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

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

From: //Repeat until we're done

To: //Repeat the steps until the remainder is 0, which means we've found the GCD.

From: // Set remainder to the remainder of a / b

To: // Set remainder to the remainder of a divided by b.

From: // If remainder is 0, we're done. Return b.

To: // If remainder is 0, we've found the GCD, so return b.

From: // Set a = b and b = remainder.

To: // Prepare for the next iteration: set a = b and 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?

They were overconfident about the readability of their code. Also, they were too informal.

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.]

Wrap the code in this:

Debug.Assert(a >= 0 && b => 0); checked { }

Problem 7.5, Stephens page 170

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

Yes, now that we're using assertions and exceptions to indicate errors, the code that calls our method needs to use exception handling to deal with those exceptions

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.

Realistically I would not drive my car to the nearest super market as it is faster to just walk but in the case that someone wanted to drive here are the steps:

Get keys from key-holder.

Find car that is street parked.

Unlock car and get in.

Start car and put into gear.

Make U-turn.

Go straight until stop sign ~300ft

Turn right.

Go straight for ~50ft.

Turn right into parking lot.

Find empty spot.

Park car.

Turn car off.

Get out of car and lock it.

Walk into supermarket.

Assumptions:

Store is open.

User can drive my car (It’s a 90’s Miata that’s manual so good luck).

Car is fully functioning and has gas.

They start in my house.

The road is open.

There are parking spots available.

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.)

```
# Testing code
def test_are_relatively_prime():
  test_cases = [
      (5, 7, True),  # 5 and 7 are relatively prime
      (8, 12, False),  # 8 and 12 are not relatively prime
      (-1, 7, True),  # -1 and 7 are relatively prime
      (0, 5, False),  # 0 and 5 are not relatively prime
      (0, 0, False)  # 0 and 0 are not relatively prime
  ]

  for a, b, expected in test_cases:
    result = are_relatively_prime(a, b)
    assert result == expected, f"Failed for ({a}, {b}): expected {expected},
got {result}"

  print("All test cases passed!")
```

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 testing was used because it only interacts with the are_relatively_prime function by providing inputs (integer pairs) and observing the outputs (boolean results) based on expected behavior, without any knowledge of the internal implementation details of are_relatively_prime.

The only technique not possible would be exhaustive as numbers are infinite, thus it's impossible.

Black-box testing: It is suitable when you want to test the software's functionality without considering its internal implementation details. Test cases are designed based on specifications or requirements.

White-box testing: It is useful when you need to test the internal logic, structure, and implementation details of the software. Test cases are designed to exercise specific paths or branches within the code.

Gray-box testing: It combines elements of both black-box and white-box testing techniques. Testers have limited knowledge of the internal workings of the software and design test cases based on a combination of external functionality and internal implementation.

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?

```
import math


def are_relatively_prime(a, b):
  # Only 1 and -1 are relatively prime to 0.
  if a == 0:
    return b == 1 or b == -1
  if b == 0:
    return a == 1 or a == -1

  gcd_val = gcd(a, b)
  return gcd_val == 1 or gcd_val == -1


def gcd(a, b):
  a = abs(a)
  b = abs(b)

  # if a or b is 0, return the other value.
  if a == 0:
    return b
  if b == 0:
    return a

  while True:
    remainder = a % b
    if remainder == 0:
      return b
    a = b
    b = remainder
```

```
# Testing code
def test_are_relatively_prime():
  test_cases = [
      (5, 7, True),  # 5 and 7 are relatively prime
      (8, 12, False),  # 8 and 12 are not relatively prime
      (-1, 7, True),  # -1 and 7 are relatively prime
      (0, 5, False),  # 0 and 5 are not relatively prime
      (0, 0, False)  # 0 and 0 are not relatively prime
  ]

  for a, b, expected in test_cases:
    result = are_relatively_prime(a, b)
    assert result == expected, f"Failed for ({a}, {b}): expected {expected},
got {result}"

  print("All test cases passed!")


# Run tests
test_are_relatively_prime()
```

Yes, initially there was a test that said that 0 and 5 were relatively prime. When testing, the test failed and I fixed the test. The code seemed to be fine, however.

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 testing falls into the category of black-box because exhaustive tests attempt every possible input and thus the test aren’t written based off of any knowledge of the code.

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?

Get the average Lincoln index of each combination:

Alice and Bob (5x4)/2=10

Alice and Carmen (5x5)/2 =12.5

Bob and Carmen (4x5)/1 = 20

(10+12.5+20)/3 = 14 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?

You would be dividing by 0 so technically you would get infinity. Thus, it would be hard to discern how many bugs there actually are because it's neither 0 bugs nor infinite bugs. An easy fix would be to just divide by 1 rather than 0.