Name - Tasmay Patel

Student Id - 202201129

IT314: Lab 7

EXERCISE-I
PROGRAM INSPECTION

I have my own code where I have written some of CP Algorithms which I have used for preparation of my SI The code is around 1625 lines

this is the link of the file

Errors identified:

Category A: Data reference Errors

- There is an unused variable named **aaa** where aaa=None but not used afterwards.

- I tried to print the last element in binary_serach_example but instead of print arr[n-1] I have written **arr[n]** which goes out of bounds where n is the size of the array.

- We define a global variable `global_reference` initialized to `None`. When `create_list()` is called, `global_reference` points to `local_list`. After the function returns, `local_list` goes out of scope, but the global reference still points to the same list.
- Since I have made a mixed **array (containing integer and floating values)** in merge_sort_example(): so an error is likely present since the machine would use the floating-point bit representation in the memory area as an integer.

- An error happened in my class circle when i am calculating area of circle
    - I have two classes, Rectangle and Circle, each with an area method. The Rectangle class has attributes for width and height, while the Circle class has an attribute for the radius.

    - There is a function designed to calculate the area of a shape, specifically expecting an instance of the Rectangle class. It attempts to call the area method on the passed shape object.

    - The first call passes a Rectangle, which works correctly.

    - The second call mistakenly passes a Circle, which does not have an area method defined in the same way as Rectangle. This will raise an AttributeError.

Category B: Data declaration Errors

- In my process_data function, I may encounter a NameError because the variable result is referenced before it has been defined. When I attempt to execute result += data[i], Python expects result to already exist, but since it hasn't been initialized, it raises an error indicating that result is not defined. To resolve this issue, I will initialize the result to a starting value, such as 0, before entering the loop. This

way, the result will hold the cumulative sum of the elements in the data list correctly.

- There might be an error somewhere I might have declared a count variable and print it out without incrementing the count. In this the value of count might be printed as the default value, to resolve this error I can increment the count as count++ in the loop or whichever fashion I want.

- There is an error of wrong initialization of an array in segment_tree_example in this function I had passed size of the array too nut instead of declaring size as n i have by chance made an array of data = size which i cant instead of i need to give n = size and array of data with n values.

- Now I have used this data array as string in future that leads to two type of error first using of same variable multiple times other using this variable as different data type

- There are many variables in my code with similar names where differences might be of one or two characters mostly differing in singular and plural names.

Category C: Computation Errors

- There is an error in my function named add(); where I might have added a string with an integer i have named a variable a=”5” and b=3 and i have added them but this won't work because 5 is declared as string I just forgot to add ounces for 3 so 3 in b became an integer this sum won't add at all to correct it i will ounces in b=3 as b=”3”

- There can be an error where in simple complex CP algos in my code fragment such as Maximum flow problem, aho corasick where there

can be addition of some integers but it may overflow due to large int present somewhere and make the answer out of bounds.

- There might be a problem in some graph algorithms that a divide by zero or floating point error may occur and this will result in floating point error and to correct it I have to debug the code using a compiler and find where this error occurs.

- There is an error in check_probability() function where actually the probability is defined between 0 and 1 both inclusive but i have entered check_probability(1,5) actually I have given 1,5 value and not given the case of what will happen if 1,5 comes.

Category D: Comparison Errors

- There is a logical error in my code in Compare function where i want to compare two strings but i have declared one variable a=3 and b=”3” result=(a==b) here the result will give false despite value of and b in numeric sense equal because b was declared in staring and i forgot to declare a as string i forgot to put ounces in a which declare it as string.

- There is an error in my boolean expression function where i have declared two variables x=5 and y=10 and i want to set some values to true using if - else statement but i might have written if statement as x==10 and y==5 set condition true this will give me error as both x and y have opposite values each to declared in if statement.

- There might be some floating point comparison such as **a==0.3**. When comparing floating-point numbers, such as checking if a == 0.3, it's important to remember that floating-point arithmetic can introduce precision errors. Due to the way numbers are represented in binary, 0.3 may not be stored exactly as 0.3, which can lead to unexpected results when making direct comparisons.

- There is a problem in the boolean_expression2 function lies in the use of the bitwise & operator instead of the logical and operator in the conditional statement. While x == 5 & y == 10 may not raise a syntax error, it evaluates y == 10 as a boolean (True or False) and then performs a bitwise operation with the result of x == 5, which can lead to unexpected behavior. To properly check if both conditions are true, the expression should use if x == 5 and y == 10:. This ensures that both conditions are evaluated as logical comparisons, allowing the correct output when both are satisfied.

Category E: Control-Flow Errors

- **Loop termination:** While most loops are designed to terminate, it is not guaranteed for every loop. A loop will terminate if it has a defined exit condition that is met during execution. For example, a for loop with a counter that increments will eventually reach its maximum limit. There can be a scenario in my code where in some of the calculated algos the while(true) condition holds and the loop goes on till infinity without ending the program.

- A loop can fail to execute if the initial conditions do not satisfy its entry requirements. For example, in the loop for (i = x; i <= z; i++), if x is greater than z, the loop body will never execute. Similarly, in while (NOTFOUND), if NOTFOUND is initially false, the loop will not run at all. This can represent a logical oversight, especially if the programmer assumes that the loop will execute under any circumstances.

- Non-exhaustive decisions occur when logic does not account for all possible input values. For instance, if an input parameter's expected values are 1, 2, or 3, and the logic assumes that any value not equal to 1 or 2 must be 3, it can lead to incorrect assumptions. If the input were 4, the program could behave unexpectedly because the logic does not handle this case appropriately.

- In programming languages with statement groups or code blocks (like C or Java), there should be a corresponding closing bracket for each opening bracket. For instance, mismatched brackets will lead to compilation errors. If the opening and closing brackets do not match, the compiler will typically raise an error indicating that the block structure is invalid.

Category F: Control-Flow Errors

- The number of parameters in a module (function) must equal the number of arguments passed by the calling module. If they don't match, Python raises a TypeError. This kind of error came in my function add(a,b) where I have added two values a and b where a and b is given as input as function and the function must require both values but I passed add(5) during my function call. I forgot to add another value which gives me this error.

- When the data types of arguments do not match the expected types of parameters, it can result in TypeError. For example, passing a string when an integer is expected can cause the function to fail unexpectedly.

- If the attributes of arguments passed to another module do not align with the expected parameter types, a TypeError will occur, indicating that the argument type is incompatible.

- A mismatch in the units of transmitted arguments and their corresponding parameters can lead to logical errors. For example, if a function assumes its input is in kilometers when it's actually in miles, the output will be incorrect.

- When invoking built-in functions, incorrect numbers or order of arguments can raise errors. For instance, calling max() without any

arguments will cause a TypeError, indicating that at least one argument is required.

- When functions modify global variables without clear intent, it can lead to unintended side effects throughout the program. This can make debugging challenging and lead to inconsistent states in the program.

Category G: Input / Output Errors

- Since there is no file used in my code so there might not be any input / output errors present there but there are some errors which can be there might be some errors present.

- In Python, errors can occur when dealing with file input/output if attributes are incorrect, such as opening a file in the wrong mode (e.g., reading when you need to write), which leads to `FileNotFoundError` or `UnsupportedOperation`.

- Memory issues may arise when attempting to read large files into memory all at once, potentially causing the system to run out of memory.

- Proper end-of-file (EOF) detection is crucial to avoid infinite loops or crashes when reading files. Handling I/O errors like file access issues through `try-except` blocks is essential for program robustness.

Category H: Other Checks

- **Unused or Rarely Used Variables**: If the compiler produces a cross-reference listing, you may find variables that are never referenced or are only referenced once. These unused variables can indicate dead code, poor design, or unnecessary memory allocation. Cleaning up unused variables can improve readability and performance.

- **Incorrect Variable Attributes:** Attribute listings from the compiler can reveal incorrect or unexpected default attributes assigned to variables. For example, a variable might unintentionally be declared as global when it should be local, or the wrong data type might be inferred. This can lead to subtle bugs or unexpected behavior during execution.

- **Compiler Warnings**: When the program compiles successfully but generates warnings or informational messages, it indicates potential issues the compiler has flagged. These might include suspicious syntax, undeclared variables, or inefficient language constructs. Even if the program runs, ignoring these warnings can lead to unpredictable behavior or degraded performance.

- **Robustness and Input Validation:** A program is not robust if it doesn't validate input properly. Without checks for invalid, out-of-range, or unexpected input values, the program might crash or produce incorrect results. Ensuring robust input handling is crucial for program stability and security.

- **Missing Functions:** A missing function in the program can lead to incomplete functionality, causing errors or program crashes when the missing function is called. For instance, if a helper function designed to validate input or handle an edge case is omitted, the program could fail in scenarios that rely on it.

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

I think data reference errors, **Data Declaration Errors and Computation Errors** is more effective in finding out program inspection because

Category A: Data Reference Errors

Effectiveness: These errors are critical in programs that deal with a large volume of data or dynamic memory management. They often lead to crashes, memory leaks, or corrupted data, which are difficult to trace. Programs that are performance-critical, like operating systems or database systems, would benefit from early detection of these errors.

Category B: Data-Declaration Errors

Effectiveness: Data declaration errors can prevent the program from compiling or cause runtime type errors. Languages with strong typing (like Java or C#) typically catch these errors at compile time, making this category less of a concern compared to dynamically typed languages (e.g., Python, JavaScript). If you work in a dynamically typed language, inspecting for these errors is very effective to prevent bugs early in development.

Category D: Comparison Errors

Effectiveness: In systems where decision-making depends on conditions (e.g., control systems, AI algorithms), catching comparison errors early is critical. These can be subtle and lead to hard-to-debug logical flaws.

Programs that involve decision-making algorithms or conditional logic benefit greatly from inspecting for comparison errors.

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

1. Runtime Errors

- **Description:** These errors occur when the program is running, such as memory leaks, out-of-bounds errors, or division by zero.
- **Why it's difficult:** During inspection, code is reviewed statically (without running it). This makes it hard to anticipate dynamic issues that arise only under specific runtime conditions.

2. Performance Issues

- **Description:** Performance bottlenecks or inefficiencies, such as slow response times or excessive memory consumption.
- **Why it's difficult:** While you can sometimes spot inefficient algorithms, inspecting the code won't reveal how well it performs under real workloads. Performance testing tools and profiling are needed to assess these issues.

3. Concurrency Issues (Race Conditions, Deadlocks)

- **Description:** Errors that arise in multi-threaded or parallel programming environments, such as race conditions, deadlocks, or improper synchronization.
- **Why it's difficult:** These issues are often context-dependent and difficult to foresee just by looking at the code. They depend on timing

and interactions between threads, which are not visible in static code inspection.

4. Logic Errors Based on Unclear Requirements

- **Description:** Misunderstandings of the system's requirements that lead to incorrect logic.
- **Why it's difficult:** Even if the code matches the written requirements, those requirements themselves may be ambiguous or incomplete. This is not a failure of the code itself but of the broader system design.

5. Security Vulnerabilities

- **Description**: Issues like SQL injection, buffer overflows, or cross-site scripting (XSS) vulnerabilities.
- **Why it's difficult:** Many security vulnerabilities can be subtle and context-dependent. While inspection can catch some obvious ones, thorough vulnerability testing often requires dedicated tools (like static analysis or penetration testing) to spot complex issues.

6. User Interface (UI) / User Experience (UX) Issues

- **Description:** Usability problems, poor layout, or inconsistent design elements.
- **Why it's difficult:** Code inspection focuses on the internal workings of the software, not how users interact with it. UI/UX issues typically emerge during testing or user feedback, rather than being caught in code reviews.

7. Integration and Environment-Specific Errors

- **Description:** Errors that occur due to interactions with external systems, APIs, or hardware, or that are dependent on the deployment environment (e.g., database connections, file systems).

- **Why it's difficult:** Program inspection looks at the code in isolation, so it may not identify errors that depend on external systems or specific environments.

4. Is the program inspection technique is worth applicable?

1. Early Detection of Defects

- **Benefit:** Program inspection can catch many types of defects early in the development process, such as syntax errors, logical flaws, and violations of coding standards.
- **Why it's valuable:** Early detection saves time and resources because the cost of fixing defects increases significantly later in the development lifecycle (during integration, testing, or after release).

2. Improvement of Code Quality and Maintainability

- **Benefit:** Code inspections ensure that code is clean, well-structured, and adheres to best practices, making it more maintainable in the long term.
- **Why it's valuable:** Well-inspected code tends to be easier to read, understand, and modify, which reduces technical debt and makes future changes or additions more efficient.

3. Knowledge Sharing and Team Collaboration

- **Benefit:** Inspections often involve multiple developers reviewing the code, fostering knowledge sharing within the team.
- **Why it's valuable:** Junior developers can learn from more experienced ones, while senior developers can catch blind spots they

may have missed. This also improves team-wide familiarity with the codebase.

4. Increased Code Consistency and Standardization

- **Benefit:** Inspections help ensure that the code follows agreed-upon coding standards and architectural guidelines.
- **Why it's valuable:** Consistent code is easier to understand and debug, even when different team members are working on it over time.

EXERCISE-II
DEBUGGING JAVA CODES

1. Armstrong Numbers

Errors Identified:

- Error in digit extraction (division instead of modulus): The variable `remainder = num / 10;` is incorrect. To extract the last digit of a number, the correct operation should be `remainder = num % 10;`. The division would give the quotient instead of the last digit.
- Incorrect reduction of `num`: The statement `num = num % 10;` should instead be `num = num / 10;` to remove the last digit from the number.
- Wrong logic for summing the cubes of digits: Since the program is calculating the cube of `remainder`, the cube sum should accumulate `check = check + (int)Math.pow(remainder, 3);`.

Breakpoints Needed to Fix Errors:

- One to check the extraction of the remainder and cube calculation.
- Another to check the reduction of the number (`num`) after processing each digit.

a. Steps Taken to Fix the Errors:

1. Correct the extraction of digits from the number using `remainder = num % 10;`.
2. Correct the reduction of the number using `num = num / 10;` to move to the next digit.
3. Ensure the Armstrong logic correctly sums the cubes of the digits.

Corrected Executable Code:

```
class Armstrong {
```

```
    public static void main(String args[]) {
        int num = Integer.parseInt(args[0]); // Input number
        int n = num; // Original number to compare at the end
        int check = 0, remainder;

        // Loop through all digits of the number
        while (num > 0) {
            remainder = num % 10; // Get the last digit
            check += Math.pow(remainder, 3); // Add cube of the digit to
check
            num = num / 10; // Remove the last digit from num
        }

        // Check if the sum of cubes equals the original number
        if (check == n)
            System.out.println(n + " is an Armstrong Number");
        else
            System.out.println(n + " is not an Armstrong Number");
    }
}
```

2. Tower of Hanoi

Errors Identified:

- Use of post-increment (`topN++`) and post-decrement (`inter--`): In the recursive function, post-increment and post-decrement are incorrectly used. The values of `topN`, `inter`, and other parameters should not change during the recursive call. You need to pass them without altering their values in the function call.
- Inappropriate character arithmetic (`from+1`, `to+1`): Characters in Java can't be incremented in this context (i.e., `from + 1`, `to + 1`). This will result in unexpected behavior and needs to be avoided. Instead, pass the characters directly to the recursive function as parameters.
- Missing semicolon after the recursive call: The second recursive call `doTowers(topN ++, inter--, from+1, to+1)` lacks a semicolon, leading to a syntax error.

Breakpoints Needed to Fix Errors:

- One to check the proper function of the recursive call (i.e., `doTowers`).
- Another to ensure correct increment/decrement of `topN`, `from`, and `to`.

a. Steps Taken to Fix the Errors:

1. Remove the post-increment and post-decrement operations and pass `topN - 1`, `from`, `inter`, and `to` directly to the recursive function.
2. Replace incorrect character arithmetic (`from + 1`, `to + 1`) with the appropriate parameters `from`, `inter`, and `to`.
3. Fix the missing semicolon after the recursive call to complete the statement.

Corrected Executable Code:

```java
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 top N-1 disks from 'from' to 'inter' using 'to' as auxiliary
         System.out.println("Disk " + topN + " from " + from + " to " + to); // Move the remaining disk
         doTowers(topN - 1, inter, from, to); // Move the N-1 disks from 'inter' to 'to' using 'from' as auxiliary
      }
   }
}
```

3. Knapsack

Errors Identified:

- Post-increment in accessing `opt[n++][w]`: The statement `int option1 = opt[n++][w];` increments `n` after the value is fetched, which will lead to incorrect behavior in the following lines. This should be `opt[n][w]` without the increment. You want to stay on the same item for both option checks.
- Error in `option2` calculation: In the calculation `option2 = profit[n-2] + opt[n-1][w-weight[n]];`, subtracting `2` from `n` is wrong. It should simply be `profit[n]` and not `profit[n-2]`. Also, ensure that `w-weight[n]` is non-negative.
- Missing logic to handle weight constraint: The check `if (weight[n] > w)` should be `if (weight[n] <= w)` to ensure that the item is only taken if it fits in the current weight limit.
- Array initialization for profit and weight: The weights of the items are being initialized using `Math.random() * W`, which may lead to weights greater than the knapsack capacity (`W`). Consider limiting the random values appropriately to make them feasible for the knapsack problem.

Breakpoints Needed to Fix Errors:

- One to check that the `opt` and `sol` arrays are updated correctly in the nested loop.
- Another to validate the computation of `option1` and `option2` during the decision-making process.

a. Steps Taken to Fix the Errors:

1. Fix the post-increment issue by replacing `opt[n++][w]` with `opt[n][w]`.

2. Update the logic for `option2` to correctly reference `profit[n]` and avoid subtracting `2` from `n`.
3. Correct the condition for checking if an item can be taken (i.e., `if (weight[n] <= w)`).
4. Adjust the random weight generation to ensure weights are smaller than or equal to the maximum weight (`W`).

Corrected Executable Code:

```
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);  // random profit
for each item

         weight[n] = (int) (Math.random() * W);     // random weight,
should be <= 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];



                // take item n if it fits

                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);

            }

        }
```

```
        // 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. Stack Implementation

Errors Identified:

- Error in the `push` method: The `push` method is decrementing `top` instead of incrementing it. When an element is added to the stack, `top` should be incremented (`top++`), not decremented (`top--`).
- Error in the `pop` method: The `pop` method is incrementing `top` instead of decrementing it. When an element is removed from the stack, `top` should be decremented (`top--`), not incremented.
- Error in the `display` method: The `for` loop condition in the `display` method is incorrect. The condition `i > top` will not work because the loop should run from `0` to `top` to display all elements in the stack. The correct condition should be `i <= top`.

Breakpoints Needed to Fix Errors:

- One to verify the `push` method and check if `top` is incremented correctly.
- Another to validate the `pop` method and ensure `top` is decremented.
- One more to check the `display` method and verify that the correct elements are being printed.

a. Steps Taken to Fix the Errors:

1. Correct the `push` method by incrementing `top` (`top++`) before assigning the value to the stack.
2. Correct the `pop` method by decrementing `top` (`top--`) when an element is removed.
3. Fix the `for` loop in the `display` method to ensure it runs from `0` to `top` and prints the correct stack elements.

Corrected Executable Code:

```
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++; // Corrected from top-- to top++
            stack[top] = value;
        }
    }

    public void pop() {
        if (!isEmpty()) {
            System.out.println("Popped: " + stack[top]);
            top--; // Corrected from top++ to top--
        } else {
            System.out.println("Can't pop...stack is empty");
        }
    }

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

    public void display() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
        } else {
            System.out.println("Stack contents:");
            for (int i = 0; i <= top; i++) { // Corrected condition from i
> top to i <= top
                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(); // Should display all the pushed elements
        newStack.pop();       // Popping elements
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.display();  // Should display remaining elements
    }
}
```

5. Quadratic probing

Errors Identified:

- Error in Insert Method (`insert`):
  - The expression `i + = (i + h / h--) % maxSize;` contains a syntax error (`i +=` should be used instead of `i + =`).
  - The increment and decrement operators in the expression `i += (i + h / h--) % maxSize;` are incorrect. This causes unexpected behavior during insertion, especially during quadratic probing.
- Error in `remove` and `rehash`:
  - The removal method decreases `currentSize` twice: once while removing the element and again when rehashing. This causes the size to be incorrectly reduced.
  - During rehashing, the logic to reinsert elements in the table is incorrect. The value `h` is incremented inside the loop, and quadratic probing is not applied correctly while reinserting elements.
- Error in Hash Function (`hash`):
  - The hash function may produce negative values due to Java's `hashCode` method. This issue can be resolved by ensuring the result is non-negative: `return (key.hashCode() & 0x7fffffff) % maxSize;`.

Breakpoints Needed to Fix Errors:

- One breakpoint at the `insert` method to check for the correct calculation of the index `i` during insertion.
- One breakpoint at the `remove` method to ensure `currentSize` is correctly updated and rehashing works properly.

a. Steps Taken to Fix the Errors:

1. Correct the `insert` method to fix the syntax issue and adjust the probing logic for quadratic probing.
2. Ensure `currentSize` is decremented correctly only once during removal and rehashing.
3. Fix the hash function to always return a non-negative index.
4. Rework the rehashing loop inside the `remove` method to ensure the correct re-insertion of keys and values using quadratic probing.

Corrected Executable 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 (key.hashCode() & 0x7fffffff) % maxSize; // Ensure
non-negative index
    }

    /** Function to insert key-value pair **/
    public void insert(String key, String val) {
        if (isFull()) {
            System.out.println("Hash Table is full, cannot insert new
key-value pair.");
            return;
        }

        int tmp = hash(key);
        int i = tmp, h = 1;

        do {
            if (keys[i] == null) {
                keys[i] = key;
                vals[i] = val;
                currentSize++;
                return;
            }

            if (keys[i].equals(key)) {
```

```
                vals[i] = val;
                return;
            }

            i = (i + h * h) % maxSize; // Quadratic probing
            h++;
        } while (i != tmp);
    }

    /** Function to get value for a given key **/
    public String get(String key) {
        int i = hash(key), h = 1;
        while (keys[i] != null) {
            if (keys[i].equals(key))
                return vals[i];

            i = (i + h * h) % maxSize;
            h++;
        }
        return null;
    }

    /** Function to remove key and its value **/
    public void remove(String key) {
        if (!contains(key)) {
            System.out.println("Key not found, cannot remove.");
            return;
        }

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

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

        /** Rehash all keys **/
        i = (i + h * h) % maxSize;
```

```
        while (keys[i] != null) {
            String tmpKey = keys[i];
            String tmpVal = vals[i];
            keys[i] = vals[i] = null;
            currentSize--;
            insert(tmpKey, tmpVal);
            i = (i + h * h) % maxSize;
        }
    }

    /** Function to print HashTable **/
    public void printHashTable() {
        System.out.println("\nHash Table: ");
        for (int i = 0; i < maxSize; i++) {
            if (keys[i] != null)
                System.out.println(keys[i] + " " + vals[i]);
        }
        System.out.println();
    }
}

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

        QuadraticProbingHashTable qpht = new
QuadraticProbingHashTable(scan.nextInt());

        char ch;
        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;
            }

            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');
    }
}
```

6. Matrix Multiply

Errors Identified:

- Logical error in the matrix multiplication loop:
  - The indexing logic inside the innermost loop for matrix multiplication is incorrect. Specifically, the expression `first[c-1][c-k]` and `second[k-1][k-d]` uses wrong indices, which will result in incorrect matrix element access and could potentially throw an `ArrayIndexOutOfBoundsException`.
  - The correct formula for matrix multiplication is:

    multiply[i][j]+ = first[i][k] + second[k][j]

  - The current loop structure is incorrect because it references matrix elements with the wrong indices.
- User input description mistake:
  - The program asks for "Enter the number of rows and columns of first matrix" again when it should ask for the second matrix.
- Variable name duplication:
  - `sum` should be reset inside the outermost loop, not just inside the innermost one. This could lead to errors in subsequent iterations.
- Matrix element input format:
  - The input format for matrix elements does not correspond to the correct dimensions mentioned (rows and columns). It needs to be clarified.

Breakpoints Needed:

- One breakpoint inside the nested loop where matrix multiplication is happening, to check if matrix elements are being correctly accessed and multiplied.

- One breakpoint before the result is printed to ensure that `multiply[][]` stores the correct product.

a. Steps Taken to Fix the Errors:

1. Correct the indexing logic for matrix multiplication to:
    - `sum += first[c][k] * second[k][d];`
2. Fix the input prompt text for the second matrix.
3. Ensure that the `sum` variable is reset after each iteration of calculating the result of one element of the resultant matrix.
4. Clarify the input format for matrix elements, ensuring the user enters elements in the correct sequence and size.

Corrected Executable Code:

```
import java.util.Scanner;

class MatrixMultiplication {
    public static void main(String args[]) {
        int m, n, p, q, sum = 0, c, d, k;

        Scanner in = new Scanner(System.in);
        System.out.println("Enter the number of rows and columns of first
matrix:");
        m = in.nextInt();
        n = in.nextInt();

        int first[][] = new int[m][n];

        System.out.println("Enter the elements of first matrix:");
        for (c = 0; c < m; c++)
            for (d = 0; d < n; d++)
                first[c][d] = in.nextInt();

        System.out.println("Enter the number of rows and columns of second
matrix:");
        p = in.nextInt();
        q = in.nextInt();
```

```
        if (n != p)
            System.out.println("Matrices with entered orders can't be
multiplied with each other.");
        else {
            int second[][] = new int[p][q];
            int multiply[][] = new int[m][q];

            System.out.println("Enter the elements of second matrix:");
            for (c = 0; c < p; c++)
                for (d = 0; d < q; d++)
                    second[c][d] = in.nextInt();

            // Matrix multiplication logic
            for (c = 0; c < m; c++) {
                for (d = 0; d < q; d++) {
                    for (k = 0; k < n; k++) { // Corrected loop bound to
`n`
                        sum += first[c][k] * second[k][d]; // Corrected
indices
                    }
                    multiply[c][d] = sum;
                    sum = 0; // Reset sum for next element
                }
            }

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

7. Magic Numbers

Errors Identified:

- Logical error in the inner while loop:
  - The condition `while(sum == 0)` is incorrect. The loop should continue until `sum` becomes 0. The condition should be `while (sum > 0)`.
- Wrong calculation of the sum of digits:
  - The expression `s = s * (sum / 10)` is incorrect for calculating the sum of digits. It should be `s += sum % 10` to get the remainder (digit) and add it to the sum.
  - The line `sum = sum % 10` should be `sum = sum / 10` to remove the last digit from the number.
- Missing semicolon:
  - There's a missing semicolon in `sum=sum%10`, which should be `sum = sum / 10;`.
- Incorrect initialization of variables:
  - The `sum` variable is being assigned as `sum = num;` at the start of the loop, which is unnecessary and will cause an infinite loop if the number is greater than 9. Instead, `sum` should be calculated inside the loop by summing the digits of `num`.

Breakpoints Needed:

- One breakpoint before the inner while loop to check if the sum of digits is calculated correctly.
- One more breakpoint before checking if `num` equals 1 after the outer while loop.

a. Steps Taken to Fix the Errors:

1. Corrected the condition in the inner loop to `while (sum > 0)`.

2. Fixed the digit sum calculation inside the inner loop by changing `s = s * (sum / 10)` to `s += sum % 10`.
3. Changed `sum = sum % 10` to `sum = sum / 10;` to correctly remove the last digit.
4. Corrected the initialization and placement of sum calculation to make sure the sum of digits is calculated properly.

Corrected Executable Code:

```
import java.util.*;

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

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

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

8. Merge sort

Errors Identified:

- Incorrect array passing in the `mergeSort` method:
  - The expressions `int[] left = leftHalf(array + 1);` and `int[] right = rightHalf(array - 1);` are incorrect. You cannot add or subtract integers from arrays. It should simply be `leftHalf(array)` and `rightHalf(array)` without modifications.
- Incorrect parameters for the `merge` method:
  - The expression `merge(array, left++, right--);` is incorrect. You don't need to modify the references to the arrays `left` and `right`. It should be `merge(array, left, right);`.
- Misuse of `array++` and `array--`:
  - `left++` and `right--` in the `merge` call are invalid operations for arrays. You cannot increment or decrement arrays like this.

Breakpoints Needed:

- One breakpoint after splitting the array into two halves, to ensure the array is split correctly.
- One breakpoint inside the `merge` method to verify that the left and right arrays are merged correctly.

a. Steps Taken to Fix the Errors:

1. Correct the passing of arrays in `mergeSort`:
   - Changed `leftHalf(array + 1)` to `leftHalf(array)`, and similarly for `rightHalf`.
2. Remove invalid array increment/decrement:

    - Fixed `merge(array, left++, right--)` to `merge(array, left, right)`.
3. Ensure correct logic in array splitting and merging:
    - Verified that arrays are properly split and merged without errors.

Corrected Executable Code:

```
import java.util.*;

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

    // Places the elements of the given array into sorted order
    // using the merge sort algorithm.
    public static void mergeSort(int[] array) {
        if (array.length > 1) {
            // split array into two halves
            int[] left = leftHalf(array);
            int[] right = rightHalf(array);

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

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

    // Returns the first half of the given array.
    public static int[] leftHalf(int[] array) {
        int size1 = array.length / 2;
        int[] left = new int[size1];
        for (int i = 0; i < size1; i++) {
            left[i] = array[i];
```

```
        }
        return left;
    }

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

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

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

9. Sorting Array

Errors Identified in the Program:

- Invalid Class Name:
  - The class name `Ascending_Order` contains a space, which is not allowed in Java. Class names should not have spaces or underscores, and by convention, they should start with an uppercase letter without spaces.
- Incorrect loop condition in the outer loop:
  - The condition `for (int i = 0; i >= n; i++);` is incorrect. This will not allow the loop to run as `i >= n` will always be false from the start. It should be `i < n - 1`.
- Extra semicolon (`;`) after the for loop:
  - There is an extra semicolon after the outer `for` loop which terminates the loop prematurely, resulting in incorrect execution.
- Incorrect conditional check in sorting logic:
  - The condition `if (a[i] <= a[j])` should be changed to `if (a[i] > a[j])` to properly sort in ascending order. Currently, the condition is set up to perform a descending order swap.

Breakpoints Needed:

- Breakpoint before the nested loops to check the values of `i` and `j`.
- Breakpoint inside the swap logic to verify if the condition for swapping is correctly implemented.

a. Steps Taken to Fix the Errors:

1. Class Name Correction:
   - Changed the class name from `Ascending _Order` to `AscendingOrder` to follow Java's naming convention.
2. Correct the loop condition:

- Changed the outer loop condition from `i >= n` to `i < n - 1` to iterate through the elements properly.

3. Remove the extra semicolon:
   - Removed the semicolon after the `for` loop to allow the loop to execute its body.
4. Fix sorting logic:
   - Changed the condition `if (a[i] <= a[j])` to `if (a[i] > a[j])` for sorting in ascending order.

Complete Executable Code:

```
import java.util.Scanner;

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

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

        // Taking input for the elements of the array
        System.out.println("Enter all the elements:");
        for (int i = 0; i < n; i++) {
            a[i] = s.nextInt();
        }

        // Sorting the array in ascending order using a basic bubble sort
        for (int i = 0; i < n - 1; i++) {
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) {
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
```

```
        }

        // Display the sorted array
        System.out.print("Ascending Order: ");
        for (int i = 0; i < n - 1; i++) {
            System.out.print(a[i] + ", ");
        }
        System.out.print(a[n - 1]);  // To print the last element without
a comma
    }
}
```

10. GCD and LCM

Errors Identified in the Program:

- Logic in GCD Calculation:
  - In the `gcd` function, the condition `while(a % b == 0)` should be replaced with `while(a % b != 0)` because the GCD calculation continues until the remainder becomes 0, not while it is 0.
- Logic in LCM Calculation:
  - The `if(a % x != 0 && a % y != 0)` condition should be `if(a % x == 0 && a % y == 0)`. The LCM is the smallest multiple of both `x` and `y`, which means both `x` and `y` must divide `a` without leaving a remainder.
- Incorrect Output Text for LCM:
  - In the output for LCM, the text should mention "LCM" instead of repeating "GCD" twice.

Breakpoints Needed:

- A breakpoint can be set inside the `while` loop in the `gcd` method to check the values of `a` and `b` as the loop runs.
- A breakpoint can also be set inside the `while(true)` loop in the `lcm` method to check the value of `a` as it is incremented.

a. Steps Taken to Fix the Errors:

1. Fixed the GCD Logic:
   - Changed the condition `while(a % b == 0)` to `while(a % b != 0)` to ensure the GCD calculation continues until the remainder becomes zero.
2. Fixed the LCM Logic:

- Corrected the condition `if(a % x != 0 && a % y != 0)` to `if(a % x == 0 && a % y == 0)` so that `a` is a common multiple of both `x` and `y`.

3. Updated the Output Message:
   - Fixed the output message to print "The LCM of two numbers is" instead of incorrectly repeating the GCD message.

Complete Executable Code:

```
import java.util.Scanner;

public class GCD_LCM {

    // Method to calculate GCD of two numbers
    static int gcd(int x, int y) {
        int r = 0, a, b;
        a = (x > y) ? x : y; // a is the greater number
        b = (x < y) ? x : y; // b is the smaller number

        // Calculating GCD using Euclidean algorithm
        while (b != 0) {
            r = a % b;
            a = b;
            b = r;
        }
        return a; // a will contain the GCD
    }

    // Method to calculate LCM of two numbers
    static int lcm(int x, int y) {
        int a = (x > y) ? x : y; // Start with the greater number
        // Increment and check until the smallest common multiple is found
        while (true) {
            if (a % x == 0 && a % y == 0)
                return a; // Return LCM when divisible by both
            ++a;
        }
    }
```

```
    public static void main(String args[]) {
        Scanner input = new Scanner(System.in);

        // Input two numbers from the user
        System.out.println("Enter the two numbers: ");
        int x = input.nextInt();
        int y = input.nextInt();

        // Output GCD and LCM of the two numbers
        System.out.println("The GCD of two numbers is: " + gcd(x, y));
        System.out.println("The LCM of two numbers is: " + lcm(x, y));

        input.close(); // Close the scanner
    }
}
```

EXERCISE-III
Static Analysis Tool

Here is the Static analysis of the file performed using pylint report of python code

Formatting Issues:

- **Unnecessary Semicolons**: Lines with unnecessary semicolons (W0301).
- **Trailing Whitespace**: Lines with extra spaces at the end (C0303).
- **Line Too Long**: Lines that exceed the maximum length (C0301).
- **Trailing Newlines**: Extra newline characters at the end of the file (C0305).
- **Bad Indentation**: Inconsistent indentation (W0311).

Naming Convention Issues:

- **Snake_case Naming Style**: Variable and module names that don't conform to snake_case (C0103).
- **Upper Case for Constants**: Constant names not in UPPER_CASE (C0103).

Missing Docstrings:

- **Missing Docstrings**: Functions, methods, and classes missing docstrings (C0115, C0116).
- **Missing Module Docstring**: No module-level docstring (C0114).

Code Efficiency/Quality Issues:

- **Unused Variables**: Variables that are declared but never used (W0612).

- **Unspecified Encoding**: `open()` called without specifying encoding (W1514).
- **Unnecessary 'elif' After 'return'**: The `elif` is redundant because the previous `return` already exits the function (R1705).
- **Inconsistent Return Statements**: Functions that inconsistently return values (R1710).

Code Redefinition:

- **Redefining Built-in Functions**: Redefining built-in functions like `sum` (W0622).
- **Redefining Variables**: Variables being redefined, possibly causing conflicts (W0621).

Errors:

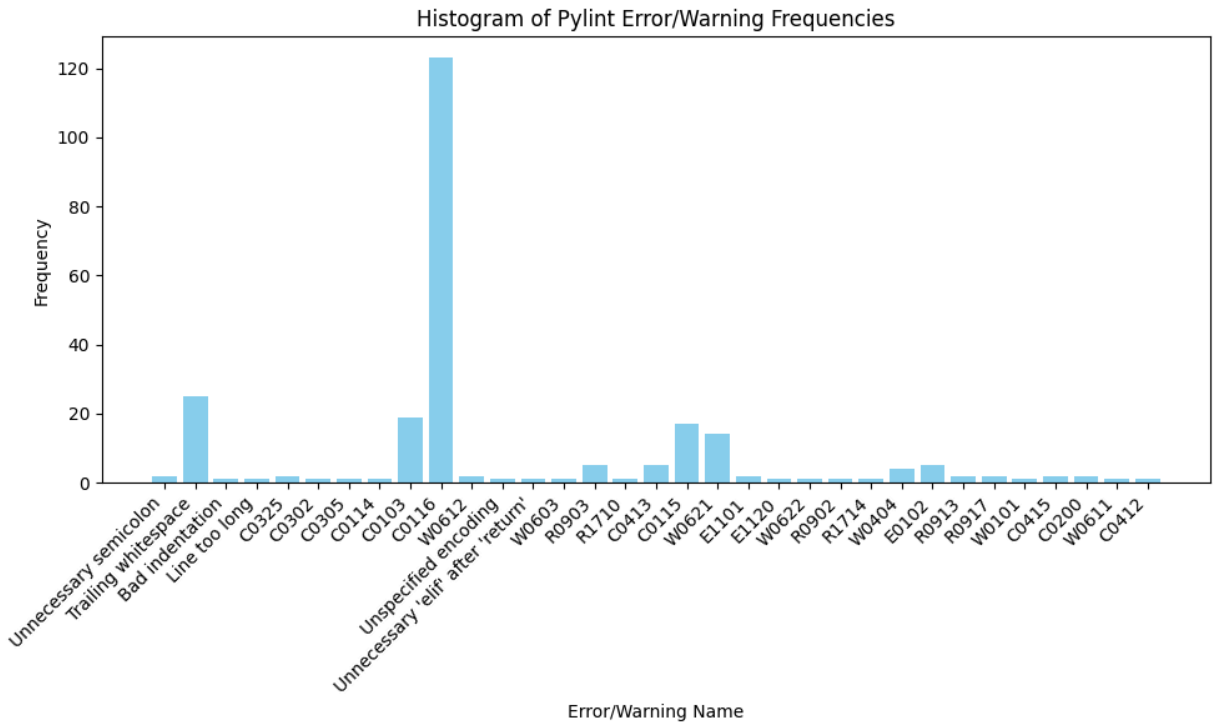
- **No Member Error**: Refers to methods or attributes that don't exist in the class (E1101).
- **Reimported Modules**: Modules like `deque` being imported multiple times (W0404).
- **Unreachable Code**: Lines of code that will never execute (W0101).

Class Design:

- **Too Few Public Methods**: Classes that don't have enough public methods (R0903).
- **Too Many Instance Attributes**: Classes that have too many instance attributes (R0902).

Logical Suggestions:

- **Consider Merging Comparisons**: Multiple comparisons that can be merged (R1714).


Histogram of Pylint Error/Warning Frequencies
Frequency
0
20
40
60
80
100
120
Unnecessary semicolon
Trailing whitespace
Bad indentation
Line too long
C0325
C0302
C0305
C0114
C0103
C0116
W0612
Unspecified encoding
Unnecessary 'elif' after 'return'
W0603
R0903
R1710
C0413
C0115
W0621
E1101
E1120
W0622
R0902
R1714
W0404
E0102
R0913
R0917
W0101
C0415
C0200
W0611
C0412
Error/Warning Name