Aidan Esposito

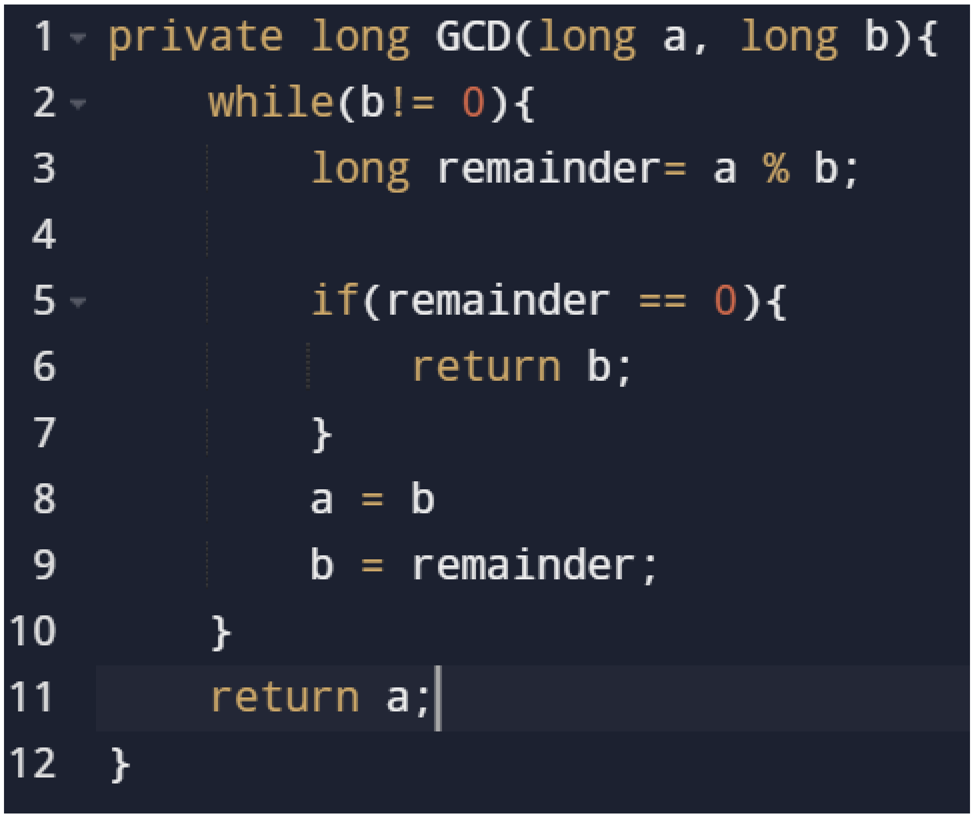
Professor Johnson

CMSI 4072

26 March 2025

Homework 3

7.1



```
private long GCD(long a, long b){
    while(b!= 0){
        long remainder= a % b;

        if(remainder == 0){
            return b;
        }
        a = b
        b = remainder;
    }
    return a;
}
```

7.2

According to the textbook, the two conditions where you might end up with bad comments shown in the previous code includes writing comments as you code and writing all of your code without comments. Updating the comments as you code can lead to hopeless disconnection from the final code and time waste. On the other hand, writing all code without comments leads to writing a bare minimum of comments at the end of the coding cycle. These are both bad approaches since they badly explain and examine what code does and how it performs.

7.4

To apply offensive programming to exercise 3, you would need to find a way to shut down the wrong functions quickly and throw 'temper tantrums' in your code if something goes wrong "offending" the code. To do this, you could use assert statements to make checks in certain parts of the code that can break the system if the code complains about a problem. You can also use the "checked" function in C# to check to make sure something follows the correct criteria, otherwise the code will not run. You can also throw exceptions throughout the code to tell the user why they are doing something wrong to hopefully prevent future insertion error. All of these together create an offensive approach to this exercise.

7.5

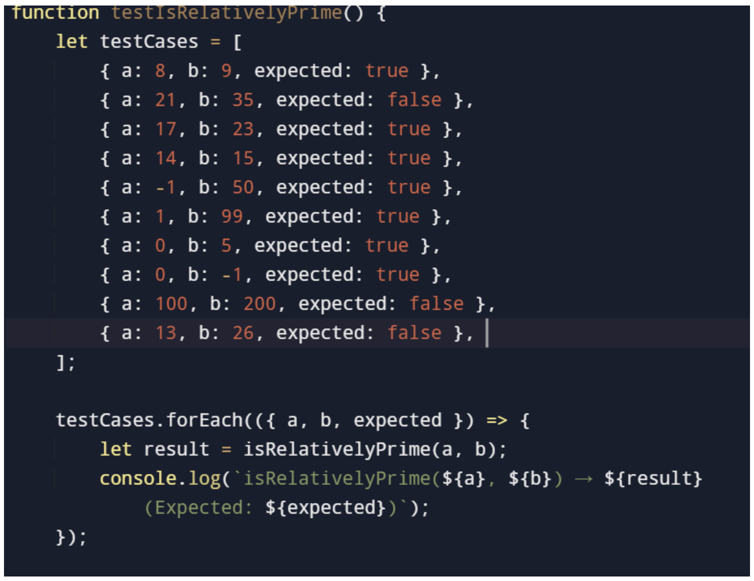
In this situation, I believe error handling should be added to the modified code in exercise 4 since it would allow for meaningful error messages to be present throughout the system preventing future errors as well as allowing for less unknown and unexplained crashes throughout the application in its construction. With error handling, these messages can be logged in order to study and use them to identify problems and find solutions to problems in your code handling. Overall, using error handling can save time and prevent confusion on code errors and allow for clearer fixes to problems to be seen more easily.

7.7

To start high level instructions on how to get to the supermarket via driving a car, you would need to start with entering and using the car itself. You would need to worry about things such as adjusting the mirrors, making sure you can start the car with keys or other methods, and turning on the engine to get the car running. Once the car is on, you would need to navigate to the supermarket and be able to not only determine the best route, but also follow traffic guides and laws as well as being aware of pedestrians and other cars that may get in your way. Once you arrive at the supermarket, you will have to find where to park your car either in a lot, on the street, or somewhere else. Once the car is parked, the car will need to be safely turned off and locked before you can leave and enter the market.

8.1

Here is a function used for testing isRelativelyPrime in Javascript:



```
function testIsRelativelyPrime() {
    let testCases = [
        { a: 8, b: 9, expected: true },
        { a: 21, b: 35, expected: false },
        { a: 17, b: 23, expected: true },
        { a: 14, b: 15, expected: true },
        { a: -1, b: 50, expected: true },
        { a: 1, b: 99, expected: true },
        { a: 0, b: 5, expected: true },
        { a: 0, b: -1, expected: true },
        { a: 100, b: 200, expected: false },
        { a: 13, b: 26, expected: false },
    ];

    testCases.forEach(({ a, b, expected }) => {
        let result = isRelativelyPrime(a, b);
        console.log(`isRelativelyPrime(${a}, ${b}) → ${result}
            (Expected: ${expected})`);
    });
```

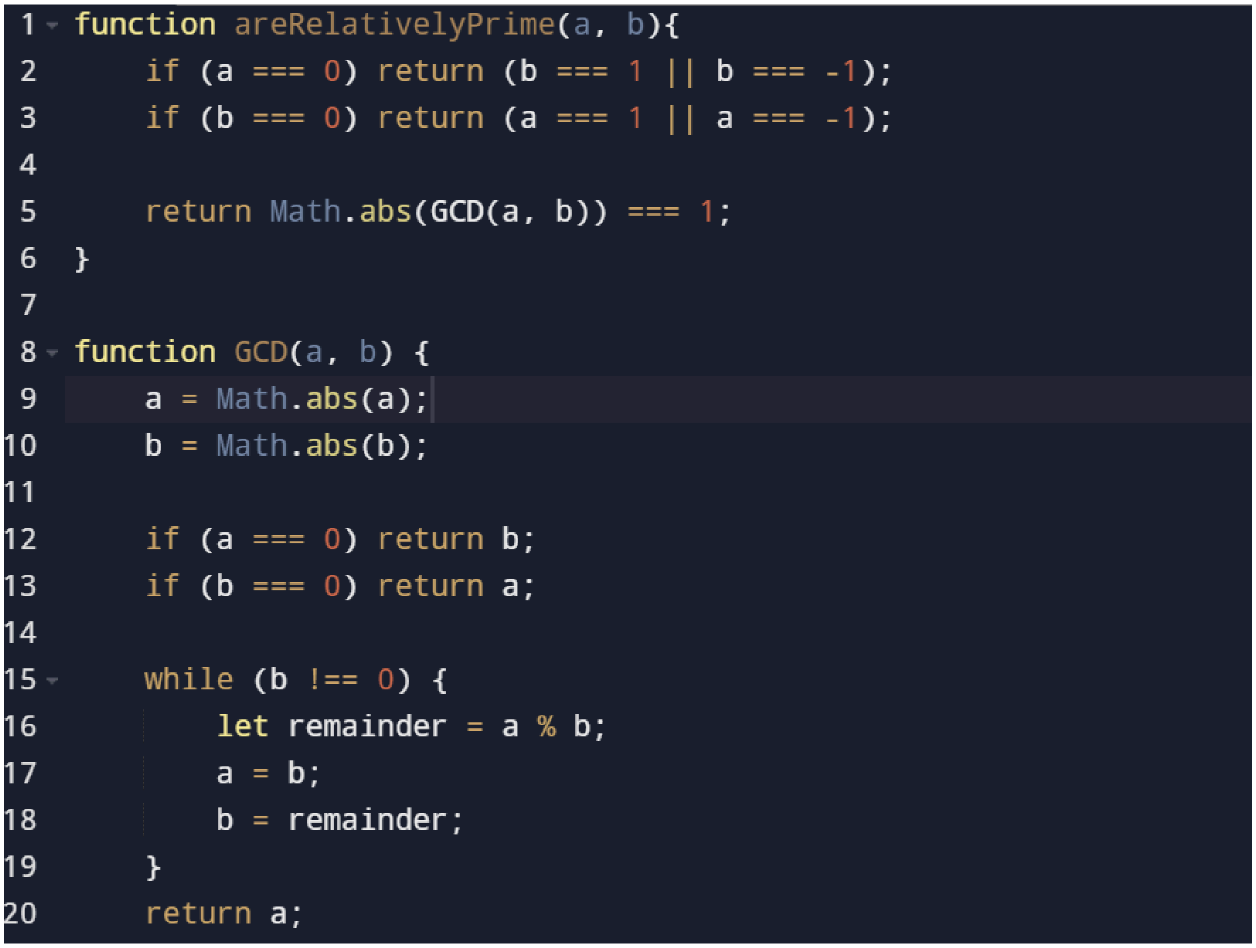
8.3

a. For this specific problem, I used a combination of white and black box testing specifically with grey testing to create the testing function. I needed to use black box testing to test the different inputs for the testing function itself but also used whitebox testing by writing a variant of the isRelativelyPrime function in order to understand how it works to prepare better tests.

b. Most of these test types can be used depending on different circumstances. Exhaustive testing is almost impossible to use because of the sheer amount of prime numbers out there which can be tested with an infinite number amount to make a theoretical infinite amount of tests. The only way for this to work is by setting a cap on the amount of prime numbers that can be tested on with the function. Black box testing can be used to try and make tests by not knowing the main method and using a lot of random values until results are found. White box testing can be used by testing the initial isRelativelyPrime function and using it to write different test cases for the test function. Grey box testing can be used as seen in my example to not only randomly try values but also create an example function to have an idea of what values correlate with the original function.

8.5

For this problem, I wrote my areRelativelyPrime and GCD methods in Javascript for testing. There were some bugs from converting from my initial code especially involving the original for loops which would not have worked well without conversion to while loops in the new language. From testing the code, I understood the function more and learned more about the problem I was working on as well as how to better test it.



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

    return Math.abs(GCD(a, b)) === 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;
```

8.9

Exhaustive testing would fall, at least in my mind, under the label of white box testing because it requires a tester to cover all possible code paths, conditions, inputs, and outputs of a situation via knowing or not knowing the general function. Blackbox testing, in comparison, lacks the knowledge of the function implementation making it impossible to test all possible outputs randomly. Gray box testing is more similar, but includes black box testing in its situation making it more aligned to it compared to general white box testing.

8.11

To use the Lincoln index to estimate the total number of bugs, you would need to start by defining Alice, Bob, and Cartmen's datasets which are given to us via the problem. Once defined, we must label and find the number of bugs in each set, those being 5, 4, and 5 specifically. With this, we find the common bugs and apply the Lincoln index formula to the values of the total bugs divided by the common bugs in the set. For this situation, the total number of bugs found would be approximately 14 bugs with none at large since we found all of the bugs in each dataset.

8.12

If there are no common bugs found in the Lincoln formula, it would make the denominator of the formula 0 which would make the final number infinite making the answer be no solution. This would mean the testers did not check what are the common bugs and possibly suggest that each tester has a different set of bugs. With this, it is still possible to provide a lower bound estimate of the bugs by combining the sum of all the bugs found, even if they do not overlap.