Assignment 3

Problem 7.1

```
// Calculates the greatest common divisor (GCD) of two numbers using Euclid's algorithm.
private long GCD(long a, long b) {
   // Loop until the remainder becomes zero
   for (;;) {
      // Compute the remainder of a divided by b
      long remainder = a % b;

      // If the remainder is zero, b is the GCD
      if (remainder == 0) return b;

      // Update a and b for the next iteration
      a = b;
      b = remainder;
   }
}
```

Problem 7.2

You might encounter poorly written comments—such as those that merely restate the code—under the following two conditions:

1. When the code is overly complex or unclear. In such cases, developers may add comments to explain what the code is doing. However, if the code were written more clearly, these explanatory comments might be unnecessary.

2. When comments are added after the code is written. This approach can lead to comments that simply describe the code line by line, rather than providing meaningful insights or explanations.

To avoid these issues, it's recommended to write clean, self-explanatory code and to use comments to clarify the intent or purpose behind complex logic, rather than to restate what the code is doing.

Problem 7.4

```
private long GCD(long a, long b) {
   // Offensive programming: check for invalid inputs
```

```
    if (a == 0 && b == 0) {
        throw new IllegalArgumentException("GCD is undefined when both numbers are zero.");
    }
    if (b == 0) {
        throw new IllegalArgumentException("Second argument (b) must not be zero.");
    }
    if (a < 0 || b < 0) {
        throw new IllegalArgumentException("Both arguments must be non-negative.");
    }

    // Euclid's algorithm
    for (;;) {
        long remainder = a % b;
        if (remainder == 0) return b;
        a = b;
        b = remainder;
    }
}
```

Problem 7.5

Absolutely, because it limits the amount of bugs early.

Problem 7.7

1. Prepare for the trip
2. Start the car
3. Navigate to the supermarket
4. Park the car
5. Turn off the car and exit

1. Prepare for the trip

- Make sure you have your driver’s license.
- Grab your wallet, reusable bags, and shopping list.
- Check that the car keys are with you.

- Ensure the car has enough fuel.

2. Start the car

- Unlock the car.
- Sit in the driver's seat and buckle your seatbelt.
- Insert the key or press the ignition button to start the engine.

3. Navigate to the supermarket

- Put the car in drive.
- Use GPS or follow familiar roads to the supermarket.
- Follow traffic rules and signals.
- Stay in the correct lane and use turn signals when needed.

4. Park the car

- Enter the supermarket parking lot safely.
- Find an available spot and signal before turning in.
- Park within the lines.
- Put the car in park and engage the parking brake if needed.

5. Turn off the car and exit

- Turn off the engine.
- Grab your keys, phone, and wallet.
- Exit the vehicle and lock it.

Problem 8.1

Pseudocode:

```
FUNCTION isRelativelyPrime(a, b)
    // Assume this function is already implemented efficiently
    // and returns TRUE if a and b are relatively prime

// Define a list of test cases (pairs of integers)
SET testCases = [
    (14, 15),
    (12, 18),
    (17, 31),
    (100, 101),
    (35, 49),
    (0, 1),
    (-25, 16),
    (-18, -35),
    (999983, 1000000)
]

FOR each (a, b) IN testCases DO
    PRINT "Testing: (", a, ", ", b, ")"
    IF isRelativelyPrime(a, b) THEN
        PRINT "→ They are relatively prime."
    ELSE
        PRINT "→ They are NOT relatively prime."
    ENDIF
    PRINT newline
```

Problem 8.3

Black-box testing

In this pseudocode, we used black-box testing because:

- We are treating `isRelativelyPrime(a, b)` as a "black box": we don't look at its internal code.
- We simply provide inputs (a, b) and check the outputs (true/false).
- We're focused on functionality — whether it gives correct results — without knowing how it’s implemented.

What other techniques could be used, and when?

1. Exhaustive testing

- Definition: Test *every possible input pair* in the domain.
- Could be used? Not practical here.
- Why not? There are over 4 trillion combinations between -1,000,000 and 1,000,000 — too many to test exhaustively.
- When it's useful: Only for very small input domains (e.g., testing all values from 1 to 10).

2. White-box testing

- Definition: Testing based on internal logic and structure of the code.
- Could be used? Yes, if you have access to the actual implementation of `isRelativelyPrime()`.
- When to use:
    - When you want to check branches, loops, and conditions inside the method.
    - Ensures code coverage (e.g., testing how the algorithm handles edge cases, like when one value is 0 or negative).

3. Gray-box testing

- Definition: A mix of black-box and white-box testing.
- Could be used? Yes, especially useful when:
    - You partially know the internal logic (e.g., it uses GCD).
    - You want to write smarter test cases (e.g., input where GCD is > 1, or known primes).
- When to use: When you have limited access to internals, like in API or library testing.

Problem 8.5

Python Version of `areRelativelyPrime()`

```
def gcd(a, b):
    a = abs(a)
    b = abs(b)

    # If either number is zero, return the other
    if a == 0:
        return b
    if b == 0:
        return a

    # Euclid's algorithm
    while True:
        remainder = a % b
        if remainder == 0:
            return b
        a, b = b, remainder



def are_relatively_prime(a, b):
    # Only 1 and -1 are relatively prime to 0
    if a == 0:
        return b == 1 or b == -1
```

```
    if b == 0:
        return a == 1 or a == -1

    return abs(gcd(a, b)) == 1
```

Did I find any bugs?

Yes — there were some small issues in the original C# version:

1. `Math.abs()` is incorrect in C# — it should be `Math.Abs()`.
2. The `GCD` method might return a negative value if the language allows (Python doesn't here because we use `abs()`), so checking for `gcd == -1` is safe but redundant once `abs(gcd) == 1` is used.
3. ## Did I get any benefit from the testing code?

Yes,

- The black-box test cases helped confirm correctness of the Python translation.
- The test set with edge cases like `(0, 1)`, `(1, 0)`, and `(0, 0)` was especially helpful.
- Seeing known relatively prime and non-prime pairs let me validate logic and assumptions, particularly with zero and negative numbers.

Problem 8.9

Exhaustive testing falls into black-box testing because black-box testing focuses on inputs and outputs without knowing how the code is implemented internally.

Exhaustive testing means testing all possible input combinations to ensure correct output for every case.

Since you're not using knowledge of how the code works inside — just checking if it gives the right result for every possible input — it fits the black-box approach.

Problem 8.11

Step 1: Count unique and overlapping bugs

Let's count:

- Total distinct bugs found = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} = 10 bugs
- Overlaps:
    - Alice ∩ Bob = {2, 5} → 2 bugs
    - Alice ∩ Carmen = {1, 2} → 2 bugs
    - Bob ∩ Carmen = {2} → 1 bug

Step 2: Estimate using pairwise Lincoln index

We'll estimate the total bug count using Alice and Bob:

- A = 5 (Alice)
- B = 4 (Bob)
- Overlap = 2

So:

nginx

CopyEdit

```
Estimate (Alice & Bob) = (5 × 4) / 2 = 20 / 2 = 10
```

Then try Alice and Carmen:

- A = 5
- C = 5

- Overlap = 2

nginx

CopyEdit

```
Estimate (Alice & Carmen) = (5 × 5) / 2 = 25 / 2 = 12.5
```

And Bob and Carmen:

- B = 4
- C = 5
- Overlap = 1

nginx

CopyEdit

```
Estimate (Bob & Carmen) = (4 × 5) / 1 = 20
```

Step 3: Average the estimates

markdown

CopyEdit

```
(10 + 12.5 + 20) / 3 = 42.5 / 3 ≈ **14.17**
```

We round to the nearest whole number:

Estimated total bugs = 14

Step 4: Bugs still at large?

We found 10 unique bugs, so:

Estimated bugs still at large = 14 – 10 = 4

Problem 8.12

...the denominator becomes zero (Overlap = 0), and you get:

A×B/0=undefined (or infinite)

- Mathematically, it means the estimate blows up — there's no meaningful upper-bound estimate of how many bugs exist.
- Practically, it implies that:
    - The bug space is large, and the testers may be exploring very different parts of the system.
    - Or the system has a lot of bugs, and the testers haven't exhausted the common ones yet.

This tells you that your sampling is too sparse or non-overlapping, and more overlap (or better coordination) is needed to make a solid estimate.

Lower bound = $|\{1, 2, 3, 4, 5, 6\}|$ = 6 bugs