IT314: Software Engineering

202201472 - Kaushik Prajapati

Lab - 7: Program Inspection, Debugging and Static Analysis

PART - I : Program Inspection

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

Code 1:

Category A: Uninitialized Variables

- The method meant to initialize the class (`__init__`) is mistakenly written as `_init_`, so the variables `self.matrix`, `self.vector`, and `self.res` are never set up.

Category C: Division by Zero

- In the `gauss` method, if the value `A[i][i]` is zero, there's a risk of dividing by zero when trying to calculate `x[i] /= A[i][i]`, which can cause errors.

Category D: Wrong Comparison Logic

- The `diagonal_dominance` method might not work properly if there are multiple maximum values in a row, leading to incorrect results.

Category E: Off-by-One Error

- In the `get_upper_permute` method, the loop condition is wrong. It runs one step too far with `k in range(i+1, n+1)` and should stop at `n`, like `k in range(i+1, n)` to prevent out-of-bounds errors.

Category F: Input Format Problems

- The methods expect specific input formats, like a list of lists for matrices and a list for vectors, but don't check if the input is in the right format, which can cause issues.

Category G: Missing Input Validation

- The code doesn't check if the input matrices have the right dimensions (e.g., square matrices) or if they match for operations, which could lead to errors.

Category H: Missing Libraries

- The code doesn't import important libraries like `numpy` and `matplotlib.pyplot`, which are necessary for it to work properly.

Code 2:

Category A: Data Reference Errors

- Constructor Naming Error: The constructor is named *init* instead of init. This means the constructor won't be called when you create an instance of the Interpolation class.
- Potential Index Errors: The code directly accesses elements in the matrix in the `cubicSpline` and `piecewise_linear_interpolation` methods without checking if the indices are valid, which could cause IndexErrors.

- No Type Checking: There are no checks to make sure the elements in the matrix are numbers (integers or floats). Passing non-numeric values could cause runtime errors.

Category C: Computation Errors

- Risk of Division by Zero: In the `piecewise_linear_interpolation` method, calculating the slope could result in division by zero if two x-values are the same.
- Uninitialized Variables: The `err` attribute is defined but not properly initialized in some methods, which could lead to inconsistencies.

Category D: Comparison Errors

- Floating Point Comparisons: Comparisons involving floating-point numbers can be inaccurate. The code should handle the precision correctly when comparing values from floating-point calculations.

Category E: Control-Flow Errors

- Loop Indexing Issues: The loop in the `mynewtonint` method needs to ensure it doesn't go out of bounds, especially with arrays, to avoid IndexErrors if n is small.
- Missing Edge Case Handling: In `cubicSpline`, there is no handling for cases where the input matrix has fewer than 4 points, which could cause errors in matrix operations.

Category F: Interface Errors

- Insufficient Documentation: The functions lack explanations (docstrings) that describe what they do, their parameters, and their return values, making them harder to use.
- No Parameter Validation: There's no validation of input parameters across methods, which could lead to incorrect results or runtime errors.

Category G: Input/Output Errors

- No Error Handling for Plotting: The plotting functions don't check if the input data is valid, which could cause runtime errors when plotting invalid data.

Category H: Other Checks

- Missing Compiler Warnings: The code might generate warnings during compilation or execution for unused variables or potential errors, but there are no checks to catch or handle these.

Code 3:

Category A: Data Reference Errors

- Redundant Function Definitions: The functions `fun` and `dfun` are defined multiple times for different equations without clear labels, making it confusing.
- Variable Reuse Issues: The `data` variable is reused to store different results, but it isn't clearly reset each time. This could cause unexpected problems when running multiple roots one after the other.

Category B: Data-Declaration Errors

- Uninitialized Variables: In the initial loop, the `next` variable is calculated before being properly set, which could lead to NaN (not a number) values in the first iteration.
- DataFrame Initialization Issue: The DataFrame `df` is created after the loop, so if the loop doesn't run (like in cases of immediate convergence), the data might not be set up correctly, causing errors.

Category C: Computation Errors

- Function Evaluation Issues: The line `fpresent = fun(present)` should check for convergence using both `|fun(present)|` and the value of `next`.
- Error Calculation Problem: The line `error.append(next_present)` may not truly reflect how close the solution is because it compares only the last two values instead of the last two iterations used in the process.

Category D: Comparison Errors

- Incorrect Error Check: The error check looks at the difference between `next` and `present`, but it might not consider that `present` could be very close to the root without actually converging.
- Weak Convergence Criteria: The convergence criteria only checks if `abs(next - present) > err`, ignoring whether `|fun(next)| < err`, which could also be important.

Category E: Control-Flow Errors

- Risk of Infinite Loop: If the initial guess is too far from the actual root or if `dfun(present)` is zero, the loop could run forever without converging.
- Missing Break Conditions: There are no limits to stop the loop after a certain number of iterations or to prevent division by zero in `next = present * (fpresent / dfpresent)`.

Category F: Input/Output Errors

- No Iteration Logging: The code doesn't log or display any output during iterations, making it hard to follow the algorithm's progress.
- Confusing Plot Titles: The plot titles don't clearly show which function or root they represent, which can confuse users when analyzing multiple roots from different functions.

Category G: Other Checks

- Unhandled Edge Cases: The code doesn't consider edge cases where a function might not have a root in the given domain or when the derivative could behave unexpectedly.
- Overlapping Plots: Each new plot is created without clearing the previous one's data, leading to cluttered visuals if multiple functions are tested one after the other.

Code 4:

Category A: Data Reference Errors

- Inconsistent Input Structure: The program expects a 2D array as input, but it doesn't check if the input shape is correct. This could cause runtime errors.
- Variable Reuse Confusion: The variables `coef` and `poly_i` are used in different places (inside and outside functions) without clear definitions, leading to confusion about their meanings.

Category B: Data-Declaration Errors

- Uninitialized Variables in Plotting: The `plot_fun` function doesn't handle situations where `y` might be empty or not set up properly, which can cause errors when trying to plot.
- No Error Handling for Matrix Inversion: There's no check to see if the matrix `ATA` can be inverted before using `np.linalg.inv(ATA)`. This could crash the program if the matrix is singular.

Category C: Computation Errors

- Potential Loss of Precision: The line `coef = coef[::-1]` reverses the coefficients, but the polynomial needs them in descending order, which could lead to unexpected results.
- Overwriting Coefficients: The coefficients for each polynomial are computed in a loop, but they aren't separated in the final output. This can make it unclear which coefficients belong to which polynomial.

Category D: Comparison Errors

- Incorrect Error Tolerance: The error tolerance `err = 1e-3` is hardcoded in the `plot_fun` function, which may not be suitable for all datasets and doesn't allow for adjustments based on input ranges.
- Poor Comparison Logic in Plotting: When plotting multiple polynomials, the code doesn't ensure that each polynomial has a clear label, making it hard to understand the plotted lines.

Category E: Control-Flow Errors

- Infinite Loop Risk in Plotting: The plotting function could get stuck in an infinite loop if it encounters incorrectly formatted data, especially if there are no points to plot.
- Missing Early Exit Conditions: The `leastSquareErrorPolynomial` function lacks conditions to stop early if it finds that the matrix is poorly conditioned or if the polynomial degree is too high for the number of points.

Category F: Input/Output Errors

- No User Feedback on Processing: There's no indication of progress or completion for polynomial fitting, making it difficult for users to track what's happening during execution.
- Misleading Variable Naming: The name `poly_i` can be confusing because it suggests a single polynomial, while it actually stores a polynomial object. A clearer name would help.

Category G: Other Checks

- No Handling of Edge Cases: The function doesn't consider edge cases where all `y` values are the same, which would create a constant polynomial and might confuse users.
- Lack of Unit Tests or Assertions: There are no tests to check input parameters or validate that the function works correctly in different situations.

Category H: General Code Quality

- Redundant Code Sections: The code for plotting multiple polynomials has some redundancy and could be improved by putting it into a function for better reuse.
- Missing Function Documentation: There's no documentation for the functions, making it harder for other users (or the author later) to understand what the code does.

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

Categories A (Data Reference Errors) and D (Comparison Errors) are the most important. Many problems come from incorrect comparisons, like ensuring diagonal dominance, and from not setting up the constructor correctly. These issues can lead to runtime errors.

Data Reference Errors are key because they deal with how inputs are managed. For instance, using the wrong constructor name (*init* instead of **init**) or having index errors can cause problems. Fixing these issues is crucial for program stability.

Also, Computation Errors (Category C) matter for getting accurate results, especially in polynomial fitting.

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

Program inspection can catch many errors, but it may miss important ones that appear only when the program runs. For example, data-specific errors like having all y-values the same might not show up until specific datasets are tested.

Hidden logical errors can also lead to wrong results. These include finding the wrong root or problems that happen during execution. Runtime errors, like floating-point issues or unexpected input types, might go unnoticed, causing unexpected behavior.

Lastly, logical errors can occur where the code runs but gives incorrect results due to flawed algorithms, especially in methods like Jacobi or Gauss-Seidel.

4. Is the program inspection technique worth applicable?

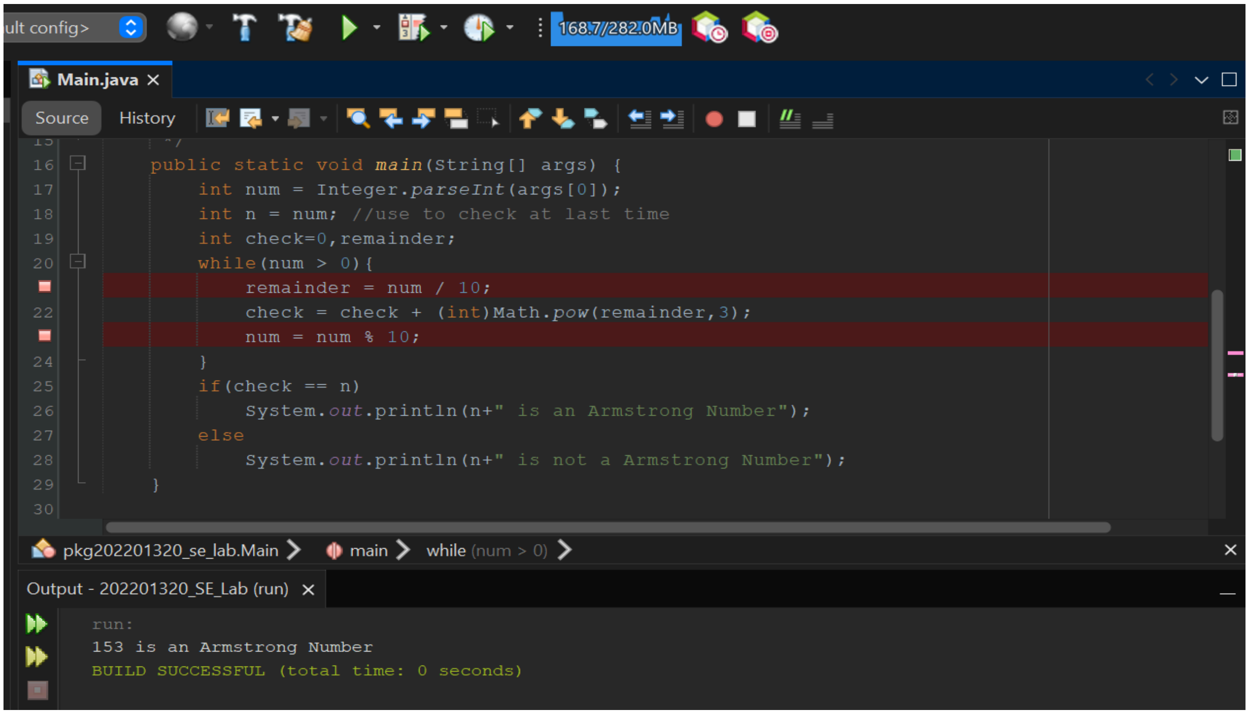
Yes, program inspection is a useful technique for finding errors in code. It helps spot common mistakes related to data, calculations, and comparisons, improving code quality.

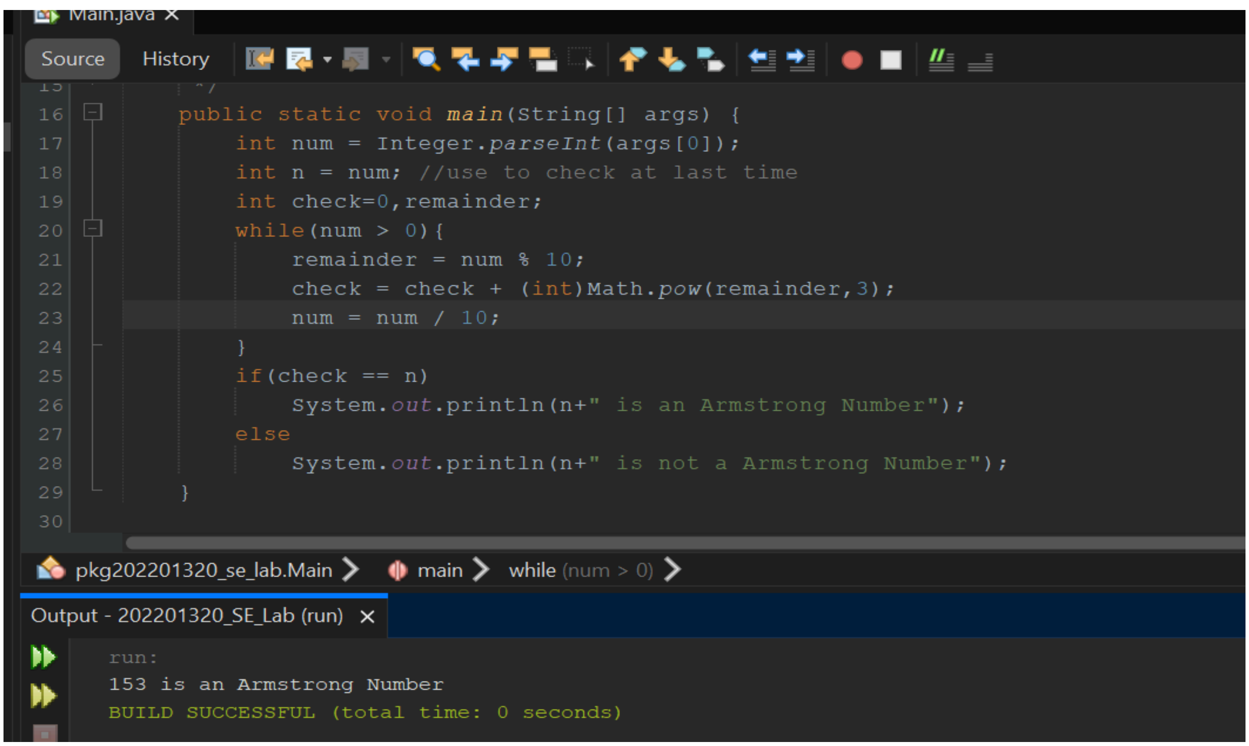
However, it’s important to use other testing methods alongside it, like unit testing and dynamic analysis. This helps catch logical errors and ensures the program works well in different situations.

Program inspection is especially beneficial in team projects, making code easier to understand and maintain. Overall, it plays a key role in improving code quality, especially for complex tasks like numerical methods and data analysis.

PART - II : Code Debugging

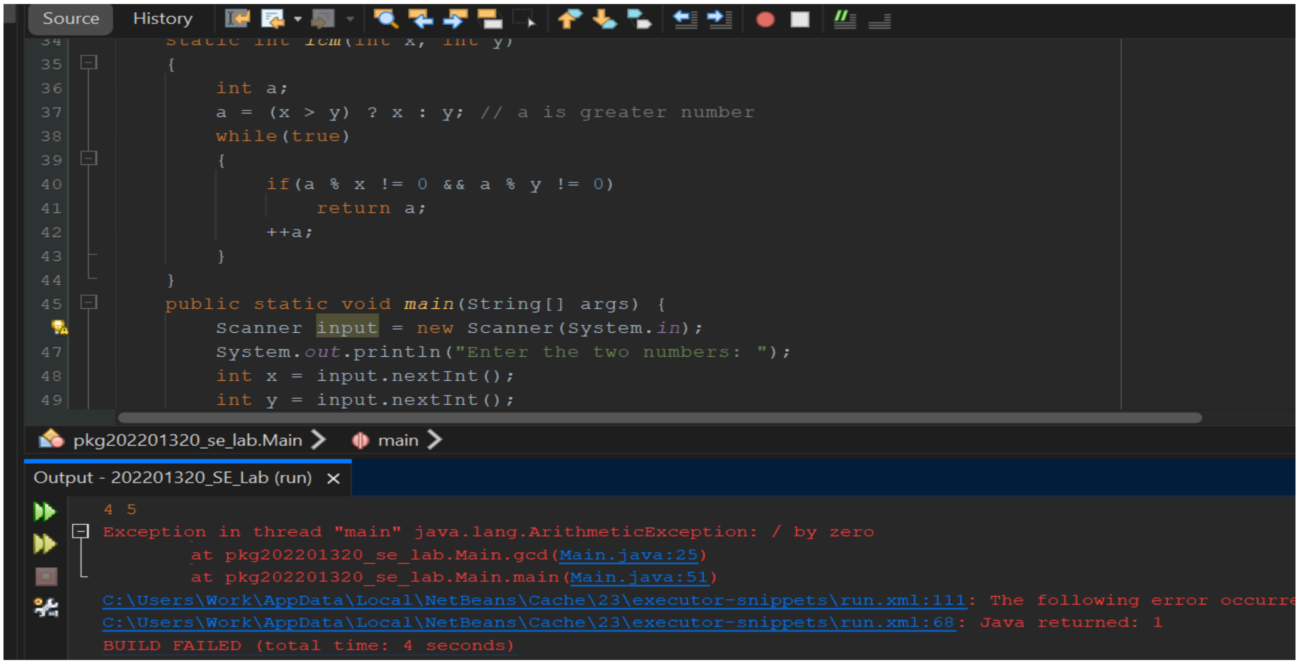
1. Armstrong Number



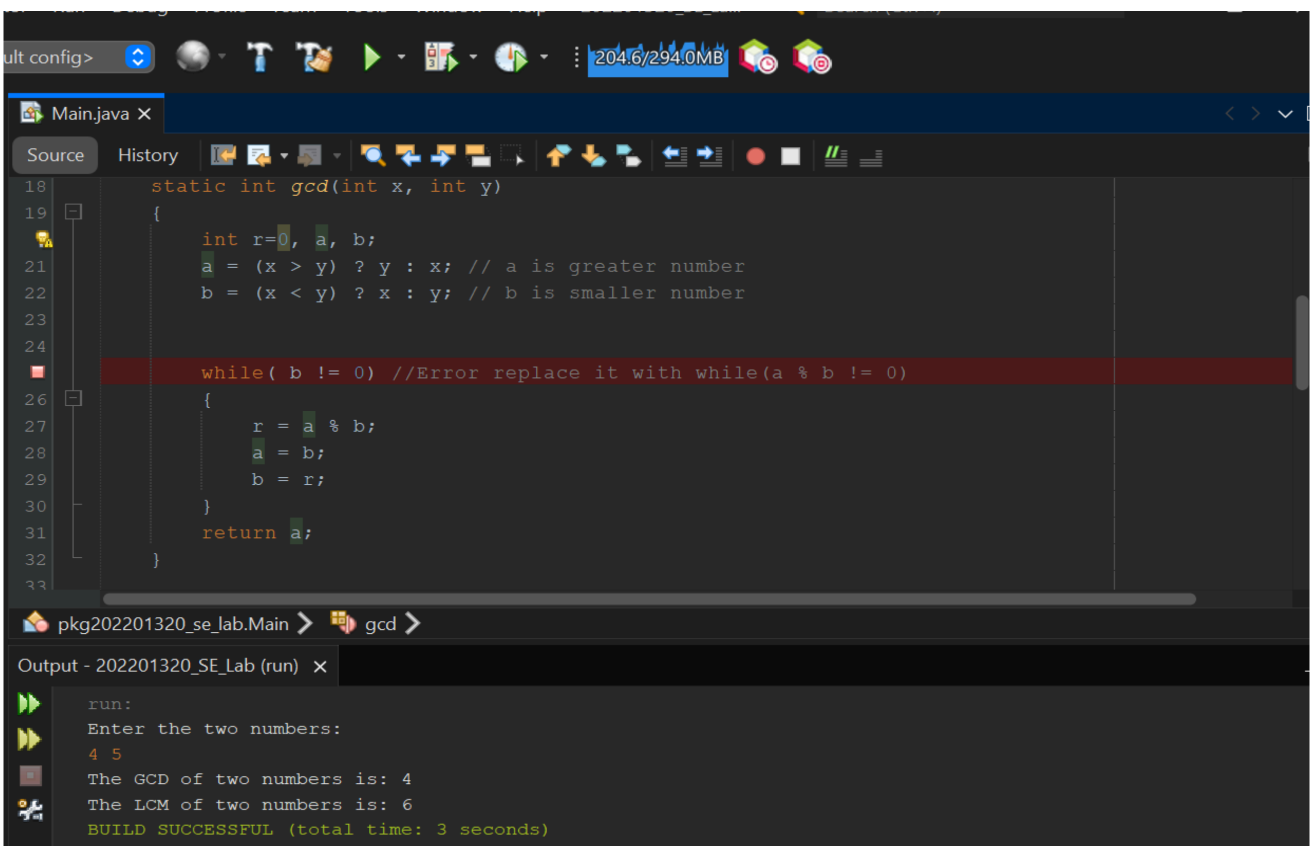


The error in this code is that remainder is calculated wrongly and this leads to error in the main().

2. GCD_LCM



Here the while loop should be a%b!=0 instead of a%b==0 and thus it throws an ArithmeticExpression error.



3. KnapSack Problem

```
run:
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index -1420 out of bounds for length 2001
        at pkg202201320_se_lab.Main.main(Main.java:44)
C:\Users\Work\AppData\Local\NetBeans\Cache\23\executor-snippets\run.xml:111: The following error occurred while executing thi
C:\Users\Work\AppData\Local\NetBeans\Cache\23\executor-snippets\run.xml:68: Java returned: 1
BUILD FAILED (total time: 0 seconds)
```

Main.java

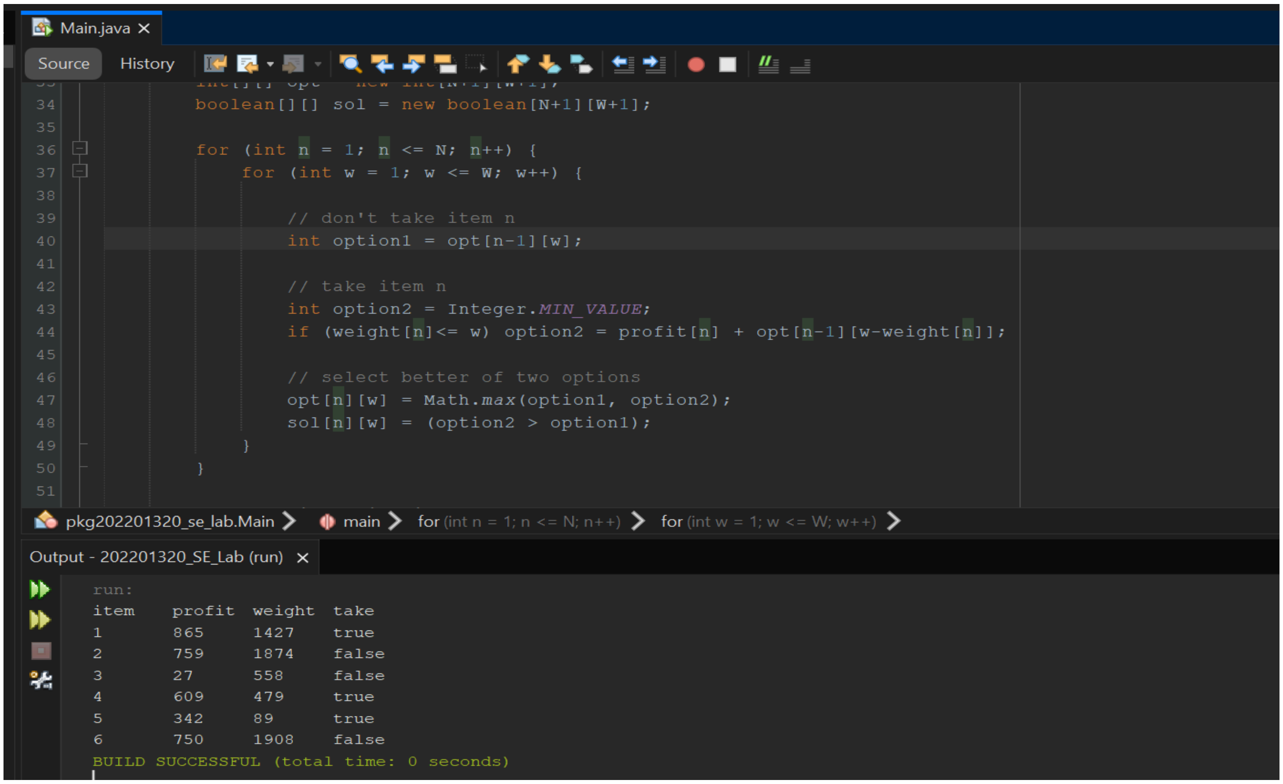
Source History

```java
        int[][] opt = new int[N+1][W+1];
        boolean[][] sol = new boolean[N+1][W+1];

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

                // don't take item n
                int option1 = opt[n-1][w];

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

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

pkg202201320_se_lab.Main > main > for (int n = 1; n <= N; n++) > for (int w = 1; w <= W; w++) >

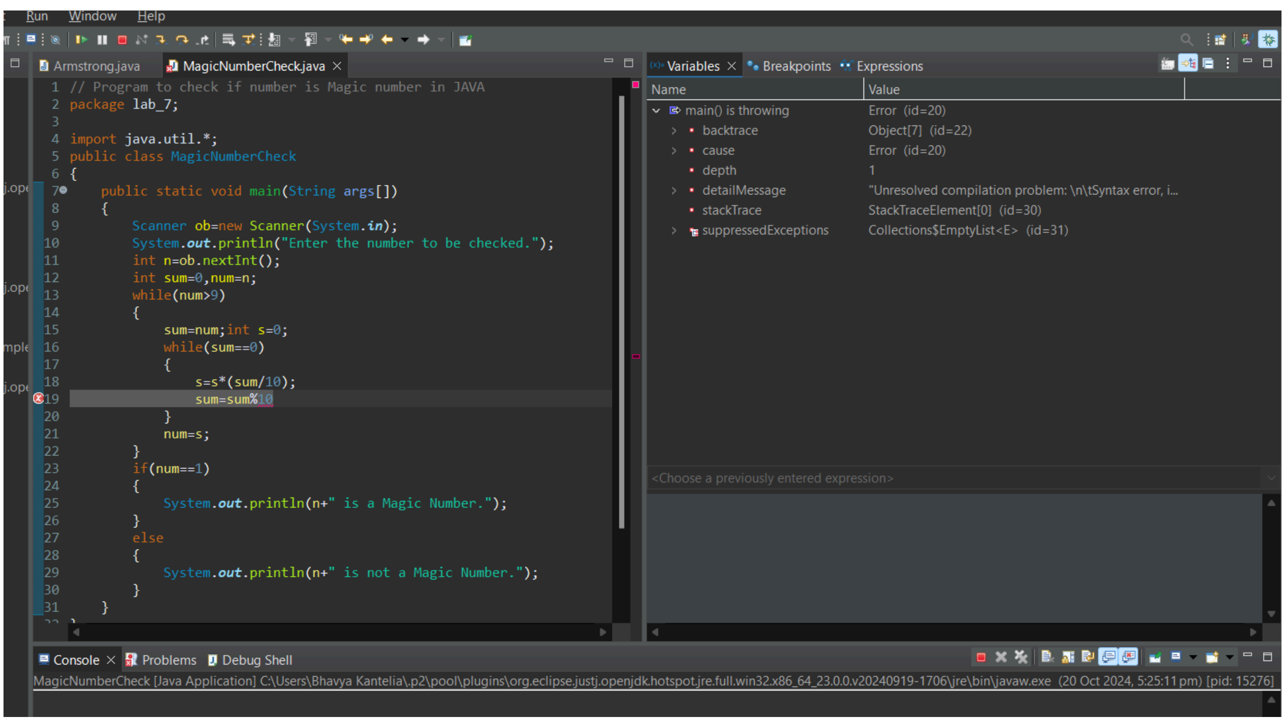
Output - 202201320_SE_Lab (run)

```
run:
item    profit  weight  take
1       865     1427    true
2       759     1874    false
3       27      558     false
4       609     479     true
5       342     89      true
6       750     1908    false
BUILD SUCCESSFUL (total time: 0 seconds)
```



Error here is an incorrect index update of n-1 which should be n++ which causes main out of bound error.

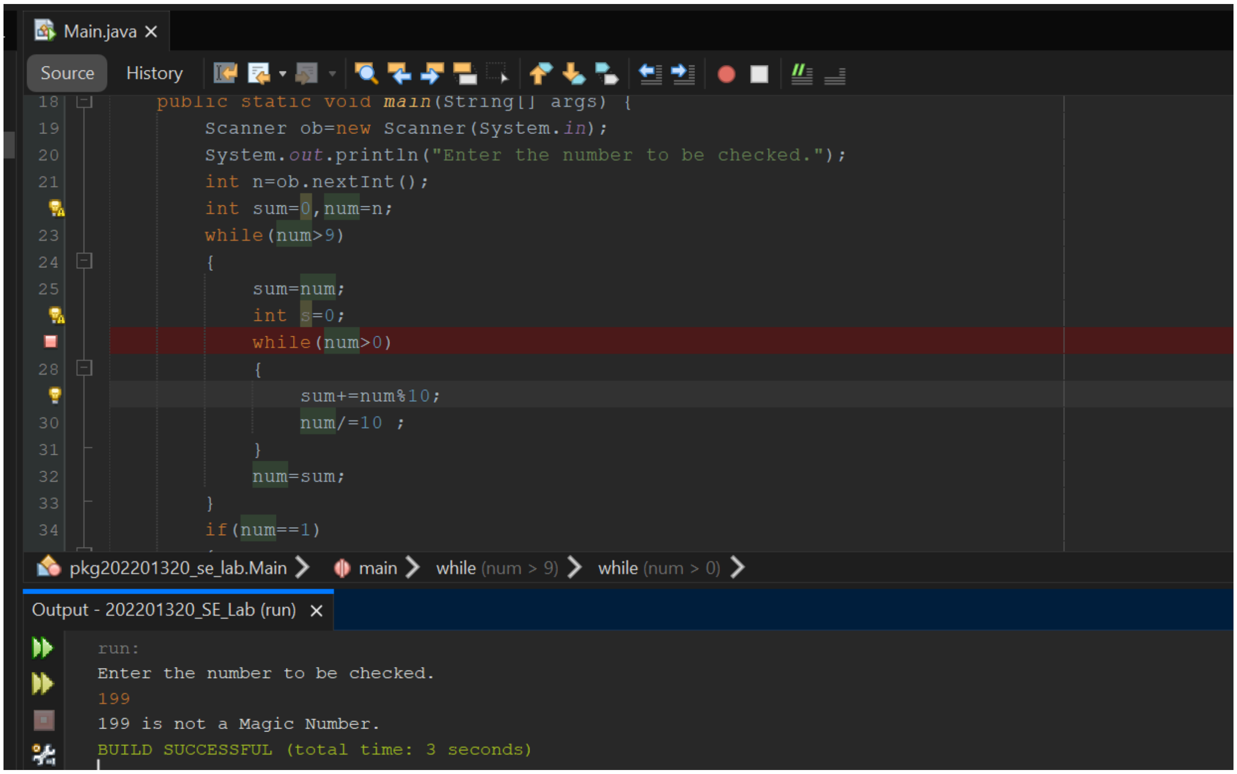
4. MagicNumberCheck



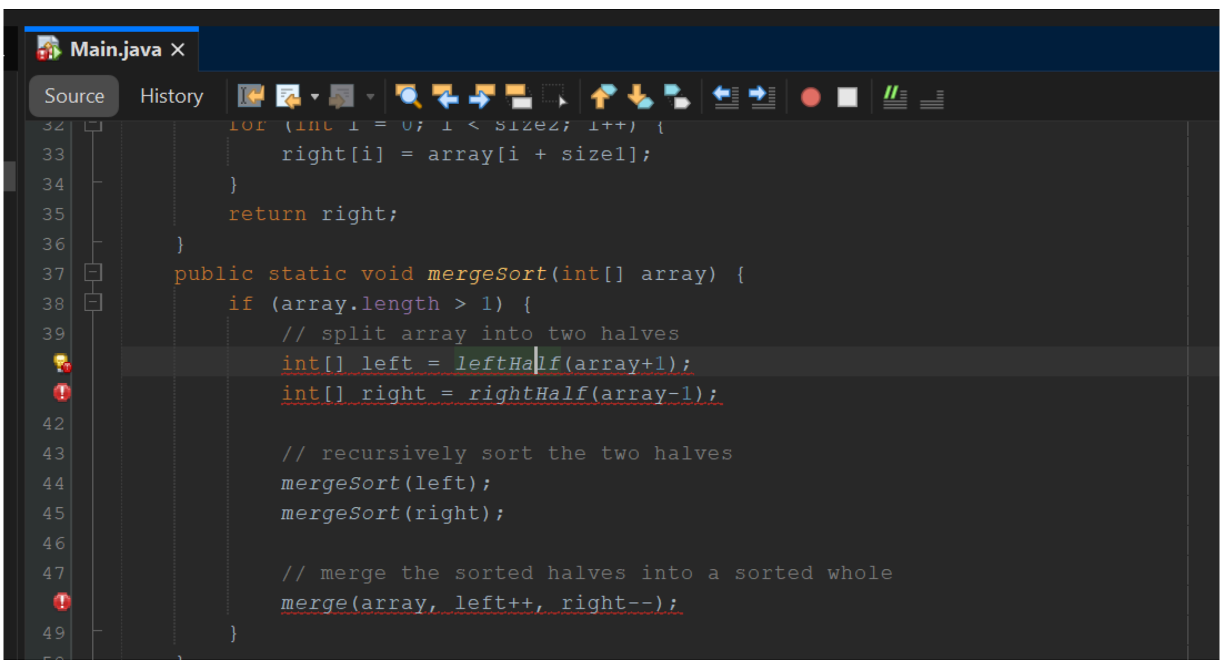
Original Condition: while (sum == 0)

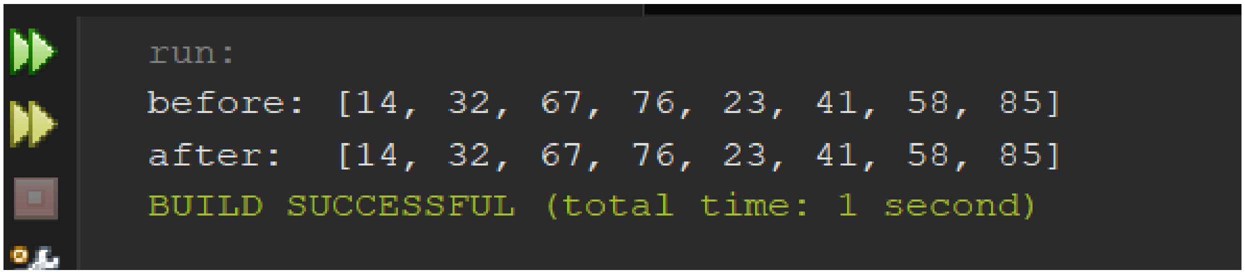
This condition would never execute if sum is initialized to a non-zero value (which it is, since it starts as num).

As a result, the loop meant to sum the digits wouldn't run at all, leading to incorrect or no



5. MergeSort

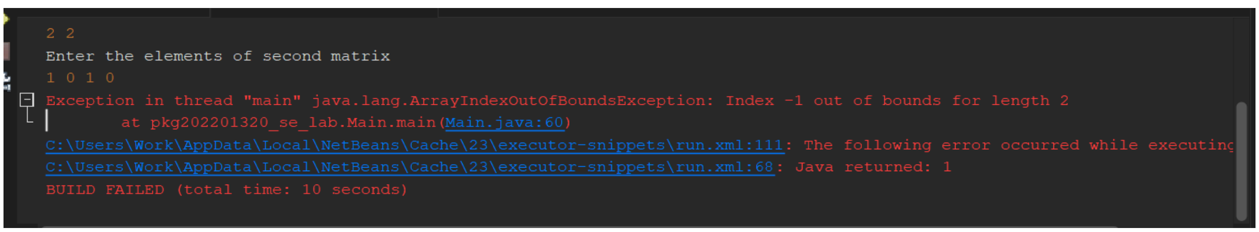


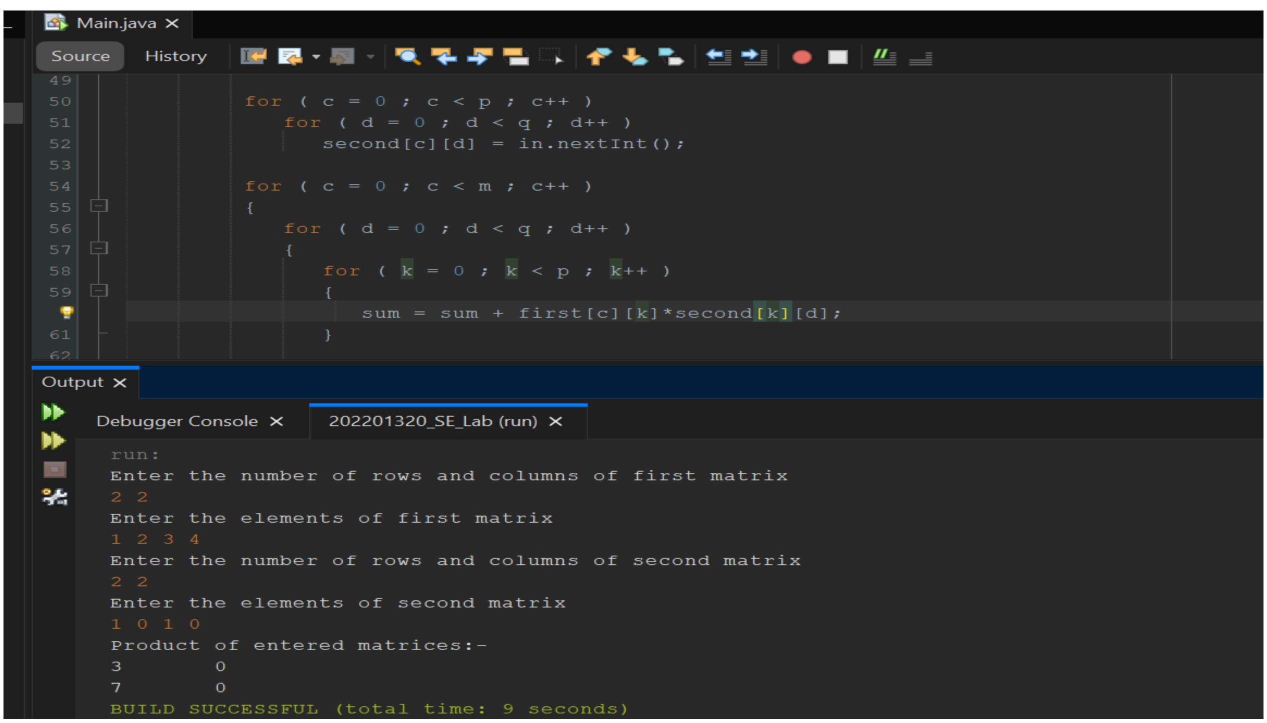


array + 1 and array -1:

This is invalid because you cannot perform arithmetic operations on an array reference. In Java (and many programming languages), adding or subtracting integers directly to/from an array reference does not make sense, as arrays are objects and not numeric types. If you want to access elements, you would use an index, like array[i]. Subtracting from an array reference does not yield a meaningful result in terms of accessing its element.

6. Matrix Multiplication





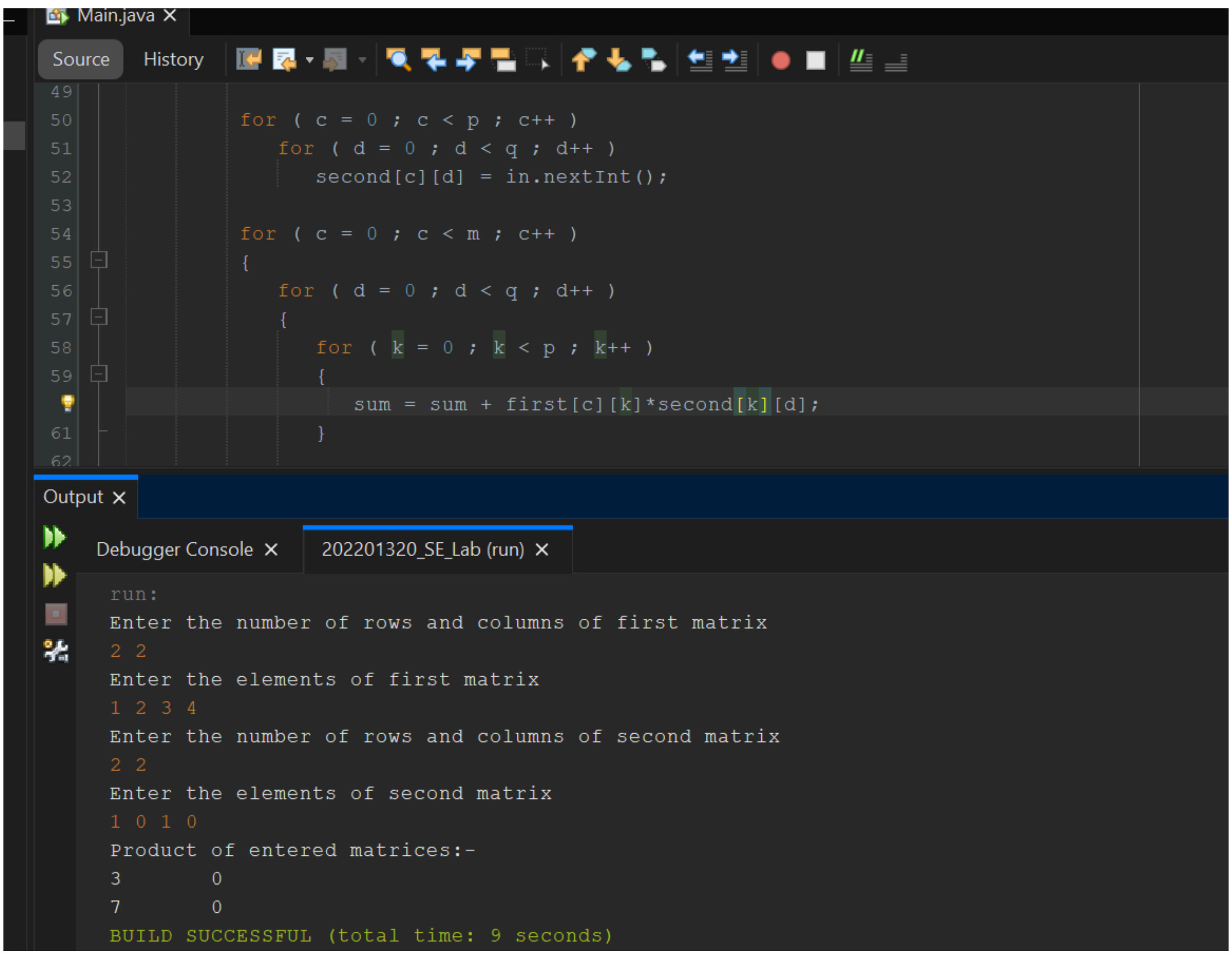
The correct indices should be:

For the first matrix, use first[c][k], which refers to the k-th element in the c-th row.

For the second matrix, use second[k][d], which refers to the k-th element in the d-th column.

Using indices like first[c-1][c-k] and second[k-1][k-d] can lead to invalid references, resulting in negative values or accessing incorrect elements. This is because matrix indices need to stay within valid bounds based on the structure of the matrices involved.

7. Quadratic Probing



i + = (i + h / h--) % maxSize; is invalid syntax. It should be i = (i + h * h++) % maxSize;. The += operator should be properly placed, and the arithmetic operation should use * for quadratic probing, not /.

8. Ascending

Incorrect Output:

```
run:
Enter no. of elements you want in array:5
Enter all the elements:
1 12 2 9 7
Ascending Order:12 1 2 9 7BUILD SUCCESSFUL (total time: 9 seconds)
```

```java
        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++);
        {
            int i = 0;
            for (int j = i + 1; j < n; j++)
            {
                if (a[i] > a[j])
                {
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
```

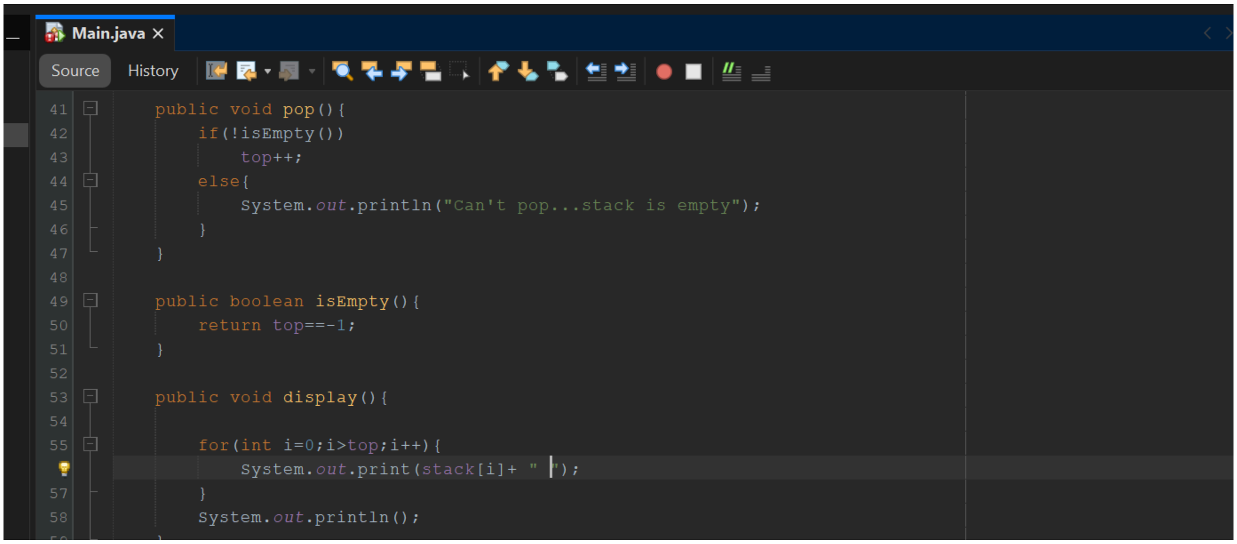
Correct Output:

```
run:
Enter no. of elements you want in array:5
Enter all the elements:
9 8 7 6 4
Ascending Order:4,6,7,8,9BUILD SUCCESSFUL (total time: 5 seconds)
```

The class name Ascending _Order has a space, which is not allowed in Java. The space should be removed or replaced with an underscore (_) if you want to separate words. The condition for (int i = 0; i >= n; i++); is incorrect because i >= n means the loop will never run, and there is an unnecessary semicolon (;) at the end of the loop. The correct condition should be i < n. In the inner if condition, you are checking if (a[i] <= a[j]), which will sort the array in descending order. You should change it to if (a[i] >

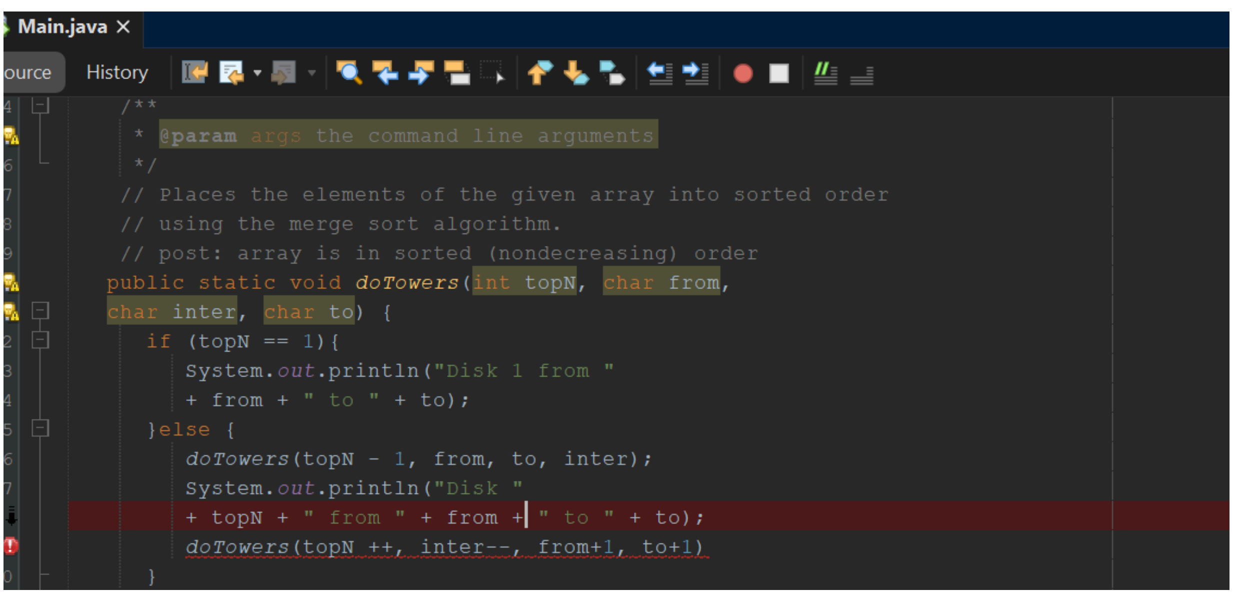
a[j]) to sort the array in ascending order. The final loop prints the array elements separated by commas but incorrectly leaves a trailing comma.

9. Stack



In the push method, top-- is used, which decrements top, but it should be top++ to increment the position for inserting a new value. In the display method, the condition for(int i=0;i>top;i++) is incorrect, as it will never execute. The condition should be i <= top to display all elements from index 0 to top. In the pop method, it only increments the top but doesn't actually remove the element or return it. For a correct stack implementation, you should return the popped value, and it should also decrement the top pointer

10. TowerOfHanoi



The expressions topN++, inter--, from + 1, and to + 1 are incorrect in the context of the recursive calls. The parameters should be passed unchanged (no increment or decrement) to maintain the correct behavior of the algorithm. The topN decrement in the recursive calls should be topN - 1, not topN ++.

Correct Output:

```
run:
Disk 1 from A to C
Disk 2 from A to B
Disk 1 from C to B
Disk 3 from A to C
Disk 1 from B to A
Disk 2 from B to C
Disk 1 from A to C
BUILD SUCCESSFUL (total time: 0 seconds)
```