Homework 3 – Arash Lari

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](https://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 comments the above code provides is not really useful as they only say what the code does and not so much on how or why it does that. It gives descriptions on sections of the code, but does not provide a general explanation of Euclid’s algorithm.

The following shows an example of the improved version of the previous code:

       // Use Euclid's algorithm to calculate the GCD (see <https://en.wikipedia.org/wiki/Euclidean_algorighm> for more details on Euclid’s algorithm)  
         private long GCD( long a, long b )  
         {  
            a = Math.abs( a );  
            b = Math.abs( b );  
  
            for( ; ; )  
            {  
               long remainder = a % b; // to find common factors  
               If( remainder == 0 ) return b;   
               a = b;  
               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?

Bad comments occur either when the programmer takes a top down approach or describes the code as he’s writing it, leading to redundant and over descriptive comments, or if the programmer comments AFTER writing the code and coming back to it leading to very simple comments that explain each individual line rather than the reason behind them.

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 can be used to validate the inputs and the result by asserting them as greater than 0. The debug assert would then throw an exception if it sees a problem.

Problem 7.5, Stephens page 170

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

Error handling takes care of cases that would crash the program, therefore it would be best to implement them.

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.

1. Go to garage
2. Find car
3. Open door
4. Get in car’s driver seat
5. Turn on car engine
6. Drive forwards and take a right
7. Take another right
8. Open garage door
9. Continue straight then turn right
10. Turn left onto La Tijera and stop at the stop sign
11. Turn right and then take another right at Loyola Blvd.
12. Turn left into Ralphs’ parking lot
13. Find parking spot
14. Turn off the car engine and get out
15. Lock car and go to 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.)

{

take in inputs for a and b;

get their absolute values;

if either value is 1, then return true;

if either is 0, then return false;

loop from 2 to the smaller of a and b looking for factors.

If a % factor AND b % factor are 0, then return false;

Otherwise return true;

}

The code will first ensure that a and b are positive integers. Then if either value is 1, it returns true because 1 and/or -1 are relatively prime to every number. In the same vein, when a and/or b is 0, the method will return false because only 1 and -1 are relatively prime to 0.

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.]

Since the statement of exercise 1 doesn’t say how the method works, this must be a black box test. If we were given how it works we could write white box and grey box tests for it. Writing an exhaustive test ranging from negative 1 million to positive 1 million would probably be too much, but if it were from -1000 to 1000 it would be manageable.

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?

The testing code did not consider how robust the program could be, and therefore didn’t test for weird values like -1, 0, and 1.

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 black-box tests as they don’t need to know how the method works to test what the expected outcome should be.

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?

You can use each pair of testers to calculate three different Lincoln indexes.

Alice and Bob: (5x4)/2 = 10

Alice and Carmen: (5x5)/2 = 12.5

Bob and Carmen: (4x5)/1 = 20

The average of all three Lincoln estimates is 14, but there are still 4 bugs unaccounted for.

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 don’t find any bugs in common, that means a division by 0, which theoretically means infinite errors, but actually just means you have no idea how many errors. You can however calculate a sort of lower bound by pretending that the coders all had 1 bug in common.