Josh Soriano

Homework 3: CMSI 402

Problem 7.1

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.)

Add the link to the explanation for the algorithm as a comment and you could do-away with the rest of them.

Problem 7.2

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

The comments could be a result writing the comments after the code has been written. The comments say what the code does and not why it does it, which is an easy mistake to make after programmers finish code because, well, they wrote it. Everything makes sense to them, so they don’t feel the need to explain it. Another possible condition is that the unnecessary information that is included in the code is due to the programmers take of a top-down design.

Problem 7.4

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.]

Adding edge cases and type checks before any computation is done would be a starting point in seeing what offends the program. For example, ensuring that the values of a and b are greater than 0 or checking back and seeing that the remainder evenly divides the original a and b values.

Problem 7.5

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

It could be done, but probably not ideal. Error handling would probably be best left to the caller, allowing the GCD method to be clean of error handling for future use.

Problem 7.7

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.

Instructions

1. Grab keys off the key hook
2. Check for wallet, keys, and phone
3. Open front door

4. Lock front door
5. Take elevator down to the street
6. Locate car
7. Open door to car
8. Put seatbelt on
9. Start the car
10. Turn left onto alleyway behind apartment complex
11. Continue into parking garage
12. Locate an empty parking space
13. Park in the parking space
14. Turn the car off
15. Unbuckle seatbelt
16. Get out of the car
17. Enter Ralphs from across the parking garage

Assumptions

- Car is parked on Rayford Drive outside of my apartment complex
- Car is in working condition (filled with gas, no flat tires, etc.)
- No construction/road work being done on my normal route
- Parking garage has available spaces
- Ralphs is open
- Ralphs' back entrance is open

Problem 8.1

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.)

```
isRelativelyPrimeValidation(a, b) {
    a = Math.abs(a);
    b = Math.abs(b);

    if ((a === 1) || (b === 1)) {
      return true;
    } else if ((a === 0) || (b === 0)) {
      return false;
    } else {
      let min = Math.min(a, b);
      for (i = 0; i < min; i++) {
        if ((a % min === 0) && (b % min === 0)) {
          return false;
        }
        return true;
      }
    }
}
```

Problem 8.3

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 I essentially wrote the test method blindly (without knowing how the actual "IsRelativelyPrime" method works), I made use of the black-box testing technique. Under the circumstances that we did know anything and everything about the method, then we would be able to design tests to attack the methods weaknesses, making white-box testing appropriate. Gray-box testing would be necessary if we were given partial knowledge of the method we are testing. Fittingly, it's the gray-area between knowing nothing and knowing everything.

Problem 8.5

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?

Coverage wise, my testing method doesn't return true if a or b = 0 and the other is -1. So adding that additional check in there ensured that that case result would be correct.

```
isRelativelyPrimeValidation(a, b) {
    a = Math.abs(a);
    b = Math.abs(b);

    if ((a === 1) || (b === 1) {
      return true;
    } else if ((a === 0) || (b === 0)) {
      If ((a === -1) || (b === -1)) {
          return true;
      }
      return false;
    } else {
      let min = Math.min(a, b);
      for (i = 0; i < min; i++) {
        if ((a % min === 0) && (b % min === 0)) {
          return false;
        }
        return true;
      }
    }
}
```

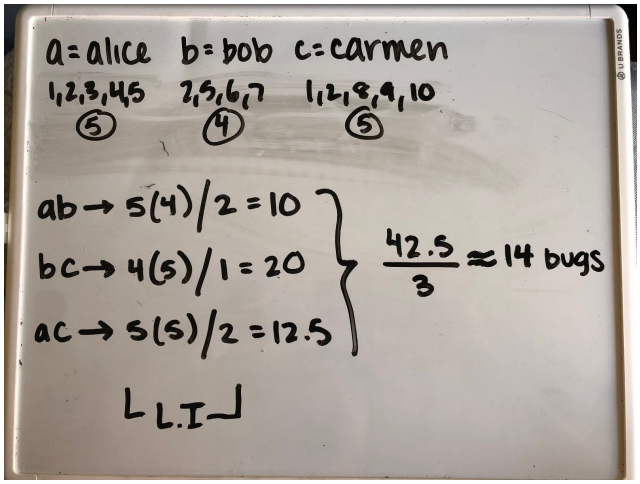
Problem 8.9

Exhaustive testing actually falls into one ot the categories black-box, white-box, or gray-box. Which one is it and why?

Since exhaustive testing doesn't require that a programmer know what's going on in a method internally, it would fall under the black-box category.

Problem 8.11

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?



Problem 8.12

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?

Assuming we were sticking to the formula, each equation would divide by 0 (infinite). Therefore, we would have no way of knowing how many bugs there are at large. In order to get some sort of lower bound, you would have to assume that the testers found one bug in common and the average of that would be your new bound.