Software Engineering

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

Program Inspection
Category A

```
2794    void CCompositor::arrangeMonitors() {
2795        static auto* const     PXWLFORCESCALEZERO = (Hyprlang::INT* const*)g_pC
2796
2797        std::vector<CMonitor*> toArrange;
2798        std::vector<CMonitor*> arranged;
2799
2800        for (auto const& m : m_vMonitors)
2801            toArrange.push_back(m.get());
2802
2803        Debug::log(LOG, "arrangeMonitors: {} to arrange", toArrange.size());
2804
2805        for (auto it = toArrange.begin(); it != toArrange.end();) {
2806            auto m = *it;
2807
2808            if (m->activeMonitorRule.offset != Vector2D{-INT32_MAX, -INT32_MAX}
2809                // explicit.
2810                Debug::log(LOG, "arrangeMonitors: {} explicit {:j}", m->szName,
2811
2812                m->moveTo(m->activeMonitorRule.offset);
2813                arranged.push_back(m);
2814                it = toArrange.erase(it);
2815
2816                if (it == toArrange.end())
2817                    break;
2818
2819                continue;
2820            }
```

Potential issues with array access: In methods such as CCompositor::arrangeMonitors(),
there are loops that reference elements of arrays or lists (e.g., m_lMonitors). There are
no explicit boundary checks for array indices, which poses a risk of out-of-bounds access,
particularly if the list is empty or shorter than anticipated.

```
1347    PHLWINDOW CCompositor::getTopLeftWindowOnWorkspace(const WORKSPACEID& id) {
1348        const auto PWORKSPACE = getWorkspaceByID(id);
1349
1350        if (!PWORKSPACE)
1351            return nullptr;
1352
1353        const auto PMONITOR = getMonitorFromID(PWORKSPACE->m_iMonitorID);
1354
1355        for (auto const& w : m_vWindows) {
1356            if (w->workspaceID() != id || !w->m_bIsMapped || w->isHidden())
1357                continue;
1358
1359            const auto WINDOWIDEALBB = w->getWindowIdealBoundingBoxIgnoreReserved();
1360
1361            if (WINDOWIDEALBB.x <= PMONITOR->vecPosition.x + 1 && WINDOWIDEALBB.y <= PMONITOR->vecPosition.y + 1)
1362                return w;
1363        }
1364        return nullptr;
1365    }
```

The pointer PMONITOR is initialized, but there is no assurance that it cannot be null, which may result in null reference issues.

Category B

```
void CCompositor::arrangeMonitors() {
    static auto* const    PXWLFORCESCALEZERO = (Hyprlang::INT* const*)g_pConfigManager->getConfigValuePtr("xwayland:force_zero_scaling");

    std::vector<CMonitor*> toArrange;
    std::vector<CMonitor*> arranged;

    for (auto const& m : m_vMonitors)
        toArrange.push_back(m.get());

    Debug::log(LOG, "arrangeMonitors: {} to arrange", toArrange.size());

    for (auto it = toArrange.begin(); it != toArrange.end();) {
        auto m = *it;

        if (m->activeMonitorRule.offset != Vector2D{-INT32_MAX, -INT32_MAX}) {
            // explicit.
            Debug::log(LOG, "arrangeMonitors: {} explicit {:j}", m->szName, m->activeMonitorRule.offset);

            m->moveTo(m->activeMonitorRule.offset);
            arranged.push_back(m);
            it = toArrange.erase(it);

            if (it == toArrange.end())
                break;

            continue;
```

Variable shadowing: In the previously mentioned snippet, the variable 'm' is utilized, which also appears in several other instances. This can lead to potential problems due to the shadowing of similar names across different scopes.

```
2739        if (FULLSCREEN)
2740            setWindowFullscreenInternal(pWindow, FSMODE_NONE);
2741
2742        if (!pWindow->m_bIsFloating) {
2743            g_pLayoutManager->getCurrentLayout()->onWindowRemovedTiling(pWindow);
2744            pWindow->moveToWorkspace(pWorkspace);
2745            pWindow->m_iMonitorID = pWorkspace->m_iMonitorID;
2746            g_pLayoutManager->getCurrentLayout()->onWindowCreatedTiling(pWindow);
2747        } else {
2748            const auto PWINDOWMONITOR = g_pCompositor->getMonitorFromID(pWindow->m_iMonitorID);
2749            const auto POSTOMON       = pWindow->m_vRealPosition.goal() - PWINDOWMONITOR->vecPosition;
2750
2751            const auto PWORKSPACEMONITOR = g_pCompositor->getMonitorFromID(pWorkspace->m_iMonitorID);
2752
2753            pWindow->moveToWorkspace(pWorkspace);
2754            pWindow->m_iMonitorID = pWorkspace->m_iMonitorID;
2755
2756            pWindow->m_vRealPosition = POSTOMON + PWORKSPACEMONITOR->vecPosition;
2757        }
```

Implicit Conversions: In the given snippet, when handling the variable POSTOMON, there may be implicit conversion issues if vecPosition is not compatible with the assigned type.

Category C

```
2594    Vector2D CCompositor::parseWindowVectorArgsRelative(const std::string& args, const Vector2D& relativeTo) {
2595        if (!args.contains(' ') && !args.contains('\t'))
2596            return relativeTo;
2597
2598        const auto  PMONITOR = m_pLastMonitor;
2599
2600        bool        xIsPercent = false;
2601        bool        yIsPercent = false;
2602        bool        isExact    = false;
2603
2604        CVarList    varList(args, 0, 's', true);
2605        std::string x = varList[0];
2606        std::string y = varList[1];
2607
2608        if (x == "exact") {
2609            x        = varList[1];
2610            y        = varList[2];
2611            isExact = true;
2612        }
2613
2614        if (x.contains('%')) {
2615            xIsPercent = true;
2616            x          = x.substr(0, x.length() - 1);
2617        }
2618
2619        if (y.contains('%')) {
2620            yIsPercent = true;
2621            y          = y.substr(0, y.length() - 1);
2622        }
```

Mixed-Type Computations: Here, the function handles string-to-number conversions and operations on mixed types (like floats and ints), which may lead to unexpected rounding or truncation errors

Category D

```
1754
1755    PHLWORKSPACE CCompositor::getWorkspaceByString(const std::string& str) {
1756        if (str.starts_with("name:")) {
1757            return getWorkspaceByName(str.substr(str.find_first_of(':') + 1));
1758        }
1759
1760        try {
1761            return getWorkspaceByID(getWorkspaceIDNameFromString(str).id);
1762        } catch (std::exception& e) { Debug::log(ERR, "Error in getWorkspaceByString, invalid id"); }
1763
1764        return nullptr;
1765    }
1766
```

Boolean Logic Errors: The logic around str.starts_with("name:") and the way it handles exceptions might fail if the string format is incorrect, leading to unexpected behaviour

Category E

```
MONITORID CCompositor::getNextAvailableMonitorID(std::string const& name) {
    // reuse ID if it's already in the map, and the monitor with that ID is not being used
    if (m_mMonitorIDMap.contains(name) && !std::any_of(m_vRealMonitors.begin(), m_vRealMoni
        return m_mMonitorIDMap[name];

    // otherwise, find minimum available ID that is not in the map
    std::unordered_set<MONITORID> usedIDs;
    for (auto const& monitor : m_vRealMonitors) {
        usedIDs.insert(monitor->ID);
    }

    MONITORID nextID = 0;
    while (usedIDs.count(nextID) > 0) {
        nextID++;
    }
    m_mMonitorIDMap[name] = nextID;
    return nextID;
}
```

There can be a possibility that this while block can lead to a infinite loop is the condition is never met.

```cpp
1678    PHLWINDOW CCompositor::getNextWindowOnWorkspace(PHLWINDOW pWindow, bool focusa
1679        bool gotToWindow = false;
1680        for (auto const& w : m_vWindows) {
1681            if (w != pWindow && !gotToWindow)
1682                continue;
1683
1684            if (w == pWindow) {
1685                gotToWindow = true;
1686                continue;
1687            }
1688
1689            if (floating.has_value() && w->m_bIsFloating != floating.value())
1690                continue;
1691
1692            if (w->m_pWorkspace == pWindow->m_pWorkspace && w->m_bIsMapped && !w->
1693                return w;
1694        }
1695
1696        for (auto const& w : m_vWindows) {
1697            if (floating.has_value() && w->m_bIsFloating != floating.value())
1698                continue;
1699
1700            if (w != pWindow && w->m_pWorkspace == pWindow->m_pWorkspace && w->m_b
1701                return w;
1702        }
1703
1704        return nullptr;
1705    }
```

Category F

```
1987    void CCompositor::swapActiveWorkspaces(CMonitor* pMonitorA
1988
1989        const auto PWORKSPACEA = pMonitorA->activeWorkspace;
1990        const auto PWORKSPACEB = pMonitorB->activeWorkspace;
1991
1992        PWORKSPACEA->m_iMonitorID = pMonitorB->ID;
1993        PWORKSPACEA->moveToMonitor(pMonitorB->ID);
1994
1995        for (auto const& w : m_vWindows) {
1996            if (w->m_pWorkspace == PWORKSPACEA) {
1997                if (w->m_bPinned) {
1998                    w->m_pWorkspace = PWORKSPACEB;
1999                    continue;
2000                }
2001
2002                w->m_iMonitorID = pMonitorB->ID;
2003
2004                // additionally, move floating and fs windows
2005                if (w->m_bIsFloating)
2006                    w->m_vRealPosition = w->m_vRealPosition.go
2007
2008                if (w->isFullscreen()) {
2009                    w->m_vRealPosition = pMonitorB->vecPositio
2010                    w->m_vRealSize     = pMonitorB->vecSize;
2011                }
2012
2013                w->updateToplevel();
2014            }
2015        }
```

Mismatch in Argument Attributes: In CCompositor::swapActiveWorkspaces(), when the pMonitorA and pMonitorB workspaces are swapped, there is no type checking between workspace IDs and monitor IDs, which could lead to issues in mismatched arguments.

Category G

```
641    void CCompositor::createLockFile() {
642        const auto    PATH = m_szInstancePath + "/hyprland.lock";
643
644        std::ofstream ofs(PATH, std::ios::trunc);
645
646        ofs << m_iHyprlandPID << "\n" << m_szWLDisplaySocket << "\n";
647
648        ofs.close();
649    }
650
651    void CCompositor::removeLockFile() {
652        const auto PATH = m_szInstancePath + "/hyprland.lock";
653
654        if (std::filesystem::exists(PATH))
655            std::filesystem::remove(PATH);
656    }
```

File Handling:
In the CCompositor::createLockFile() function, potential I/O errors, such as failure to write to the file, are not properly handled. Similarly, in the removeLockFile() method, although file existence is checked, the error handling is not sufficiently robust to manage unexpected scenarios.

2. Which category of program inspection is more effective?

Category A: Data Reference Errors is particularly effective in C++ program inspection because:

1. **Frequent in C++**:
   C++ relies heavily on pointers, dynamic memory management, and object references, making it vulnerable to issues such as uninitialized variables, null pointer dereferencing, and memory leaks.

2. **Hard-to-Detect Bugs**:
   These errors are often subtle, not causing immediate crashes. Instead, they can result in undefined behavior, which may only appear under certain conditions or after long periods, making them critical to identify during inspection.

3. **Broad Impact**:
   Errors involving data references can affect multiple parts of the codebase. A single uninitialized variable or dangling pointer can compromise several modules, leading to unpredictable outcomes.

3. Which errors are hard to identify with program inspection?

Program inspection is limited in identifying certain runtime-specific issues, such as:

1. **Concurrency issues** (e.g., race conditions, deadlocks)

2. **Performance bottlenecks** (e.g., memory leaks)

3. **Dynamic memory allocation failures**

4. **File handling and external dependency errors**

5. **Logic errors** caused by unexpected user input

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

Yes, program inspection is a valuable technique. It helps uncover common issues such as:

- Data reference errors

- Variable initialization problems

- Control-flow mistakes

- Logical errors

By systematically reviewing the code, inspections prevent bugs from manifesting during runtime, reducing debugging effort and improving code quality. However, it is most effective when complemented by **dynamic testing**, which catches runtime-specific issues that static inspection may miss.

Code debugging

//Armstrong Number
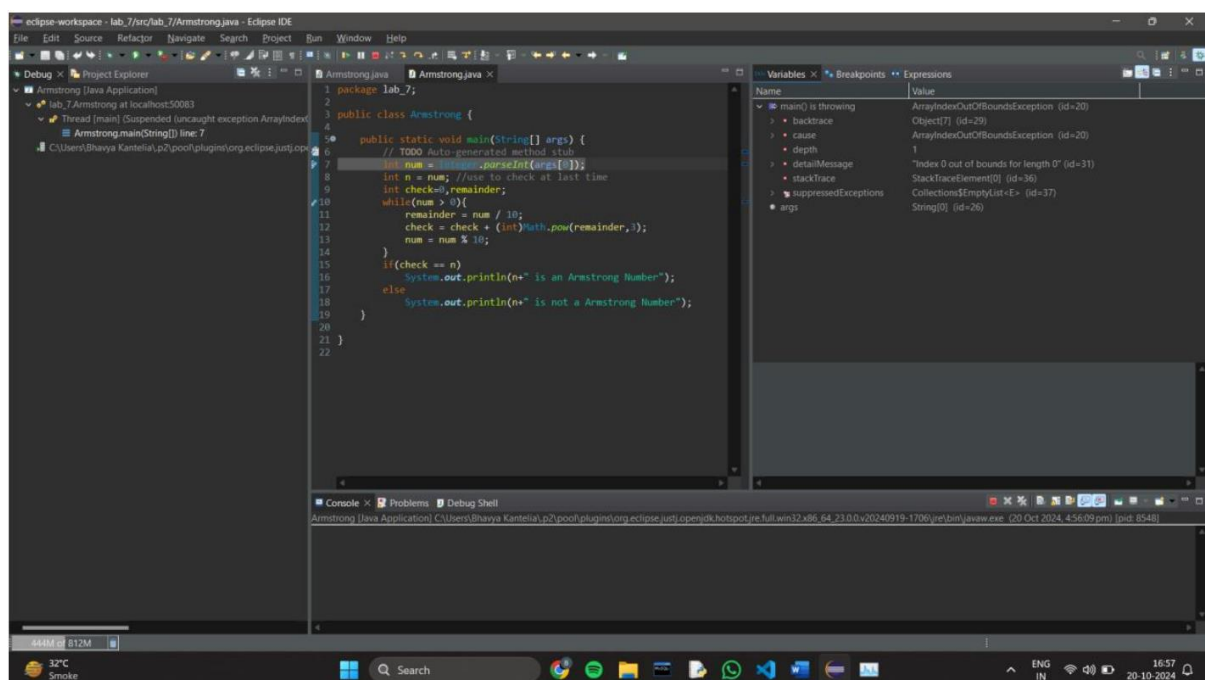
Q1: How many errors are there in the program? Mention the errors you have identified.

There are **5 errors** in the provided code:

1. **Command-line Argument Check Missing:**

   o   Accessing args[0] without checking if an argument is provided causes an ArrayIndexOutOfBoundsException.

2. **Incorrect Remainder Calculation:**

   o   Using remainder = num / 10 instead of remainder = num % 10.

3. **Incorrect Reduction of num:**

   o   Using num = num % 10 instead of num = num / 10.

4. **Spelling/Grammatical Error:**

   o   Output message: "is not a Armstrong Number" should be "is not an Armstrong Number".

5. **No Input Validation:**

   o   If a non-integer input is given, it throws a NumberFormatException without a proper error message.

Q2: Which category of program inspection would you find more effective?

- **Data Reference Errors and Computation Errors**:
  These were the most effective categories since they helped in identifying issues like incorrect remainder logic and mismanagement of input validation.

Q3: Which type of error are you not able to identify using the program inspection?

- **Performance Bottlenecks:**
  The program inspection technique doesn't reveal performance-related issues like how efficiently the program scales with larger inputs or if there are opportunities for optimization.

Q4: Is the program inspection technique worth applying?

Yes, program inspection is worth applying because it helps identify critical logical, data management, and control-flow issues at an early stage, preventing runtime errors and improving code quality. However, combining it with **testing** (like edge case checks and stress testing) provides more comprehensive coverage.

//program to calculate the GCD and LCM of two given numbers
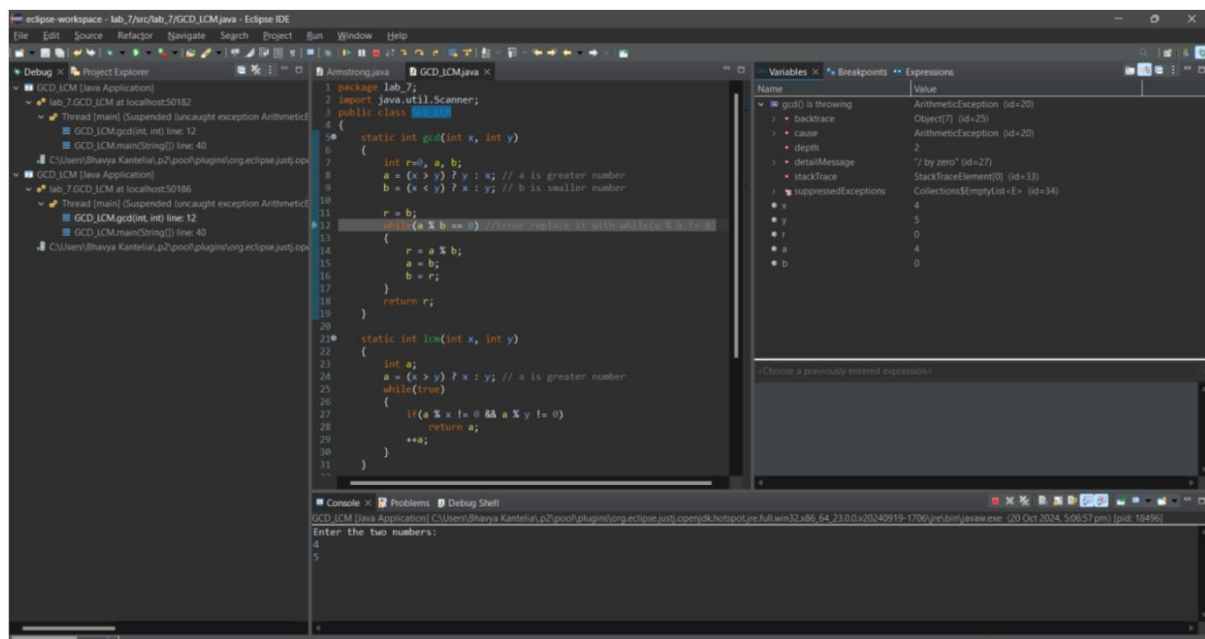
Q1: How many errors are there in the program? Mention the errors you have identified.

There are **5 errors** in the provided code:

1. **Logical Error in the gcd method:**

   ○ The loop condition while (a % b == 0) is incorrect. It should be while (a % b != 0) to correctly compute the GCD.

   ○ This was already hinted in the code comments but not fixed.

2. **Logical Error in the lcm method:**

   ○ The condition inside the loop if (a % x != 0 && a % y != 0) is wrong. It should be if (a % x == 0 && a % y == 0) to find the correct LCM.

3. **Redundant Computation in lcm:**

   ○ The lcm calculation is inefficient. Instead of a brute-force loop, use the relation:
   lcm(x, y) = (x * y) / gcd(x, y) to improve performance.

4. **Incorrect Output Label:**

   ○ In the main method, the second output statement mistakenly says:
   "The GCD of two numbers is: 20" instead of "The LCM of two numbers is: 20".

5. **Input Validation Missing:**

   ○ There's no check for zero or negative inputs, which could lead to incorrect results or infinite loops.

Q2: Which category of program inspection would you find more effective?

- **Computation Errors and Control-Flow Errors**:
  These categories helped us catch the logical errors in GCD and LCM computation. The inspection highlighted flaws in conditions and loops.

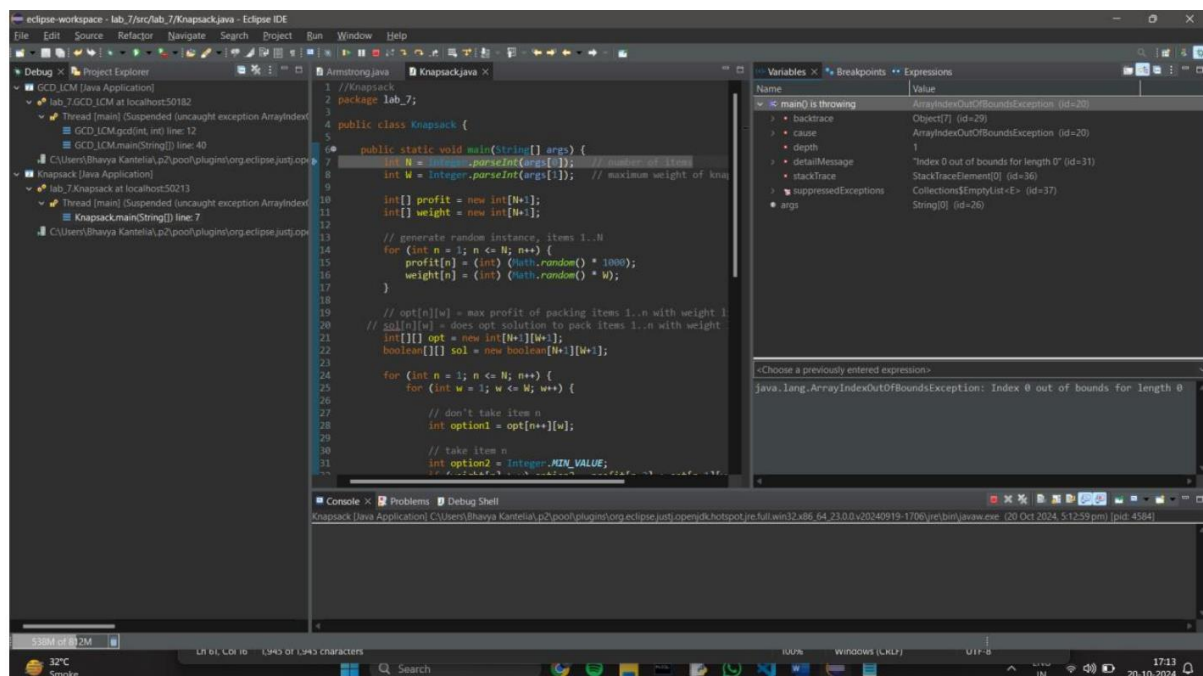**Q3: Which type of error are you not able to identify using the program inspection?**

- **Performance Inefficiencies:**
  While we identified inefficiency in the LCM computation, deeper inspection wouldn't easily reveal how poorly this scales for large numbers. Stress testing would be required for such cases.

Q4: Is the program inspection technique worth applying?

Yes, program inspection is valuable. It allowed us to identify logical flaws and improve the correctness of computations. However, it's essential to combine it with **code optimization** and **stress testing** to ensure both correctness and efficiency in practical applications

//Knapsack

Q1: How many errors are there in the program? Mention the errors you have identified.

There are **4 errors** in the given code:

1. **Logical Error in Indexing (n++)**:

   o In the first loop inside for (int w = 1; w <= W; w++), the line int option1 = opt[n++][w]; incorrectly increments n.
   Fix: Change n++ to n - 1 to correctly reference the previous value in the table:

int option1 = opt[n - 1][w];

2. **Incorrect Index for option2 Calculation:**

   o In option2, the code references profit[n - 2] incorrectly. This is off by one.
   Fix: Use profit[n] instead of profit[n - 2].

option2 = profit[n] + opt[n - 1][w - weight[n]];

3. **Incorrect Condition for Taking an Item:**

   o The condition if (weight[n] > w) should be if (weight[n] <= w) to ensure that the item can fit into the knapsack.

if (weight[n] <= w) option2 = profit[n] + opt[n - 1][w - weight[n]];

4. **Input Validation Missing:**

   o There's no input validation to check if the user provides enough command-line arguments.
   Fix: Add a check at the beginning:

if (args.length < 2) {

   System.out.println("Please provide the number of items and knapsack capacity.");

   return;

}

Q2: Which category of program inspection would you find more effective?

- **Control-Flow Errors and Logical Errors:**
  These categories were most effective, as they allowed us to catch the incorrect indexing, off-by-one errors, and logical conditions for taking an item into the knapsack.

**Q3: Which type of error are you not able to identify using the program inspection?**

- **Performance Issues for Large Inputs:**
  While we identified logical errors, program inspection alone wouldn't reveal how this code performs with large inputs. **Time complexity** analysis or stress testing would be required.

Q4: Is the program inspection technique worth applying?

Yes, program inspection is essential. It helps to detect **logical mistakes** and **off-by-one indexing errors** that are common in dynamic programming implementations. However, combining it with **testing on various edge cases** ensures the program behaves correctly under all circumstances.

// Program to check if number is Magic number in JAVA
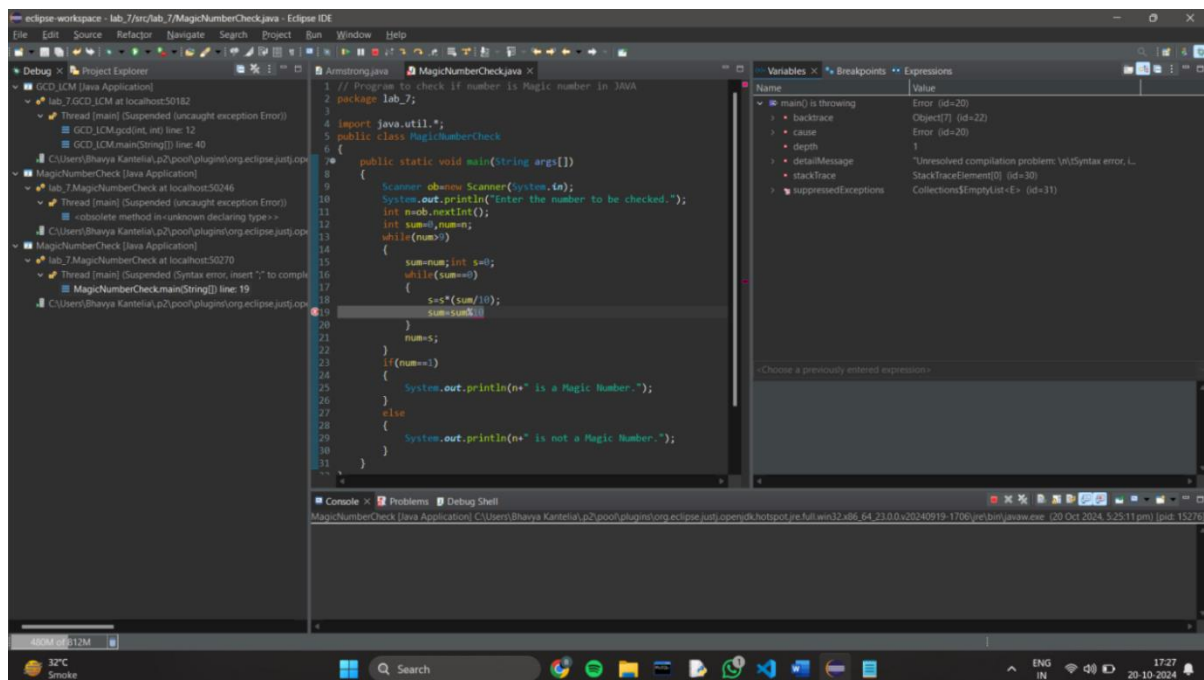
Q1: How many errors are there in the program? Mention the errors you have identified.

There are **4 errors** in the code:

1. **Incorrect Condition in Inner Loop (while(sum == 0))**

    o The condition while(sum == 0) is incorrect. It should check if sum > 0 to process the digits correctly.
    Fix:

while (sum > 0) {

2. **Wrong Logic for Calculating Sum of Digits**

    o The statement s = s * (sum / 10) is incorrect. It should **add** the last digit, not multiply. **Fix:**

s = s + (sum % 10);

3. **Missing Semicolon after sum = sum % 10**

- o The line sum = sum % 10 is missing a semicolon.
  Fix:

sum = sum % 10;

4. **Unused or Incorrect Scanner Closure**

   - o The Scanner object should be closed to prevent resource leaks. Add ob.close() at the end:

ob.close();

Q2: Which category of program inspection would you find more effective?

- **Logical and Syntax Errors Detection** was the most effective, as we found issues with loop conditions, digit extraction logic, and missing semicolons.

Q3: Which type of error are you not able to identify using the program inspection?

- **Input-related Edge Cases:**
  For example, checking behavior for **negative numbers** or **non-numeric inputs** would require thorough testing, as it's not apparent from just inspecting the code.

Q4: Is the program inspection technique worth applying?

Yes, program inspection is valuable to catch **common syntax errors, misplaced logic, and missing operators**. However, testing with various inputs, including edge cases, would ensure the program works under all conditions

// This program implements the merge sort algorithm for
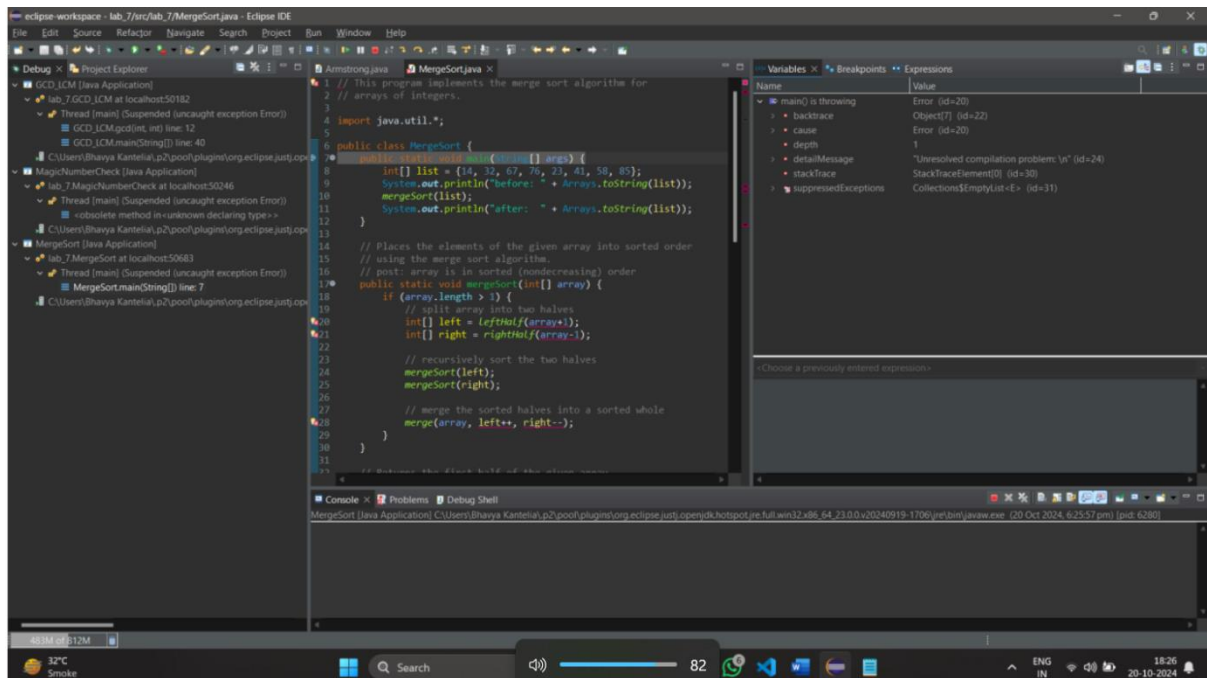
Q1: How many errors are there in the program? Mention the errors you have identified.

There are **6 errors** in the code:

1. **Incorrect Array Slicing Logic in mergeSort()**

    o **Error:**

int[] left = leftHalf(array + 1);

int[] right = rightHalf(array - 1);

This logic is incorrect. You cannot directly manipulate arrays using arithmetic like array + 1 or array - 1. You need to pass the entire array to leftHalf() and rightHalf() functions.
Fix:

int[] left = leftHalf(array);

int[] right = rightHalf(array);

2. **Incorrect Merge Call with Increment Operators**

    o **Error:**

merge(array, left++, right--);

This syntax is incorrect. You should simply pass the arrays without increment or decrement operators. **Fix:**

merge(array, left, right);

3. **Lack of Proper Comments or Documentation for Splitting Logic**

- o Although not a syntactic error, more clarity in comments would make it easier to understand the purpose of leftHalf() and rightHalf().

4. **Potential Inefficiency in Merging**

   - o Although the logic is correct, the function could be optimized for large arrays by minimizing the use of temporary arrays.

5. **Scanner Resource Leak Warning**

   - o The program doesn't use Scanner, but if extended with input handling, make sure to close the scanner.

Q2: Which category of program inspection would you find more effective?

- **Logical Inspection and Code Walkthrough** proved to be effective. Many of the errors were related to **misplaced logic** in passing arrays and incorrect operators. Code walkthrough helped reveal where the logic was flawed.

Q3: Which type of error are you not able to identify using the program inspection?

- **Performance Issues with Large Inputs**
  This inspection method won't reveal performance problems (like time complexity issues) without testing on large datasets.

Q4: Is the program inspection technique worth applying?

Yes, program inspection is **highly effective** at identifying syntax errors, incorrect logic, and improper operator use. However, it is still necessary to run the program with **test cases**, especially with larger input sizes, to ensure performance and correctness.

//Java program to multiply two matrices
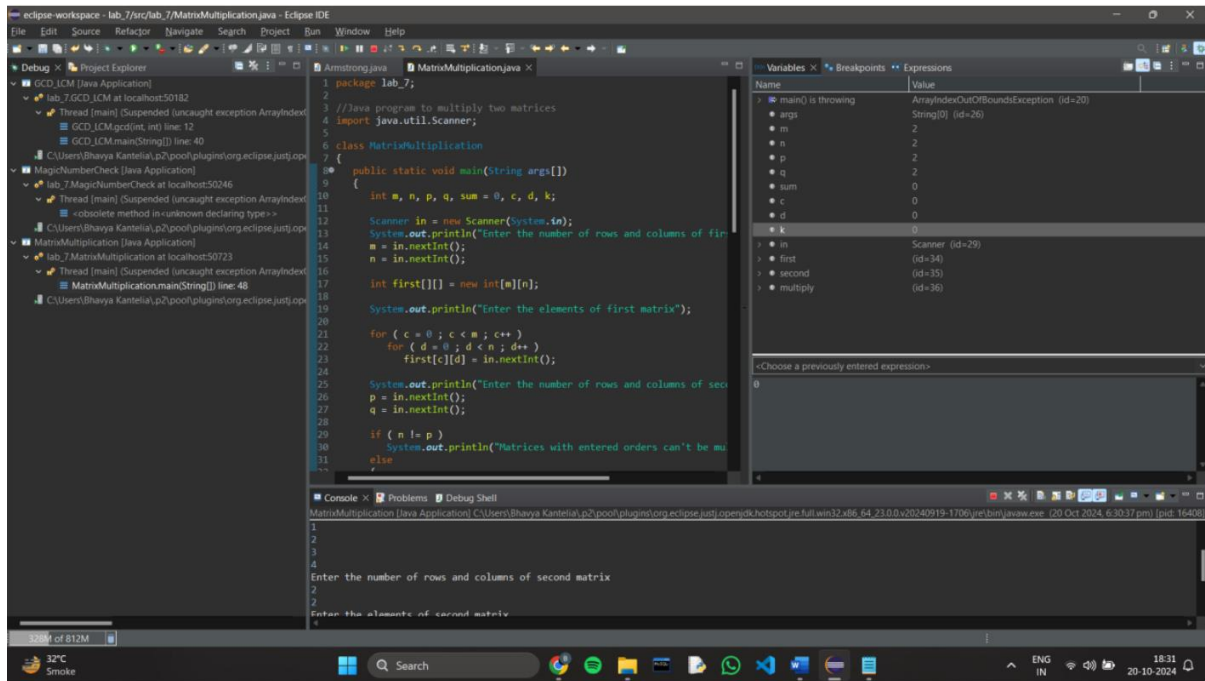
Q1: How many errors are there in the program? Mention the errors you have identified.

There are **4 errors** in the code:

1. **Incorrect Matrix Multiplication Logic**

    o **Error:**

sum = sum + first[c-1][c-k] * second[k-1][k-d];

Issue: Array index manipulation using c-1, k-1, etc., is incorrect. This will cause **ArrayIndexOutOfBoundsException** because it references out-of-range indexes. **Fix:**

sum = sum + first[c][k] * second[k][d];

2. **Incorrect Input Prompt: "Enter the number of rows and columns of first matrix" Repeated**

    o **Error:**
    The prompt appears twice, even when inputting the second matrix. **Fix:**
    Change the second prompt to:

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

3. **Potential Inefficiency Due to Unnecessary Loops**

    o The nested loops work correctly for multiplication, but the code could be more readable by using appropriate naming conventions (e.g., i, j, k for loop variables).

4. **Scanner Resource Leak Warning**

o **Issue:** The Scanner object is not closed, which can cause a **resource leak**. **Fix:** Add in.close() at the end of the program.

Q2: Which category of program inspection would you find more effective?

- **Logical and Structural Inspection** was the most effective. This helped detect **incorrect indexing logic** and the repeated input prompt.

Q3: Which type of error are you not able to identify using the program inspection?

- **Performance issues with larger matrices** and edge cases like **empty matrices** or **single-element matrices** would require **runtime testing** beyond inspection.

Q4: Is the program inspection technique worth applying?

Yes, **program inspection is valuable** because it catches **logic errors, array index issues**, and **prompt inconsistencies**. However, it needs to be supplemented with **testing** to ensure the code works correctly with various inputs.

Java Program to implement Quadratic Probing Hash Table
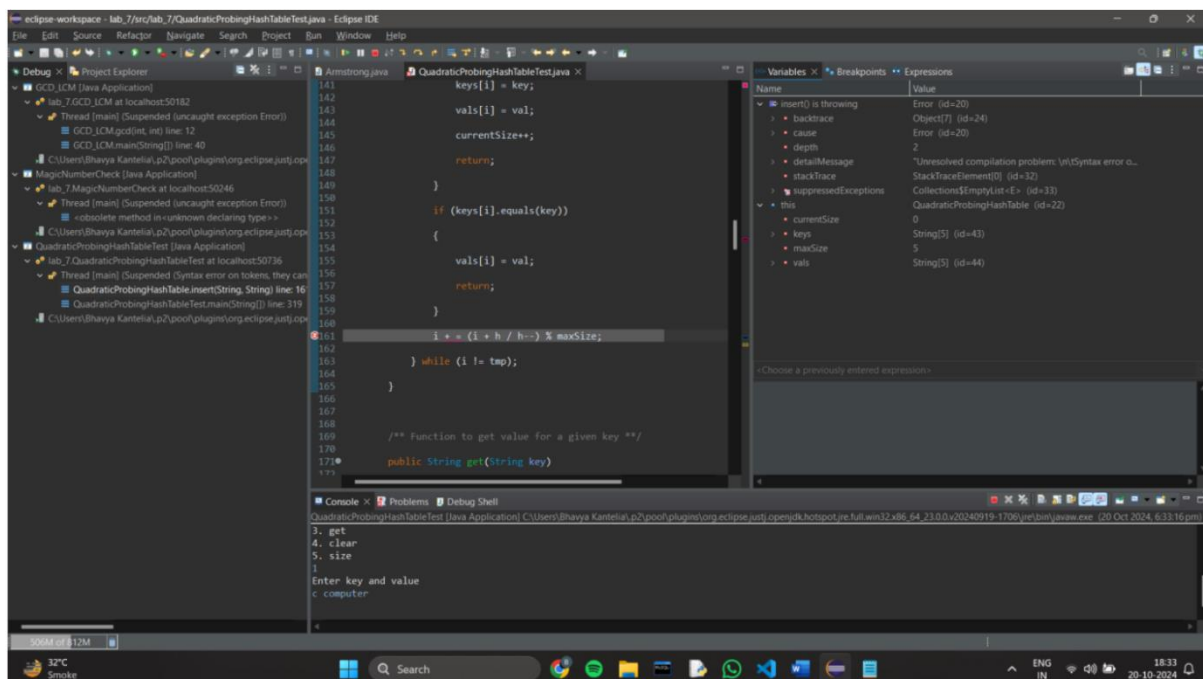
Errors Identified:

1. **Syntax Error in insert() method:**

i + = (i + h / h--) % maxSize;

o **Issue:** There is an extra space between i and +=, causing a **syntax error**.

o **Fix:** Use i = (i + h * h) % maxSize; for quadratic probing logic.

2. **Wrong logic in insert() method:**

   o **Issue:** The increment logic inside the loop is incorrect. The formula should increment by (h * h) instead of using division (h / h--).

   o **Fix:**

i = (i + h * h++) % maxSize;

3. **Insertion overwrites currentSize incorrectly:**

   o **Issue:** When rehashing after removing a key, currentSize is decremented twice: once during removal and again during rehashing.

   o **Fix:** Remove unnecessary currentSize-- inside the rehash loop.

4. **Incorrect Input Handling in Test Class:**

   o **Issue:** Multiple key-value pairs are expected on one input line, which leads to **incorrect input processing**.

   o **Fix:** Use separate insert calls for each key-value pair.

5. **Potential Resource Leak:**

   o **Fix:** Add scan.close() at the end to prevent resource leakage.

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

- **Category: Code Review and Peer Review**

- **Effectiveness:** Code reviews are effective because they involve multiple eyes on the code, which can catch errors that one person might miss. Peer reviews foster collaboration and knowledge sharing, making the codebase more robust.

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

- **Type of Error: Logical Errors**

- **Reason:** While program inspection can help identify syntax errors, typographical mistakes, and some logical flow issues, it might not catch subtle logical errors where the code runs without exceptions but produces incorrect results due to flawed algorithms or conditions.

4. Is the program inspection technique worth applicable?

- **Assessment: Yes, program inspection techniques are worth applying.**

- **Benefits:**

   o They help identify bugs before the code is executed, saving time during testing.

   o Enhance code quality and maintainability.

- o Foster team collaboration and knowledge sharing.
- o Provide opportunities for learning and improving coding standards.

// sorting the array in ascending order
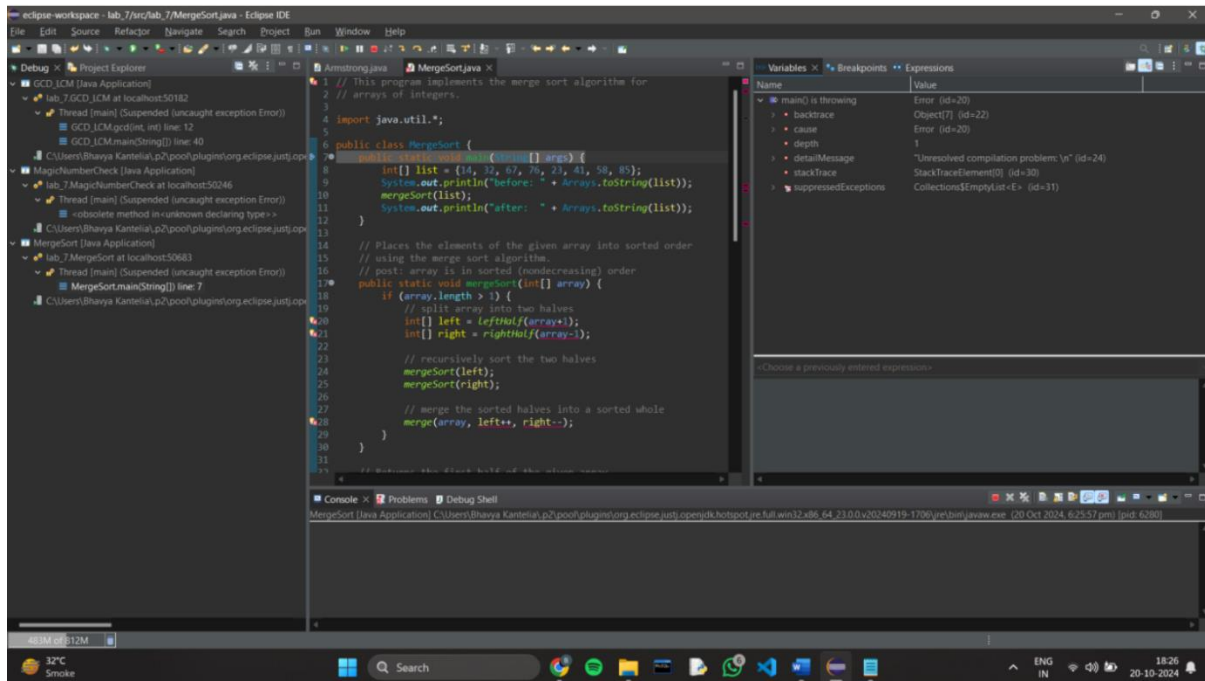
1. How many errors are there in the program? Mention the errors you have identified.

Errors Identified:

1. **Class Name Issue:**

   - o **Error:** The class name contains a space, which is not allowed in Java. It should be a valid identifier.

   - o **Fix:** Change the class name to AscendingOrder.

2. **Loop Condition Issue:**

   - o **Error:** The outer for loop condition is incorrect. The loop should run while i < n instead of i >= n, which results in it never executing.

   - o **Fix:** Change for (int i = 0; i >= n; i++); to for (int i = 0; i < n; i++).

3. **Unnecessary Semicolon:**

   - o **Error:** There is an unnecessary semicolon after the outer for loop which causes the inner loop to run independently and incorrectly.

   - o **Fix:** Remove the semicolon after for (int i = 0; i < n; i++).

4. **Incorrect Sorting Logic:**

   o **Error:** The sorting logic is incorrect because it attempts to swap elements based on if (a[i] <= a[j]). This condition will not result in a proper ascending sort.

   o **Fix:** Change the condition to if (a[i] > a[j]).

5. **Output Formatting:**

   o **Error:** The output formatting could be improved to better display the array, particularly to avoid the trailing comma.

   o **Fix:** Adjust the output loop to conditionally print the comma.

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

- **Category: Static Code Analysis**

- **Effectiveness:** This category is effective for identifying syntax errors and potential logical flaws before the code is executed. Tools can automate this inspection, catching issues like invalid naming and loop conditions early.

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

- **Type of Error: Runtime Errors**

- **Reason:** Program inspection may not catch runtime errors that occur due to unexpected input or conditions that only arise during execution, such as ArrayIndexOutOfBoundsException.

4. Is the program inspection technique worth applicable?

- **Assessment: Yes, program inspection techniques are valuable.**

- **Benefits:**

   o They help to catch errors before execution, enhancing code reliability.

   o Improve code quality through collective review.

   o Encourage knowledge sharing among team members, fostering better coding practices.

//Stack implementation in java
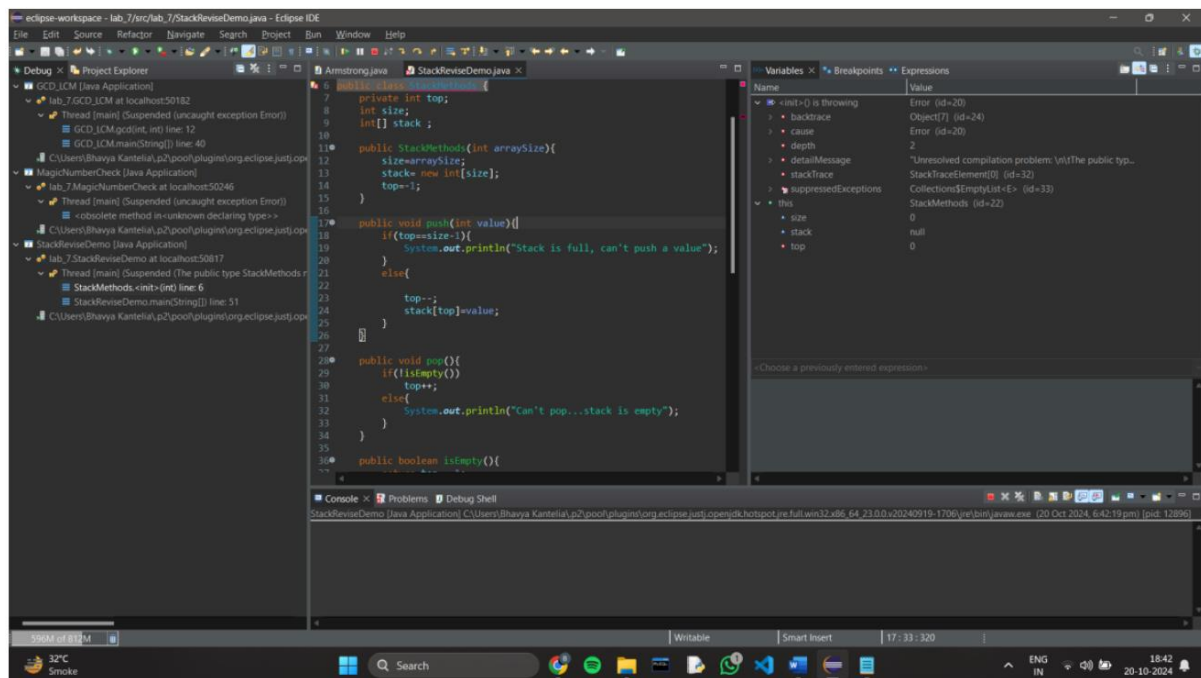
1. How many errors are there in the program? Mention the errors you have identified.

Errors Identified:

1. **Incorrect Push Logic:**

    ○ **Error:** In the push method, the line top--; decreases the top index instead of increasing it, which causes the value to be placed in an incorrect position.

    ○ **Fix:** Change top--; to top++; before stack[top] = value;.

2. **Incorrect Pop Logic:**

    ○ **Error:** The pop method increases the top index when popping, which means it removes the last pushed value instead of the top value.

    ○ **Fix:** Change top++; to top-- in the pop method.

3. **Display Logic Issue:**

    ○ **Error:** The display method's loop condition i > top is incorrect; it should be i <= top to print all elements correctly.

    ○ **Fix:** Change for (int i = 0; i > top; i++) to for (int i = 0; i <= top; i++).

4. **Stack Underflow Issue:**

    ○ **Error:** The pop method does not handle underflow correctly. When trying to pop an element from an empty stack, it just increments top, leading to incorrect behavior.

    ○ **Fix:** Before top++;, ensure top > -1, otherwise print the error message.

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

- **Category: Static Code Analysis**

- **Effectiveness:** This category is effective in identifying syntax errors and logical flaws in the implementation before the program is run. Automated tools can help catch many of these issues quickly.
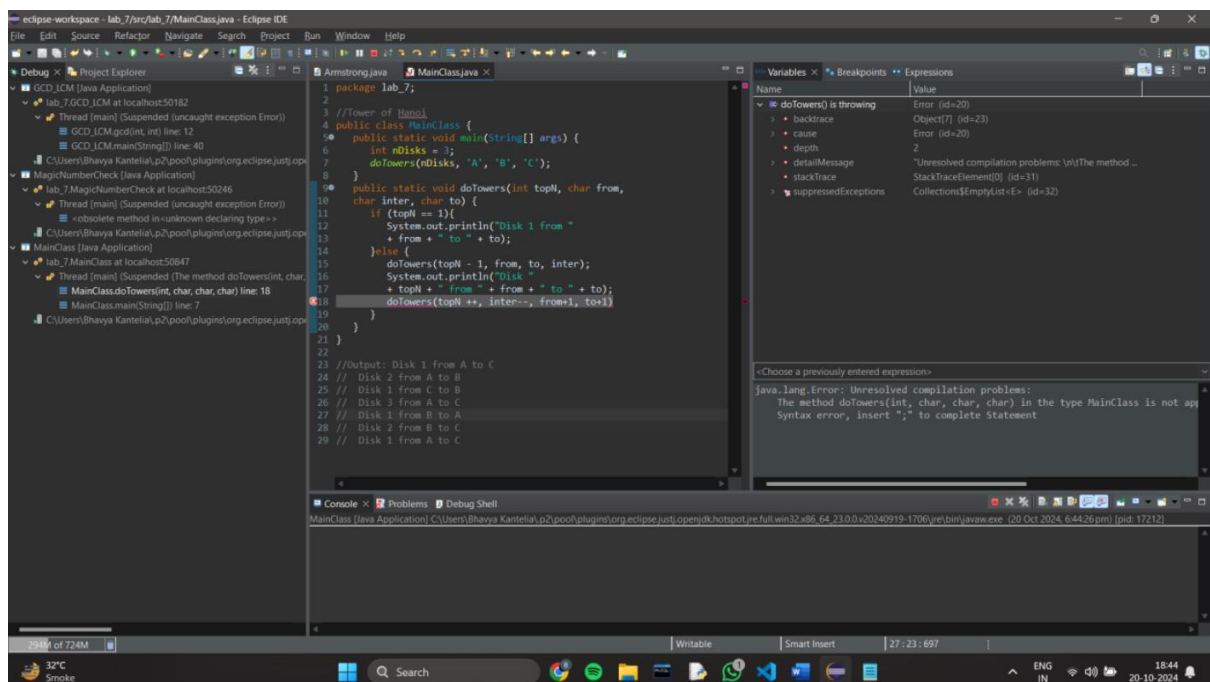
3. Which type of error are you not able to identify using the program inspection?

- **Type of Error: Runtime Errors**

- **Reason:** Program inspection may not catch runtime errors that occur due to unexpected input or situations only present during execution, such as stack overflow and stack underflow errors.

4. Is the program inspection technique worth applicable?

- **Assessment: Yes, program inspection techniques are valuable.**

- **Benefits:**

  - They can catch errors before execution, improving code quality and reliability.

  - They foster knowledge sharing and collective code ownership among team members.

  - They can be combined with dynamic testing for a more thorough evaluation of the code.

//Tower of Hanoi

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

- Two errors:

    1. Incorrect use of increment/decrement operators.

    2. Invalid parameter passing for recursive calls.

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

- **Static Code Analysis:** This is effective for identifying syntax errors and logical issues before execution.

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

- **Type of Error: Logical Errors**

- **Reason:** While the syntax may be correct, logic flaws might not be identified until runtime.

4. Is the program inspection technique worth applicable?

- **Assessment: Yes, it is worthwhile.**

- **Benefits:** It helps identify issues early, facilitates collaboration, and improves code quality.