Elizabeth Shen
402 HW #3

Problem 7.1, Stephens page 169

The greatest common divisor (GCD) of two integers is the largest integer that evenly divides them both. For example, the GCD of 84 and 36 is 12, because 12 is the largest integer that evenly divides both 84 and 36. You can learn more about the GCD and the Euclidean algorithm, which you can find at en.wikipedia.org/wiki/Euclidean_algorithm. (Don't worry about the code if you can't understand it. Just focus on the comments.)(Hint: It should take you only a few seconds to fix these comments. Don't make a career out of it.)

```
// Use Euclid's algorithm to calculate the GCD.
// See 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;
   };
 }
```

These comments are better because they are short and succinct and don't clutter the code with too much explanation.

Problem 7.2, Stephens page 170

Under what two conditions might you end up with the bad comments shown in the previous code?

One condition that could have led to bad comments is that the programmer added the comments after writing the code itself. Another condition is that the programmer took a top-down approach when writing the comments.

Problem 7.4, Stephens page 170

How could you apply offensive programming to the modified code you wrote for exercise 3? [Yes, I know that problem wasn't assigned, but if you take a look at it you can still do this exercise.]

The validation code is already fairly offensive since it validates the input and the result.

Problem 7.5, Stephens page 170

Should you add error handling to the modified code you wrote for Exercise 4?

You don't really need to add error handling code here, since the calling code should handle any errors. If the code throws exceptions, they should be passed up and handled by the calling code.

Problem 7.7, Stephens page 170

Using top-down design, write the highest level of instructions that you would use to tell someone how to drive your car to the nearest supermarket. (Keep it at a very high level.) List any assumptions you make.

a. **Find the car**
b. **Get in the car**
c. **Start the car**
d. **Drive East down the current street**
e. **Turn left at the first stop sign**
f. **Turn left at the third stop light**
g. **Turn right into the supermarket parking lot**
h. **Park in an available parking space**
i. **Stop the car**
j. **Get out of the car**

This list of instructions holds the assumptions that you begin parked at my apartment on Blackburn Avenue, Los Angeles. It also assumes that the supermarket is open, that there is gas in the car, and you follow all necessary precautions when driving such as checking your mirrors, and watching out for pedestrians and moving aside for sirens.

Problem 8.1, Stephens page 199

Two integers are *relatively prime* (or *coprime*) if they have no common factors other than 1. For example, 21 = 3 X 7 and 35 = 5 X 7 are *not* relatively prime because they are both divisible by 7. By definition -1 and 1 are relatively prime to every integer, and they are the only numbers relatively prime to 0. Suppose you've written an efficient IsRelativelyPrime method that takes two integers between -1 million and 1 million as parameters and returns true if they are relatively prime. Use either your favorite programming language or pseudocode (English that sort of looks like code) to write a method that tests the IsRelativelyPrime method. (Hint: You may find it useful to write another method that also tests two integers to see if they are relatively prime.)

Pseudocode Below:

```
// Convert two values to positive numbers
// Return true if either value is 1
// Return false if either value is 0
```

// Loop from 2 to the smaller of a and b while looking for factors.

Problem 8.3, Stephens page 199

What testing techniques did you use to write the test method in Exercise 1? (Exhaustive, black-box, white-box, or gray-box?) Which ones *could* you use and under what circumstances? [Please justify your answer with a short paragraph to explain.]

This was a black-box test, since Exercise 1 did not specify how the AreRelativelyPrime method works. If it was specified, then you could write white-box and gray-box tests. You could probably do this by testing values ranged from -1000 to 1000, which would give a million pairs to test.

Problem 8.5, Stephens page 199 - 200

the following code shows a C# version of the AreRelativelyPrime method and the GCD method it calls.

```
// Return true if a and b are relatively prime.
 private bool AreRelativelyPrime( int a, int 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 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;
    };
  }
```

The AreRelativelyPrime method checks whether either value is 0. Only -1 and 1 are relatively prime to 0, so if a or b is 0, the method returns true only if the other value is -1 or 1.
The code then calls the GCD method to get the greatest common divisor of a and b. If the greatest common divisor is -1 or 1, the values are relatively prime, so the method returns true. Otherwise, the method returns false.
Now that you know how the method works, implement it and your testing code in your favorite programming language. Did you find any bugs in your initial version of the method or in the testing code? Did you get any benefit from the testing code?

```
private boolean AreRelativelyPrime( int a, int b )
{
 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));
}

public static int GCD(int m, int n) {
        if (m == 0 && n == 0) {
                return 0;
        }
        if (n == 0) {
                return m;
        } else if (m >= n && n > 0) {
                return GCD(n, m%n);
        } else {
```

```
                    return GCD(n, m);
            }
    }
```

When I first tested the code, I got a bug because of a typo in my variable names. There was a benefit in the testing because I caught this bug.

Problem 8.9, Stephens page 200

Exhaustive testing actually falls into one ot the categories black-box, white-box, or gray-box. Which one is it and why?

Exhaustive tests are black-box since they do not rely on any knowledge of the actual method they are testing and how it works.

Problem 8.11, Stephens page 200

Suppose you have three testers: Alice, Bob, and Carmen. You assign numbers to the bugs so the testers find the sets of bugs {1, 2, 3, 4, 5}, {2, 5, 6, 7}, and {1, 2, 8, 9, 10}. How can you use the Lincoln index to estimate the total number of bugs? How many bugs are still at large?

You can calculate the three different Lincoln indexes using each pair of testers:
Alice/Bob: 5 x 4 / 2 = 10
Alice/Carmen: 5 x 5 / 2 = 12.5
Bob/Carmen: 4 x 5 / 1 = 20

Then take the average of the three solutions to get an estimate of 14 bugs. Then, continue to revise the estimate as new bugs are found.

Problem 8.12, Stephens page 200

What happens to the Lincoln estimate if the two testers don't find any bugs in common? What does it mean? Can you get a "lower bound" estimate of the number of bugs?

If the two testers don’t find any bugs in common, then the equation for the Lincoln index divides by 0, giving you an infinite result. Then you don’t know how many bugs there are. You can kind of get of lower bound estimate by pretending the testers found 1 bug in common. For example, if the testers found 3 and 4 bugs in common, then the lower bound index would be 3 x 4 / 1 = 12 bugs.