CMSI 402 Homework #3

- Eileen Choe
- 3/19/18

Problem 7.1 p 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.)

```
        // Implementation of Euclid's algorithm to calculate the GCD.
        private long GCD( long a, long b )
        {
           a = Math.abs( a );
           b = Math.abs( b );

           for( ; ; )
           {
              long remainder = a % b;
              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?

The programmer may have added the the code to explain the algorithm in detail, however this is not necessary because the variable and function names self-document the functionality of the code and is thus repetitive.

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.]

The validation code is offensive, because it validates the inputs and results. Additionally, the `Debug.Assert` method will throw an exception if there is an error.

Problem 7.5, Stephens page 170

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

Error handling can be added in this method, but it is better for the function that calls the GCD method to handle the error.

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. Look up directions to nearest supermarket
2. Locate car keys
3. Walk to car
4. Get in the driver's seat
5. Follow driving instructions to nearest supermarket
6. Park
7. Enter the supermarket

Assumptions:

- There is adequate gas in the car
- The driver has all necessary certifications and documentation to drive
- The supermarket is open
- There is parking at the 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.)

```
const isRelativelyPrimeTester = (numberOfTests) => {
  for (x in [...Array(numberOfTests).keys()]) {
    const firstRandom = getRandomIntInclusive(-1000000, 1000000);
    const secondRandom = getRandomIntInclusive(-1000000, 1000000);
    assert.equals(isRelativelyPrime(firstRandom, secondRandom),
validateIsRelativelyPrime(firstRandom, secondRandom));
  }
};

const getRandomIntInclusive = (min, max) => {
  min = Math.ceil(min);
  max = Math.floor(max);
  return Math.floor(Math.random() * (max - min + 1)) + min; //The maximum is
inclusive and the minimum is inclusive
}

const validateIsRelativelyPrime = (x, y) => {
  x = Math.abs(x);
  y = Math.abs(y)
  const smaller = Math.min(x, y);

  if (x === 1 || y === 1) { return true; }
  if (x === 0 || y === 0) { return false; }

  for (let i = 2; i <= smaller; i++){
    if ((a % i === 0) && (b % i === 0)) { return false; }
  }
  return true;
};

isRelativelyPrimeTester(1000000);
```

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, because I don't have access to the source code for the method being tested.

If this information was provided, then white box testing can be performed.

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?

Based on my testing, the `AreRelativelyPrime` method worked well. The benefit of testing the code was explicitly considering different edge cases, and the assurance that this method works well under these cases.

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?

It falls under the category of black-box testing, because it does not assume any knowledge of the source code of the method being tested.

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?

There are three pairs of testers, with the following corresponding Lincoln indexes:

- Alice & Bob `(5 * 4) / 2 = 10`
- Bob & Carmen `(5 * 5) / 2 = 12.5`
- Alice & Carmen `(4 * 5) / 1 = 20`

It will take an average about 14 bugs, or a worst case of 20 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 there are no bugs that are found in common, then we divde by zero in the formula leading to an infinite result. In this case, we cannot ascertain how many bugs that there are.

A lower bound can be found by assuming that the testers all found 1 common bug. If the testers find 2 and 3 bugs, the lower bound would be `2 * 3 / 1 = 6 bugs`