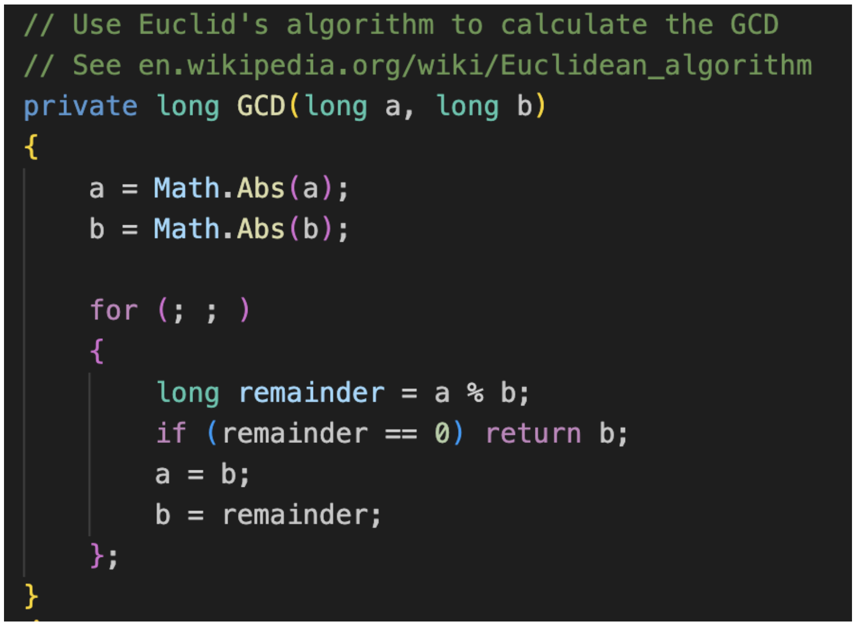
Senior Project Group EVE

Professor Johnson

3-19-25

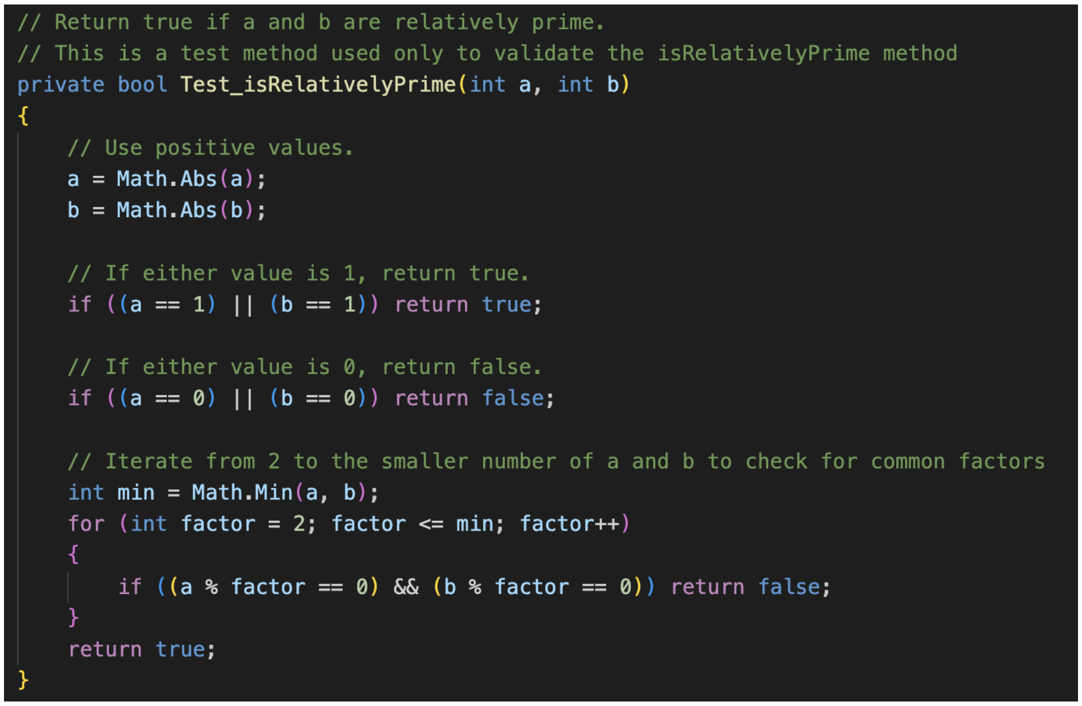
Assignment #3

- Problem 7.1:
  - 

```
// Use Euclid's algorithm to calculate the GCD
// See en.wikipedia.org/wiki/Euclidean_algorithm
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:
  - First, the programmer may have fully applied a top-down design, leading to overly detailed and redundant descriptions of each line of code. When using this approach, stopping before writing the actual code is essential. In this case, the last step would be: Use Euclid's algorithm to find the GCD. -> See en.wikipedia.org/wiki/Euclidean_algorithm. The second reason could be that the programmer added comments after completing the code, resulting in mere descriptions of each line rather than explanations of their purposes.
- Problem 7.4:
  - The code for validation created in Exercise 3 is already offensive. It checks both the inputs and the output, as well as the Debug.Assert method will throw an exception if any issues arise.
- Problem 7.5:
  - The intention is for the calling code to manage any errors. Currently, if any exceptions are thrown, they are sent to the calling code for it to address. This indicates no need to include error-handling code at this level.
- Problem 7.7:
  - Getting to the supermarket

        - A. Find the car
        - B. Open the car door
        - C. Start the car
        - D. Back halfway out of the long driveway into the free space (to turn out)
        - E. Turn left and continue up the driveway to exit
        - F. Turn right. Drive to the first stoplight
        - G. Turn right again. Drive to the first stoplight.
        - H. Turn left. Drive until you see the supermarket.
        - I. Turn into the supermarket parking lot.
        - J. Find an empty parking space and park in it.
        - K. Stop the car and get out.
        - L. Enter the supermarket
        - M. Buy Calypso Lemonade and brownie bites.
    - Assumptions
        - The car is parked facing away from the driveway (on the way down, you're going to be facing that way, so that's how it's parked)
        - You adjust the seat and mirrors
        - You have gas
        - You know how to drive
        - No people or objects are blocking the way
        - There is at least one free space in the parking lot
        - Supermarket is open
- Problem 8.1:
    - Program to test
        - 

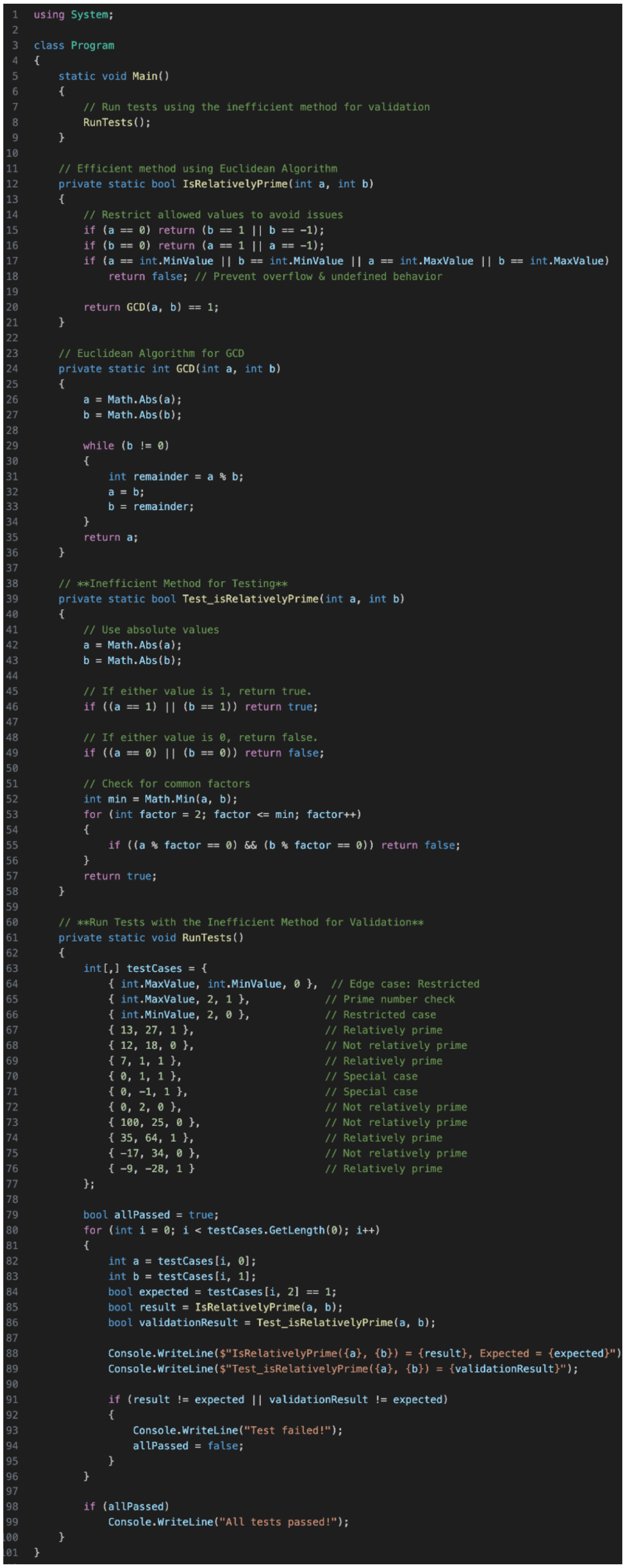
```
// Return true if a and b are relatively prime.
// This is a test method used only to validate the isRelativelyPrime method
private bool Test_isRelativelyPrime(int a, int b)
{
    // Use positive values.
    a = Math.Abs(a);
    b = Math.Abs(b);

    // If either value is 1, return true.
    if ((a == 1) || (b == 1)) return true;

    // If either value is 0, return false.
    if ((a == 0) || (b == 0)) return false;

    // Iterate from 2 to the smaller number of a and b to check for common factors
    int min = Math.Min(a, b);
    for (int factor = 2; factor <= min; factor++)
    {
        if ((a % factor == 0) && (b % factor == 0)) return false;
    }
    return true;
}
```

- Problem 8.3:
    - a.) Exercise 1 doesn't give you the isRelativelyPrime method, so it has to be a black-box test.
    - b.) If you were told how it works, then white and gray-box tests could be written. You could attempt to conduct an exhaustive test. Still, with permissible values extending from -1 million to 1 million, there would be trillions of combinations of values to examine, which is likely excessive. If the permissible values were limited to –1,000 and 1,000, you would need to test around 1 million combinations, making it feasible.
- Problem 8.5:



```csharp
using System;

class Program
{
    static void Main()
    {
        // Run tests using the inefficient method for validation
        RunTests();
    }

    // Efficient method using Euclidean Algorithm
    private static bool IsRelativelyPrime(int a, int b)
    {
        // Restrict allowed values to avoid issues
        if (a == 0) return (b == 1 || b == -1);
        if (b == 0) return (a == 1 || a == -1);
        if (a == int.MinValue || b == int.MinValue || a == int.MaxValue || b == int.MaxValue)
            return false; // Prevent overflow & undefined behavior

        return GCD(a, b) == 1;
    }

    // Euclidean Algorithm for GCD
    private static int GCD(int a, int b)
    {
        a = Math.Abs(a);
        b = Math.Abs(b);

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

    // **Inefficient Method for Testing**
    private static bool Test_isRelativelyPrime(int a, int b)
    {
        // Use absolute values
        a = Math.Abs(a);
        b = Math.Abs(b);

        // If either value is 1, return true.
        if ((a == 1) || (b == 1)) return true;

        // If either value is 0, return false.
        if ((a == 0) || (b == 0)) return false;

        // Check for common factors
        int min = Math.Min(a, b);
        for (int factor = 2; factor <= min; factor++)
        {
            if ((a % factor == 0) && (b % factor == 0)) return false;
        }
        return true;
    }

    // **Run Tests with the Inefficient Method for Validation**
    private static void RunTests()
    {
        int[,] testCases = {
            { int.MaxValue, int.MinValue, 0 },  // Edge case: Restricted
            { int.MaxValue, 2, 1 },            // Prime number check
            { int.MinValue, 2, 0 },            // Restricted case
            { 13, 27, 1 },                     // Relatively prime
            { 12, 18, 0 },                     // Not relatively prime
            { 7, 1, 1 },                       // Relatively prime
            { 0, 1, 1 },                       // Special case
            { 0, -1, 1 },                      // Special case
            { 0, 2, 0 },                       // Not relatively prime
            { 100, 25, 0 },                    // Not relatively prime
            { 35, 64, 1 },                     // Relatively prime
            { -17, 34, 0 },                    // Not relatively prime
            { -9, -28, 1 }                     // Relatively prime
        };

        bool allPassed = true;
        for (int i = 0; i < testCases.GetLength(0); i++)
        {
            int a = testCases[i, 0];
            int b = testCases[i, 1];
            bool expected = testCases[i, 2] == 1;
            bool result = IsRelativelyPrime(a, b);
            bool validationResult = Test_isRelativelyPrime(a, b);

            Console.WriteLine($"IsRelativelyPrime({a}, {b}) = {result}, Expected = {expected}")
            Console.WriteLine($"Test_isRelativelyPrime({a}, {b}) = {validationResult}");

            if (result != expected || validationResult != expected)
            {
                Console.WriteLine("Test failed!");
                allPassed = false;
            }
        }

        if (allPassed)
            Console.WriteLine("All tests passed!");
    }
}
```

○

    - The areRelativelyPrime method had some problems. The first version didn't have restrictions on a and b, and it couldn't handle the maximum and minimum possible values. So, I just restricted the values. The rest of it was fine. The testing code made me consider edge cases such as -1, 0, and 1.
- Problem 8.9:
    - Exhaustive tests are black-box tests because they don't rely on knowing how the method they are testing works.
- Problem 8.11:
    - You can use each pair of testers to calculate three different Lincoln indexes.
        - Alice/Bob: $5 \times 4 \div 2 = 10$
        - Alice/Carmen: $5 \times 5 \div 2 = 12.5$
        - Bob/Carmen: $4 \times 5 \div 1 = 20$
        - You might calculate an average of the three numbers to arrive at a rough estimate of $(10 + 12.5 + 20) \div 3 \approx 14$ bugs. Another option would be to prepare for the maximum by assuming there will be 20 bugs. In both scenarios, it's essential to keep monitoring the number of bugs discovered to adjust your estimate as you gather more data.
- Problem 8.12:
    - If the testers fail to identify any shared bugs, the calculation for the Lincoln index results in division by zero, leading to an infinite solution. This indicates that you have no information regarding the total number of bugs present. You can estimate a kind of lower limit for the bug count by assuming that the testers discovered one bug in common. For instance, if the testers reported 3 and 4 bugs, respectively, then the lower bound index would be $(3 \times 4) \div 1 = 12$ bugs.