Owen Hunger, Evan Yu, Mitchell Cootauco, Connor Savage.
-NOTE: important, this is actually chapter 10/13 in the book.

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;
            };
         }
```

The problem with the gcd code comments, is that it's basically just reiterating what is happening in the code, not actually why it happens or why it works. So like its basically just copying the code, which only makes it more cluttered and confusing. Comments should probably not have any coding syntax and be more close to english to help readers understand quickly. You could maybe link the wikepedia link instead in the comments, as that can probably explain faster than you.

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 might be the programmer was coding in a style where he described each aspect of the code in super detail as he/she was coding. So that's why they ended up with redundant comments. The other scenario is that the coder added the comments after the code was written in an attempt to look like it was documented. Because the coder already knows how the code functions, they probably just wanted to get some comments out fast and lazily.

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.]

You could make sure to validate the data that is coming in, and the data that is leaving, so both input and results. You could also add assert methods to make sure everything is correct without throwing an error.

Problem 7.5, Stephens page 170

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

I think the error catching I talked about in problem 4 would cover this, since the error would actually happen when the method is called, so basically if there is an error it will be passed up to the method call. So there really wouldn't be any point I feel. I believe if my understanding of C is correct.

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.

Exit your house door.

Walk to the car.

Start the car.

Look behind you, pull out (if safe).

Turn right and head to the end of the street. (if no cars)

Turn left, enter the parking lot.

When possible, find an empty spot and park in it.

Turn the car off.

Lock the car.

Grab your keys, enter supermarket.

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.)

I will write pseudo code here.

#TRUE CASE

For 100 rounds, generate random a (positive int)

Assert IsRelativelyPrime(1,a) = true

Assert IsRelativelyPrime(-1,a) = true

Assert IsRelativelyPrime(a,1) = true

Assert IsRelativelyPrime(a,-1) = true

#FALSE CASE

For 100 rounds, generate random a (positive int, excluding 1,-1)

Assert IsRelativelyPrime(0,a) = false

Assert IsRelativelyPrime(a,0) = false

From here you could also test set inputs x,y that you know the result of to see if they match, and potentially the limitations of the method like largest value possibly inputted etc.

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 is black box, I wrote the test code with no understanding of how the function works, and basically just tested on how a method like that should work. So in the context of this test this would be like if a coworker told me about his method and told me to create test cases. Possibly if you used white box testing you would have full knowledge of it, so you could hand pick test cases that would really push and test the capabilities, but you also might unconsciously make tests that you know will work. Grey box is probably ideal because it is in the middle.

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?

Hmm, it seems to work well. I did it in C++. Took me a bit to get my testing code to work correctly. Especially in C++ I didn't have much knowledge about max values for certain variables so some tests kind of got messed up. So maybe I would have to add certain restrictions on the method and add appropriate error messaging. Yes I would say I got benefit from testing the code, even though it mostly works.

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 really black box testing if you think about it, because you either could have/ or have no knowledge of what is being tested, and you are just aggressively testing it.

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?

Using the lincoln index:

You could pair testers to calculate the three different indexes-

Alice + Carmen = 5*5/2 = 12.5

Bob/Carmen = 4*5 /1 = 20

Alice + Bob = 5*4 /2 = 10

Average would be = 14.666, maybe rounging up to 15 to plan for 15 bugs would make sense in my opinion.

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?

I think based off my reading: if they dont find any bugs in common then it would divide by zero, so the result would be undefined or infinite, so you would have no idea how many bugs there are.