George Sarantinos
3/19/18
CMSI 402
Homework #3

7.1:

```
// 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 GCD is found.
   for ( ;  ;  )
   {
      long  remainder = a % b;
      if (remainder == 0) return b;
      a = b;
      b = remainder;
   };
}
```

7.2:

One condition is caused by the developer trying to comment while writing the code. Due to the fact that the code is revised so many times, the comments have to be revised as well. Overtime, the developer maybe did not update the comments to the best of their abilities, which lead to bad comments. The next condition is caused when the developer tries to write all of the code, and then add the comments when they are finished. Unfortunately, when it comes to adding the comments, the developer will often add the bare minimum amount of comments and/or lazy comments. Therefore, the comments may not fully explain what the code should do and/or explain poorly.

7.4:

```
// 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 GCD is found.
   checked
   {
      for ( ;  ;  )
      {
         long  remainder = a % b;
         if (remainder == 0) return b;
         a = b;
         b = remainder;
      };
   }
}
```

7.5:

No, you do not need to add error handling. This is because the checked block, which I added for 7.4, already throws an exception if there is an integer overflow. Furthermore, the parameters must be longs, so there is no need to check if the input is anything but a number.

7.7:

*Assuming the person knows how to drive a car.
1. Obtain the key to the car
*Assuming the car is locked.
2. Unlock the car
3. Get into the driver's seat.
4. Buckle your seat belt.
5. Put the key on the ignition
6. Turn on the car.
7. Put the car in drive.
8. Take off the parking brake.
*Assuming the driver knows how to get to the nearest supermarket.
9. Drive to the nearest supermarket.

10. Park in a parking spot.
11. Put on the parking brake.
12. Put the car in park.
13. Turn off the car

8.1:

```
function  RelativelyPrimeMethod(int a, int b) {

   //pseudo code
   if (a and b are relatively prime) {
      return true;
   }

   return false;
}


function test1(int a, int b) {

   if (IsRelativelyPrime(a, b) == RelativelyPrimeMethod(a, b)) {
      return true;
   }

   return false;
}
```

8.3:

I used white-box testing for the test method in Exercise 1. This is because according to the problem, I wrote the method; therefore, I know how the method works. I couldn't use exhaustive testing because it is not feasible to test all the pairs of numbers from -1 million to 1 million. Furthermore, I couldn't use black-box testing because I have complete knowledge about how the method works. For this same reason, I did not use gray-box testing either. If I didn't know how the method works, I would have used black-box testing. In addition, if I had some knowledge about the method, I would have used gray-box testing. Finally, if the range of integers that the IsRelativelyPrime method could take in were smaller, exhaustive testing would have worked to test all possible inputs.

8.5:

```
//JavaScript

function AreRelativelyPrime(a, b) {
  if(a == 0 && (b == -1 || b == 1)) {
    console.log("true")
    return true;
  }
  if(b == 0 && (a == -1 || a == 1)) {
    console.log("true")
    return true;
  }

  if(a == 0 && b == 0) {
    console.log("false");
    return false;
  }

  let gcd = GCD(a, b);

  if (gcd == 1 || gcd == -1) {
    console.log("true");
    return true;
  }

  console.log("false")
  return false;
}

function GCD(a, b) {
  a = Math.abs(a);
  b = Math.abs(b);

  while(true) {
    let remainder = a % b;
    if (remainder == 0) {
      return b;
    }
    a = b;
    b = remainder
  }
}

AreRelativelyPrime(7, 5);    //true
AreRelativelyPrime(0, 1);    //true
AreRelativelyPrime(-1, 0);    //true
AreRelativelyPrime(5, 7);    //true
AreRelativelyPrime(10, 5);    //false
AreRelativelyPrime(5, 10);    //false
AreRelativelyPrime(1002847, 937);    //true
AreRelativelyPrime(0, 0);    //false
```

One bug that I found is when the inputs are 0 and 0, which causes the program to infinite loop. To fix this I return false when 0 and 0 are the inputs.

8.9:

Exhaustive testing falls into black-box testing. This is because it is testing every possible input, which negates the need to know how the method works. Instead it is focusing only on the inputs and outputs.

8.11:

One can find the Lincoln index for each pair that can be made from Alice, Bob, and Carmen. After that, the Lincoln indexes are averaged. In this situation:

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

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

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

(10 + 12.5 + 20) / 3 = 14.16 -> about 14

So, there are still about 14 bugs at large.

8.12

If none of the testers find any of the same bugs, this means that S would be equal to 0. 0 cannot divide a number; therefore, the equation does not produce an answer. This means that no information about the remaining bugs is provided from the equation. To find a lower bound, one can simply, set S = 1 instead of 0. The answer that is returned from the equation is then the lower bound.