#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;
        };
}
```

#7.2

There are two probable explanations for the poor quality of comments in the original GCD code. Firstly, the programmer might have pursued a top-down design approach meticulously, resulting in overly detailed descriptions of the code. While this is a commendable programming practice, it often leads to redundant comments as each comment corresponds to a code statement. Secondly, such comments may have been added after the code was written. It's common in this scenario for the comments to merely describe what each line of code does without providing insight into why it's executed that way. This resembles a simplistic book report where you recount the story without delving into deeper analysis akin to a professional book reviewer.

#7.4

The validation code written for Exercise 3 is already fairly offensive. It validates the inputs and the result, and the Debug.Assert method throws an exception if there is a problem.

#7.5

Error handling code could be incorporated into the GCD method, but the preferred approach is for the calling code to manage any potential errors. Currently, if the code encounters exceptions, they are propagated up to the calling code for handling. Consequently, there's no necessity to introduce error handling code here.

#7.7

a. Go down to the lobby using the elevator
b. Exit the elevator
c. Turn right
d. Turn another right in the intersection
e. Exit through the door

f. Turn right
g. Walk straight and enter the supermarket

Assumption:
- The elevator is working
- The supermarket is still open

#8.1

```
def test_is_relatively_prime(integer1, integer2):
   if is_relatively_prime(integer1, integer2):
      print(f"The integers {integer1} and {integer2} are relatively prime.")
   else:
      print(f"The integers {integer1} and {integer2} are not relatively prime.")

def is_relatively_prime(integer1, integer2):
   # Check if either integer is 1 or -1
   if integer1 == 1 or integer1 == -1 or integer2 == 1 or integer2 == -1:
      return True  # By definition, 1 and -1 are relatively prime to every integer

   # Check if either integer is 0
   if integer1 == 0 or integer2 == 0:
      return False  # 0 is not relatively prime to any integer except 1 and -1

   # Find the greatest common divisor (GCD) using the Euclidean algorithm
   gcd = compute_gcd(integer1, integer2)

   # If the GCD is 1, the integers are relatively prime
   if gcd == 1:
      return True
   else:
      return False

def compute_gcd(a, b):
   # Compute the greatest common divisor (GCD) using the Euclidean algorithm
   while b != 0:
      temp = b
      b = a % b
      a = temp
   return a
```

#8.3

Since the Exercise 1 prompt lacks details on the inner workings of the AreRelativelyPrime method, it necessitates a black-box test approach. However, if the workings of the AreRelativelyPrime method were disclosed, it would enable the creation of both white-box and gray-box tests.Attempting an exhaustive test might seem feasible, but considering the permitted values ranging from -1 million to 1 million, it would entail testing approximately 4 trillion pairs of values [(1,000,000 - (-1,000,000) + 1)^2 ≈ 4 trillion], which is likely impractical. Alternatively, if the permitted values were limited to -1,000 to 1,000, the number of pairs to test would reduce to around 1 million, making such testing viable.

#8.5

```
import math

def are_relatively_prime(a, b):
   if a == 0:
      return abs(b) == 1
   if b == 0:
      return abs(a) == 1

   gcd = compute_gcd(a, b)
   return gcd == 1 or gcd == -1

def compute_gcd(a, b):
   a = abs(a)
   b = abs(b)

   if a == 0:
      return b
   if b == 0:
      return a

   while True:
      remainder = a % b
      if remainder == 0:
         return b
      a = b
      b = remainder
```

While developing this program, I encountered issues with the AreRelativelyPrime method. The initial version lacked constraints on the values of a and b, causing difficulties in handling the maximum and minimum possible integer values. This prompted me to impose restrictions on the allowed values. Such limitations often arise during testing, leading to adjustments in the implementation.

#8.9

Exhaustive testing falls under the category of black-box testing. The reason for this is that in exhaustive testing, the internal structure, design, and implementation details of the software are not taken into consideration. Instead, exhaustive testing focuses solely on testing the software's functionality against its specified requirements or expected behaviors. It’s not white box testing where we have the knowledge of the internal structure and code and definitely not gray box testing (which incorporates both white and black box testing).

<u>#8.11</u>

You can calculate three different Lincoln indexes using each pair of testers:

- Alice/Bob: 5 × 4 ÷ 2 = 10
- Alice/Carmen: 5 × 5 ÷ 2 = 12.5
- Bob/Carmen: 4×5÷1=20

You could take an average of the three to get a rough estimate of (10 + 12.5 + 20) ÷ 3 ≈ 14 bugs.

<u>#8.12</u>

If testers fail to find any bugs in common, the equation for the Lincoln index results in division by zero, yielding an infinite result. Essentially, this implies a complete lack of information regarding the total number of bugs.

To derive a lower bound for the number of bugs, you can assume that the testers found at least one bug in common. For instance, if the testers discovered 5 and 6 bugs, respectively, the lower bound index would be calculated as (5 × 6) ÷ 1= 30 bugs.