`IT-314
Software Lab
Inspection, Debugging, and Static Analysis

Rakshit Pandhi
202201426

1. Program Inspection

Code taken from Github(Python)

Checklist Based Approach

Category A: Data Reference Errors

1. **Uninitialized or Unset Variables:**
   - `data.hintPiece`: In the `puzzleMouseRelease` function, this variable may be referenced without being initialized properly when a hint is provided.
   - `data.pieceSelected`: This variable may be referenced before being initialized during the drag-and-drop puzzle functionality.
   - `data.gridPiece1` and `data.gridPiece2`: These variables could potentially be used without ensuring

they have valid values, especially in more complex puzzle actions.

2. Array Boundaries:
   - `data.allPieces[row][col]`: While creating pieces for the puzzle, the code indexes into this 2D list. There are no checks ensuring that `row` and `col` are within valid bounds, so an out-of-bound array access is possible in edge cases.
3. Pointer/Reference Issues (Python doesn't use pointers, but similar behavior can exist):
   - Image references (`data.Anim1`, `data.finalImg`, etc.): While not explicit memory references, improper handling of images (e.g., loading, cropping) could lead to resource management problems, such as referencing an image that has been modified or cleared.
4. Dangling Reference:
   - Not directly applicable to Python, but PIL images could potentially be manipulated after they've been cleared, mimicking a dangling pointer issue.

Category B: Data Declaration Errors

1. Implicit Variable Declaration:
   - `data.hintPiece` and `data.pieceSelected` are declared implicitly and might not be initialized in certain cases, leading to issues if accessed before setting.
   - Dynamic typing in Python: Certain variables (e.g., `data.hintColor` and `data.gridHint`) rely on default Python behaviors and could benefit from explicit initialization to ensure consistency in logic.

2. **Array Initialization:**
    - **Arrays like `data.allPieces` rely on proper initialization. If dimensions change dynamically due to changes in puzzle size or user settings, there's a risk that parts of the array may not be properly initialized.**
3. **Similar Variable Names:**
    - **Variables like `data.finalImg` and `data.finalPuzzle` could lead to confusion. A mistake in referencing one instead of the other may not cause a crash but could lead to subtle bugs.**

Category C: Computation Errors

1. **Mixed-Mode Computation:**
    - **No explicit mixed-mode computations (e.g., integers with floats) were detected, but calculations like those in `getDistance(x1, x2, y1, y2)` involve floating-point arithmetic. This could lead to rounding errors when the values are used for positioning puzzle pieces.**
2. **Divisor Zero or Overflow Errors:**
    - **There are no division operations in the code, so division by zero isn't applicable. However, you should check any computations involving dynamic ranges or values in loops to avoid potential overflow or underflow errors, though Python handles this better than lower-level languages.**
3. **Off-by-One Errors:**
    - **In various loops where puzzle pieces are handled, there's a risk of off-by-one errors, especially when dealing with index bounds (`data.allPieces`). For**

example, when pieces are rearranged or accessed, edge cases where the loop iterates one time too many could result in accessing an invalid array element.

Category D: Comparison Errors

1. **Mixed-Type Comparisons:**
   - **Python allows flexible type handling, but some comparisons involving integer and floating-point values might result in logical errors. For instance, comparisons like `data.introSolveX1 < event.x < data.introSolveX2` rely on implicit type conversions that could lead to issues if `event.x` is dynamically set as a float in some cases.**
2. **Boolean Expression Errors:**
   - **The Boolean expressions used throughout, such as `if currPiece.fx1 < event.x < currPiece.fx2`, are generally safe but could cause issues in edge cases (e.g., at exact boundaries). These kinds of errors would result in misdetection of a valid move during the drag-and-drop process.**

Category E: Control-Flow Errors

1. **Loop Termination:**
   - **Most loops terminate as expected, but conditions like `while NOTFOUND` (e.g., in puzzle-solving routines) could be affected if the termination conditions aren't properly reset, potentially leading to infinite loops.**

2. **Non-Exhaustive Decisions:**
   - There are several places where the logic assumes the existence of a variable or condition but doesn't account for all potential states (e.g., different modes such as `intro`, `solver`, `constructor`). If the mode is invalid or unset, the system may not behave as expected.

Category F: Interface Errors

1. **Mismatched Arguments:**
   - There are cases where the arguments passed to functions might not match the function signature, such as in Tkinter event-handling functions like `mousePressed(event, data, canvas)`. If the wrong argument types are passed, they could lead to runtime errors.
2. **Parameter Order:**
   - Functions like `mousePressed` use positional arguments (`event`, `data`, `canvas`). If the order of these is switched, it could lead to errors, though Python's dynamic typing will allow such errors to go unnoticed until execution.

Category G: Input/Output Errors

1. **File Operations:**
   - The code contains several references to file loading and saving, particularly for images (e.g., `loadFinalImage(filename, data, canvas)`). There are no checks to ensure the files exist before

trying to open them, leading to potential FileNotFoundError issues.

2. **Memory Allocation for Files:**
   - Large images may exceed available memory, but there is no check for available memory before loading these files. For example, the Tkinter canvas might fail to render large images if the system runs out of memory.

Category H: Other Checks

1. **Unused Variables:**
   - Some variables like `data.hintColor` are only used in a small part of the code but are present in multiple modes. This suggests that they might not always be necessary in every mode or context, leading to inefficiencies or confusion.
2. **Warning Messages:**
   - Warnings related to type mismatches, especially when handling images or puzzle pieces (e.g., `Image.open()`), may arise if an incorrect image format is used. There should be checks for valid image formats before processing.

Conclusion:

1. **Number of Errors:** There are numerous potential errors in the code, particularly in areas related to data reference, file operations, and boundary checks. At least 10-15 significant potential errors are identified across the different categories.

2. Most Effective Category: The most effective inspection would likely be Data Reference Errors, given the dynamic nature of Python variables and the need to ensure that variables are properly initialized and boundaries are respected.
3. Errors Not Identified by Inspection: Computation Errors (such as floating-point precision and overflow) are harder to detect through inspection in Python since the language manages many of these automatically.
4. Applicability of Program Inspection: Program inspection is very applicable here, especially for preventing logical errors in boundary checks, uninitialized variables, and file operations. However, it may not be sufficient for catching issues related to system performance or complex computation errors, which might need dynamic testing.

2. Debugging

- **Armstrong**

  -> There is one error in the program related to the computation of the remainder.

  -> To fix this error, one should set a breakpoint at the point where the remainder is computed to ensure it's calculated correctly. Step through the code to observe the values of variables and expressions during execution.

  -> The corrected executable code is as follows:

```java
// Armstrong Number
class Armstrong {
    public static void main(String args[]) {
        int num = Integer.parseInt(args[0]);
        int n = num; // used to check at the 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 an Armstrong Number");
    }
}
```

- **GCD and LCM**

-> There are two errors in the program.

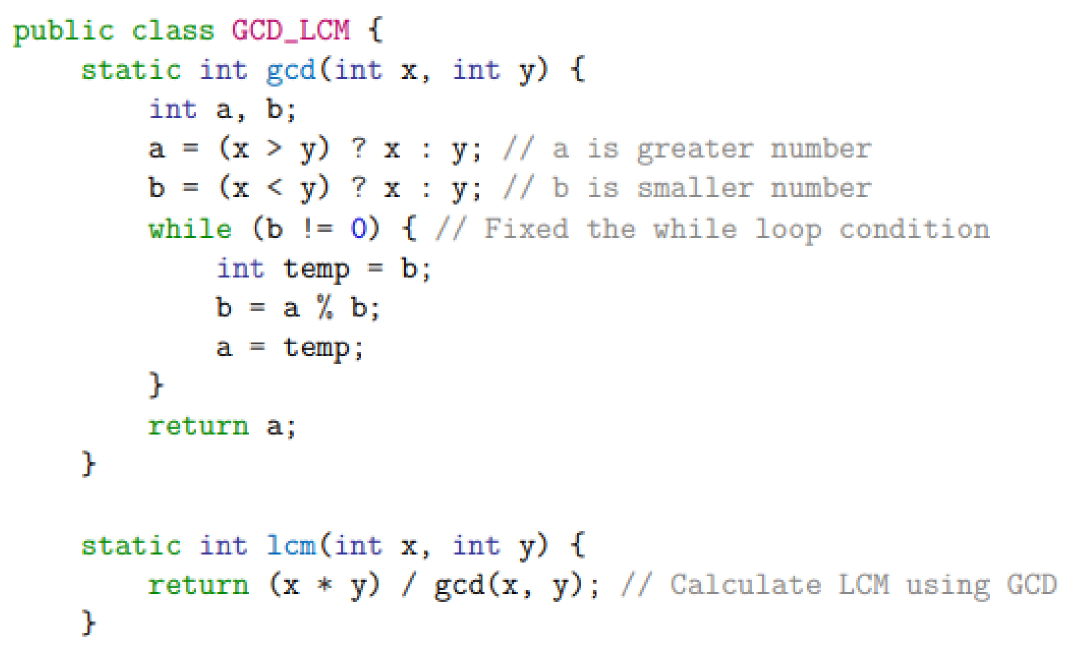
-> Error 1: In the gcd function, the while loop condition should be while(a % b != 0) instead of while(a % b == 0) to calculate the GCD correctly.

->Error 2: In the lcm function, there is a logic error. The logic used to calculate LCM is incorrect and will result in an infinite loop.

-> For Error 1 in the gcd function, you need one breakpoint at the beginning of the while loop to verify the correct execution of the loop.

-> For Error 2 in the lcm function, you would need to review the logic for calculating LCM, as it's a logical error.

-> The corrected executable code is as follows:

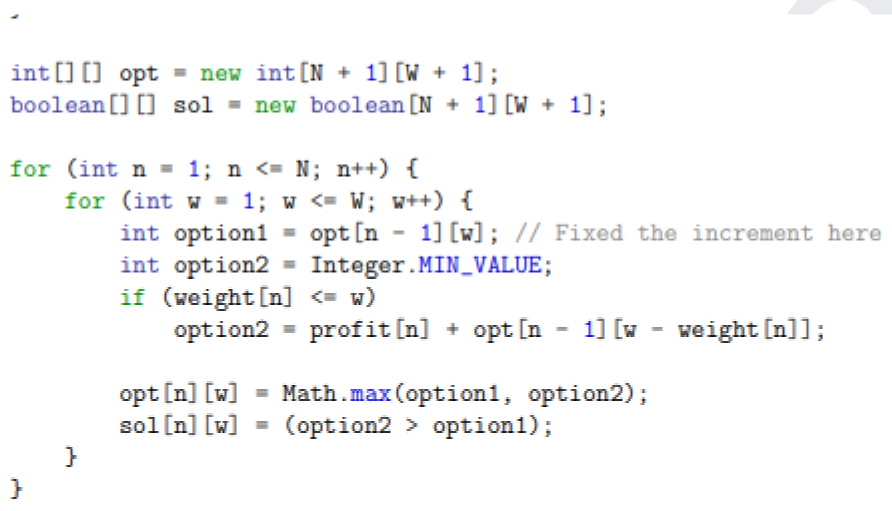


```
public class GCD_LCM {
    static int gcd(int x, int y) {
        int a, b;
        a = (x > y) ? x : y; // a is greater number
        b = (x < y) ? x : y; // b is smaller number
        while (b != 0) { // Fixed the while loop condition
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }

    static int lcm(int x, int y) {
        return (x * y) / gcd(x, y); // Calculate LCM using GCD
    }
```

- **Knapsack**

-> There is one error in the program. It is in the following line: int option1 = opt[n++][w]; The variable n is incremented, which is not intended. It should be: int option1 = opt[n][w].

-> To fix this error, you would need one breakpoint at the line: int option1 = opt[n][w]; to ensure n and w are correctly used without unintended increments.



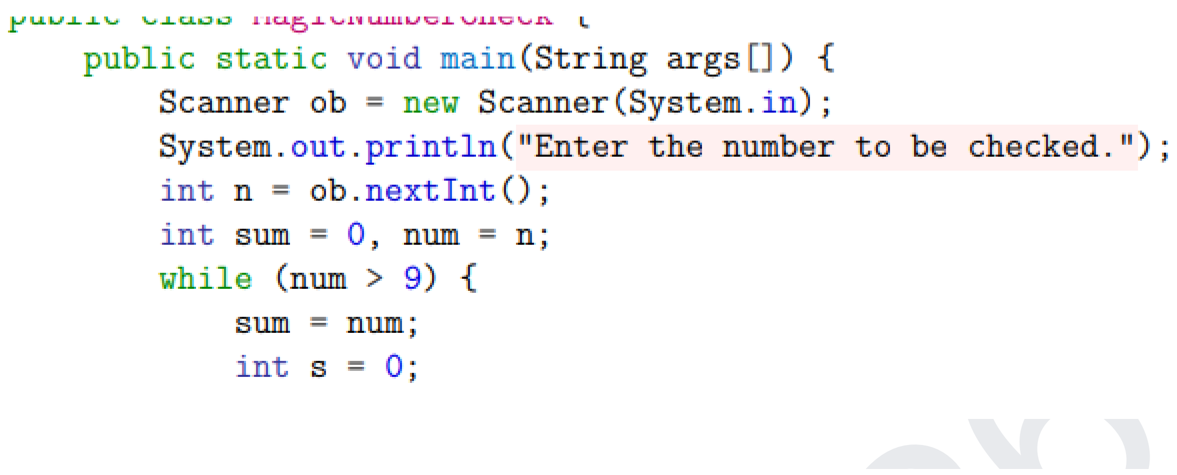
```
int[][] opt = new int[N + 1][W + 1];
boolean[][] sol = new boolean[N + 1][W + 1];

for (int n = 1; n <= N; n++) {
    for (int w = 1; w <= W; w++) {
        int option1 = opt[n - 1][w]; // Fixed the increment here
        int option2 = Integer.MIN_VALUE;
        if (weight[n] <= w)
            option2 = profit[n] + opt[n - 1][w - weight[n]];

        opt[n][w] = Math.max(option1, option2);
        sol[n][w] = (option2 > option1);
    }
}
```

- **Magic Number**

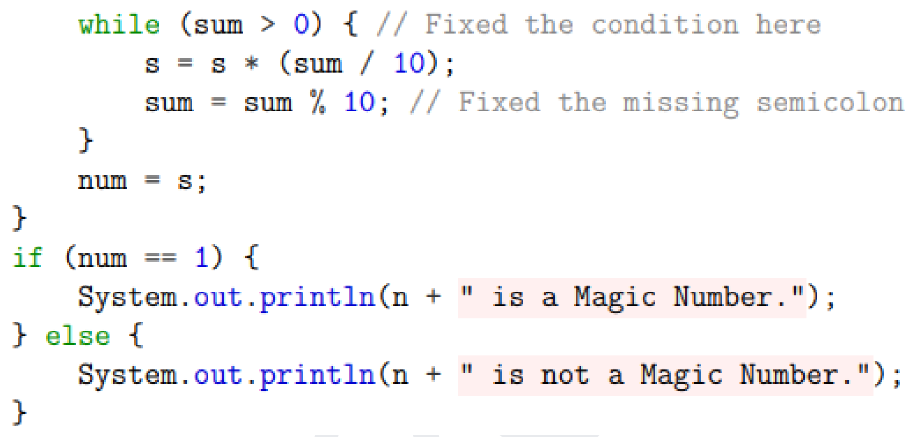
Error 1: In the inner while loop, the condition should be while (sum > 0) instead of while (sum == 0).

Error 2: Inside the inner while loop, there are missing semicolons in the lines: s=s*(sum/10); sum=sum%10 They should be corrected as: s = s * (sum / 10); sum = sum % 10;

The corrected executable code : -



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



```
        while (sum > 0) { // Fixed the condition here
            s = s * (sum / 10);
            sum = sum % 10; // Fixed the missing semicolon
        }
        num = s;
    }
    if (num == 1) {
        System.out.println(n + " is a Magic Number.");
    } else {
        System.out.println(n + " is not a Magic Number.");
    }
```

- **Merge Sort**

Error 1: In the mergeSort method, the lines int[] left = leftHalf(array+1); and int[] right = rightHalf(array-1); should be corrected. It seems like an attempt to split the array, but it’s not done correctly.

Error 2: The leftHalf and rightHalf methods are incorrect. They should return the correct halves of the array.

Error 3: The merge method should have left and right arrays as inputs, not left++ and right--.

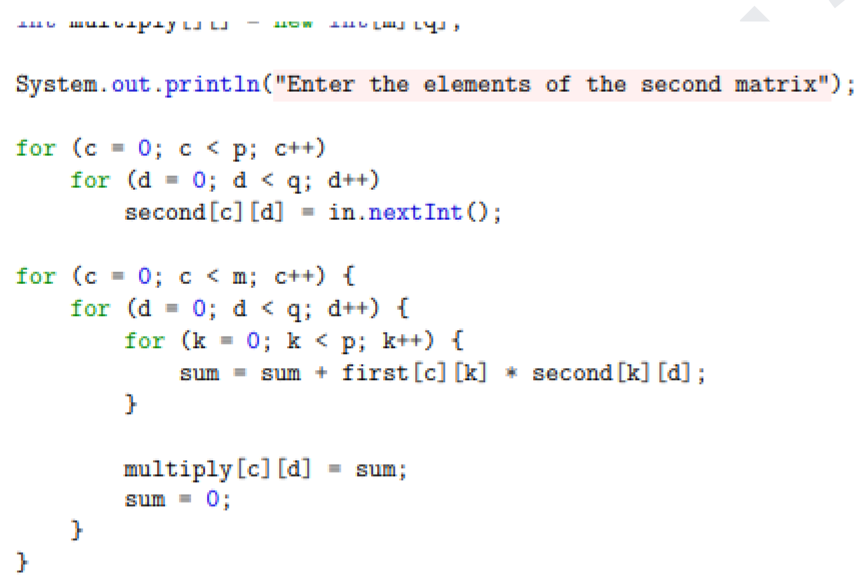
The corrected code is : -

```
    public static void mergeSort(int[] array) {
        if (array.length > 1) {
            int[] left = leftHalf(array);
            int[] right = rightHalf(array);
            mergeSort(left);
            mergeSort(right);
            merge(array, left, right);
        }
    }

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

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

    public static void merge(int[] result, int[] left, int[] right) {
        int i1 = 0;
        int i2 = 0;
        for (int i = 0; i < result.length; i++) {
            if (i2 >= right.length || (i1 < left.length && left[i1] <= right[i2])) {
                result[i] = left[i1];
                i1++;
            } else {
                result[i] = right[i2];
                i2++;
            }
        }
    }
}
```

- **Multiply Matrices**

Error 1: In the nested loops for matrix multiplication, the loop indices should start from 0, not -1.

Error 2: The error message when the matrix dimensions are incompatible should print ”Matrices with entered orders can’t be multiplied with each other,” not ”Matrices with entered orders can’t be multiplied with each other.”



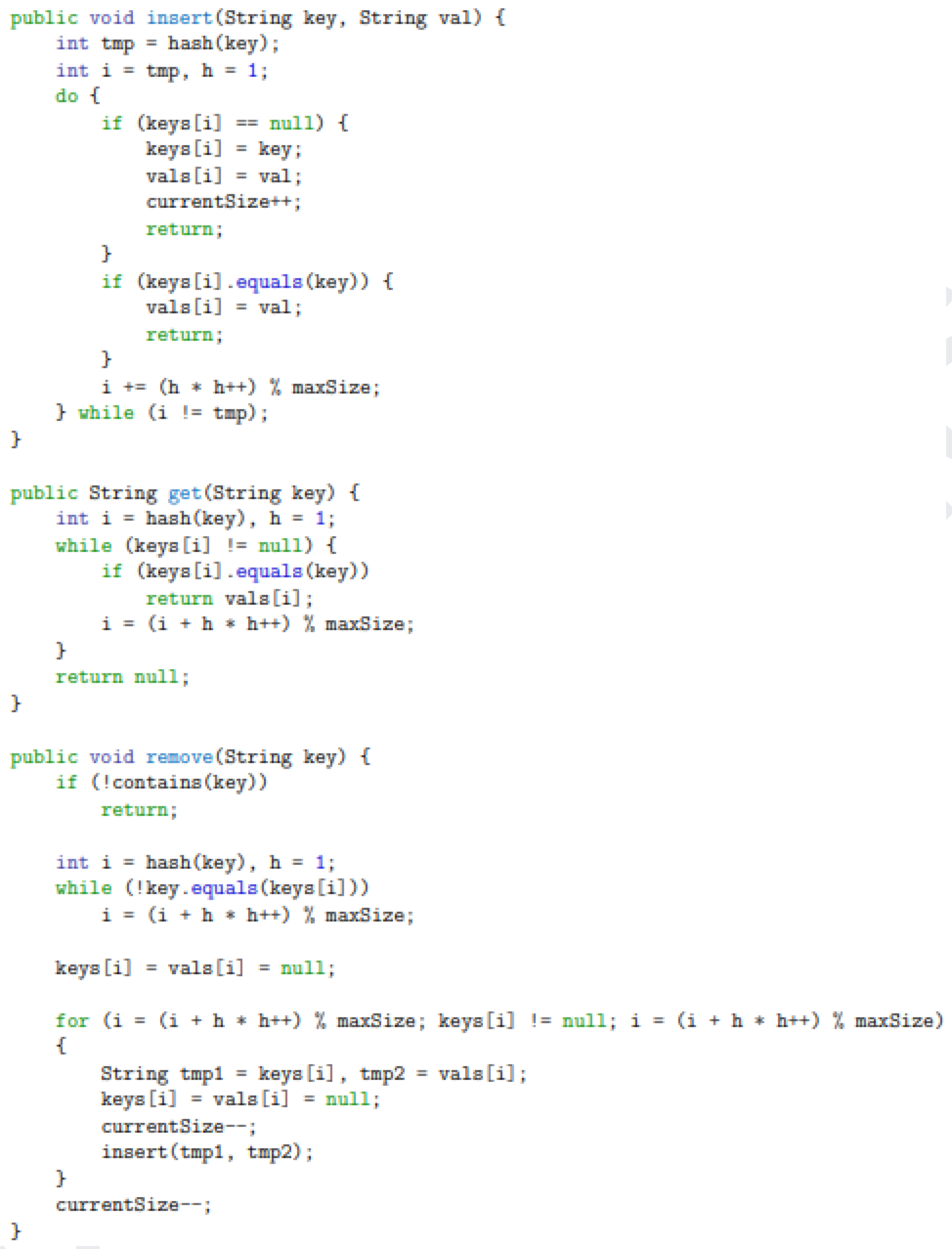
```
int multiply[][] = new int[m][q];

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

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

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

        multiply[c][d] = sum;
        sum = 0;
    }
}
```

- **Quadratic Probing**

Error 1: The insert method has a typo in the line i + = (i + h / h–) 3.

Error 2: In the remove method, there is a logic error in the loop to rehash keys. It should be i = (i + h * h++).

Error 3: In the get method, there is a logic error in the loop to find the key. It should be i = (i + h * h++).



```
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;
    } while (i != tmp);
}

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;
    }
    return null;
}

public void remove(String key) {
    if (!contains(key))
        return;

    int i = hash(key), h = 1;
    while (!key.equals(keys[i]))
        i = (i + h * h++) % maxSize;

    keys[i] = vals[i] = null;

    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--;
}
```

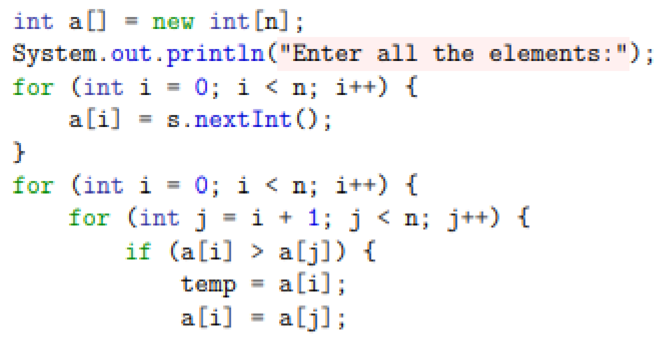
- **Sorting Array**

Error 1: The class name "Ascending Order" contains an extra space and an underscore. The class name should be corrected to "AscendingOrder."

Error 2: The first nested for loop has an incorrect loop condition for (int i = 0; i ¿= n; i++);, which should be modified to for (int i = 0; i ¡ n; i++).

Error 3: There is an extra semicolon (;) after the first nested for loop, which should be removed.

To fix these errors, you need to set breakpoints and step through the code. You should focus on the class name, the loop conditions, and the unnecessary semicolon.



```
int a[] = new int[n];
System.out.println("Enter all the elements:");
for (int i = 0; i < n; i++) {
    a[i] = s.nextInt();
}
for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        if (a[i] > a[j]) {
            temp = a[i];
            a[i] = a[j];
```

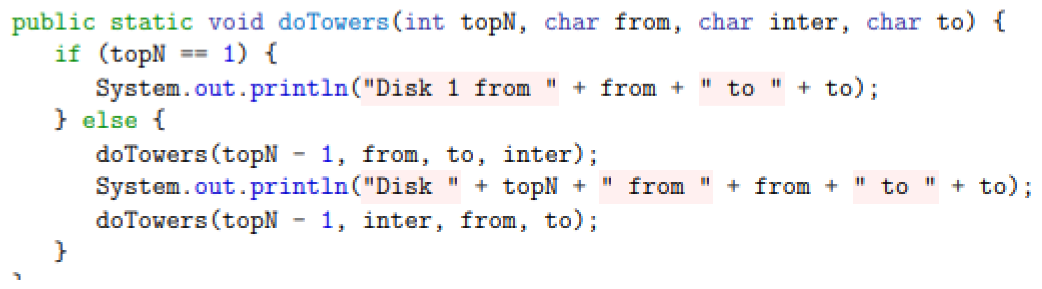
- **Tower of Hanoi**

Error 1: In the line doTowers(topN ++, inter–, from+1, to+1), there are errors in the increment and decrement operators. It should be corrected to doTowers(topN - 1, inter, from, to).

To fix this error, you need to replace the line:

doTowers(topN ++, inter--, from+1, to+1);   with

doTowers(topN - 1, inter, from, to); .



```
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);
      System.out.println("Disk " + topN + " from " + from + " to " + to);
      doTowers(topN - 1, inter, from, to);
   }
}
```

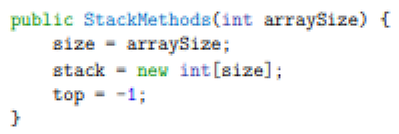
- **Stack Implementation**

Error 1: The push method has a decrement operation on the top variable (top–) instead of an increment operation. It should be corrected to top++ to push values correctly.

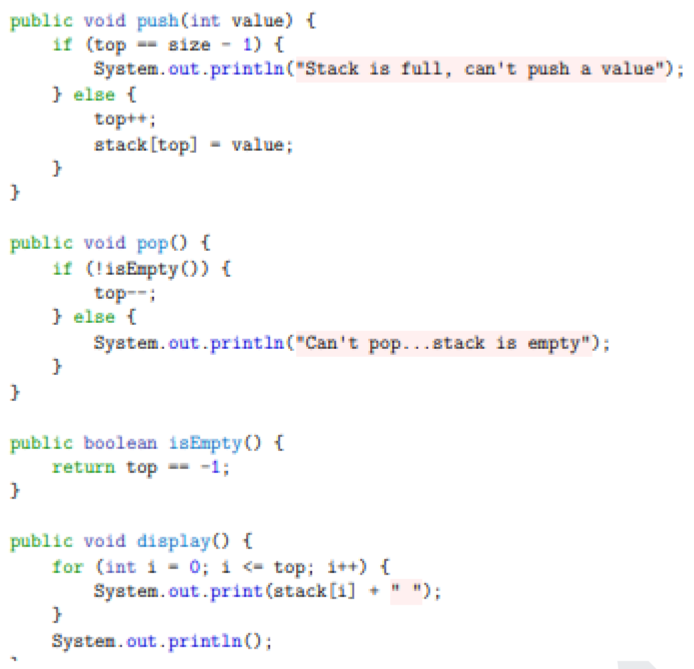
Error 2: The display method has an incorrect loop condition in for(int i=0; i ¿ top; i++). The loop condition should be for (int i = 0; i ¡= top; i++) to correctly display the elements.

Error 3: The pop method is missing in the StackMethods class. It should be added to provide a complete stack implementation.

To fix these errors, you would need to set breakpoints and step through the code, focusing on the push, pop, and display methods. Correct the push and display methods and add the missing pop method to provide a complete stack implementation.



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



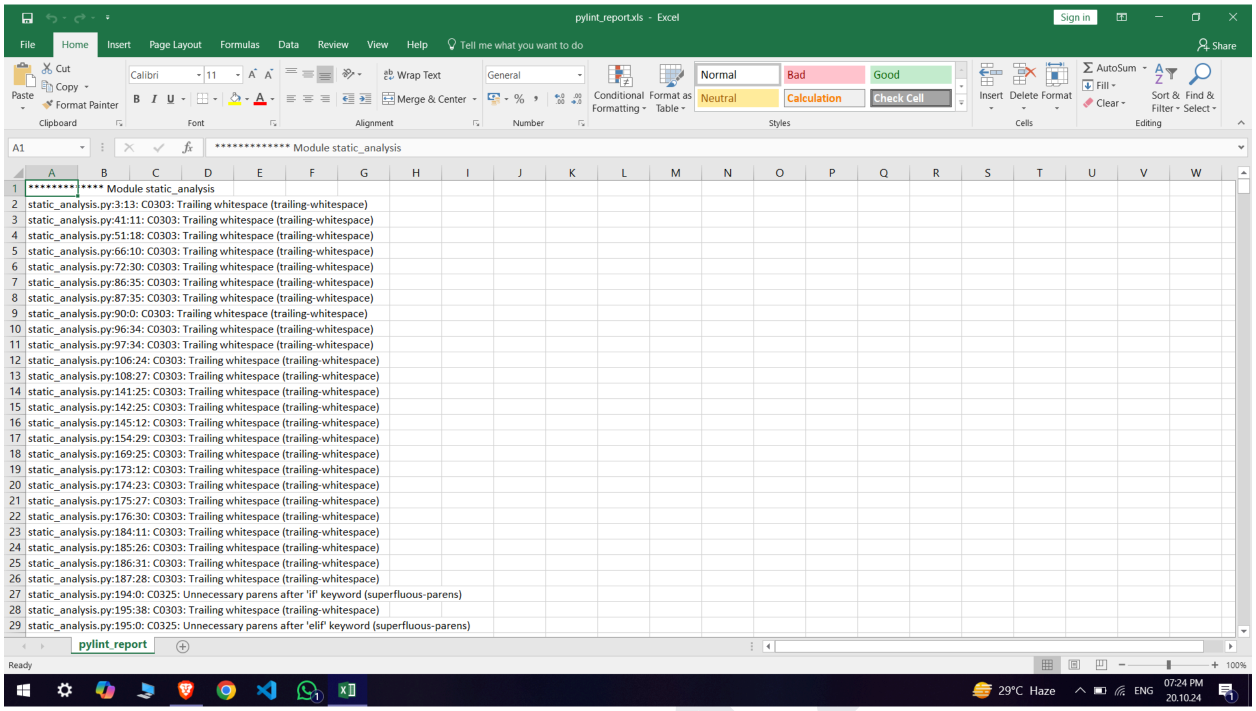
```
public void push(int value) {
    if (top == size - 1) {
        System.out.println("Stack is full, can't push a value");
    } else {
        top++;
        stack[top] = value;
    }
}

public void pop() {
    if (!isEmpty()) {
        top--;
    } else {
        System.out.println("Can't pop...stack is empty");
    }
}

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

public void display() {
    for (int i = 0; i <= top; i++) {
        System.out.print(stack[i] + " ");
    }
    System.out.println();
```

3. Static Analysis

Code taken from Github(Python)

Used Pylint to perform the static analysis

The excel file for the same is attached along side in Github.



```
************* Module static_analysis
static_analysis.py:3:13: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:41:11: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:51:18: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:66:10: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:72:30: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:86:35: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:87:35: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:90:0: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:96:34: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:97:34: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:106:24: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:108:27: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:141:25: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:142:25: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:145:12: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:154:29: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:169:25: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:173:12: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:174:23: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:175:27: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:176:30: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:184:11: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:185:26: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:186:31: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:187:28: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:194:0: C0325: Unnecessary parens after 'if' keyword (superfluous-parens)
static_analysis.py:195:38: C0303: Trailing whitespace (trailing-whitespace)
static_analysis.py:195:0: C0325: Unnecessary parens after 'elif' keyword (superfluous-parens)
```

It looks like this.