Problem 7.1, Stephens page 169

The greatest common divisor (GCD) of two integers is the largest integer that evenly divides them both. For example, the GCD of 84 and 36 is 12, because 12 is the largest integer that evenly divides both 84 and 36. You can learn more about the GCD and the Euclidean algorithm, which you can find at en.wikipedia.org/wiki/Euclidean_algorithm. (Don't worry about the code if you can't understand it. Just focus on the comments.)(Hint: It should take you only a few seconds to fix these comments. Don't make a career out of it.)

```
        // Use Euclid's algorithm to calculate the GCD.
         private long GCD( long a, long b )
         {
            // Get the absolute value of a and b
            a = Math.abs( a );
            b = Math.abs( b );

            //Repeat until we're done
            for( ; ; )
            {
               // Set remainder to the remainder of a / b
               long remainder = a % b;
               // If remainder is 0, we're done.  Return b.
               If( remainder == 0 ) return b;
               // Set a = b and b = remainder.
               a = b;
               b = remainder;
            };
         }
```

The comments aren’t as helpful as they could be by describing why the code does what it does or explaining how the algorithm works. The code is only stating what the code does which is pretty obvious.

Problem 7.2, Stephens page 170

Under what two conditions might you end up with the bad comments shown in the previous code?

1. If the person added the comments after the code was written

2. If the person based their code off a top-down design that already broke down the algorithm into smaller pieces and explained the “whys”

Problem 7.4, Stephens page 170

How could you apply offensive programming to the modified code you wrote for exercise 3? [Yes, I know that problem wasn't assigned, but if you take a look at it you can still do this exercise.]

It could be applied, as inputs and results are validated, and an exception error is thrown if it discovers an issue.

Problem 7.5, Stephens page 170

Should you add error handling to the modified code you wrote for Exercise 4?

Error handling does not need to be added as the calling code should already handle it.

Problem 7.7, Stephens page 170

Using top-down design, write the highest level of instructions that you would use to tell someone how to drive your car to the nearest supermarket. (Keep it at a very high level.) List any assumptions you make.

1. Go to car
2. Open car door
3. Start car
4. Pull out of parking spot
5. Drive out of parking lot
6. Turn out of parking lot
7. Drive in the direction of the nearest supermarket
8. Stop at stop signs or any stop lights
9. Turn into supermarket
10. Find parking spot
11. Turn into parking spot
12. Turn car off

13. Get out of car

Assumptions:

- Car is backed into parking spot
- Car has gas
- Driver knows how to drive, has license
- Driver knows how to use car (start, stop, adjust seat, adjust mirror)
- Driver stops for pedestrians and drives safely
- Supermarket has open parking lot
- Supermarket is open

Problem 8.1, Stephens page 199

Two integers are *relatively prime* (or *coprime*) if they have no common factors other than 1. For example, 21 = 3 X 7 and 35 = 5 X 7 are *not* relatively prime because they are both divisible by 7. By definition -1 and 1 are relatively prime to every integer, and they are the only numbers relatively prime to 0.

Suppose you've written an efficient `IsRelativelyPrime` method that takes two integers between -1 million and 1 million as parameters and returns `true` if they are relatively prime. Use either your favorite programming language or pseudocode (English that sort of looks like code) to write a method that tests the `IsRelativelyPrime` method. (Hint: You may find it useful to write another method that also tests two integers to see if they are relatively prime.)

```
import math


# Function to check if two integers are relatively prime
def IsRelativelyPrime(a, b):
    return math.gcd(a, b) == 1


# Function to test the IsRelativelyPrime method
def TestIsRelativelyPrime():
```

```
    # Test cases
    test_cases = [
        (21, 35),  # Not relatively prime
        (17, 31),  # Relatively prime
        (0, 7),    # Not relatively prime (0 is not relatively prime to any integer)
        (-1, 100), # Relatively prime (-1 is relatively prime to every integer)
    ]

    # Iterate through test cases
    for a, b in test_cases:
        result = IsRelativelyPrime(a, b)
        if result:
            print(f"{a} and {b} are relatively prime.")
        else:
            print(f"{a} and {b} are not relatively prime.")

# Call the test method
TestIsRelativelyPrime()
```

Problem 8.3, Stephens page 199

What testing techniques did you use to write the test method in Exercise 1? (Exhaustive, black-box, white-box, or gray-box?) Which ones *could* you use and under what circumstances? [Please justify your answer with a short paragraph to explain.]

The testing technique used for Exercise 1 was black-box. White-box or gray-box testing could be implemented if it was said how the AreRelativelyPrime method

works. If an exhaustive test were to be attempted, the value ranges would have to be high in the negative/positive millions, or thousands ranges depending of if millions is too large for all pairs of values to be tested.

Problem 8.5, Stephens page 199 - 200

the following code shows a C# version of the **AreRelativelyPrime** method and the **GCD** method it calls.

```
        // Return true if a and b are relatively prime.
         private bool AreRelativelyPrime( int a, int b )
         {
            // Only 1 and -1 are relatively prime to 0.
            if( a == 0 ) return ((b == 1) || (b == -1));
            if( b == 0 ) return ((a == 1) || (a == -1));

            int gcd = GCD( a, b );
            return ((gcd == 1) || (gcd == -1));
         }

         // Use Euclid's algorithm to calculate the
         // greatest common divisor (GCD) of two numbers.
         // See https://en.wikipedia.org/wiki/Euclidean_algorighm
         private int GCD( int a, int b )
         {
            a = Math.abs( a );
            b = Math.abs( b );

            // if a or b is 0, return the other value.
            if( a == 0 ) return b;
            if( b == 0 ) return a;

            for( ; ; )
            {
               int remainder = a % b;
               if( remainder == 0 ) return b;
               a = b;
               b = remainder;
            };
         }
```

The **AreRelativelyPrime** method checks whether either value is 0. Only -1 and 1 are relatively prime to 0, so if a or b is 0, the method returns **true** only if the other value is -1 or 1.

The code then calls the `GCD` method to get the greatest common divisor of `a` and `b`. If the greatest common divisor is -1 or 1, the values are relatively prime, so the method returns `true`. Otherwise, the method returns `false`.

Now that you know how the method works, implement it and your testing code in your favorite programming language. Did you find any bugs in your initial version of the method or in the testing code? Did you get any benefit from the testing code?

There weren't any large bugs or errors, but when trying to input larger values, the program couldn't handle it, so setting value limits is helpful to solve the issue. There seems to always be a benefit to testing code as it brings about errors that you didn't realize were there, or possible ways to improve upon what's already there.

Problem 8.9, Stephens page 200

Exhaustive testing actually falls into one of the categories black-box, white-box, or gray-box. Which one is it and why?

It is in the black-box category because it doesn't rely on information from the method that is currently being tested, only the results.

Problem 8.11, Stephens page 200

Suppose you have three testers: Alice, Bob, and Carmen. You assign numbers to the bugs so the testers find the sets of bugs {1, 2, 3, 4, 5}, {2, 5, 6, 7}, and {1, 2, 8, 9, 10}. How can you use the Lincoln index to estimate the total number of bugs? How many bugs are still at large?

Use each pair to calculate three different Lincoln indexes.

Alice and Bob: 10 (5*4 / 2)

Alice and Carmen: 12.5 (5*5 / 2)

Bob and Carmen: 20 (4*5 / 1)

Then you can take an average of the three to get 14 bugs. You could also plan for 20 bugs because that will be the max amount of bugs found.

Problem 8.12, Stephens page 200

What happens to the Lincoln estimate if the two testers don't find any bugs in common? What does it mean? Can you get a "lower bound" estimate of the number of bugs?

If there are no bugs in common, then the equation will always be divided by 0, giving an infinite answer, therefore meaning no conclusions can be drawn about the number of bugs. No, you can’t get a lower bound if none are found in common, the closest answer would be to divide everything by 1, which still isnt accurate.