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. [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 )
         {
            // Repeat until we find the GCD
            for( ; ; )
            {
               // Set 'remainder' var to the remainder of a / b
               long remainder = a % b;
               // If the remainder is 0, we're done.  Return b.
               if( remainder == 0 ) return b;
               // Set a = b and b = remainder.
               a = b;
               b = remainder;
            };
         }
```

Problem 7.2, Stephens page 170

According to your textbook, under what two conditions might you end up with the bad comments shown in the previous code?

First, the programmer may have taken a top-down design where the code is described in too many details. It can result in redundant comments which describe each line of code individually which is not necessary.

Second, if the programmer added the comments after writing the code. After the code is written, it's easy to just say what each line of code does and not why it is doing it.

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.]

Offensive Programming is a strategy that prioritizes catching and surfacing bugs early. We can implement this in exercise 3 by only allowing ‘long’ data types (or ints/floats if we’re using python). If there is any other type of input, either try to convert it to a long or return an error. We can also make both a and b positive by using the math.abs method which is shown in the exercise.

Problem 7.5, Stephens page 170

Should you add error handling to the modified code you wrote for Exercise 4? Explain your reasoning.

Yes, it would be helpful for the user to know why their input did not work as intended. For example, if the user didn’t use the correct data type for the variables a or b, then the error handler should tell them of this problem.

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.

Assumptions:

1. **You know how to drive a car (have your license)**
2. **You have access to a navigation system** (GPS, Apple maps).
3. **You know which supermarket to go to.**

High-Level Instructions:

1. **Start the car**
   Wear a seatbelt ,make sure you can see the road and the mirrors are all working :)
2. **Use Navigation to put in the address of the supermarket**
   Open your navigation system (phone or GPS), Input the name of supermarket, pick the fastest route
3. **Drive to the Supermarket**

   Make sure that you follow all road signs and rules, follow the speed limit, stay on track of the GPS

4. **Arrive at the Supermarket**
   Park the car and turn it off.

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. However, integers 8 = 2 X 4 and 9 = 3 X 3 ARE relatively prime because the only common factor they have is 1 [even though NEITHER of them is prime by itself!] 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 to write a program that *tests* the `isRelativelyPrime()` method. [Hint: You may find it useful to write the `isRelativelyPrime()` method itself as well.]

```
def test_isRelativelyPrime():
    test_cases = [
        (21, 35),  # Not relatively prime, common factor is 7 (example)
        (8, 9),    # Relatively prime, no common factor other than 1
        (7, 5),    # Relatively prime, no common factor other than 1
        (12, 18),  # Not relatively prime, common factor is 6
```

```
        (0, 9432),    # Not relatively prime, because 0 is not relatively prime to any
integer
        (1, 100),  # Relatively prime, 1 is relatively prime to any number
        (100, 100), # Not relatively prime, because they are the same number
        (-8, 9),   # Relatively prime, no common factor other than 1
        (-15, 10)  # Not relatively prime, common factor is 5
    ]

    for a, b in test_cases:
        result = isRelativelyPrime(a, b)
        print(f"isRelativelyPrime({a}, {b}) = {result}")

# Run the tests
test_isRelativelyPrime()

# check results based off what we have written in the comments
```

Problem 8.3, Stephens page 199

1. What testing techniques did you use for the program in Exercise 8.1? [Exhaustive, black-box, white-box, or gray-box?]

   I would say it is gray-box testing. I say this because I come to this exercise with background information of what the function is trying to do. However, I do not have access to the actual function and see the complexities of the code. What matters the most in this regard is the output, either true or

false. It is certainly not exhaustive because we have not tried all the combinations of inputs, in fact, I have only tried using ints.

2. Which ones *could* you use and under what circumstances? [Justify your answer with a short paragraph to explain.]

   I could use blackbox testing if we simply just focus on inputs/outputs and without writing the function itself. We can use whitebox testing if we specifically write out the function and know how the code works. Finally, we could use graybox testing in our current circumstances by knowing generally what the function does but not explicitly writing the code. Although it is not ideal, we could use exhaustive testing as the integers are specified to be between -1 million and 1 million, therefore there is a finite number of possible inputs 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?

areRelativelyPrime(21, 35) = False
areRelativelyPrime(8, 9) = True
areRelativelyPrime(7, 5) = True
areRelativelyPrime(12, 18) = False
areRelativelyPrime(0, 9) = False
areRelativelyPrime(1, 100) = True
areRelativelyPrime(100, 100) = False
areRelativelyPrime(-8, 9) = True
areRelativelyPrime(-15, 10) = False

This is the expected output from the above testing function. From the code, I do not see any specific errors which stand out to me. By tracing the algorithm with my tests, I think we receive the expected output. The testing code provides confidence that the areRelativelyPrime() method works correctly across a variety of input cases.

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?

I would think black-box testing because in blackbox testing, we do not know exactly what the code does. Therefore, we do not come in with preconceived notions of how the code is supposed to work. In white box testing, when we create/know what the code does, we may be biased by online using values that we know will work instead of trying literally any input.

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?

From the Lincoln index, we multiply the number of bugs and divide by the common number of bugs. We see from the sets that only ‘2’ is common between the sets, therefore the Lincoln index is (5 x 4 x 5)// 1 = 100.

From the union of all three testers, we see 10 bugs, so 100 - 10 = 90 bugs still at large.

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 testers do not find any bugs in common, the formula would divide by 0, which would not be possible. The lower bound estimate for the number of bugs is union of the two testers’ bugs.