Thomas O'Brien

CMSI 402

3/19/18

Homework Assignment #3

7.1 While none of the comments in the program are wrong, they are not very helpful at the end of the day. All the comments do is provide a literal explanation of what individual lines of code are doing. It would be more appropriate to provide a brief explanation of the logic behind the program. This would allow the user to quickly understand the purpose of the code and understand the implementation. Down below is an example of better comments.

```
// Here Euclid's algorithm is being used to calculate the GCD for two numbers. This algorithm
// uses a series of steps that uses the output of the previous step to be used as input for the
// next step. A basic example of this algorithm is as follows.
// a = q0b + r0
// b = q1r0 + r1
// r0 = q2r1 + r2
// This continues until a remainder of 0 is found.
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 A common mistake when writing comments for code is taking a top down approach while explaining the code. While this is commonly how people tend to program (and is good programming practice), this won't necessarily accurately capture the behavior of the program. Not only that, but this kind of habit can lead to comments that elaborate on the wrong parts of the program. Another common mistake would be when the programmer adds the comments after they have written the program. Writing comments like this leads to the habit of simply explaining what individual lines of code do, and not how the individual lines of code interactive with each other and create a working solution.

7.4 The best way to make the program more offensive is to add in a few safety checks. These safety checks can include ensuring that the input provided to the function is the valid data type that the function is capable of handling. This includes checking to see if the input uses positive numbers only. Not only that, but it would be a good idea to have another check to ensure that the result is the valid data type as well. The method **Debug.Assert()** should be used if any of these checks fail. By using this method, the function will throw a desired exception.

7.5 You could add error handling to the GCD method, but it is better to have error handling implemented as a part of the calling code. The GCD method should be restricted to solely the logic behind Euclid's algorithm. It is better programming practice to have the calling code be response for invalid data and handle any potential errors that might be thrown.

7.7 First, the user needs to locate the keys to the car that they are going to use to get to the super market. Next, the user must then locate then car and head to it. Once the car has been located, the user must use the keys to unlock the car and enter the driver's seat. Then the user must use the keys to turn the car on. If the car is low on gas, then the user must locate a gas station and fill the car up with gas. If not, then the user must find directions to the supermarket. This can be done by either a navigation application or my memory. Once this is done, the user must begin using their driving skills and head to the super market. Eventually when the user gets to the supermarket, they must find a parking spot and park in it. Then the user must get out of the car, lock the car, and walk inside the supermarket. Some assumptions being made here is that the user has a smartphone, has a driver’s license, easy access to a car, and money to spend on both gas and the supermarket.

8.1 The following is written in pseudo code. It would also be required to have a method that we know for sure works called isRelativelyPrimeWorking().

```
checkIsRelativelyPrime {
    for loop that runs 10000 times {
        a = Math.random(-1000000, 1000000)
        b = Math.random(-1000000, 1000000)
        assert(isRelativelyPrime(a, 1000000) == isRelativelyPrimeWorking(a, 1000000))
        assert(isRelativelyPrime(a, -1000000)==isRelativelyPrimeWorking(a, -1000000))
        assert(isRelativelyPrime(a, b) == isRelativelyPrimeWorking(a, b))
        assert(isRelativelyPrime(1000000, b) == isRelativelyPrimeWorking(1000000, b))
        assert(isRelativelyPrime(-1000000, b)==isRelativelyPrimeWorking(-1000000, b))
    }
}
```

8.3 The kind of testing I chose here would be black-box testing. Black-box testing is where you know what the function is meant to do, but you don’t know how it works. The function above essentially runs thousands of tests that specifically tests for the edge cases. If these tests are to pass, then it is likely that the program is working as desired.

8.5 I tried implementing the program in the language Python3. Since for loops are implemented differently in Python3, I had to use while loops instead (in reference to the line that contains the code **for (; ;)**). There were some bugs that I found when dealing with the edge cases. This included me forgetting to include the check for when a or b is equal to 0. Not only that, but in my GCD function, I had forgotten to get the absolute value of a. Having the

8.9 Exhaustive testing is a form of black box testing. The reason for this is due to exhaustive testing not relying on any previous knowledge of how the program was implemented in order to write the tests. This is the foundation of how black box testing is conducted.

8.11 First we need to calculate the Lincoln index of the 3 potential pairs. These values are 10, 20 and 12.5. After that, we can take the average of these values to get around 14. It is important to understand that given the sample size of this average, the actual bug count could vary significantly. It would be best practice to consider the standard deviation as well.

8.12 This would mean that the Lincoln index calculation would force a division by 0. Essentially in this scenario, the Lincoln index does not hold much value to the user. To get an artificial lower bound, the user could recalculate the Lincoln index, but pretend that there is at least one bug in common. While this may not be entirely accurate, it at least gives the user some kind of ball park to play around with.