Assignment #3

7.1)

The comments don’t really enhance the understanding of the code, rather *show* what the code does, it *tells* what the code does. Comments are not really required if the algorithm is explained to the reader.

       // Use Euclid's algorithm to calculate the GCD (see <https://en.wikipedia.org/wiki/Euclidean_algorighm> for more details on Euclid’s 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 condition where the programmer may have taken a top-down approach where the code is described in extreme detail that results in redundant information. And the second condition, by the opposite where the programmer added the comments after writing the code. In this case, it’s easy to just say what each line of code does and not why it is doing it.

7.4)

Offensive programming may be applied to validate the inputs of a and b by asserting them to be greater than 0. By using offensive programming, you can easily locate the source of error and make appropriate corrections.

7.5)

With error handling, it takes care of certain cases that would cause the code to crash. Whenever the program is vulnerable to the type of cases it receives, error handling should be implemented.

7.7)*.*

1. Drive West until you reach La Brea.
2. Make a left turn on La Brea.
3. Drive straight until you reach market on your left side.
4. Turn left into market’s parking lot.

8.1)

bool IsRelativelyPrimeTest(){

int n1 = some\_number;

int n2 = some\_other\_number;

int constraint = 1000000;

// Test when both values are the same

if(!IsRelativelyPrime(n1,n1))

// Success

} else {

// Error

}

// Test where parameters exceed constraints

if(n1 > constraint || n1 < -constraint || n2 > contraint || n2 < -constraint){

// throw invalid integer exception

}

// Loop through random values to ensure functionality correctness

for(int i =0; i < 100; i ++){  
 int a = Random.Range(-1000000, 1000000);

int b = Random.Range(-1000000, 1000000);

IsRelativelyPrime(a, b)

}

}

8.3).

For the last exercise, I used the black-box technique. Since the implementation of the function was assumed to be given and correct, the test I curated was focused on exception handling and defensive programming. However had I known the source code, other techniques such as white-box testing would’ve also been used to ensure accuracy of answers as well.

8.5)

The testing code does not take into account the idea and robustness of the code in 8.1. The test exceeds in testing constraints and validity but not accuracy.

8.9)

Exhaustive tests fall under black-boxes because they are used independent to how a method or function is implemented.

8.11)

Possible combinations are:

A and B: (5x4)/2 = 10

A and C: (5x5)/2 = 12.5

B and C: (4x5)/1 = 20

By taking the average, total number of bugs is approximately 14, leaving 4 unknown bugs.

8.12)

If testers do not find any bugs in common, then essentially, the Lincoln index equation uses division by 0, which is basically infinite therefore there’s no idea how many bugs exist.

To calculate a lower bound, you must assume that the testers found 1 bug in common. So, dividing by one gives the lower bound.