Elijah Senior

1. The issue with the original GCD code's comments is that they merely describe what the code does rather than explaining the reasoning behind it. They do not provide any additional insight beyond what is already evident from the code itself.

// Implements Euclid's algorithm to compute the GCD.
// For more details, refer to en.wikipedia.org/wiki/Euclidean_algorithm.

2. The original GCD code likely has ineffective comments for two reasons. One possibility is that the programmer took a top-down design approach too far, explaining each part of the code in excessive detail. Although this approach is generally considered good practice, it often results in redundant comments that simply repeat what the code already makes clear.

3. The validation code for Exercise 3 is already quite strict. It checks both the inputs and the result, and if any issue arises, the Debug.Assert method triggers an exception.

4. While you could add error handling to the GCD method, it's better to let the calling code manage any errors. As it stands, any exceptions thrown by the method are propagated to the caller, allowing them to be handled there. This eliminates the need for additional error handling within the method itself.

5.

a. Locate the car.
b. Unlock the car and get inside.
c. Start the engine.
d. Reverse out of the parking space carefully.
e. Turn right while exiting the parking lot and drive to the first intersection.
f. Take a left turn and continue straight until the road ends.
g. Turn right and drive until you reach a traffic light.
h. At the light, turn left and continue driving until the supermarket is visible.
i. Enter the supermarket parking lot.
j. Look for an available parking space and pull into it.

k. Turn off the car and step out.
l. Head inside and grab some snacks, maybe some cinamon buns.

Assumptions:
You properly adjust the seat and mirrors when you enter the car.

There's gas in the car.

You know how to drive.

There's nothing behind/infront the car when you pull out/drive. No cars, people, dogs, trees, or other objects jump in the way.

6.

```
def validate_are_relatively_prime(a: int, b: int) -> bool:
    # Use positive values
    a, b = abs(a), abs(b)

    # If either value is 1, return True
    if a == 1 or b == 1:
        return True

    # If either value is 0, return False
    # (Only 1 and -1 are relatively prime to 0.)
    if a == 0 or b == 0:
        return False

    # Loop from 2 to the smaller of a and b looking for common factors
    min_value = min(a, b)
    for factor in range(2, min_value + 1):
        if a % factor == 0 and b % factor == 0:
            return False

    return True
```

7.

Since Exercise 1 does not specify how the AreRelativelyPrime method functions internally, it qualifies as a black-box test.

If I were to explain how the method works, you could then design both white-box and gray-box tests.

Attempting an exhaustive test would be impractical due to the vast number of possible input pairs. With values ranging from -1 million to 1 million, there would be roughly 4 trillion combinations to test—far too many to handle efficiently. However, if the range were limited to -1,000 to 1,000, the number of possible pairs would be closer to 1 million, making comprehensive testing more feasible.

8. While developing this program, I encountered some issues with the AreRelativelyPrime method. The initial version lacked restrictions on the values of a and b, which caused problems when handling the largest and smallest possible integers. This led me to impose limits on the allowed values—a common outcome of thorough testing.

Aside from that, the AreRelativelyPrime method functioned well. However, fine-tuning the ValidateAreRelativelyPrime method and the test-breaking code required some effort to get everything working as intended. It wasn't particularly difficult, but it did make me think carefully about edge cases involving -1, 0, and 1. This highlights another advantage of writing tests: they encourage deeper consideration of unusual values that could potentially cause issues in the program.

9. Exhaustive tests are considered black-box tests since they do not depend on any understanding of the internal workings of the method being tested.

10. You can calculate three different Lincoln indexes using each pair of testers:

- Alice/Bob: $5 \times 4 \div 2 = 10$
- Alice/Carmen: $5 \times 5 \div 2 = 12.5$
- Bob/Carmen: $4 \times 5 \div 1 = 20$

One option is to take the average of these three values to get an estimated total of $(10 + 12.5 + 20) \div 3 \approx 14$ bugs. Alternatively, you could plan for the worst-case scenario and assume there are 20 bugs. In either case, it's important to continue monitoring the number of bugs found, so you can adjust your estimate as more information becomes available.

11. If the testers don't find any bugs in common, the Lincoln index formula would involve division by 0, resulting in an infinite value. This indicates that there's no reliable estimate of the total number of bugs.

To get a lower bound for the bug count, you can assume that the testers found at least one bug in common. For instance, if one tester found 5 bugs and another found 6, the lower bound estimate for the total number of bugs would be $(5 \times 6) \div 1 = 30$ bugs.