CMSI 4072: Senior Project II Written Assignment #3

Problem 7.1, Stephens page 169

1. Typo in “provote”, should be “private”
2. “If” should be lowercase
3. The loop can be simplified using a while loop instead of if
4. Semicolon is not needed after the closing bracket

Problem 7.2, Stephens page 170

1. **Inconsistent or Incorrect Comments**: Comments that are not aligned with the functionality of the code can lead to confusion. In the code example, the comments talk about things that are either inaccurate or overly simplistic. For instance, the comment "Repeat until we're done" is too vague and doesn't clearly explain the algorithm or what condition the loop is checking for
2. **Outdated Comments**: If the comments are not updated when the code is changed, they can become misleading or incorrect. For example, if the logic of the code changes but the comments remain the same, it leads to a mismatch between what the comments suggest and what the code actually does.

Problem 7.4, Stephens page 170

1. **Check for Invalid Input (Non-positive Integers)**: One of the most common errors in GCD calculations can occur if the input values are non-positive (e.g.., a <= 0 or b <= 0). Since the GCD is typically only defined for positive integers, you could add checks to ensure the inputs are valid.
2. **Handle Edge Case for GCD(0, 0)**: The GCD of (0, 0) is undefined, and most GCD algorithms do not handle this case. We can include a check to throw an exception or return a specific error code.
3. **Check for Overflow/Underflow**: If a or b are very large, there could be an overflow when calculating the remainder

```
private long GCD(long a, long b)
{
   // Check for invalid input (negative numbers or zero)
```

```
    if (a <= 0 || b <= 0)
    {
        throw new IllegalArgumentException("Both numbers must be positive integers.");
    }

    // Handle the edge case where GCD(0, 0) is called
    if (a == 0 && b == 0)
    {
        throw new IllegalArgumentException("GCD of (0, 0) is undefined.");
    }

    // Proceed with the Euclidean algorithm
    while (b != 0)
    {
        long remainder = a % b;
        a = b;
        b = remainder;
    }
    return a;
}
```

Problem 7.5, Stephens page 170

Yes, error handling should be added to the modified code.

Input Validation and Edge Cases Are Already Handled:

- The modified code already checks for invalid inputs such as non-positive integers (a <= 0 || b <= 0) and the edge case where both a and b are 0 (a == 0 && b == 0).
- These checks prevent the algorithm from running into unexpected or undefined behavior.
- If any of these error conditions are encountered, an IllegalArgumentException is thrown, providing clear feedback about what went wrong.

Why Error Handling is Important in This Case:

- **Clear Feedback for the User**: If invalid values are passed in, throwing an exception with a descriptive error message ensures the user understands exactly what went wrong. Without this error handling, if someone tried to run the function with invalid inputs, the program might produce incorrect results or even crash without any explanation

- **Handling Undefined Behavior**: By explicitly handling the case of (0, 0), you're ensuring that the function doesn't run with a condition that is mathematically undefined. This is especially important to ensure that no incorrect results are returned or undefined operations are performed
- **Reliability**: With proper error handling, you make the function more reliable, as users of the code can be confident that it will always behave correctly, even when given unexpected inputs

Problem 7.7, Stephens page 170

Highest Level of Instructions:

1. **Get in your car and prepare to drive**
    - Ensure the car is in working condition (e.g., check fuel, ensure engine is running, and seatbelt is fastened).
    - Start the car
2. **Drive to the nearest supermarket.**
    - Determine the location of the nearest supermarket
    - Identify the best route from your current location to the supermarket
3. **Park the car at the supermarket.**
    - Find a parking spot near the supermarket entrance
    - Park the car safely
4. **Enter the supermarket and proceed with your shopping**

Assumptions:

1. You know where the nearest supermarket is (either through personal knowledge or an address)
2. You have access to a working car with a full gas tank or sufficient fuel for the trip
3. You have a driver’s license and the legal ability to drive
4. You know how to operate the car, including starting the engine, driving, and parking.

5. There are no significant road closures, accidents, or other obstructions that would prevent you from getting to the supermarket
6. The car is equipped with basic safety features, such as functioning brakes, headlights, and indicators
7. You have access to a map or GPS, or you are familiar with the route to the supermarket.
8. There is a parking spot available at the supermarket
9. You plan to shop once you get there

Problem 8.1, Stephens page 199

```
public class RelativelyPrimeTest {

   // Method to calculate the GCD of two numbers using Euclid's algorithm
   public static long GCD(long a, long b) {
      // Ensure that a is always the larger number
      if (a < b) {
         long temp = a;
         a = b;
         b = temp;
      }

      // Use Euclid's algorithm to find the GCD
      while (b != 0) {
         long remainder = a % b;
         a = b;
         b = remainder;
      }
```

```
        return a; // GCD is in 'a' when b becomes 0
    }

    // Method to check if two numbers are relatively prime
    public static boolean isRelativelyPrime(long a, long b) {
        // Special case: -1 and 1 are relatively prime to every number, and the only numbers relatively prime
to 0
        if (a == 0 || b == 0) {
            return false;
        }

        // Calculate the GCD of a and b
        long gcd = GCD(a, b);

        // If GCD is 1, they are relatively prime
        return gcd == 1;
    }

    // Main method to test the isRelativelyPrime() method
    public static void main(String[] args) {
        // Test pairs of integers
        long[][] testCases = {
            {8, 9},   // Relatively prime (GCD = 1)
            {21, 35}, // Not relatively prime (GCD = 7)
            {14, 25}, // Relatively prime (GCD = 1)
            {0, 5},   // Not relatively prime (GCD = 5)
```

```
        {1, -1},  // Relatively prime (GCD = 1)
        {-12, 15} // Relatively prime (GCD = 3, so not relatively prime)
    };

    // Test and output the results
    for (long[] testCase : testCases) {
        long a = testCase[0];
        long b = testCase[1];
        boolean result = isRelativelyPrime(a, b);
        System.out.println("Are " + a + " and " + b + " relatively prime? " + result);
    }
  }
}
```

Problem 8.3, Stephens page 199

Testing technique used is a mix of black-box and gray-box testing

- **Black-box testing**
    - The program was tested with various input pairs (e.g., (8, 9), (21, 35), (14, 25)) to determine if the integers were relatively prime. In black-box testing, we do not consider the internal workings of the program (such as the GCD calculation), but only focus on providing input and observing the output
    - This testing is based on the functional requirements: the program should correctly determine if two numbers are relatively prime. The test cases were chosen without looking at how the program calculates the result, but based on the expected outcome (whether the two numbers are relatively prime)
- **Gray-box testing**

- In gray-box testing, some knowledge of the internal workings of the system is used, but not all. In this case, we have a basic understanding that the isRelativelyPrime() function calculates the GCD, but we focus on testing it from an external perspective (input-output behavior).
- While testing the GCD function itself, you may want to consider how the algorithm works under different scenarios (e.g., small vs. large numbers), ensuring that the implementation is efficient, even though we aren't directly focusing on code coverage.

Techniques that could be used

1. **Black-box Testing**:

    - The program was tested with various input pairs (e.g., (8, 9), (21, 35), (14, 25)) to determine if the integers were relatively prime. In black-box testing, we do not consider the internal workings of the program (such as the GCD calculation), but only focus on providing input and observing the output.
    - This testing is based on the functional requirements: the program should correctly determine if two numbers are relatively prime. The test cases were chosen without looking at how the program calculates the result, but based on the expected outcome (whether the two numbers are relatively prime).

2. **Gray-box Testing**:

    - In **gray-box testing**, some knowledge of the internal workings of the system is used, but not all. In this case, we have a basic understanding that the `isRelativelyPrime()` function calculates the GCD, but we focus on testing it from an external perspective (input-output behavior).

- While testing the GCD function itself, you may want to consider how the algorithm works under different scenarios (e.g., small vs. large numbers), ensuring that the implementation is efficient, even though we aren't directly focusing on code coverage.

Techniques that Could Be Used

1. **Exhaustive Testing**:
    - testing all possible input combinations. For relatively prime testing, exhaustive testing would involve testing all pairs of integers in the range of -1 million to 1 million. However, this approach is impractical due to the large input space.
    - useful when the input space is small and manageable, but for large ranges of integers (as in this case), exhaustive testing would be computationally expensive and impractical. It's better to use representative test cases to cover various scenarios.
2. **Black-box Testing**:
    - where we focus on the outputs (whether the two numbers are relatively prime) based on given inputs, without considering the internal workings of the algorithm.
    - useful when you want to test the functionality of a system without knowing its internals. For this program, you can test different input pairs to ensure the program behaves as expected.
3. **White-box Testing**:
    - involves testing the internal logic and structure of the program, such as ensuring that the GCD() function works correctly, checking the efficiency of the algorithm (e.g., handling edge cases or large inputs), and testing all possible paths through the code (e.g., checking how the function handles very small or very large numbers).
    - appropriate when you need to verify the correctness of the implementation of algorithms or the structure of the code. For example, you might write test cases to cover all possible

branches of the GCD() function, including handling large numbers or edge cases like (0, 0).

4. **Gray-box Testing**:
    - uses a combination of both black-box and white-box testing. It might involve testing the program with knowledge of the algorithm (such as knowing it uses the GCD algorithm), but without delving into the details of the implementation.
    - useful when you have partial knowledge of the internal workings of the program but still want to test from an external perspective. For example, you may want to test the performance of the algorithm by inputting very large numbers or edge cases to see if the program handles them efficiently without needing to directly inspect the code.

Technique justification

- **Black-box testing** was used in the program because we tested the functionality of the isRelativelyPrime() method without any concern for the implementation details, such as how the GCD is calculated.
- **White-box testing** would be helpful to test the implementation itself, such as the GCD() method. You would focus on the logic of the method, ensuring that it behaves as expected for edge cases (e.g., (0, 0) or negative numbers).
- **Gray-box testing** could be useful if you know about the internal algorithm but still want to focus on higher-level functional tests, including performance tests or verifying the handling of extreme values.
- **Exhaustive testing** would be useful in a case where the input space is small enough that every combination can be tested, but it's not feasible for this case due to the large range of possible integer inputs

Problem 8.5, Stephens page 199-200

No bugs were found in the implementation or test cases. The method behaves correctly according to the test cases, returning true when the numbers are relatively prime and false otherwise. The testing code provided a clear way to validate the implementation of the AreRelativelyPrime method. Testing the edge cases, such as when one or both numbers are zero or negative, helped ensure that the program handles all scenarios as expected. The results of the tests confirmed that the method correctly implements the logic for checking relative primality and handles all edge cases properly.

Problem 8.9, Stephens page 200

Exhaustive testing is considered black-box testing:

Black-box testing focuses on testing the functionality of a program based on its input and expected output, without any knowledge of its internal workings or code. The goal is to validate that the program behaves as expected under various conditions, regardless of how it achieves the results internally. Exhaustive testing, which involves testing all possible input combinations to ensure that every possible scenario is covered, aligns with black-box testing because you are focused purely on the program's behavior based on input-output pairs. The objective is to ensure that for every possible input, the output matches the expected result.

Problem 8.11, Stephens page 200

Step 1: Identify the Sets of Bugs Found by Each Tester

From the problem, we know the bugs found by each tester:

- Alice found the bugs: {1, 2, 3, 4, 5}
- Bob found the bugs: {2, 5, 6, 7}
- Carmen found the bugs: {1, 2, 8, 9, 10}

Step 2: Calculate the Overlap Between Testers

We need to find the overlap in bugs between different testers:

- m12: The overlap between Alice and Bob, i.e., the bugs found by both Alice and Bob:
    - Overlap: {2, 5} → m12 = 2
- m13: The overlap between Alice and Carmen, i.e., the bugs found by both Alice and Carmen:
    - Overlap: {1, 2} → m13 = 2
- m23: The overlap between Bob and Carmen, i.e., the bugs found by both Bob and Carmen:
    - Overlap: {2} → m23 = 1
- m123: The overlap between all three testers, i.e., the bugs found by Alice, Bob, and Carmen:
    - Overlap: {2} → m123 = 1

Step 3: Estimate the Total Number of Bugs Using the Lincoln Index

Using the formula for three testers:

$$T = \frac{(m_1+1)(m_2+1)(m_3+1)}{(m_{12}+1)(m_{13}+1)(m_{23}+1)} - 1$$

Where:

- m1 = 5 (Alice found 5 bugs),
- m2 = 4 (Bob found 4 bugs),
- m3 = 5 (Carmen found 5 bugs),
- m12 = 2 (Overlap between Alice and Bob),
- m13 = 2 (Overlap between Alice and Carmen),
- m23 = 1 (Overlap between Bob and Carmen).

The **estimated total number of bugs** in the system is **9**.

Step 4: Determine the Bugs Still at Large

To find the number of bugs still at large, we subtract the number of bugs discovered from the total estimated bugs:

- **Bugs discovered** = Alice's bugs ∪ Bob's bugs ∪ Carmen's bugs = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} (a total of 10 distinct bugs).
- **Estimated total number of bugs** = 9.

Since the estimate suggests there are only 9 bugs in total, but 10 distinct bugs were discovered, this indicates a discrepancy due to the overlap calculation. The Lincoln Index is only an estimate, and in this case, it suggests that the 10th bug is likely a result of over-reporting or some redundancy in the discovery process. Therefore, **there are no bugs still at large**, as the estimation of 9 bugs is lower than the 10 distinct bugs reported.

Problem 8.12, Stephens page 200

Formula of Lincoln estimate formula (m = number of defects found in common):

$$N = \frac{(n_1 \times n_2)}{m}$$

If m = 0, the formula becomes undefined since division by 0 is not possible

If no defects are found in common, it suggests that:

1. The testers are exploring completely different parts of the software.
2. The testing methods or approaches might be significantly different.
3. There could be a large number of defects in the system, making it unlikely for testers to overlap in their findings.

A lower bound for the number of defects can be estimated by simply taking the total number of unique defects found

$$N \geq n_1 + n_2$$

This is because, at minimum, the system contains at least as many defects as those already discovered. To improve the estimate, additional testing should be conducted, potentially using a different testing strategy to increase overlap and obtain a more reliable Lincoln estimate.