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.
  private 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;
    };
  }
```

```
private long GCD(long a, long b)
{
   a = Math.abs(a);
   b = Math.abs(b);
   while (b != 0)
   {
      long remainder = a % b;
      a = b;
      b = remainder;
   }
   return a;
}
```

Problem 7.2, Stephens page 170

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

Bad comments could arise from copy-pasting code without updating the comments, or a lack of understanding of the code's functionality.

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 involves actively preparing for and handling unexpected or erroneous states in your code. In the context of the GCD function, this could involve adding checks for invalid inputs (like negative numbers) and throwing appropriate exceptions or handling edge cases (e.g., when one of the parameters is zero).

Problem 7.5, Stephens page 170

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

Yes, error handling should be added to ensure the robustness of the code. This might include handling cases where inputs are not valid integers, or ensuring that division by zero does not occur.

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.

- Determine the location of the nearest supermarket.
- Get into the car and start the engine.
- Follow the best route to the supermarket, obeying all traffic laws.
- Park the car in the supermarket parking lot.
- Assumptions: The car is functional and has enough fuel, the driver knows how to operate the car, can operate a map/direction app, and the route to the supermarket is known.

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.)

(Python)

```
def gcd(a, b):
    """Compute the greatest common divisor using Euclid's algorithm."""
    while b != 0:
        a, b = b, a % b
    return a

def is_relatively_prime(a, b):
    """Determine if two integers are relatively prime."""
    return gcd(a, b) == 1

# Tests for the is_relatively_prime function
test_cases = [
    (13, 18, True),  # relatively prime
    (9, 28, True),   # relatively prime
    (21, 35, False), # not relatively prime
    (12, 18, False), # not relatively prime
    (0, 1, True),    # edge case, relatively prime
    (0, 0, False),   # edge case, not relatively prime
    (-13, 18, True), # negative numbers, relatively prime
]

# Running the tests
test_results = []
for a, b, expected in test_cases:
    result = is_relatively_prime(a, b)
    test_results.append((a, b, result == expected))

test_results
```

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 would be an example of black-box testing, as it involves testing the function without knowing its internal workings, focusing solely on input-output behavior. White-box testing could also be used if the internal structure of the method is considered to create the test cases.

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?

No bugs found in the code written above. Testing the code did help verify the correctness of the implementation.

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 under white-box testing, as it requires knowledge of the internal structure of the code to ensure all paths and scenarios are tested. However, it's often impractical for complex systems due to the immense number of possible inputs and states.

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?

The Lincoln index can estimate the total number of bugs by using the formula: (number of bugs found by the first tester × number of bugs found by the second tester) / number of bugs found by both. Applying this to Alice and Bob: (5 × 4) / 3 = 6.67. Adjusting for the third tester, Carmen, can complicate the estimate, as the Lincoln index typically considers only two testers.

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 two testers find no common bugs, the Lincoln index cannot be used, as it relies on an overlap to estimate the total population. This scenario could indicate either a very large number of bugs or very different testing methods or areas of focus by the testers. A lower bound estimate would be the sum of unique bugs found by each tester.