Carter Esparza

Assignment #3

7.1

```
private long GCD(long a, long b) {
   // Repeat until remainder is zero.
   while (true) {
      // Calculate the remainder of a divided by b.
      long remainder = a % b;
      // If the remainder is 0, then b is the GCD; return b.
      if (remainder == 0) {
         return b;
      }
      // Set a to b and b to remainder, then continue.
      a = b;
      b = remainder;
   }
}
```

Here the comments give actual meaning behind the code, ensuring that both future programmers and readers of the code are aware of each step.

7.2

1. When the comments simply restate what the code does in a redundant or trivial way, adding little value for someone reading the code.

2. When the programmer fails to update the comments after changing the code, resulting in misleading or incorrect commentary.

7.4

There is not really a need for extra offensive programming checks in the function as the necessary checkers are already in place for the inputs and result.

7.5

I do not believe so, as the method already allows for the calling function to handle the errors.

7.7

1. Enter the Car and Prepare to Drive: Ensure you have your keys, seatbelt, and necessary personal items. Start the car.
2. Determine Your Location: Use a GPS or map application to establish your current location.
3. Select the Nearest Supermarket: Locate the nearest supermarket on your map application or based on local knowledge.
4. Plan Your Route: Program the GPS with the supermarket address, and review the route, noting any major turns or highways.
5. Drive to the Supermarket: Follow the directions provided, obeying traffic signals and speed limits.
6. Arrive and Park: Arrive at the supermarket, find a parking spot, and turn off the car.

Assumptions made:

- The car is in good working condition with a full fuel tank.
- The GPS or map application used is accurate and up-to-date.

- Traffic conditions are normal, and no unexpected events occur during the trip.

8.1

```
import math


def isRelativelyPrime(a, b):
    """
    Returns True if integers a and b are relatively prime.
    According to the definition, if either integer is 0, only 1 and -1 are considered relatively prime to 0.
    """
    # Handle the special case where one value is 0.
    if a == 0 or b == 0:
        return (abs(a) == 1 or abs(b) == 1)

    # Compute the greatest common divisor using math.gcd (which returns a non-negative integer).
    return math.gcd(a, b) == 1


# Testing the isRelativelyPrime() method with a set of test cases.
def test_isRelativelyPrime():
    test_cases = [
        (8, 9, True),        # Relatively prime: only common factor is 1.
        (84, 36, False),     # Not relatively prime: common factor is 12.
```

```
        (0, 1, True),        # 0 is relatively prime to 1.
        (0, -1, True),       # 0 is relatively prime to -1.
        (0, 2, False),       # 0 is not relatively prime to 2.
        (-12, 5, True),      # -12 and 5 are relatively prime.
        (27, 36, False)      # 27 and 36 are not relatively prime (common factor 9).
    ]

    for a, b, expected in test_cases:
        result = isRelativelyPrime(a, b)
        print(f"isRelativelyPrime({a}, {b}) = {result} (expected: {expected})")

if __name__ == "__main__":
    test_isRelativelyPrime()
```

8.3

For the program in Exercise 8.1, I primarily used black-box testing, where the focus was on providing various input values and verifying that the output met the expected results without considering the internal workings of the function.

Additionally, white-box testing could be applied to ensure that all branches of the Euclidean algorithm are executed, while gray-box testing would combine both approaches.

8.5

```
import math

def gcd(a, b):
```

```
    """
    Return the greatest common divisor of a and b using Euclid's algorithm.
    Both a and b are converted to non-negative values.
    """
    a = abs(a)
    b = abs(b)
    if a == 0:
        return b
    if b == 0:
        return a
    while True:
        remainder = a % b
        if remainder == 0:
            return b
        a, b = b, remainder


def areRelativelyPrime(a, b):
    """
    Return True if integers a and b are relatively prime.
    Only 1 and -1 are relatively prime to 0, as defined.
    """
    # Handle the special case of 0.
    if a == 0:
```

```
        return abs(b) == 1
    if b == 0:
        return abs(a) == 1
    # Use the gcd function to determine if the gcd is 1.
    return gcd(a, b) == 1

def test_areRelativelyPrime():
    test_cases = [
        (8, 9, True),
        (84, 36, False),
        (0, 1, True),
        (0, -1, True),
        (0, 2, False),
        (-12, 5, True),
        (27, 36, False)
    ]

    for a, b, expected in test_cases:
        result = areRelativelyPrime(a, b)
        print(f"areRelativelyPrime({a}, {b}) = {result} (expected: {expected})")

if __name__ == "__main__":
    test_areRelativelyPrime()
```

8.9

Exhaustive testing is considered a form of black-box testing because it involves providing every possible valid input (or a very comprehensive sample of inputs) to the system without regard for the internal structure of the code. In exhaustive testing, the system is treated as a "black box" where only input and output are examined.

8.11

Alice: {1, 2, 3, 4, 5} (5 bugs)

Bob: {2, 5, 6, 7} (4 bugs)

Carmen: {1, 2, 8, 9, 10} (5 bugs)

Alice and Bob:

- Common bugs: {2, 5} (m = 2)
- Estimate = (5 × 4) / 2 = 10

Alice and Carmen:

- Common bugs: {1, 2} (m = 2)
- Estimate = (5 × 5) / 2 = 12.5, which we can approximate as 13

Bob and Carmen:

- Common bugs: {2} (m = 1)
- Estimate = (4 × 5) / 1 = 20

Rounded, we estimate about 14 total bugs, which shows about 4 are still uncovered.

8.12

If two testers do not find any bugs in common (i.e., the intersection is zero), the Lincoln index formula, which divides by the number of common bugs, becomes undefined. This situation implies that the samples are completely disjoint, which in turn suggests that there might be a

very high number of bugs overall or that the sampling process was not representative. However, a lower-bound estimate of the total number of bugs would be at least the sum of the bugs found by both testers, because no bug was counted twice.