Jackson Watkins
Joey Martinez
John Goocher

402: Assignment 3

7.1

```
// Use Euclid's algorithm to calculate the GCD.
         provate long GCD( long a, long b )
         {
            // Make sure there are no non-negative values to get the GCD properly
            a = Math.abs( a );
            b = Math.abs( b );

            // Repeat Euclid’s algorithm until there is no remainder
            for( ; ; )
            {
               // Get the remainder to use later in the algorithm
               long remainder = a % b;
               // If remainder is 0 return the previous remainder
               If( remainder == 0 ) return b;
               // make the dividend the new divisor and the divisor the remainder
               // and continue the algorithm
               a = b;
               b = remainder;
            };
         }
```

7.2

Under the two conditions of writing comments when you code and waiting to write the comments until after the code, one might end up with bad comments shown in the previous code.

7.4

The validation code I would write in Exercise 3 covers offensive programming principles since it would validate the inputs and results.

7.5

Since the primary purpose of the GCD function is to find the GCD, error handling should be taken care of in the context of the calling function if the function fails for some odd reason.

7.7

Top-down design of how to go to the nearest supermarket:

1. Go to your car
2. Open the car
3. Start the car
4. Pull out of the parking space
5. Turn left on to Ignatian Circle
6. Turn left at the roundabout on to loyola blvd
7. Turn right on to W 83rd St.
8. Turn left on to Lincoln Blvd.
9. Turn right to drive into the parking lot of Ralphs
10. Find a parking spot
11. Park in that parking spot
12. Turn off the engine
13. Get out of the car
14. Walk into the super market

8.1

Below is python code for the ValidateAreRelativelyPrime method:

Python

```
def validate_are_relatively_prime(a, b):


    a = abs(a)

    b = abs(b)
```

```
    if a == 1 or b == 1:
        return True

    if a == 0 or b == 0:
        return False

    # Loop from 2 to the smaller of a and b looking for factors.
    smallest = min(a, b)

    for factor in range(2, smallest + 1):
        if a % factor == 0 and b % factor == 0:
            return False

    return True
```

Below is the test code for ensuring that are_relatively_prime is the same as the method we wrote for ‘validate_are_relatively_prime()’.

Python

```
import unittest
from validate_are_relatively_prime import validate_are_relatively_prime
from are_relatively_prime import are_relatively_prime
import random
```

```
class test_relatively_prime(unittest.TestCase):


    def test_for_random_a_and_b_1000_trials(self):
        for i in range(0,1000):
            rand_a = random.randint(-1000000, 1000000)
            rand_b = random.randint(-1000000, 1000000)
            self.assertEqual(validate_are_relatively_prime(
                rand_a, rand_b), are_relatively_prime(rand_a, rand_b))


    def test_for_random_a_and_1(self):
        for i in range(0, 1000):
            rand_a = random.randint(-1000000, 1000000)
            self.assertEqual(validate_are_relatively_prime(rand_a, 1), are_relatively_prime(rand_a,
1))
            self.assertEqual(validate_are_relatively_prime(1, rand_a), are_relatively_prime(1,
rand_a))
            self.assertEqual(validate_are_relatively_prime(rand_a, -1), are_relatively_prime(rand_a,
-1))
            self.assertEqual(validate_are_relatively_prime(-1, rand_a), are_relatively_prime(-1,
rand_a))


    def test_for_random_a_and_0(self):
        for i in range(0, 1000):
            rand_a = random.randint(-1000000, 1000000)
```

```
        self.assertEqual(validate_are_relatively_prime(rand_a, 0), are_relatively_prime(rand_a,
0))

        self.assertEqual(validate_are_relatively_prime(0, rand_a), are_relatively_prime(0,
rand_a))


    def test_for_random_a_and_1000000(self):

        for i in range(0, 1000):

            rand_a = random.randint(-1000000, 1000000)

            self.assertEqual(validate_are_relatively_prime(rand_a, 1000000),
are_relatively_prime(rand_a, 1000000))

            self.assertEqual(validate_are_relatively_prime(rand_a, -1000000),
are_relatively_prime(rand_a, -1000000))

            self.assertEqual(validate_are_relatively_prime(1000000, rand_a),
are_relatively_prime(1000000, rand_a))

            self.assertEqual(validate_are_relatively_prime(-1000000, rand_a),
are_relatively_prime(-1000000, rand_a))


if __name__ == '__main__':

    unittest.main()
```

8.3

The testing technique we used to test the method in Exercise 1 was white-box/gray-box testing since we came up with the method “validate_are_relatively_prime” so we know the makeup of the method but we do not know the logic behind the the original “are_relatively_prime” method that we are testing “validate_are_relatively_prime” against.

8.5

```
using System;


class Program
{
	// Return true if a and b are relatively prime.
	private static bool AreRelativelyPrime( int a, int b )
	{
	//if the values are too large or too small the function throw an error
	if( a < -1000000000 || a > 1000000000 || b < -1000000000 || b > 1000000000)
	{
	throw new System.NotFiniteNumberException("Parameter cannot be large", a);
	}

	if(b < -1000000000 || b > 1000000000)
	{
	throw new System.NotFiniteNumberException("Parameter cannot be large", b);
	}

	// Only 1 and -1 are relatively prime to 0.
	if( a == 0 ) return ((b == 1) || (b == -1));
	if( b == 0 ) return ((a == 1) || (a == -1));

	int gcd = GCD( a, b );
	return ((gcd == 1) || (gcd == -1));
```

```
}

// Use Euclid's algorithm to calculate the
// greatest common divisor (GCD) of two numbers.
// See https://en.wikipedia.org/wiki/Euclidean_algorighm
private static int GCD( int a, int b )
{
a = Math.Abs( a );
b = Math.Abs( b );

// if a or b is 0, return the other value.
if( a == 0 ) return b;
if( b == 0 ) return a;

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

private static void Main()
```

```
        {
        System.Console.WriteLine(AreRelativelyPrime(2,4));
        System.Console.WriteLine(AreRelativelyPrime(3,5));
        System.Console.WriteLine(AreRelativelyPrime(-2,4));
        System.Console.WriteLine(AreRelativelyPrime(-3,5));
        System.Console.WriteLine(AreRelativelyPrime(-2,-4));
        System.Console.WriteLine(AreRelativelyPrime(-3,-5));
        System.Console.WriteLine(AreRelativelyPrime(2,int.MaxValue));
        System.Console.WriteLine(AreRelativelyPrime(-2,int.MaxValue));
        System.Console.WriteLine(AreRelativelyPrime(2,int.MinValue));
        System.Console.WriteLine(AreRelativelyPrime(-2,int.MinValue));
        System.Console.WriteLine(AreRelativelyPrime(int.MaxValue,int.MinValue));
        System.Console.WriteLine(AreRelativelyPrime(int.MaxValue,int.MaxValue));
        System.Console.WriteLine(AreRelativelyPrime(int.MinValue,int.MinValue));
        }
}
```

Writing the tests made me think about edge cases and even revealed that very large and very small values caused the function to error out. To combat this, the inputs are now limited to -1bn and 1bn any values outside this range are unsupported.

8.9

Exhaustive testing is a form of black box testing, because it only tests inputs and outputs without checking or testing the internal structure of the program.

8.11

There are 3 potential lincoln indexes for this problem,

Alice X Bob = 10

Alice X Carmen = 12.5

Bob X Carmen = 20

Averaging the lincoln indexes provides the estimate 14.17, so about 14 bugs are present in the program. Therefore about 4 bugs are still at large. These estimates will change over time as more testing is performed.

8.12

If two testers find zero bugs in common, then the Lincoln index divides by zero which given the context of the problem approaches infinity. To get an estimate just divide by 1 as if they found a bug in common and keep testing to get a more accurate prediction.