7.1:

```
// Use Euclid's algorithm to calculate the GCD.
private long GCD(long a, long b) {
   // Repeat until we're done
   while (b != 0) {
      // Set remainder to the remainder of a / b
      long remainder = a % b;
      // Set a = b and b = remainder
      a = b;
      b = remainder;
   }
   // Return the final GCD value
   return a;
}
```

7.2:

Bad comments can come about due to the need to just have barely good enough comments to quickly move on to the next part of the software project, or the code gets edited and updated so that the comments lose their meaning.

7.4:

You could add explicit debug statements that check for proper typing and invalid inputs, then throw specific kinds of errors given invalid types of input. That way, information that produces invalid inputs are made clear to the programmer.

7.5:

Yes, if the program is going to throw an error without crashing, it needs to be able to handle it somehow.

7.7:

(assuming you own a car) get in your car and start it
Drive to the nearest supermarket

8.1:

```
public class RelativePrimeTest {

   // Method to compute the Greatest Common Divisor (GCD) using Euclid's algorithm
   private static long GCD(long a, long b) {
      while (b != 0) {
         long remainder = a % b;
         a = b;
         b = remainder;
      }
      return Math.abs(a);
```

```
    }

    // Method to check if two numbers are relatively prime
    private static boolean isRelativelyPrime(long x, long y) {
        return GCD(x, y) == 1;
    }

    // Test the isRelativelyPrime method
    public static void main(String[] args) {
        // Sample test cases
        long[][] testCases = {
            {21, 35},   // Not relatively prime (GCD = 7)
            {8, 9},     // Relatively prime (GCD = 1)
            {-1, 100},  // -1 is relatively prime to any number
            {17, 31},   // Both are prime, so they are relatively prime
            {0, 5},     // 0 is only relatively prime to ±1
            {14, 25},   // 14 = 2x7, 25 = 5x5, so relatively prime
            {49, 77}    // Not relatively prime (GCD = 7)
        };

        // Run tests
        for (long[] pair : testCases) {
            System.out.printf("isRelativelyPrime(%d, %d) = %b%n", pair[0], pair[1],
isRelativelyPrime(pair[0], pair[1]));
        }
    }
}
```

8.3:

a) I used the black-box method for testing because I designed test cases based on the expected behavior of the isRelativelyPrime() function without looking at the internal implementation details.

b) Alternatively, white-box testing could be used if we wanted to ensure that every branch and loop in the GCD and isRelativelyPrime methods are covered, analyzing control flow to make sure that different paths are properly executed. If we only had partial knowledge of the implementation, gray-box testing could be used, such as if we knew it used Euclid's algorithm without seeing the exact code. We could create cases that specifically challenge the efficiency and edge cases of the algorithm. Testing every case exhaustively would not be very practical, however, due to the sheer number of possible cases. We instead rely on a representative subset of cases to cover key scenarios.

8.5:

Below is an implementation of the testing and method in C#:

```
using System;
```

```
class RelativePrimeTest
{
    // Returns true if a and b are relatively prime
    private static 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);

        // Compute GCD and check if it's 1
        return GCD(a, b) == 1;
    }

    // Compute the Greatest Common Divisor using Euclid's algorithm
    private static int GCD(int a, int b)
    {
        a = Math.Abs(a);
        b = Math.Abs(b);

        // Handle zero cases correctly
        if (a == 0) return b;
        if (b == 0) return a;

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

    // Test the AreRelativelyPrime method
    static void Main()
    {
        // Define test cases
        int[,] testCases = {
            {21, 35},  // Not relatively prime (GCD = 7)
            {8, 9},    // Relatively prime (GCD = 1)
            {-1, 100}, // -1 is relatively prime to any number
            {17, 31},  // Prime numbers are always relatively prime
            {0, 5},    // 0 is only relatively prime to ±1 (should return false)
            {14, 25},  // 14 = 2x7, 25 = 5x5, relatively prime (GCD = 1)
            {49, 77}   // Not relatively prime (GCD = 7)
```

```
        };

        // Run tests
        for (int i = 0; i < testCases.GetLength(0); i++)
        {
            int a = testCases[i, 0];
            int b = testCases[i, 1];
            Console.WriteLine($"AreRelativelyPrime({a}, {b}) = {AreRelativelyPrime(a, b)}");
        }
    }
}
```

Fixes:
Fixed Math.abs(a) → Math.Abs(a) and Math.abs(b) → Math.Abs(b)

C# uses Math.Abs() with an uppercase "A".
Fixed improper return condition (gcd == -1)

The GCD is always non-negative after applying Math.Abs(), so checking for -1 is unnecessary.
Replaced for(; ;) loop with while (b != 0)

A while loop is more readable and idiomatic.
Fixed infinite loop risk when b == 0

The previous GCD function would proceed into the loop without handling b == 0 properly.

8.9:
Exhaustive testing technically falls under white-box testing, since you are checking every possible control flow explicitly in terms of a program's possible inputs and execution paths.

8.11:
Estimate the total number of bugs by taking the product of the cardinality of each tester's set of bugs, and dividing them all by the number of overlapping bugs. In this case, it would be (5 * 4 * 5)/ 5 = 20 bugs - 10 bugs already found (unique in sets) = the Lincoln index estimates that there are about 20 total bugs in the system, meaning 10 more bugs are yet to be discovered.

8.12:
If two testers don't find any overlapping bugs, the formula becomes undefined because the denominator of the number of shared bugs is zero, meaning the estimate blows up to infinity or is not computable.