Homework #3

Problem 7.1: The 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.

```
    // Use Euclid's algorithm to calculate the GCD.
    provate long GCD( long a, long 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;
       };
    }
```

```
// Use Euclid's algorithm to calculate the GCD
// Read this article:
// https://en.wikipedia.org/wiki/Euclidean_algorithm
private long GCD (long a, long b) {
     // find the absolute values of a and b
     a = Math.Abs(a);
     b = Math.Abs(b);

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

Problem 7.2: According to your textbook, under what two conditions might you end up with the bad comments shown in the previous code?

In the previous code, one might end up with those bad comments if they added the comments after writing the entire code or if they wrote "just barely good enough" or JBGE comments. Another condition that would result in the bad comments in the above code is when

using a top-down design in which the code is described in excruciating detail with redundant comments that are not necessary or helpful.

Problem 7.4: How could you apply offensive programming to the modified code you wrote for exercise 3?

The code written for Exercise 3 already applies offensive programming by validating the inputs and the result. In addition, the Debug.Assert method will throw an exception if it sees an issue.

Problem 7.5: Should you add error handling to the modified code you wrote for Exercise 4? Explain your reasoning.

Typically, it is best to write error handling before writing the code. For the modified code for Exercise 4, it is not necessary to add any error handling because any exceptions that are thrown are passed up to the calling code to be handled there.

Problem 7.7: 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.

- Grab the car keys
- Unlock the car
- Get inside the car and put your seatbelt on
- Start the car
- Back out of the driveway and drive to the left
- At the second stop, turn right
- After the third stop sign, turn left into the parking lot
- Park the car in one of the open spots
- Get out of the car
- Walk into the supermarket
- Buy some groceries!

Problem 8.1: … Use either your favorite programming language or pseudocode to write a program that tests the isRelativelyPrime() method.

```
For i from 1 to 1,000:
     a = random integer
     b = random integer
     Assert isRelativelyPrime(a, b) ==
     validate_IsRelativelyPrime(a, b)
```

```
For i from 1 to 1,000:
     a = random integer
     Assert isRelativelyPrime(a, a) ==
     validate_IsRelativelyPrime(a, a)

For i from 1 t 1,000:
     a = random integer (excluding -1 and 1)
     Assert isRelativelyPrime(a, 0) == True
     Assert isRelativelyPrime(0, a) == True

For i from 1 to 1,000:
     a = random integer
     Assert isRelativelyPrime(a, -1) == True
     Assert isRelativelyPrime(a, 1) == True
     Assert isRelativelyPrime(-1, a) == True
     Assert isRelativelyPrime(1, a) == True

For i from 1 to 1,000:
     a = random integer
     Assert isRelativelyPrime(a, -1,000,000) ==
     validate_IsRelativelyPrime(a, -1,000,000)
     Assert isRelativelyPrime(a, 1,000,000) ==
     validate_IsRelativelyPrime(a, 1,000,000)
     Assert isRelativelyPrime(-1,000,000, a) ==
     validate_IsRelativelyPrime(-1,000,000, a)
     Assert isRelativelyPrime(1,000,000, a) ==
     validate_IsRelativelyPrime(1,000,000, a)
```

Problem 8.3:

a. What testing techniques did you use for the program in Exercise 8.1?

For the program in Exercise 1, I used gray-box testing, so I used a little bit of both white-box and black-box testing. I did not use exhaustive testing, since I allowed the variables a and b to range from -1,000,000 to 1,000,000 which has too many possible inputs.

b. Which ones could you use and under what circumstances?

If the range of values for the variables a and b was smaller, I could have used exhaustive testing to test the isRelativelyPrime() method.

Problem 8.5: …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?

Yes, I did find a few bugs in my initial version of the method as well as in the testing pseudocode that I created. The initial version of the method had some difficulty handling the minimum and maximum integer values that were possible.

Problem 8.9: Exhaustive testing actually falls into one of the categories black-box, white-box, or gray-box. Which one is it and why?

In black-box testing, a method is viewed as a black box that you can't look inside of. You know what the method is supposed to do, but do not know how it works. Exhaustive testing tests methods with every possible input. This type of testing falls into black-box testing since the method is also tested by sending it numerous random inputs and does not rely on knowing what happens in the method being tested.

Problem 8.11: … How can you use the Lincoln index to estimate the total number of bugs? How many bugs are still at large?

Alice and Bob: 5 x 4 / 2 = 10
Bob and Carmen: 4 x 5 / 1 = 20
Alice and Carmen: 5 x 5 / 2 = 12.5
I can use all the pairs of individuals to find the three main Lincoln indexes. Once I have these, I can take the average of the three Lincoln indexes to estimate the total number of bugs.
(10 + 20 + 12.5) / 3 = 14.167
This shows that there are still around 14 or 15 bugs.

Problem 8.12: 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 the two testers do not find any bugs in common, the Lincoln estimate is divided by zero which provides an infinite answer. This means that the testers do not know how many bugs there are in the code. It is possible to get a "lower bound" estimate of the number of bugs by saying that the testers found 1 common bug even though no bugs were found in common.