Jay Patel

Homework 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 aten.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.
         private long GCD( long a, long b )
         {
            // Get the absolute value of a and b
            a = Math.abs( a );
            b = Math.abs( b );

            //Repeat Euclid’s Algorithm until we're done
            for( ; ; )
            {
               long remainder = a % b;
               // If remainder is 0, we're done.  Return b.
               If( remainder == 0 ) return b;
               // Otherwise Set a = b and b = remainder.
               a = b;
               b = remainder;
            };
         }
```

Problem 7.2, Stephens page 170

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

Programmer took the top-down design to the point where the code is described in too much detail or added comments after writing the program.

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.]

There is no need to really added offensive programming because it has the Debug.Assert method and throws exceptions if there is a problem

Problem 7.5, Stephens page 170

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

It depends on the user, but it can go both ways. One could add error handling in the method which or add it during 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.

Get your keys and then exit your house. Go to your car and enter it. Once inside use gps and enter in the market address. Put on your seatbelt and start the car After the car is on get yourself on the road, depending on where you parked either make a u-turn, just pull out of the spot, reverse and go down the road, but whatever it is just head down the road in which direction the market is. Follow the gps till you reach the market. Park the car and turn it off and don’t forget your keys. Then go to the entrance of the supermarket and go inside.

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.)

```
Def isRPrime(a, b):
        A = math.abs(a)
        B = math.abs(b)
        If a == 1 || b == 1 return True
        Else if a == 0 || b == 0 return False
        Else if a > 1m || b > 1mill return False
        Else: minVal = math.Min(a,b)
                For factor in range(2, minVal):
                        If a%factor == 0 and b%factor == 0: return false
        Return True
Def checkTwoValues():
        Count = 0;
        while(count < 100):
                A = random number b/w -1million-1million
                B = randome number b/w -1million-1million
                Return IsRelativelyPrime(a.b) == isRPrime(a,b)
Def checkConstraints():
        Count = 0
        While count < 100:
                A = random number b/w -1million-1million
```

If IsRelativelyPrime(a.1) != isRPrime(a,1) or IsRelativelyPrime(a.-1) != isRPrime(a,-1) or IsRelativelyPrime(a.0) != isRPrime(a,0)

Return False

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.]

Black-box because we don't actually know how AreRelativePrime works. The reason we need black box is cause we don't know what the method we are testing against does, but we know what the value should be so if we write a working version of isRPrime then we should be able to check the two values and make sure they are equivalent. Can't do white or grey cause we don't know how AreRelativePrime works and Exhaustive would be too many pairs.

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?

The only bug is that if the value is 0 it should return true, but my test wouldn't do that, but besides that even if it did do that it would be wrong because we use the same exact method so its like testing if True == True there is no real testing or checking going on.

Problem 8.9, Stephens page 200

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

Exhaustive tests are black-box tests because they don’t rely on knowledge of what’s going on inside the method.

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?

Alice/Bob = 5*4/2=10 Alice/Carmen = 5*5/2 = 12.5 and Bob/Carmen = 20.

(12 + 12.5 + 20)/3 = 14.8 or roughly 15 bugs there could still be 5 bugs at large because the largest value is 20

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 there are no bugs in common it means that the value is divided by 0 and it gives us no clue about how many bugs there are. The way to get a lower bound is pretending they have 1 bug in common and work with that.