Teddy Chu, Justin Kyle Torres, Alejandro Zapata

Homework #3

Problem 7.1:

```
// An explanation can be found here:https://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;
            };
         }
```

Problem 7.2: Under what two conditions might you end up with the bad comments shown in the previous code?

One condition might be that the programmer finished the code and then went back later to comment it. In their rush, instead of making descriptive comments they just went line by line to explain the statement. They also might've commented the code from a top-down approach. This is a good way of coding but not for making useful comments.

Problem 7.4: 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.]

One method of offensive programming is attempting to see if there are any cases where exceptions are thrown. For example, there is a spot where input is validated and an exception is thrown. That might be a good target.

Problem 7.5: Should you add error handling to the modified code you wrote for Exercise 4?

You probably do not need to add any error handling code because it is necessary for errors to propagate up to whomever is calling the code so *they* can handle it.

Problem 7.7: Using top-down design, write 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.

I assume they know where my car is and what turns they need make to get to the supermarket. I also assume they have the appropriate amount of cash in their wallets. The car should also have gas, and the car should be parked with in the head in. I also assume there is ample parking in the parking lot. I assume the market is open.

1. Get to my car.
2. Get inside my car.
3. Close the door.
4. Turn the car on.
5. Back out.
6. Get onto the appropriate side of the road.
7. Drive to the supermarket.
8. Turn into the parking lot.
9. Make the appropriate amount of turns to park in a spot.
10. Next, go to the market and spend lots of money.

Problem 8.1

```
Num, num -> boolean
AlternativeRelativelyPrime(x, y):
  Num X_abs := math.abs(x)
  Num Y_abs := math.abs(y)
  If x_abs == 1 or y_abs == 1:
     Return true
  If x_abs or y_abs == 0:
     Return false
  Smaller_value := math.min(x, y)
  For i in range(2, min):
  If x % i == 0 and y %  i == 0:
     Return false
   ; if we get here then we must have two relatively prime numbers
   Return true
```

You would want to test the method against our method under a number of different conditions.

For some number of trials (maybe 1000)

X = random()

Y = random()

Assert alternativeRelativelyPrime(x, y) == a IsRelativelyPrime(x, y)

Assert alternativeRelativelyPrime(y, y) == IsRelativelyPrime(y, y)

Assert alternativeRelativelyPrime(1, x) == IsRelativelyPrime(1, x)

Assert alternativeRelativelyPrime(-1, x) == IsRelativelyPrime(-1, x)

Assert alternativeRelativelyPrime(x, 1) == IsRelativelyPrime(x, -1)

You would want to this for every permutation of 0s and 1s and -1s

Assert alternativeRelativelyPrime(0, 1) == IsRelativelyPrime(0, 1)

Assert alternativeRelativelyPrime(-1, 0) == IsRelativelyPrime(-1, 0)

Assert alternativeRelativelyPrime(-1, 1) == IsRelativelyPrime(-1, 1)

…

You would want to do something similar to above but testing with the max and min values aka 1,000,000, and -1,000,000.

Problem 8.3:

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.]

If the number of pairs of numbers were smaller we could do an exhaustive test, but the order of magnitude of pairs is far too high. We are not given information about how the IsRelativelyPrime method works so we must use the black-box method. It would be possible to use the white-box or gray-box method if we were given information though.

Problem 8.5:

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?

It was fairly straightforward to port it. Testing was important though to validate that it was working and that there weren't any idiosyncrasies between languages that break the implementation.

```
; implementation in jen-lang
num, num -> bool
AreRelativelyPrime(a, b):
    if a == 0:
        return b == 1 or b == -1

    if b == 0:
        return a == 1 or a == -1

    num gcd = GCD(a, b)
    return gcd == 1 || gcd == -1

num, num -> num
GCD(a, b)
    num a_abs := math.abs(a)
    num b_abs = math.abs(b)

    while true:
        num remainder := a % b
        if remainder == 0:
            return b
        a = b
        b = remainder
```

Problem 8.9

Exhaustive testing is a black box since they do not rely on the contents of the method being tested.

Problem 8.11

Take each pair of testers and calculate their Lincoln indexes.

Alice/Bob = 10

Alice/Carmen = 12.5

Bob/Carmen = 20

You can roughly estimate by adding these 3 together and dividing by 3. So (10+12.5+20) / 3 ≈ 14 bugs

Problem 8.12

No bugs in common means the Lincoln index would divide by 0, meaning you don't know how many bugs there are. You can get a lower bound by assuming the testers found 1 bug in common, then dividing by 1 (ends up being tester 1 x tester 2 bugs)