Owen Dewing
CMSI 4072
Homework 3

7.1
Most of the comments in the function do not add much information that isn't already clear. For example, adding a comment saying: "repeat until we're done" for a for loop is self explanatory. That is the definition of a for loop; it isn't adding any helpful information about the content of the loop or why we need a for loop. I think that the only necessary comment is the first comment, and then you could probably add a comment under that saying to visit "en.wikipedia.org/wiki/Euclidean_algorithm" for more information about the GCD algorithm.

7.2
You might end up with the bad comments shown in the GCD function due to the programmer taking a top-down approach to the coding, adding comments as they go along. This could lead to redundant comments and explaining each line of code in *too* much detail.

Another approach that can lead to bad comments is if the programmer adds comments after writing all of the code. This can be problematic as code changes a lot; thus, you must always remember what comments you need to adjust. Furthermore, it's very easy to simply state the programs function and ignore the "why".

7.4
To add offensive programming to the GCD function, you could add an explicit check for the case when both inputs are zero (which would make the GCD undefined). It could throw an error message. You could also add a debug.assert that ensures at least one number is positive after taking the absolute values.

Ex: if (a == 0 && b == 0) { throw new IllegalArgumentException("The GCD is undefined!"); }
Debug.Assert(a > 0 || b > 0, "One number must be non-zero");

7.5
To add more error handling, you could add a try-catch block to log exceptions, and catch any other errors the program might make.

7.7
(directions to ralphs from my apartment)

a. Find your keys.
b. Find your car.
c. Unlock your car.
d. Start your car.
e. Back out of the parking garage..
f. Turn to the right. Drive until the alleyway ends.

g. Turn right. Drive to the first stoplight.
h. Turn left. Drive past the first stoplight, and stop at the next.
i. Turn right. Drive until you see the supermarket.
j. Turn into the parking lot.
k. Find an empty parking space.
l. Park.
m. Stop the car.
n. Turn off the car.
o. Walk into the supermarket.

Assumptions:

a. It is day time (no need for car headlights).
b. Your car has gas.
c. You know how to drive a car.
d. The garage door opener works.
e. Nobody (cars, people, pets, etc) gets in your way when driving.
f. There is no construction or obstacles blocking the usual route to the supermarket.
g. The stoplights are working.
h. The supermarket sign is working/visible (its not foggy outside).
i. There are empty parking spots.
j. The supermarket is open.

8.1

Pseudocode:

```
FUNCTION TestRelativelyPrimeFunction:
// Test cases
    TEST(21, 35, false)
    TEST(8, 9, true)
// Special cases
    TEST(-1, any_number, true)
    TEST(1, any_number, true)
    TEST(-1, 0, true)
    TEST(1, 0, true) // 1 is relatively prime to 0
// Edge cases
    TEST(0, 0, false)
    TEST(number, number, false)
// Boundary cases TEST(-1000000, 1000000, true)
END FUNCTION
```

8.3

a. I applied black-box testing to evaluate the relatively prime function, focusing solely on its functionality without knowing the internal implementation.
b. I could use white box and gray box testing if I knew the exact step by step of the isRelativelyPrime function. Additionally, an exhaustive test could be performed, covering a range from -1 million to 1 million.

8.5
```
function areRelativelyPrime(a, b) {
    if (a === 0) return (b === 1 || b === -1);
    if (b === 0) return (a === 1 || a === -1);

    let gcd = GCD(a, b);
    return (gcd === 1 || gcd === -1);
}

function GCD(a, b) {
    a = Math.abs(a);
    b = Math.abs(b);

    if (a === 0) return b;
    if (b === 0) return a;

    while (b !== 0) {
        let remainder = a % b;
        a = b;
        b = remainder;
    }
    return a;
}

function TestRelativelyPrimeFunction() {
    function TEST(a, b, expected) {
        const result = areRelativelyPrime(a, b);
        console.assert(result === expected, `Test failed: areRelativelyPrime(${a}, ${b}) expected
${expected}, got ${result}`);
    }

    // Test cases
    TEST(21, 35, false);
    TEST(8, 9, true);

    // Special cases
    TEST(-1, 5, true);
    TEST(1, 5, true);
    TEST(-1, 0, true);
    TEST(1, 0, true);

    // Edge cases
    TEST(0, 0, false);
    TEST(7, 7, false);
    // Boundary cases
    TEST(-1000000, 1000000, true);
}

TestRelativelyPrimeFunction();
```

The testing code helped me think of different edge or special cases that I wouldn't have necessarily thought of just from reading the prompt.

8.9
Exhaustive testing is classified as a black-box testing approach because it does not rely on understanding the internal workings of the method being tested.

8.11
Number of bugs found by A * number of bugs found by B / number of common bugs found by A and B

Alice and Bob
5 * 4 / 2 = 10

Alice and Carmen
5 * 5 / 2 = 12.5

Bob and Carmen
4 * 5 / 1 = 20

10 + 12.5 + 20 / 3 = 14 bugs in total.

To find out # of bugs still at large, subtract unique bugs from 14
There are 10 unique bugs
14 - 10 = 4 bugs remaining.

8.12
If the testers don't find any bugs in common, then that means that the equation divides by 0, giving an undefined result. This means that nobody knows the total number of bugs. A lower bound could be the number of unique bugs found by both testers. For example, for Alice and Bob, the lower bound would be 7, because that is the number of unique bugs between the 2.