Marissa Alvarez
CMSI 402
3/19/19

Assignment 3

7.1.
It is essential to be clear and straight to the point in commenting your code.

```
//Get the absolute value of a and b
a =  Math.Abs(a);
b =  Math.Abs(b);

//Use Euclid's algorithm to calculate the GCD
// Check out en.wikipedia.org/wiki/Euclidean_algorithm private long GCD(long a, long b)
{
    a =  Math.Abs(a);
    b =  Math.Abs(b);

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

};
```

7.2. One might end up with bad comments if the individual who is writing the code does not understand the process thoroughly. Also, the individual could be in a rush to complete the assignment.

7.4.
The code is very offensive because it validates the inputs and the result, and the Debug.Assert method throws an exception if there is a problem.

7.5.
One could add error handling code to the GCD method, but one will want to have the code to handle errors. Although, one will not need to add error handling code if it throws exceptions.

7.7.
Go to the parked car located in front of the house
Unlock the drivers car door
Open the car door
Enter the car
Enter the closest grocery store address to the GPS
Start the car
Put the gear shift into reverse
Back out of the parking spot
Change the gear shift to drive
Begin driving to your destination listening to the GPS instructions
Arrive at your destination
Find a parking spot
Change the gear shift to park
Turn off the car
Exit the car and close door
Enter the grocery store
Purchase anything you would like.
Assumptions:
The driver knows how to drive
Knows how to use a GPS
Good driving experience
New to the town
The market is open
There are parking spots at the grocery store

8.1.
First, it is important to pass inputs to both methods to verify that they agree.
For example, Validate _ AreRelativelyPrime method that uses an algorithm for determining whether two values are relatively prime.

```
// Return true if a and b are relatively prime.
// This is a test method that is less efficient than
// AreRelativelyPrime and is used only to validate that method. private bool
Validate_AreRelativelyPrime(int a, int b)
{
// Use positive values. a = Math.Abs(a);
b = Math.Abs(b);
```

```
// If either value is 1, return true. if ((a == 1) || (b == 1)) return true;
// If either value is 0, return false.
// (Only 1 and -1 are relatively prime to 0.) if ((a == 0) || (b == 0)) return false;
// Loop from 2 to the smaller of a and b looking for factors. int min = Math.Min(a, b);
for (int factor = 2; factor <= min; factor++)
{
if ((a % factor == 0) && (b % factor == 0)) return false; }
return true; }
```

Next, I tested the original AreRelativelyPrime method.

For 1,000 trials, pick random a and b and: Assert AreRelativelyPrime(a, b) = Validate_AreRelativelyPrime(a, b)
For 1,000 trials, pick random a and: Assert AreRelativelyPrime(a, a) = Validate_AreRelativelyPrime(a, a)
For 1,000 trials, pick random a and:
Assert AreRelativelyPrime(a, 1) relatively prime Assert AreRelativelyPrime(a, -1) relatively prime
Assert AreRelativelyPrime(1, a) relatively prime Assert AreRelativelyPrime(-1, a) relatively prime
For 1,000 trials, pick random a (not 1 or -1) and: Assert AreRelativelyPrime(a, 0) relatively prime
Assert AreRelativelyPrime(0, a) relatively prime
For 1,000 trials, pick random a and:
Assert AreRelativelyPrime(a, -1,000,000) = Validate_AreRelativelyPrime(a, -1,000,000) Assert AreRelativelyPrime(a, 1,000,000) = Validate_AreRelativelyPrime(a, 1,000,000) Assert AreRelativelyPrime(-1,000,000, a) = Validate_AreRelativelyPrime(-1,000,000, a) Assert AreRelativelyPrime(1,000,000, a) = Validate_AreRelativelyPrime(1,000,000, a)
Assert AreRelativelyPrime(-1,000,000, -1,000,000) = Validate_AreRelativelyPrime(-1,000,000, -1,000,000)
Assert AreRelativelyPrime(1,000,000, 1,000,000) = Validate_AreRelativelyPrime(1,000,000, 1,000,000)
Assert AreRelativelyPrime(-1,000,000, 1,000,000) = Validate_AreRelativelyPrime(-1,000,000, 1,000,000)
Assert AreRelativelyPrime(1,000,000, -1,000,000) = Validate_AreRelativelyPrime(1,000,000, -1,000,000)

The code verifies the method's results for pairs of random values, pairs of identical numbers, 1, –1, 0, and the smallest and largest values supported by the AreRelativelyPrime method –1 million and 1 million.

8.3.
Since it does not say that AreRelativelyPrime method works this can result in a black-box-test. If it does work one could write a white-box and gray-box test for it. Therefore, one could test the values from -1,000 - 1,000.

8.5.
The AreRelativelyPrime method works reasonably well but did need some tweaking. The values of a and b do not have restrictions. Therefore it was best to restrict the allowed values. The Validate_AreRelativelyPrime method was more tricky. It was important to write tests to check the special cases.

8.9.
Exhaustive tests are black box tests because they don't rely on the inside of the method that they are testing.

8.11.
Testing the three different Lincoln indexes:
Alice/Bob 5*4/2 =10
Alice/Carmen 5*5/2 = 12.5
Bob/Carmen 4*5/1 = 20

To estimate the bug you could take the average of the three. In this case, there are 14 bugs.

8.12.
If the testers don't find any bugs in common then and when you divide the indexes by zero, then you will be revealing an infinite result. This means that you have no idea how many bugs there are in your code. One can kind of get a lower bound for the number of bugs pretending the testers found one bug in common. For example, if the testers found 3 and 4 bugs, then the lower bound would be 3*4 /1 = 12 bugs!