounds error. To avoid this, bounds checking must be added to ensure the n and w values are within the limits of the arrays.
   - **Data Initialization**: Arrays opt and sol are initialized properly, so no unset values are used.
   - **Type safety**: When generating random values for profit and weight, ensure they're appropriate for their intended use.

4. **Magic Number**:
   - **Reference Error**: In the inner loop while(sum == 0), the condition seems illogical since sum is initially set to num. This condition should be while(sum != 0) to proceed with the calculation.
   - **Array Use**: No arrays are used here, so no indexing issues.
   - **Type mismatch**: The program uses integer division where floating-point division might be more accurate for certain calculations.

5. **Merge Sort**:
   - **Reference Error**: In the leftHalf and rightHalf methods, array+1 and array-1 are used incorrectly, causing potential reference errors. Instead, appropriate index-based array splitting should be used.
   - **Array Indexing**: Array indexing must be carefully checked when accessing left and right arrays to avoid out-of-bounds errors during merging.

6. **Matrix Multiplication**:
   - **Array Boundaries**: When multiplying matrices, accessing first[c-1][c-k] and second[k-1][k-d] can cause array out-of-bounds errors, especially when looping through matrix dimensions. Ensure indices stay within matrix bounds.
   - **Uninitialized Data**: Arrays are properly initialized before use.

7. **Quadratic Probing**:
   - **Array Indexing**: The operation i += (i + h / h--) % maxSize; could cause an out-of-bounds error if i exceeds the array size maxSize. Proper checks need to ensure that i stays within bounds.

- **Uninitialized Data**: Arrays keys and vals are initialized properly.
- **Dangling references**: Although Java manages memory, the hash table implementation should ensure that deleted items are properly handled to avoid referencing non-existent data.

8. **Sorting Array**:
   - **Array Indexing**: The loops in the array sorting method have incorrect bounds (for (int i = 0; i >= n; i++);). This loop will never run and can cause index errors. It should be for (int i = 0; i < n; i++).
   - **Array Initialization**: The array a is correctly initialized by user input.

9. **Stack Implementation**:
   - **Array Indexing**: The loop for(int i=0; i>top; i++) in the display method is incorrect. The condition should be i<top to avoid referencing out-of-bound indices.
   - **Uninitialized Data**: The stack array is initialized properly before use.

10. **Tower of Hanoi**:
    - **Reference Error**: In doTowers(topN++, inter--, from+1, to+1), improper increment and decrement operations lead to reference errors. Recursive function calls should properly manage topN, from, to, and inter.
    - **No Arrays**: Since no arrays are used, no indexing issues exist here.

Category B: Data-Declaration Errors

1. **Armstrong Number:**
   - The variable 'num' is used for calculations but isn't updated correctly in the while loop.
   - The 'remainder' calculation is wrong - it should be num % 10, not num / 10.
   - The input is taken as a command-line argument, which might be confusing for beginners.

2. **GCD and LCM**:
   - There's a comment about an error in the GCD function, but it's not fixed.
   - The LCM function might run forever if it can't find a common multiple.
   - Variable names like 'a' and 'b' are not very descriptive.

3. **Knapsack:**
   - There are a few off-by-one errors in array indexing.
   - Some variable names are confusing (like 'n' and 'w').
   - The random number generation might create items too heavy for the knapsack.

4. **Magic Number:**
   - The while loop condition is wrong (should be while(sum>0)).
   - The calculation inside the inner while loop is incorrect.
   - Variable names are not very clear (like 'ob' for the Scanner).

5. **Merge Sort:**
   - In the mergeSort function, there are errors in the leftHalf and rightHalf calls (array+1 and array-1 are wrong).
   - The merge function doesn't handle the case where one array is emptied before the other.

6. **Matrix Multiplication:**
   - The matrix multiplication logic has errors in indexing.
   - There's no check to ensure the inner dimensions of the matrices match for multiplication.
   - The sum variable isn't reset to 0 after each element calculation.

7. **Quadratic Probing Hash Table**:
   - The insert method has an error in its quadratic probing formula.
   - The remove method might cause an infinite loop if the key isn't found.
   - Some variable names are unclear (like 'tmp1' and 'tmp2').

8. **Sorting Array:**
   - The outer for loop condition is wrong (i >= n should be i < n).
   - The inner loop compares elements in the wrong order for ascending sort.
   - The final print loop doesn't handle the last element correctly.

9. **Stack Implementation**:
   - The push method decrements top instead of incrementing it.
   - The display method prints elements in reverse order.
   - There's no check for stack overflow in push.

10. **Tower of Hanoi**:
    - The recursive call at the end has several errors (topN++, inter--, from+1, to+1).
    - There's no base case to stop the recursion.
    - Variable names could be more descriptive (like 'from', 'to', 'inter').

Category C: Computation Errors

Code 1 : Armstrong

1. **Mixed-mode Computations**: Yes. Math.pow(remainder, 3) returns a double, which is then cast to int. Ensure understanding of this conversion.
2. **Overflow/Underflow**: Yes. The computation Math.pow(remainder, 3) can exceed the limit of an int if remainder is too large (greater than 12).
3. **Order of Evaluation**: Yes. The expression check + (int)Math.pow(remainder, 3) could lead to unexpected results due to operator precedence.

Code 2 : GCD & LCM

- **Mixed-mode Computations**: Yes. The comparison between x and y in the gcd and lcm methods does not present a mixed-mode computation issue, but ensure that data types are consistent when calculating results.
- **Overflow/Underflow**: Yes. The lcm method can potentially result in overflow if a grows large, especially for large values of x and y.
- **Division by Zero**: Yes. If either x or y is zero, it can lead to division by zero in the gcd method.
- **Order of Evaluation**: Yes. In the lcm method, the order of operations when evaluating if(a % x != 0 && a % y != 0) could lead to an infinite loop if the conditions are not properly checked.

Code 3 : Knapsack

- **Mixed-mode Computations**: Yes. The increment n++ in int option1 = opt[n++][w]; affects the loop's control variable, which can lead to unexpected behavior.
- **Overflow/Underflow**: Yes. The calculation for option2 using profit[n-2] may lead to an ArrayIndexOutOfBoundsException if n is less than 2.
- **Order of Evaluation**: Yes. The expression Math.max(option1, option2) should be carefully examined to ensure the correct values are being compared, especially since option2 is initialized to Integer.MIN_VALUE.

Code 4 : magic number

- **Mixed-mode Computations**: Yes. The variable sum is incorrectly initialized with the value of num, which leads to improper calculations.
- **Overflow/Underflow**: Yes. The operation s = s * (sum / 10); could lead to issues with integer multiplication if not handled properly.
- **Division by Zero**: Yes. In the inner while loop, sum == 0 is used, which will never execute. It should be sum != 0 instead, but this also may lead to an infinite loop if sum is not being decremented properly.
- **Order of Evaluation**: Yes. The statement s = s * (sum / 10); may not yield the expected results due to incorrect logic.

Code 5 : merge sort

- **Mixed-mode Computations**: Yes. The expressions leftHalf(array + 1) and rightHalf(array - 1) are incorrect as array is an array reference, and arithmetic operations cannot be performed on it directly.
- **Overflow/Underflow**: Yes. The merge sort algorithm's recursive calls may lead to a stack overflow for very large arrays due to deep recursion if not handled correctly.
- **Division by Zero**: Yes. The code does not handle edge cases, such as an empty array, which may lead to unexpected behavior or exceptions.
- **Order of Evaluation**: Yes. The merge(array, left++, right--); usage is incorrect. The increment/decrement operations on left and right should not occur here, as they modify the array references instead of passing the arrays as intended.

Code :6 multiply matrics

- **Variables going outside their meaningful range**: The expressions first[c-1][c-k] and second[k-1][k-d] may lead to array index out of bounds errors.
- **Invalid integer arithmetic operations**: The indexing in first[c-1][c-k] and second[k-1][k-d] is incorrect and should be revised to prevent out-of-bounds access.

Code : 7 Quardatic Probing

- **Inconsistent or non-arithmetic data types**: The hash function does not handle negative hash codes, which could lead to inconsistent behavior when the key is null or has a negative hash code.
- **Possibilities for overflow/underflow**: The calculation i += (i + h / h--) % maxSize; can lead to unexpected results due to the order of operations and potential division by zero if h starts at zero.
- **Variables going outside their meaningful range**: The indexing used in the insert and get functions can lead to out-of-bounds access if the hash code is negative or if maxSize is improperly managed.
- **Invalid integer arithmetic operations**: The use of h / h-- is incorrect and can cause division by zero, and h++ in i = (i + h * h++) % maxSize; can lead to unintended increments, affecting the probing mechanism.

Code :8 Sorting array

- **Variables going outside their meaningful range**: The loop condition for (int i = 0; i >= n; i++); is incorrect. It should be for (int i = 0; i < n; i++). This mistake can lead to an infinite loop or not executing the sorting logic at all.
- **Incorrect assumptions about operator precedence**: The semicolon at the end of the loop for (int i = 0; i >= n; i++); effectively creates an empty loop. This may lead to unexpected behavior in the sorting process, as the intended nested loop will not execute correctly.

Code 9 : Stack Implementation

- **Mismatched variable lengths within computations**: The push method incorrectly decrements top before assigning the value, which can lead to an ArrayIndexOutOfBoundsException when trying to push elements.
- **Data type mismatches between target variables and right-hand expressions**: No data type mismatches are present in this code.
- **Potential division by zero**: There are no division operations in the provided code, so this does not apply.
- **Variables going outside their meaningful range**: The condition in the display method is incorrect: for (int i = 0; i > top; i++). It should be i <= top to properly iterate over the stack elements.

Code 10 : Tower of Hanoi

- **Inconsistent or non-arithmetic data types**: The parameters from, to, and inter are of type char, but the doTowers call in the else block uses expressions like from + 1, which attempts to perform arithmetic on char types.
- **Mismatched variable lengths within computations**: In the recursive call doTowers(topN ++, inter--, from+1, to+1), topN is incorrectly modified. topN++ should be topN - 1 to avoid an infinite recursion.
- **Data type mismatches between target variables and right-hand expressions**: The variables inter-- and from + 1 are not appropriate as they mix types and may lead to unexpected behavior.

Category D : Comparison Errors

code 1: Armstrong

- **Mixed-mode comparisons**: The program attempts to compare the check variable (of type int) with n (also int), but the logic is reversed in how check is calculated.
- **Incorrect use of operators**: The calculation of remainder is done incorrectly. The remainder should be obtained by using num % 10 instead of num / 10, leading to incorrect logic in verifying the Armstrong number.

Code 2: GCD and LCM

- **Incorrect use of operators**:In the gcd method, the condition in the while loop should be while(a % b != 0) instead of while(a % b == 0). This is a critical logical error that will result in an incorrect calculation of GCD.
- **Order of evaluation**: In the lcm method, the condition if(a % x != 0 && a % y != 0) should be if(a % x == 0 && a % y == 0) to correctly identify when a is a multiple of both x and y. This represents a logic error, as the current implementation would never return a valid LCM.

Code 3: Knapsack

- **Incorrect use of operators**: The line int option1 = opt[n++][w]; modifies the value of n within the loop, leading to incorrect indexing for subsequent iterations. The use of n++ here is a mistake; it should be replaced with just n to avoid incrementing n prematurely.
- The condition in if (weight[n] > w) is valid, but when calculating option2, the expression profit[n-2] should likely be profit[n] to ensure the correct profit of the current item is being considered.
- **Order of evaluation**: The logic used to determine which items to take may not account correctly for cases where the weights and profits do not line up correctly due to previous errors, although the specific issue isn't directly in comparison

Code 4 : magic number

- **Incorrect use of operators**:In the inner while loop, the condition while(sum==0) is incorrect. It should be while(sum > 0) to correctly process the digits of the number.
- Inside the loop, the statement s=s*(sum/10); is also incorrect. The correct operation should be to extract the last digit and add it to s, such as s += sum % 10;. The current logic does not correctly accumulate the sum of the digits.
- **Order of evaluation**: The logic in the loops may lead to an incorrect evaluation of the magic number condition because of the previous errors in calculations.

code 5: merge sort

Incorrect use of operators:In the mergeSort method, the lines int[] left = leftHalf(array+1); and int[] right = rightHalf(array-1); are incorrect. The array should not be incremented or decremented like this; instead, you need to pass subarrays. You should replace them with:

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

- The merging operation merge(array, left++, right--); incorrectly attempts to modify the arrays during the merge. It should be merge(array, left, right); without incrementing or decrementing.
- **Order of evaluation**: The use of array.length in leftHalf and rightHalf is correct, but the method implementations themselves do not reflect accurate splitting of the input array due to the earlier mistakes. This may lead to incorrect results when sorting.

code 6 : multiply matrics

- **Incorrect use of operators**:In the multiplication logic, the lines first[c-1][c-k] and second[k-1][k-d] are incorrect. The indices should be:
  - first[c][k] (using k instead of c-k for the first matrix).
  - second[k][d] (using d instead of k-d for the second matrix).
- This incorrect indexing will lead to array index out of bounds errors and incorrect calculations.
- **Order of evaluation**: The logic in the nested loops for matrix multiplication is flawed due to the previous indexing issues, leading to potential incorrect results.

code 7: Quardatic Probing

- **Incorrect operator usage**: In the line i + = (i + h / h--) % maxSize;, the operator should be += instead of + =. It will cause a compilation error due to incorrect spacing.
- **Incorrect order of operations**: In both the insert and get methods, the line i = (i + h * h++) % maxSize; increments h after using it, which may lead to unintended behavior. Instead, it should be i = (i + h * h) % maxSize; h++; to correctly update the index based on the current value of h.
- **Potential Null Pointer Exception**: When using key.equals(keys[i]) in the remove method without checking if keys[i] is null first can lead to a NullPointerException. A check should be added to ensure that keys[i] is not null before the comparison.

Code 8: Sorting array

Class Name Formatting: The class name Ascending _Order contains a space, which is not allowed in Java. It should be changed to AscendingOrder.

- **Incorrect Loop Condition**: In the line for (int i = 0; i >= n; i++);, the loop condition should be i < n instead of i >= n. This will prevent the loop from executing, resulting in no sorting.
- **Extra Semicolon**: The semicolon at the end of the first for loop (for (int i = 0; i >= n; i++);) effectively creates an empty loop. Remove the semicolon to ensure that the inner loop is executed as part of the outer loop.
- **Sorting Logic Flaw**: The comparison if (a[i] <= a[j]) should be if (a[i] > a[j]). The current condition swaps elements in the wrong order, resulting in a non-sorted array.
- **Incorrect Printing of Array**: The final output loop uses for (int i = 0; i < n - 1; i++), which does not include the last element in the printed output. The last element is printed separately. It would be better to print all elements in a single loop without special handling for the last element.

Code 9: Stack Implementation

- **Push Method Logic**: The top index is decremented when pushing a new value instead of incrementing it. This will result in the new value being placed in an incorrect position, causing data loss or errors.
- **Pop Method Logic**: The top index is incremented when popping a value, which is incorrect. It should decrement the top to correctly remove the last value from the stack.
- **Display Method Loop Condition**: The loop in the display method uses a condition that prevents it from executing, as it checks if i > top. This should be changed to i <= top to ensure it correctly prints the elements in the stack.

Code 10 : Tower of Hanoi

Incorrect Recursive Call Arguments: In the second recursive call, doTowers(topN ++, inter--, from+1, to+1) is incorrect.

- topN++ and inter-- will cause unintended side effects. It should simply be topN - 1 for the recursive call to maintain the correct number of disks.
- from + 1 and to + 1 should be replaced with from and to, as the tower names (characters) should not be incremented.
- **Disk Count Condition**: The base case only handles when topN == 1. The recursive structure is set correctly to handle multiple disks but needs proper handling of the recursive parameters.
- **Output Logic**: The implementation doesn't correctly display the moves for all disks due to the incorrect recursive logic, which may lead to logical errors in the output.

Category E : Control-Flow Errors

Armstrong Number Code:

- **Issue:**
  Incorrect remainder calculation
  - In the while loop, you need to use the modulus (%) operator to find the remainder and the division (/) operator to reduce the number.

GCD and LCM Code:

- **Issue:**
  LCM logic
  - The LCM loop should return the number when both a % x == 0 and a % y == 0, not when they are not divisible.

  GCD logic:

  while(a % b != 0) should be replaced with != .

Knapsack Code:

- **Issue:**
  Invalid array indexing
  - Incrementing n++ and using profit[n-2] will cause index errors.

Magic Number Code:

- **Issues:**
  - while(sum == 0) should be while(sum > 0).
  - Multiplication in s = s*(sum/10) should be addition, and missing semicolons.

Merge Sort Code:

- **Issue:**
  Invalid array manipulation
  - Adding/subtracting integers in array references (leftHalf(array+1)) will cause errors.

Matrix Multiplication Code:

- **Issue:**
  Invalid array indexing
  - In the product calculation first[c-1][c-k] and second[k-1][k-d], indices go out of bounds.

Quadratic Probing Hash Table Code:

- **Issue:**

Invalid syntax in i += (i + h / h--) and other logical errors.

- Correct the insertion logic and rehashing logic.

Ascending Order Sorting

- Issue :
  - The outer loop condition (i >= n) should be i < n.
  - In the condition of the inner loop, it should swap if a[i] > a[j] (to sort in ascending order).
  - There's an extra semicolon after the first for loop which causes a logic error.

Stack Implementation

- Issue :-
  - The top-- should be top++ as we are pushing elements, so the top of the stack should increase.
  - The display method should loop from 0 to top (not with a > operator but <).

Tower of Hanoi

- Issue :-
  - There are syntax errors like topN++, which should be topN, and the inter--, from+1 expressions should be removed. You should just pass from, inter, and to directly.

Category F : Interface Errors

1. Armstrong Number Calculation

- **Number of Arguments Transmitted:**
  - The main method does not transmit any arguments to another method or module.
- **Attributes of Transmitted Arguments:**
  - No arguments are transmitted to other modules in this code.
- **Units System for Transmitted Arguments:**
  - There are no arguments transmitted to another module.
- **Global Variables:**
  - There are no global variables in the provided code.

GCD and LCM Code:

- **Global Variables:**
  - There are no global variables in the provided code.

Knapsack Code:

- **Number of Arguments Transmitted:**
  - The main method does not transmit any arguments to another method or module. All processing is done within the main method.
- **Attributes of Transmitted Arguments:**
  - No arguments are transmitted to other methods.
- **Units System for Transmitted Arguments:**
  - There are no arguments transmitted to another module.
- **Parameter Alteration:**
  - In the nested loops for filling the opt and sol arrays, n is modified, which can lead to unexpected behaviour because the loop continues incrementing n:
    i. **Original:** int option1 = opt[n++][w];
    ii. This modifies n during iteration, which is incorrect. It should be int option1 = opt[n][w]; (no increment).
  - **Correctness:** This is a critical error as it affects the logic of the algorithm.
- **Global Variables:**
  - There are no global variables in the provided code.

Magic Number Code:

- **Number of Arguments Transmitted:**
  - The main method does not transmit any arguments to another method or module.
- **Attributes of Transmitted Arguments:**
  - No arguments are transmitted to other modules in this code.
- **Units System for Transmitted Arguments:**
  - There are no arguments transmitted to another module.
- **Global Variables:**
  - There are no global variables in the provided code.

Merge Sort Code:

- **Number of Arguments Transmitted:**
  - The main method does not transmit any arguments to another method or module.
- **Attributes of Transmitted Arguments:**
  - No arguments are transmitted to other modules in this code.
- **Units System for Transmitted Arguments:**
  - There are no arguments transmitted to another module.
- **Global Variables:**
  - There are no global variables in the provided code.

6. Multiply Matrices

- **Number of Parameters and Arguments:**
  - The program does not have functions that require parameter checks as it works directly in main.
- **Number of Arguments Transmitted:**
  - The main method does not transmit any arguments to another method or module.
- **Attributes of Transmitted Arguments:**
  - No arguments are transmitted to other modules in this code.
- **Units System for Transmitted Arguments:**
  - There are no arguments transmitted to another module.
- **Global Variables:**
  - There are no global variables in the provided code.

7. Quadratic Probing

- **Number of Parameters and Arguments:**
  - The program does not have methods that require multiple parameters, so this point is not applicable.
- **Number of Arguments Transmitted:**
  - There are no calls to methods with parameters outside the insert, remove, and get functions, which are appropriately defined.
- **Global Variables:**
  - There are no global variables; all variables are appropriately scoped within the class.

8. Sorting Array

- **Number of parameters:**
  - Yes, the parameters (size of the array, elements of the array) are correctly handled.
- **Attributes:**
  - The array uses the correct int data type for sorting.

9. Stack Implementation

- **Number of parameters:**
  - Yes, the parameters (value in push and pop methods) match the arguments.
- **Attributes:**
  - The int data type used for the stack elements matches between parameters and arguments.
- **Number of arguments transmitted:**
  - All methods (push, pop, isEmpty) handle the correct number of arguments.

10. Tower of Hanoi

- **Number of parameters:**
  - Yes, the number of parameters (disks and characters) matches the arguments passed to doTowers.
- **Attributes:**
  - The attributes (integer for the number of disks, characters for the rods) are correctly matched between parameters and arguments.

Category G : Input / Output Errors

Armstrong :

1.If files are explicitly declared, are their attributes correct?

- **Not applicable**: This program does not involve file I/O. No file attributes need to be declared.

2.Are the attributes on the file's OPEN statement correct?

- **Not applicable**: No file operations (like open statements) are present in this code.

3.Is there sufficient memory available to hold the file your program will read?

- **Not applicable**: Since no files are being read, memory requirements for file handling don't apply here. However, ensure your system can handle the maximum value of integers if very large numbers are passed.

4.Have all files been opened before use?

- **Not applicable**: No file operations are required in this program.

5.Have all files been closed after use?

- **Not applicable**: No file operations are present, so no need to close files.

6.Are end-of-file conditions detected and handled correctly?

- **Not applicable**: There are no files being read, so EOF conditions do not apply. However, input validation is important. Make sure to handle exceptions if args[0] is missing or not an integer.

7.Are I/O error conditions handled correctly?

- **Description**: Since there is no file I/O, this point refers to handling invalid input from the command line. The program can throw an error if a non-integer is passed as

input. Consider adding exception handling to catch NumberFormatException and prompt the user with an appropriate error message.

8.Are there spelling or grammatical errors in any text that is printed or displayed by the program?

- **Description**: The output messages in the program are mostly correct, but there is a small issue: "not a Armstrong Number" should be corrected to "not an Armstrong Number" for proper grammar.
-

GCM and LCD :

1.If files are explicitly declared, are their attributes correct?

- **Not applicable**: This program does not involve file I/O. No file attributes need to be declared.

2.Are the attributes on the file's OPEN statement correct?

- **Not applicable**: No file operations (like open statements) are present in this code.

3.Is there sufficient memory available to hold the file your program will read?

- **Not applicable**: Since no files are being read, memory requirements for file handling don't apply. However, sufficient memory should be available to store integer values. If the input numbers are extremely large, consider using long instead of int.

4.Have all files been opened before use?

- **Not applicable**: No file operations are required in this program.

5.Have all files been closed after use?

- **Not applicable**: No file operations are present, but the program correctly closes the Scanner object to prevent potential resource leaks.

6.Are end-of-file conditions detected and handled correctly?

- **Not applicable**: There are no files being read, so EOF conditions do not apply. However, input validation should be added to ensure that the inputs are positive integers.

7.Are I/O error conditions handled correctly?

- **Explanation**: The program should handle cases where the user provides invalid input, such as non-integer values or no input at all. Without proper error handling, the program may crash or behave unexpectedly.
- **Solution**: Add input validation and exception handling to ensure that only valid integers are processed. If the input is invalid (e.g., non-numeric characters or negative numbers), the program should display an appropriate error message to guide the user and avoid unexpected crashes. Additionally, check that the numbers

entered are positive integers, as GCD and LCM are typically defined for positive values.

8. Are there spelling or grammatical errors in any text that is printed or displayed by the program?

- **Explanation**: The program output should be clear, accurate, and free of spelling or grammatical mistakes. Incorrect or unclear messages can lead to confusion for users, making it difficult to understand the results or issues.
- **Solution**: Review the output statements for any errors. In the original program, there was a copy-paste mistake where both outputs were labeled as "The GCD of two numbers is". This should be corrected to properly label each result: one for GCD and one for LCM. Ensuring correct spelling and grammar will make the program's output more professional and easier to understand for users.

Knapsack:

1.If files are explicitly declared, are their attributes correct?

- **Check**: The program does not explicitly declare or use files. It reads input values directly from the command line arguments and does not perform file I/O operations.
- **Conclusion**: Not applicable.

2.Are the attributes on the file's OPEN statement correct?

- **Check**: There are no OPEN statements in the program because it does not handle any files.
- **Conclusion**: Not applicable.

3.Is there sufficient memory available to hold the file your program will read?

- **Check**: The program generates arrays (profit, weight, opt, sol) based on N and W. Ensure that N and W are not excessively large to avoid memory issues. The arrays grow in size based on these values, so excessively large input could lead to OutOfMemoryError.
- **Conclusion**: Generally, the program should work with reasonable inputs. However, very large values of N or W might lead to memory concerns.

4.Have all files been opened before use?

- **Check**: Since there are no file operations in this program, no explicit open operations are needed.
- **Conclusion**: Not applicable.

5.Have all files been closed after use?

- **Check**: No file operations mean no closing operations are required.

- **Conclusion**: Not applicable.

6.Are end-of-file conditions detected and handled correctly?

- **Check**: This is not relevant as there is no file reading. If the program were modified to read from files, it would need to handle end-of-file conditions.
- **Conclusion**: Not applicable.

7.Are I/O error conditions handled correctly?

- **Description**: The program does not handle cases where the command line inputs might be invalid (e.g., non-integer values or missing inputs). Consider adding exception handling to check for valid input types and appropriate values (e.g., positive integers for N and W). An error message could be displayed if invalid inputs are detected.

8.Are there spelling or grammatical errors in any text that is printed or displayed by the program?

- **Description**: The printed output looks mostly fine. However, adding column headers that are consistent in case (e.g., "Item", "Profit", "Weight", "Take") would enhance readability. Consider changing "item" to "Item" for uniformity.

Magic number :

1.If files are explicitly declared, are their attributes correct?

- **Check**: This program does not explicitly declare or use any files. It only reads input from the user via the console.
- **Conclusion**: Not applicable.

2.Are the attributes on the file's OPEN statement correct?

- **Check**: No OPEN statements are used in this program as there are no file operations.
- **Conclusion**: Not applicable.

3.Is there sufficient memory available to hold the file your program will read?

- **Check**: The program does not read from files but works with integer values provided by the user. It does not use much memory, so memory availability should not be an issue for normal integer inputs.
- **Conclusion**: Memory usage is minimal and should be sufficient for typical cases.

4.Have all files been opened before use?

- **Check**: There are no file operations in the program, so there is no need for any open operation.
- **Conclusion**: Not applicable.

5.Have all files been closed after use?

- **Check**: No files are being used, so no explicit close operations are necessary. However, since a Scanner object is used, it is a good practice to close it after usage using ob.close().
- **Recommendation**: Add ob.close(); at the end of the program to properly release the Scanner resource.

6.Are end-of-file conditions detected and handled correctly?

- **Check**: Since there are no file operations, this condition does not apply. If the program were modified to read input from a file, proper checks would be necessary to handle end-of-file scenarios.
- **Conclusion**: Not applicable.

7.Are I/O error conditions handled correctly?

- **Description**: The program does not handle invalid inputs from the user (e.g., non-integer values). Adding exception handling using a try-catch block can help in dealing with invalid inputs and provide appropriate error messages.

8.Are there spelling or grammatical errors in any text that is printed or displayed by the program?

- **Description**: The output text is mostly correct. Ensure that the phrasing is consistent; for example, it would be clearer to use "Enter a number to check if it is a Magic Number." instead of "Enter the number to be checked." to make the prompt more descriptive. Additionally, the output could be enhanced to say "is not a Magic Number" instead of "is not a Magic number" for consistent capitalization.

Merge sort algorithm:

1.If files are explicitly declared, are their attributes correct?

- **Check**: This program does not declare or use any files. It sorts an integer array declared within the program. Therefore, there are no file attributes to check.
- **Conclusion**: Not applicable.

2.Are the attributes on the file's OPEN statement correct?

- **Check**: There are no OPEN statements because the program does not involve file I/O.
- **Conclusion**: Not applicable.

3.Is there sufficient memory available to hold the file your program will read?

- **Check**: The program processes an in-memory integer array. Memory usage mainly depends on the array size. For typical input sizes, there should be no issues, but very large arrays could potentially lead to OutOfMemoryError.

- **Conclusion**: Memory usage should be adequate for normal inputs. Ensure that the array size is reasonable for the system's available memory.

4.Have all files been opened before use?

- **Check**: Since there are no file operations, no explicit open operations are needed.
- **Conclusion**: Not applicable.

5.Have all files been closed after use?

- **Check**: No files are being used, so no close operations are required.
- **Conclusion**: Not applicable.

6.Are end-of-file conditions detected and handled correctly?

- **Check**: No file reading is performed, so end-of-file conditions do not apply. If the program were modified to read from a file, appropriate handling would be necessary to avoid errors when the end of the file is reached.
- **Conclusion**: Not applicable.

7.Are I/O error conditions handled correctly?

- **Description**: The program does not handle potential input errors (like invalid array size or unexpected inputs). If you modify the program to read from external sources, adding proper error handling would be important to manage cases such as file not found, empty files, or improperly formatted data.

8.Are there spelling or grammatical errors in any text that is printed or displayed by the program?

- **Description**: The output messages ("before" and "after") are simple and correctly spelled. However, to enhance clarity, consider changing the format to "Before sorting: " and "After sorting: " for better readability.

Multiply two matrices

1.If files are explicitly declared, are their attributes correct?

- **Check**: The program does not explicitly declare or use files. It takes input from the user via the console and does not perform file I/O operations.
- **Conclusion**: Not applicable.

2.Are the attributes on the file's OPEN statement correct?

- **Check**: There are no OPEN statements because the program does not involve any file operations.
- **Conclusion**: Not applicable.

3.Is there sufficient memory available to hold the file your program will read?

- **Check**: The program processes matrices in memory using integer arrays. Memory usage depends on the size of the matrices. For typical inputs, the memory requirement is manageable, but excessively large matrices may lead to OutOfMemoryError.
- **Conclusion**: Memory should be sufficient for normal-sized matrices.

4.Have all files been opened before use?

- **Check**: Since there are no file operations in the program, no explicit open operations are needed.
- **Conclusion**: Not applicable.

5.Have all files been closed after use?

- **Check**: There are no files being used, so no closing operations are required. However, the Scanner object created should be closed after use to free resources.
- **Recommendation**: Add in.close(); at the end of the program to properly release the Scanner resource.

6.Are end-of-file conditions detected and handled correctly?

- **Check**: This is not relevant as there are no file reading operations. If the program were modified to read from files, proper checks would need to be added to handle end-of-file conditions.
- **Conclusion**: Not applicable.

7.Are I/O error conditions handled correctly?

- **Description**: The program does not handle cases where the user might enter invalid data types (e.g., non-integer values). Adding exception handling using a try-catch block can improve robustness by catching input errors and prompting the user appropriately.

8.Are there spelling or grammatical errors in any text that is printed or displayed by the program?

- **Description**: The prompts and output messages are mostly clear, but there is a repetition in the message "Enter the number of rows and columns of first matrix" when it should read "Enter the number of rows and columns of second matrix" for the second matrix input. This could lead to confusion for users. The output format could also be enhanced for readability, possibly by clarifying that the output is the product of the two entered matrices.

Quadratic Probing

1.If files are explicitly declared, are their attributes correct?

- **Check**: The program does not declare or use any files. It solely interacts with the user via the console for input and output.
- **Conclusion**: Not applicable.

2.Are the attributes on the file's OPEN statement correct?

- **Check**: There are no OPEN statements in this program since it doesn't involve any file operations.
- **Conclusion**: Not applicable.

3.Is there sufficient memory available to hold the file your program will read?

- **Check**: The program uses a hash table that is initialized based on user input, meaning memory consumption depends on the size specified by the user. For typical sizes, this should be adequate, but excessively large values may lead to OutOfMemoryError.
- **Conclusion**: Memory should be sufficient for standard sizes, but user-defined sizes should be monitored.

4.Have all files been opened before use?

- **Check**: There are no files involved, so no opening operations are needed.
- **Conclusion**: Not applicable.

5.Have all files been closed after use?

- **Check**: Since there are no files, there's nothing to close. However, the Scanner object created for input should be closed to release system resources.
- **Recommendation**: Add scan.close(); at the end of the main method to properly close the Scanner.

6.Are end-of-file conditions detected and handled correctly?

- **Check**: This program does not read from files, so end-of-file handling is not applicable. If it were modified to read from files, checks for reaching the end of input would need to be included.
- **Conclusion**: Not applicable.

7.Are I/O error conditions handled correctly?

- **Description**: The program does not handle cases where the user might input an invalid type or an unexpected value (e.g., non-integer when expecting an integer for the size). Implementing exception handling (using try-catch blocks) would improve the robustness of the program, allowing it to catch and handle such input errors gracefully.

8.Are there spelling or grammatical errors in any text that is printed or displayed by the program?

- **Description**: There are some minor inconsistencies in the printed output. For instance, the options in the menu lack consistent capitalization (e.g., "Insert" vs

"insert"). Also, in the makeEmpty method, the comment should be "Function to clear hash table" rather than "Functiont to clear hash table." Improving the consistency and accuracy of these texts will enhance readability and professionalism in the output.

Sorting the array in ascending order

1.If files are explicitly declared, are their attributes correct?

- **Check**: The program does not declare or use any files. It solely interacts with the user via console input and output.
- **Conclusion**: Not applicable.

2.Are the attributes on the file's OPEN statement correct?

- **Check**: There are no OPEN statements in this program since it doesn't involve any file operations.
- **Conclusion**: Not applicable.

3.Is there sufficient memory available to hold the file your program will read?

- **Check**: The program uses an integer array initialized based on user input, so memory consumption depends on the size specified by the user. For typical sizes, this should be adequate, but excessively large values may lead to OutOfMemoryError.
- **Conclusion**: Memory should be sufficient for standard sizes, but user-defined sizes should be monitored.

4.Have all files been opened before use?

- **Check**: There are no files involved, so no opening operations are needed.
- **Conclusion**: Not applicable.

5.Have all files been closed after use?

- **Check**: Since there are no files, there's nothing to close. However, the Scanner object created for input should be closed to release system resources.
- **Recommendation**: Add s.close(); at the end of the main method to properly close the Scanner.

6.Are end-of-file conditions detected and handled correctly?

- **Check**: The program does not read from files, so end-of-file handling is not applicable. If it were modified to read from files, checks for reaching the end of input would need to be included.
- **Conclusion**: Not applicable.

7.Are I/O error conditions handled correctly?

- **Description**: The program does not handle cases where the user might input an invalid type (e.g., a non-integer) when prompted for the number of elements or the array values. Implementing exception handling (using try-catch blocks) would improve the robustness of the program, allowing it to catch and handle such input errors gracefully.

8.Are there spelling or grammatical errors in any text that is printed or displayed by the program?

- **Description**: There are minor issues, such as the class name Ascending _Order which contains a space and should not have one; it should be AscendingOrder. Additionally, the prompt "Enter no. of elements you want in array:" could be improved to "Enter the number of elements you want in the array:" for clarity and correctness. The output formatting could also be enhanced for better readability, such as adding a newline before displaying the "Ascending Order" message.

Stack implementation

1.If files are explicitly declared, are their attributes correct?

- **Check**: The program does not declare or use any files. It performs operations solely in memory.
- **Conclusion**: Not applicable.

2.Are the attributes on the file's OPEN statement correct?

- **Check**: Since no files are being used, there are no OPEN statements in this program.
- **Conclusion**: Not applicable.

3.Is there sufficient memory available to hold the file your program will read?

- **Check**: The program uses an integer array to implement the stack. Memory usage depends on the size defined at instantiation. For standard sizes, the memory should generally be sufficient, but extremely large sizes could lead to OutOfMemoryError.
- **Conclusion**: Memory should be sufficient for standard sizes.

4.Have all files been opened before use?

- **Check**: There are no files involved, so no opening operations are necessary.
- **Conclusion**: Not applicable.

5.Have all files been closed after use?

- **Check**: As there are no file operations in the program, there is nothing to close.
- **Conclusion**: Not applicable.

6.Are end-of-file conditions detected and handled correctly?

- **Check**: The program does not read from files, so end-of-file handling is not applicable. If it were to read from a file, it should include checks to handle EOF conditions.
- **Conclusion**: Not applicable.

7.Are I/O error conditions handled correctly?

- **Description**: The program does not handle cases where invalid input may occur, such as trying to push a non-integer value onto the stack. Adding exception handling (using try-catch blocks) could improve its robustness against unexpected input types.

8.Are there spelling or grammatical errors in any text that is printed or displayed by the program?

- **Description**: There are no spelling or grammatical errors in the printed output. However, there could be improvements in the clarity of the error messages. For example, the message "Stack is full, can't push a value" could be rephrased for better readability as "Stack is full; cannot push the value." Additionally, the display logic in the display method has a mistake in the for loop condition that will cause it to never execute (i > top should be i <= top), leading to incorrect behavior when displaying the stack contents.

Tower of Hanoi

1.If files are explicitly declared, are their attributes correct?

- **Check**: The program does not declare or use any files. It performs operations entirely in memory.
- **Conclusion**: Not applicable.

2.Are the attributes on the file's OPEN statement correct?

- **Check**: There are no files being used, so there are no OPEN statements in this program.
- **Conclusion**: Not applicable.

3.Is there sufficient memory available to hold the file your program will read?

- **Check**: The program does not read from files but uses recursion to solve the Tower of Hanoi problem. The memory usage depends on the number of disks. For small values of n, memory usage is manageable, but a larger n could lead to a StackOverflowError due to deep recursion.
- **Conclusion**: Memory usage should be sufficient for small to moderate values of n.

4.Have all files been opened before use?

- **Check**: There are no file operations in the program, so no opening operations are necessary.
- **Conclusion**: Not applicable.

5.Have all files been closed after use?

- **Check**: Since there are no file operations, there is nothing to close.
- **Conclusion**: Not applicable.

6.Are end-of-file conditions detected and handled correctly?

- **Check**: The program does not read from files, so end-of-file handling is not applicable. If the program were reading from a file, it should include checks to handle EOF conditions appropriately.
- **Conclusion**: Not applicable.

7.Are I/O error conditions handled correctly?

- **Description**: The program does not handle any potential I/O error conditions, as it does not perform any file I/O operations. If it were to include user input or file reading, exception handling (using try-catch blocks) would be necessary to handle errors gracefully.

8.Are there spelling or grammatical errors in any text that is printed or displayed by the program?

- **Description**: There are no spelling or grammatical errors in the output of the program. However, the implementation contains issues with variable increments and decrements, specifically in the line doTowers(topN ++, inter--, from+1, to+1), which has incorrect syntax for the recursive calls and could lead to compilation errors. It should be corrected to maintain the intended logic of the Tower of Hanoi solution.

Category H : Other Checks

1. Cross-Reference Listing of Identifiers

- **Unused Variables**: In the provided code, all variables are utilized appropriately, but the logic for calculating check is flawed. The variable remainder is not used correctly, leading to potential confusion.

2. Attribute Listing

- **Variable Attributes**: All variables are initialized correctly and used appropriately, but their logic could lead to incorrect results due to the calculation errors. The check variable accumulates the sum of the cubes of digits, which is expected, but the calculation of remainder is wrong.

3. Compiler Warnings/Informational Messages

- While the program compiles successfully, there could be warnings about:
    - **Integer Division**: If there were any floating-point operations, warnings might suggest that integer division could lead to unexpected results.
    - **Math.pow Usage**: Using Math.pow with integers may raise performance concerns since it returns a double. Instead, using direct multiplication (e.g., remainder * remainder * remainder) for cubes is preferred.

4. Input Validity Checks

- The program lacks input validation. It should check whether the input is a valid integer and handle cases where the input is negative or non-integer. This would help prevent runtime errors and ensure the program behaves correctly with different inputs.

5. Missing Functions

- The program could benefit from better modularity. Extracting the logic for checking if a number is an Armstrong number into a separate method would enhance readability and maintainability. This makes the code cleaner and allows for easier testing of the Armstrong logic in isolation.

Knapsack :

1. **If the compiler produces a cross-reference listing of identifiers, examine it for variables that are never referenced or are referenced only once.**
    a. **Check**: All variables in the program are referenced at least once. However, the option2 variable can lead to issues since it uses profit[n-2], which may access an invalid index when n=1. This needs to be handled to ensure proper indexing.
    b. **Conclusion**: The variable usage looks appropriate overall, but careful attention is needed for proper indexing.
2. **If the compiler produces an attribute listing, check the attributes of each variable to ensure that no unexpected default attributes have been assigned.**
    a. **Check**: All variables have been declared correctly. They are initialized either directly or in a loop, and there are no unexpected attributes such as incorrect data types.
    b. **Conclusion**: The attributes of the variables are correct.
3. **If the program compiled successfully, but the computer produced one or more "warning" or "informational" messages, check each one carefully.**
    a. **Check**: There are potential warnings related to the logic in the handling of array indexing, especially concerning profit[n-2], which can lead to an ArrayIndexOutOfBoundsException. It's also worth checking if any warnings pertain to the use of uninitialized variables or any deprecated methods.

b. **Conclusion**: Ensure proper array indexing and check for any warnings related to this issue.

4. **Is the program or module sufficiently robust? That is, does it check its input for validity?**
    a. **Check**: The program does not validate inputs for N and W. If a non-numeric input is provided or if N or W are negative, the program will throw an exception. Implementing checks to ensure inputs are positive integers and handling exceptions would improve robustness.
    b. **Conclusion**: The program lacks input validation and should be enhanced to check for valid inputs.
5. **Is there a function missing from the program?**
    a. **Check**: The program could benefit from modularity. For example, separating the logic for generating random items, calculating the optimal knapsack solution, and printing results into distinct functions would improve readability and maintainability.
    b. **Conclusion**: Consider implementing functions to enhance the modularity of the program.

Magic number :

1. **If the compiler produces a cross-reference listing of identifiers, examine it for variables that are never referenced or are referenced only once.**
    a. **Check**: The variable s is declared and used but can be considered unnecessary because its value is not used outside the inner loop. Instead, the calculation logic inside the loop has flaws. The variable sum is also incorrectly used. The calculation should accumulate digits rather than use it as it is.
    b. **Conclusion**: Both s and sum have questionable usages that could lead to confusion.
2. **If the compiler produces an attribute listing, check the attributes of each variable to ensure that no unexpected default attributes have been assigned.**
    a. **Check**: All variables are properly declared. However, ensure that the initialization of s is appropriate. It starts at 0, which is fine, but its assignment within the inner loop is incorrect. There are no unexpected default attributes observed.
    b. **Conclusion**: No unexpected attributes; however, the logic needs fixing.
3. **If the program compiled successfully, but the computer produced one or more "warning" or "informational" messages, check each one carefully.**
    a. **Check**: The code may generate warnings regarding unreachable code or potential logic errors due to improper use of operators. Review all compiler warnings to ensure proper logic implementation, especially in the while loop where sum is intended to aggregate digits.
    b. **Conclusion**: Potential for warnings due to logic errors.
4. **Is the program or module sufficiently robust? That is, does it check its input for validity?**
    a. **Check**: The program does not handle invalid inputs (e.g., non-integer values, negative numbers). Adding checks to ensure that the input is a positive integer would enhance robustness.
    b. **Conclusion**: Input validation is missing; the program is not sufficiently robust.

5. **Is there a function missing from the program?**
    a. **Description**: The main logic for checking a magic number could be encapsulated in a separate function to improve modularity and readability. The inner loop logic should also be corrected to properly sum the digits of num and calculate the sum of digits correctly rather than accumulating incorrectly. Additionally, the method should consider zero cases correctly and provide meaningful messages or responses when encountering edge cases.

Merge sort algorithm :

1. **If the compiler produces a cross-reference listing of identifiers, examine it for variables that are never referenced or are referenced only once.**
    a. **Check**: In the current code, all defined variables are used appropriately. However, result in the merge method is never referenced or modified before being passed into the method. It is worth reviewing how the merge function is invoked.
    b. **Conclusion**: All relevant variables are used correctly.
2. **If the compiler produces an attribute listing, check the attributes of each variable to ensure that no unexpected default attributes have been assigned.**
    a. **Check**: The attributes of variables seem appropriate for their usage (e.g., integers, arrays). There are no unexpected attributes found.
    b. **Conclusion**: The attributes are appropriate and consistent with expected behavior.
3. **If the program compiled successfully, but the computer produced one or more "warning" or "informational" messages, check each one carefully. Warning messages are indications that the compiler suspects that you are doing something of questionable validity; all of these suspicions should be reviewed. Informational messages may list undeclared variables or language uses that impede code optimization.**
    a. **Check**: The current code will likely generate warnings or errors due to the improper usage of array indexing and the merge function. Specifically:
        i. leftHalf(array + 1) and rightHalf(array - 1) are incorrect usages for slicing arrays. It should not be attempting to add or subtract indices in this way.
        ii. The merge call uses left++ and right--, which are incorrect since these are not array references but array variables that need to remain unchanged.
    b. **Conclusion**: The code will likely produce compilation errors or warnings related to array manipulation and improper increment/decrement usage.
4. **Is the program or module sufficiently robust? That is, does it check its input for validity?**
    a. **Check**: The program does not currently validate input for the array or handle edge cases such as an empty array or null values. Adding checks for these conditions would improve robustness.
    b. **Conclusion**: The program lacks input validation and error handling, making it less robust.
5. **Is there a function missing from the program?**

a. **Description**: The program is missing a correct implementation for dividing the array and passing the correct subarrays to the merge method. Additionally, there should be a proper invocation of the merge function after sorting the left and right halves. Currently, it tries to use incorrect parameters that will lead to runtime errors.
b. **Conclusion**: The program needs corrections to the mergeSort method to ensure proper handling of array slicing and merging. The implementation of merging sorted halves must also be corrected to combine the sorted arrays properly into the result.

Multiply two matrices :

1.If the compiler produces a cross-reference listing of identifiers, examine it for variables that are never referenced or are referenced only once.

- **Check**: The variable sum is initialized and used multiple times within the matrix multiplication logic. The variables m, n, p, q, c, d, and k are also referenced appropriately throughout the code. No variables appear to be unused or referenced only once.
- **Conclusion**: All variables are effectively used.

2.If the compiler produces an attribute listing, check the attributes of each variable to ensure that no unexpected default attributes have been assigned.

- **Check**: The variables are appropriately defined with expected types (integers for counting and storing values). There are no unexpected attributes assigned to the variables in this context.
- **Conclusion**: Variable attributes are correct.

3.If the program compiled successfully, but the computer produced one or more "warning" or "informational" messages, check each one carefully. Warning messages are indications that the compiler suspects that you are doing something of questionable validity; all of these suspicions should be reviewed. Informational messages may list undeclared variables or language uses that impede code optimization.

- **Check**: If compiled, warnings might arise from using uninitialized values or incorrect array indexing (as seen in the code logic). Specifically, first[c-1][c-k] and second[k-1][k-d] may lead to ArrayIndexOutOfBoundsException when c or d is 0, since array indices in Java start from 0.
- **Conclusion**: Review for any potential warnings related to array indexing.

4.Is the program or module sufficiently robust? That is, does it check its input for validity?

- **Check**: The program checks if the number of columns in the first matrix matches the number of rows in the second matrix before proceeding with the multiplication. However, it does not validate that the user inputs for matrix dimensions and elements are positive integers. This could lead to runtime errors if invalid inputs are provided.
- **Conclusion**: Input validation could be improved to ensure robustness. Consider adding checks for valid integer inputs and dimensions.

5.Is there a function missing from the program?

- **Description**: The program currently lacks functions to modularize its logic. For instance, separate methods could be created for input handling, matrix multiplication, and output formatting. This would improve code readability and maintainability. Additionally, a method could be added to validate inputs before processing.

Quadratic Probing :

1. **If the compiler produces a cross-reference listing of identifiers, examine it for variables that are never referenced or are referenced only once.**
    a. **Check**: Review the code for variables that are declared but never used. For example, int tmp is used in the insert method but is not effectively utilized beyond its initial assignment.
    b. **Conclusion**: There might be variables that are defined but not needed; checking for their usage can help clean up the code.
2. **If the compiler produces an attribute listing, check the attributes of each variable to ensure that no unexpected default attributes have been assigned.**
    a. **Check**: Ensure that all variables have the expected data types. For instance, String[] keys and String[] vals are correctly declared as arrays of Strings. Ensure that no variables are unintentionally assigned a default value that could lead to errors in logic or runtime.
    b. **Conclusion**: All attributes seem correct, but it's good practice to double-check the initialization and data types.
3. **If the program compiled successfully, but the computer produced one or more "warning" or "informational" messages, check each one carefully.**
    a. **Check**: Review any warning messages generated during compilation. Potential issues could arise from the use of == to compare strings (use .equals() instead). Additionally, check for possible integer overflow in hash calculations or any warnings related to unoptimized code.
    b. **Conclusion**: Address warnings by ensuring string comparisons are done correctly and handling potential overflow or optimization issues.
4. **Is the program or module sufficiently robust? That is, does it check its input for validity?**
    a. **Check**: The program currently does not validate user input, such as checking for null or empty strings when inserting keys. Implementing checks to ensure valid inputs before processing them can prevent runtime errors or unwanted behavior.
    b. **Conclusion**: The program lacks robustness. Input validation should be added, especially when inserting and retrieving keys.
5. **Is there a function missing from the program?**
    a. **Check**: The program could benefit from a method to resize the hash table when it becomes full (dynamic resizing) or when the load factor exceeds a certain threshold. This would enhance performance and prevent collisions from becoming a bottleneck.
    b. **Conclusion**: A resizing function is advisable to improve the efficiency of the hash table and accommodate more entries without collisions.

sorting the array in ascending order :

1. **If the compiler produces a cross-reference listing of identifiers, examine it for variables that are never referenced or are referenced only once.**
    a. **Check**: The program uses the following variables: n, temp, s, a, i, and j. All are referenced adequately within their scope.
    b. **Conclusion**: No unused variables are found.
2. **If the compiler produces an attribute listing, check the attributes of each variable to ensure that no unexpected default attributes have been assigned.**
    a. **Check**: The variables have appropriate types: n (int for array size), temp (int for swapping values), and s (Scanner for input). The array a is initialized with a size defined by n.
    b. **Conclusion**: All attributes appear correct with no unexpected defaults.
3. **If the program compiled successfully, but the computer produced one or more "warning" or "informational" messages, check each one carefully.**
    a. **Check**: The code has a semicolon after the first for loop (for (int i = 0; i >= n; i++);), which results in an empty loop that does nothing. This could cause a logical error. The condition in this loop should be i < n instead of i >= n. The compiler may not flag this as an error, but it's a significant logical mistake.
    b. **Conclusion**: Review the warning about the semicolon and ensure it is removed to avoid an empty loop.
4. **Is the program or module sufficiently robust? That is, does it check its input for validity?**
    a. **Check**: The program does not validate the input for the number of elements or the elements themselves. For instance, it assumes the user will always enter valid integers. Additional checks could be added to ensure n is non-negative and that all input values are integers.
    b. **Conclusion**: The program lacks input validation and could be improved for robustness.
5. **Is there a function missing from the program?**
    a. **Check**: The program lacks modularity; the sorting functionality should ideally be encapsulated in a separate method (e.g., sortArray(int[] a)), improving code organization and reusability.
    b. **Conclusion**: Adding a function to handle the sorting logic would enhance the program structure.

Stack implementation:

1.If the compiler produces a cross-reference listing of identifiers, examine it for variables that are never referenced or are referenced only once.

- **Check**: In this program, all variables declared are utilized. The variable size and top are used in the stack operations, and stack is referenced throughout. However, top is improperly managed in the push method.

- **Conclusion**: No unused variables detected, but top is decremented incorrectly during push.

2.If the compiler produces an attribute listing, check the attributes of each variable to ensure that no unexpected default attributes have been assigned.

- **Check**: The attributes of all declared variables seem appropriate; however, ensure that integer variables are initialized correctly (e.g., top starts at -1).
- **Conclusion**: Attributes appear correct, but the initialization and manipulation of top may lead to errors.

3.If the program compiled successfully, but the computer produced one or more "warning" or "informational" messages, check each one carefully.

- **Check**: There are likely no warnings since the program compiles, but issues in logic may raise runtime errors. Compiler warnings regarding unchecked operations could appear, especially with incorrect stack operations.
- **Conclusion**: Review warning messages related to stack operations and ensure that all stack methods adhere to expected behavior.

4.Is the program or module sufficiently robust? That is, does it check its input for validity?

- **Check**: The program does check whether the stack is full before pushing and whether it is empty before popping. However, it does not handle cases where invalid data types are pushed or where user input is required.
- **Conclusion**: The stack implementation checks for basic conditions but could benefit from enhanced robustness, such as handling invalid inputs more gracefully.

5.Is there a function missing from the program?

- **Check**: The program implements basic stack functions (push, pop, display, and check if empty) but lacks other typical stack functionalities like:
    - peek(): To view the top element of the stack without removing it.
    - A method to return the current size of the stack.
    - A method to clear the stack.
- **Conclusion**: Adding these functionalities would enhance the stack implementation.

Tower of Hanoi :

1.If the compiler produces a cross-reference listing of identifiers, examine it for variables that are never referenced or are referenced only once.

- **Description**: In the provided code, there are no variables that are declared but never used. All variables, such as topN, from, inter, and to, are referenced appropriately within the doTowers method. However, there are issues with the use of topN, inter, and from in the recursive call, as they are incorrectly incremented and decremented.

2.If the compiler produces an attribute listing, check the attributes of each variable to ensure that no unexpected default attributes have been assigned.

- **Description**: The attributes of the variables are consistent with their intended use. The int variables (topN) are initialized and used correctly. The char variables (from, inter, to) are also appropriately defined. No unexpected default attributes are present.

3.If the program compiled successfully, but the computer produced one or more "warning" or "informational" messages, check each one carefully.

- **Description**: The code as provided contains potential syntax issues, particularly in the line doTowers(topN ++, inter--, from+1, to+1). The use of ++ and -- operators in this context could lead to compilation errors or warnings, as they are used incorrectly. Additionally, the intended logic of recursion may not work as expected due to the modifications of these parameters. If compiled, the code may raise warnings related to incorrect usage of increment and decrement operators.

4.Is the program or module sufficiently robust? That is, does it check its input for validity?

- **Description**: The program lacks input validation. It assumes that the number of disks is always a positive integer (in this case, nDisks is hardcoded as 3). A more robust implementation would allow the user to specify the number of disks, check for negative or zero values, and handle invalid inputs appropriately.

5.Is there a function missing from the program?

- **Description**: The program functions correctly in terms of implementing the Tower of Hanoi logic, but it could benefit from additional functionality, such as:
    - Input handling: A method to allow user input for the number of disks.
    - Validation checks: A mechanism to ensure the number of disks is a positive integer.
    - Improved output formatting: Clearer separation between the steps of the Tower of Hanoi solution for better readability.

Program Inspection:

1. Armstrong Number

1. How many errors are there in the program?

- There is 1 error in the code: remainder = num / 10; should be remainder = num % 10; since the remainder needs to be calculated using the modulus operator, not division.

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

- **Data Reference Errors**: The error is related to how the remainder is calculated and referenced.

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

- Logical errors related to the Armstrong number logic itself may not be easily caught unless tested with multiple inputs.

4. Is the program inspection technique worth applying?

- Yes, it helps in identifying simple reference issues like incorrect remainder calculations early.

2. GCD and LCM

1. How many errors are there in the program?

- There is 1 error: The condition while(a % b == 0) should be while(a % b != 0) in the GCD function.

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

- **Data Reference Errors**: The error is related to the incorrect condition for finding GCD.

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

- Possible issues with variable types or edge cases when using large numbers might not be identified without actual testing.

4. Is the program inspection technique worth applying?

- Yes, it catches small but critical errors in conditions that affect the logic of the program.

3. Knapsack

1. How many errors are there in the program?

- There are 2 errors:
  - opt[n++][w] should be opt[n][w].
  - The logic for option2 contains incorrect array access profit[n-2] and should be corrected to profit[n-1].

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

- **Data Reference Errors**: Out-of-bounds array access and incorrect use of increment operators are key issues.

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

- Performance or optimization issues may not be easily identified through program inspection alone.

4. Is the program inspection technique worth applying?

- Yes, especially for spotting incorrect array accesses and ensuring loops are within bounds.

4. Magic Number

1. How many errors are there in the program?

- There is 1 error: The condition while(sum == 0) should be while(sum != 0) in the inner loop to calculate the sum correctly.

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

- **Data Reference Errors**: This helps to identify the incorrect condition that references the wrong variable.

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

- Errors related to specific input cases (e.g., extremely large numbers) would require runtime testing.

4. Is the program inspection technique worth applying?

- Yes, it helps correct logical flaws in conditions before runtime errors occur.

5. Merge Sort

1. How many errors are there in the program?

- There are 2 errors:
  - Incorrect references like array+1 and array-1.

- The merge method's logic may access arrays incorrectly with left++ and right--.

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

- **Data Reference Errors**: This is helpful in spotting incorrect references and array accesses.

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

- Performance inefficiencies in the sorting process may not be identified through inspection alone.

4. Is the program inspection technique worth applying?

- Yes, it ensures the merge logic is correct and arrays are properly referenced.

6. Matrix Multiplication

1. How many errors are there in the program?

- There is 1 error: Accessing first[c-1][c-k] and second[k-1][k-d] incorrectly, which could lead to out-of-bounds errors.

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

- **Data Reference Errors**: Helpful in identifying incorrect array indexing.

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

- Numerical errors during the matrix multiplication process might not be caught through inspection alone.

4. Is the program inspection technique worth applying?

- Yes, it ensures that matrix access is within bounds and there are no logical errors with index usage.

7. Quadratic Probing

1. How many errors are there in the program?

- There is 1 major error: The statement i += (i + h / h--) % maxSize; can cause array out-of-bounds errors.

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

- **Data Reference Errors**: Key in identifying issues related to incorrect array indexing.

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

- Hashing inefficiencies or load factor issues in larger datasets are not easily detected.

4. Is the program inspection technique worth applying?

- Yes, it prevents out-of-bounds errors and ensures proper indexing in the hash table.

8. Sorting Array

1. How many errors are there in the program?

- There is 1 error: The loop condition for (int i = 0; i >= n; i++) is incorrect and should be for (int i = 0; i < n; i++).

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

- **Data Declaration Errors**: This helps in identifying improper loop conditions and boundaries.

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

- Logical issues in the sorting algorithm may not be caught without actually running the program.

4. Is the program inspection technique worth applying?

- Yes, especially for checking loop boundaries and array access before execution.

9. Stack Implementation

1. How many errors are there in the program?

- There are 2 errors:
  - top-- should be top++ in push().
  - The loop condition for(int i=0;i>top;i++) in display() is incorrect and should be i<top.

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

- **Data Reference Errors**: Useful for catching incorrect increments and decrements of top.

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

- Issues related to the stack's maximum capacity during runtime (e.g., stack overflow) may not be easily detected.

4. Is the program inspection technique worth applying?

- Yes, it helps ensure the stack operations are logically sound before running the program.

10. Tower of Hanoi

1. How many errors are there in the program?

- There are 2 errors:
  - The recursive function parameters topN++, inter--, from+1, to+1 are incorrect.
  - The recursion is not handled correctly in the second part of the method.

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

- **Data Reference Errors**: Helps identify logical mistakes in the recursive function calls.

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

- Specific runtime issues related to the recursion depth may not be caught.

4. Is the program inspection technique worth applying?

- Yes, it ensures recursive functions are structured correctly and helps avoid infinite recursion.

II. CODE DEBUGGING:

1. Armstrong Number

- There was an infinite loop in the code, so we need two breakpoints for error-free debugging.
- Corrected 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;
         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");
   }
```

2. GCD and LCM

- While debugging we got to know that there is exception in main thread, which is in method of GCD where variable a is divided by zero.
- This condition is incorrect because it checks when a is not divisible by both x and y, which is the opposite of what should be done for calculating the Least Common Multiple (LCM). The correct condition should check when a is divisible by both x and y.
- There are four breakpoint in the code.

```
public class GCD_LCM {
   static int gcd(int x, int y) {
      int r, a, b;
      a = (x > y) ? x : y;
      b = (x < y) ? x : y;
      while (b != 0) {  // Corrected loop condition
         r = a % b;
         a = b;
```

```
            b = r;
        }
        return a;
    }

    static int lcm(int x, int y) {
        int a = (x > y) ? x : y;
        while (true) {
            if (a % x == 0 && a % y == 0)  // Corrected LCM condition
                return a;
            ++a;
        }
    }

    public static void main(String args[]) {
        System.out.println("GCD: " + gcd(4, 5));
        System.out.println("LCM: " + lcm(4, 5));
    }
}
```

3. Knapsack

- The line int option1 = opt[n++][w]; is incorrect. Using n++ increments n after accessing opt[n][w], which can cause unintended behaviour in the loop. It should access the previous state without incrementing n.
- Fix: Change n++ to n-1 so it correctly references the previous state.
- The condition if (weight[n] > w) is incorrect because it prevents option2 from being calculated when the item's weight is less than or equal to the remaining capacity.
- Fix: The condition should check weight[n] <= w instead, allowing the item to be taken when its weight fits within the current capacity.
- The term profit[n-2] should be profit[n]. The profit should be calculated based on the current item being evaluated, not an earlier one.
- Fix: Replace profit[n-2] with profit[n].
- There are three break point needed for error correction.
- Corrected code :

```
//Knapsack
public class Knapsack {

    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);   // number of items
        int W = Integer.parseInt(args[1]);   // maximum weight 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]; // Fix: use n-1 to reference the previous state

                // take item n
                int option2 = Integer.MIN_VALUE;
                if (weight[n] <= w)  // Fix: weight[n] <= w to check if item fits
                    option2 = profit[n] + opt[n - 1][w - weight[n]]; // Fix: use profit[n] for current item

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

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

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

4. Magic Number

- The inner while loop checks if sum is zero. It should instead process the digits of sum to get the digit sum.
- The line s=s*(sum/10); is incorrect. This should accumulate the sum of the digits instead of performing multiplication.
- The line sum=sum%10 is missing a semicolon at the end.
- The condition in the inner while loop should continue while sum > 0, not sum == 0.
- If num never gets reduced to a single digit (which it will if the logic is correct), it could lead to an infinite loop.
- There are four break point in the code for error correction.
- Corrected code :

```
import java.util.*;

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

        while (num > 9) {
            int sum = 0;
            while (num > 0) {
                sum += num % 10; // Add the last digit to sum
                num /= 10;       // Remove the last digit
            }
            num = sum; // Set num to the sum of digits
        }

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

5.Merge Sort :

- The methods leftHalf and rightHalf are being called with incorrect parameters. You need to pass the array directly, not with adjustments like array + 1 or array - 1.
- The merge function call uses left++ and right--, which are invalid for arrays. You should pass the arrays directly.
- The merge function needs to accept the result array as an argument, which should be the original array passed to mergeSort.
- The result array should be initialized with the same length as the original array.
- There are five break point in the code for error correction.
- Corrected code :

```java
import java.util.*;

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

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

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

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

   // 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. **Matrix Multiplication**

- The expression first[c-1][c-k] and second[k-1][k-d] use incorrect indexing. The correct indices should refer to the current row and column, not decrementing them.
- The variable sum is being reset inside the innermost loop. It should be reset after calculating the product for each cell in the resulting matrix.
- The input prompts are repeated without clarity. The prompt for the second matrix incorrectly uses the prompt for the first matrix.
- The dimensions of the multiplication should check if the number of columns in the first matrix (n) equals the number of rows in the second matrix (p), which is already done correctly, but the subsequent code should make sure the calculation follows the matrix multiplication rules.
- There are six break point in the code for error correction.
- Corrected code :

```
import java.util.Scanner;

class MatrixMultiplication {
   public static void main(String args[]) {
      int m, n, p, q;

      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 (int c = 0; c < m; c++)
         for (int 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 (int c = 0; c < p; c++)
            for (int d = 0; d < q; d++)
               second[c][d] = in.nextInt();
```

```
        // Matrix multiplication logic
        for (int c = 0; c < m; c++) {
            for (int d = 0; d < q; d++) {
                int sum = 0; // Reset sum for each cell in the product matrix
                for (int k = 0; k < n; k++) { // Iterate over the columns of first and rows of second
                    sum += first[c][k] * second[k][d];
                }
                multiply[c][d] = sum; // Assign the computed sum to the product matrix
            }
        }

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

7. Quadratic Probing

- The line i + = (i + h / h--) % maxSize; has a syntax error due to an incorrect spacing. It should be i += (h * h) % maxSize; instead.
- The logic in these methods for calculating the index using h * h++ is incorrect. It should be h++ to increment h after each iteration, maintaining the correct probe sequence.
- The sum variable is used in the insert method but not properly reset. It should be declared and initialized within the method.
- The method printHashTable could print an empty table if no entries exist. It should provide feedback in such cases.
- In the remove method, if the key does not exist, it may lead to an infinite loop if keys[i] is not null. This can be resolved by adding a break condition.
- The prompt for entering the size in the main method has a typo: "maxSizeake" should be changed to "make".
- There are 12 break point in the code for error correction.
- Corrected code :

```
import java.util.Scanner;

/** Class QuadraticProbingHashTable **/
class QuadraticProbingHashTable {
    private int currentSize, maxSize;
    private String[] keys;
    private String[] vals;
```

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

/** Function to clear hash table **/
public void makeEmpty() {
    currentSize = 0;
    keys = new String[maxSize];
    vals = new String[maxSize];
}

/** Function to get size of hash table **/
public int getSize() {
    return currentSize;
}

/** Function to check if hash table is full **/
public boolean isFull() {
    return currentSize == maxSize;
}

/** Function to check if hash table is empty **/
public boolean isEmpty() {
    return getSize() == 0;
}

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

/** Function to get hash code of a given key **/
private int hash(String key) {
    return Math.abs(key.hashCode()) % maxSize;  // Use absolute value for positive indices
}

/** 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 += (h * h) % maxSize;  // Correct increment logic
        h++;
        i %= maxSize;  // Ensure index wraps around
    } 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 key and delete **/
    int i = hash(key), h = 1;
    while (!key.equals(keys[i])) {
        i = (i + h * h) % maxSize;
        h++;
        if (keys[i] == null) // Break if we find an empty spot
            return; // Key not found
    }

    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: ");
        if (isEmpty()) {
            System.out.println("Hash table is empty.");
        } else {
            for (int i = 0; i < maxSize; i++)
                if (keys[i] != null)
                    System.out.println(keys[i] + " " + vals[i]);
        }
        System.out.println();
    }
}

/** Class QuadraticProbingHashTableTest **/
public class QuadraticProbingHashTableTest {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Hash Table Test\n\n");
        System.out.println("Enter size");

        /** make object of QuadraticProbingHashTable **/
        QuadraticProbingHashTable qpht = new QuadraticProbingHashTable(scan.nextInt());

        char ch;

        /** Perform QuadraticProbingHashTable operations **/
        do {
            System.out.println("\nHash Table Operations\n");
            System.out.println("1. insert ");
            System.out.println("2. remove");
            System.out.println("3. get");
            System.out.println("4. clear");
            System.out.println("5. size");

            int choice = scan.nextInt();
            switch (choice) {
                case 1:
                    System.out.println("Enter key and value");
                    qpht.insert(scan.next(), scan.next());
                    break;
                case 2:
                    System.out.println("Enter key");
                    qpht.remove(scan.next());
                    break;
```

```
            case 3:
                System.out.println("Enter key");
                System.out.println("Value = " + qpht.get(scan.next()));
                break;
            case 4:
                qpht.makeEmpty();
                System.out.println("Hash Table Cleared\n");
                break;
            case 5:
                System.out.println("Size = " + qpht.getSize());
                break;
            default:
                System.out.println("Wrong Entry \n ");
                break;
        }

        /** Display hash table **/
        qpht.printHashTable();

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

8. Sorting Array

- The class name Ascending _Order contains a space, which is not allowed in Java. It should be changed to AscendingOrder.
- The outer loop condition in for (int i = 0; i >= n; i++) is incorrect. It should be i < n to iterate through the array elements.
- There is an unnecessary semicolon at the end of the outer loop declaration (for (int i = 0; i < n; i++);). This causes the loop body to be executed only once after the loop finishes.
- The sorting logic in the inner loop should swap elements when a[i] > a[j], not when a[i] <= a[j]. This ensures that elements are ordered correctly.
- The output format prints a comma after every element except the last one. It can be improved to format the output better.
- There are six break point in the code for error correction.
- Corrected code :

```
import java.util.Scanner;

public class AscendingOrder {
   public static void main(String[] args) {
      int n, temp;
      Scanner s = new Scanner(System.in);
      System.out.print("Enter no. of elements you want in array: ");
```

```
        n = s.nextInt();
        int a[] = new int[n];
        System.out.println("Enter all the elements:");

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

        for (int i = 0; i < n; i++) {  // Changed to i < n
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) {  // Corrected to >
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }

        System.out.print("Ascending Order: ");
        for (int i = 0; i < n; i++) {
            System.out.print(a[i]);
            if (i < n - 1) {
                System.out.print(", ");  // Improved output formatting
            }
        }
    }
}
```

9. Stack Implementation

- In the push method, top-- should be top++ to increment the index for the next value being pushed onto the stack. The current implementation decrements top, causing it to point to the wrong index.
- In the display method, the loop condition is i > top, which should be i <= top. This means the method won't display any elements since the condition is never true.
- In the pop method, top++ should be used to retrieve the value from the stack before incrementing it. Additionally, if the stack is not empty, the popped value should be stored or displayed.
- The current pop method does not indicate what value is being popped. It should return or print the popped value.
- The display method does not consider whether there are elements in the stack after popping. If the stack is empty, it should display a message.
- There are six break point in the code for error correction.
- Corrected code :

```
import java.util.Arrays;
```

```java
public class StackMethods {
    private int top;
    int size;
    int[] stack;

    public StackMethods(int arraySize) {
        size = arraySize;
        stack = new int[size];
        top = -1;
    }

    public void push(int value) {
        if (top == size - 1) {
            System.out.println("Stack is full, can't push a value");
        } else {
            top++;
            stack[top] = value;  // Corrected increment for pushing
        }
    }

    public Integer pop() {  // Changed return type to Integer for possible null
        if (!isEmpty()) {
            return stack[top--];  // Return popped value and decrement top
        } else {
            System.out.println("Can't pop...stack is empty");
            return null;  // Return null if stack is empty
        }
    }

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

    public void display() {
        if (isEmpty()) {
            System.out.println("Stack is empty.");
            return;
        }
        for (int i = 0; i <= top; i++) {  // Changed condition to <=
            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();

        // Pop values and display them
        System.out.println("Popped value: " + newStack.pop());
        System.out.println("Popped value: " + newStack.pop());
        System.out.println("Popped value: " + newStack.pop());
        System.out.println("Popped value: " + newStack.pop());

        newStack.display();
    }
}
```

10. Tower of Hanoi :

- In the recursive calls, the code attempts to modify parameters using topN++ and inter--. This doesn't work as intended because Java does not allow modification of primitive types like char and int in this way.
- The use of topN++ in doTowers will lead to an infinite loop because the value of topN is not decreased properly. Instead, it should be topN - 1.
- The expressions from + 1 and to + 1 do not yield the correct characters. Instead, you should pass the same characters (or appropriate ones) without modifying them.
- The output for moving disks must include all recursive calls, and since these calls are incorrect, it results in improper sequences being printed.
- There are four break point in the code for error correction.
- Corrected code :

```
public class MainClass {
    public static void main(String[] args) {
        int nDisks = 3;
        doTowers(nDisks, 'A', 'B', 'C');
    }

    public static void doTowers(int topN, char from, char inter, char to) {
        if (topN == 1) {
            System.out.println("Disk 1 from " + from + " to " + to);
        } else {
            doTowers(topN - 1, from, to, inter); // Move topN-1 disks from 'from' to 'inter'
            System.out.println("Disk " + topN + " from " + from + " to " + to); // Move the largest disk
            doTowers(topN - 1, inter, from, to); // Move the topN-1 disks from 'inter' to 'to'
        }
    }
}
```