Kevin Metelus

Homework 3

**Problem 7.1, Stephens page 169**

// Should use Euclid's algorithm to calculate the GCD.  
 private long GCD( long a, long b )  
 {  
 // Must use the absolute value to calculate the GCD  
 a = Math.abs( a );  
 b = Math.abs( b );  
  
 // Should loop until the algorithm returns the answer  
 for( ; ; )  
 {  
 // Should calculate the remainder to determine whether or not the algorithm is done  
 long remainder = a % b;  
 // Should return b if the remainder is 0  
 If( remainder == 0 ) return b;  
 // Should set a to the previous remainder and b to the newest  
 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?

When you all of the code and then write comments, or you comment as you code and just stick comments at the top of every block of code. In either case, you are writing comments about what the code does rather than what the code should do.

**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.]

**Problem 7.5, Stephens page 170**  
  
Should you add error handling to the modified code you wrote for Exercise 4?

**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. Find the car.
2. Open the car.
3. Start the car.
4. Back out of the parking space.
5. Turn to the left (as you look at the parking space).
6. Drive down the street and turn right onto Vermont Ave.
7. Keep left on Vermont Ave until you reach Florence Ave
8. Turn left onto Florence Ave and keep right.
9. Turn left into the parking lot of the supermarket.
10. Find a parking spot in the parking lot.
11. Park the car.
12. Turn off the car.
13. Get out of the car and go to the market.

Assumptions made:

1. Car is parked head-in.
2. The car has gas.
3. There is no roadwork blocking this path.
4. You know how to drive a car.
5. There is an open parking spot.

**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.)

let validateIsRelativelyPrime = (a , b) => {

a = Math.abs(a);

b = Math.abs(b);

if ((a==1) || (b==1)) {

return true;

}

if ((a==0) || (b==0)) {

return false;

}

let min = Math.min(a,b);

for (let i = 2; i <= min; i++) {

if ((a % i == 0) && (b % i == 0)) {

return false;

}

}

return true;

}

\*Pseudocode

Let isRelativelyPrimeTest = () => {

For1000 trials {

A = random int

B = random int

assert.equal(isRelativelyPrime(a, b), validateIsRelativelyPrime(a, b));

}

A = random int

B = random int

assert.equal(isRelativelyPrime(a, a), validateIsRelativelyPrime(a, a));

assert.equal(isRelativelyPrime(a, -1), validateIsRelativelyPrime(a, -1));

assert.equal(isRelativelyPrime(a, 1), validateIsRelativelyPrime(a, 1));

assert.equal(isRelativelyPrime(1000000, a), validateIsRelativelyPrime(1, a));

assert.equal(isRelativelyPrime(-1000000, a), validateIsRelativelyPrime(1, a));

assert.equal(isRelativelyPrime(1000000, 1000000), validateIsRelativelyPrime(1, a));

assert.equal(isRelativelyPrime(-1000000, -1000000), validateIsRelativelyPrime(1, a));

assert.equal(isRelativelyPrime(1000000, -1000000), validateIsRelativelyPrime(1, a));

assert.equal(isRelativelyPrime(-1000000, 1000000), validateIsRelativelyPrime(1, a));

}

**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.]

Definitely not an exhaustive tests because that would mean testing every pair of values between -1000000 and 1000000. Instead i used black-box testing. I did not care what goes on inside of the methods, just the output.

**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?

**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?

Exhaustive tests are black-box tests because it does not matter what is going on inside the function.

**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/Bob: (5 \* 4) / 2 = 10

Bob/Carmen: (4 \* 5) = 20

Alice/Carmen: (5 \* 5) / 2 = 12.5

You could use the average of the three to get a rough estimate of (10 + 12.5 + 20) / 3 = 14 bugs.

**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, then the equation for the Lincoln index divides by 0 which means you have no way of knowing how many bugs there actually are. A lower bound for the amount of bugs to be found can be assumed if we divide by 1 instead of 0.