Problem 7.1, Stephens page 169

```
// Use Euclid's algorithm to calculate the GCD.
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, Stephens page 170

The bad comments resulted from excessive top-down design, where each step was broken down too rigidly, making the comments redundant. Additionally, the comments described what the code does instead of why, lacking valuable clarification.

Problem 7.4, Stephens page 170

To apply offensive programming, add preconditions to validate inputs, ensuring they are non-negative and not both zero, and throw exceptions for invalid cases, then use assertions during execution to verify that intermediate values remain valid and the GCD result is always positive, therefore catching errors immediately rather than silently introducing bugs.

Problem 7.5, Stephens page 170

Yes, you should add error handling to the modified code to prevent crashes when encountering invalid input. Proper error handling, such as throwing exceptions for invalid values, helps catch issues early, making debugging easier while preventing unexpected failures during execution.

Problem 7.7, Stephens page 170

1. Start the car
2. Navigate to the supermarket
3. Park the car
4. Turn off the car
5. Enter the supermarket

Assumptions:

- Driver knows how to operate the car
- Supermarket is within a reasonable driving distance

- Driver has access to directions if unfamiliar with the route
- Parking lot has available spaces
- Car has enough gas

Problem 8.1, Stephens page 199

```
def is_relatively_prime(a, b):

    # If either number is ±1, then by definition it's relatively prime to everything.

    if a == 1 or a == -1 or b == 1 or b == -1:

        return True

    # If either number is 0, then they're not relatively prime (unless the other is ±1).

    if a == 0 or b == 0:

        return False

    # Otherwise, compute gcd and check if it is ±1

    return abs(gcd(a, b)) == 1

def gcd(x, y):

    while y != 0:

        x, y = y, x % y

    return x


def test_is_relatively_prime(x, y):

    if is_relatively_prime(x, y):

        print(f"{x} and {y} ARE relatively prime.")

    else:

        print(f"{x} and {y} are NOT relatively prime.")
```

Problem 8.3, Stephens page 199

For my relatively prime program from Exercise 8.1, I mostly treated it like a black-box test, feeding in different pairs of integers and checking whether the output matched my expectations. I mainly focused on the edge cases such as 0, ±1, and numbers that share divisors. If I knew more about how the function worked internally such as where it checks for zero or how it calculates the GCD, I could then perform white-box tests that target each specific path in the code. A gray-box approach would be somewhere in between, using my little knowledge of the code structure to pick out tricky test values, while I am not too sure about Exhaustive as in theory it is possible to run all possible pairs from -1000000 to 1000000 but that would involve billion checks which is not really suitable in practice.

Problem 8.5, Stephens page 199 - 200

```
def are_relatively_prime(a, b):
    if a == 0:
        return b == 1 or b == -1
    if b == 0:
        return a == 1 or a == -1
    return abs(gcd(a, b)) == 1

def gcd(a, b):
    a, b = abs(a), abs(b)
    if a == 0:
        return b
    if b == 0:
        return a
    while True:
        r = a % b
        if r == 0:
            return b
        a, b = b, r

def test_are_relatively_prime(a, b):
    if are_relatively_prime(a, b):
        print(f"{a} and {b} are relatively prime.")
    else:
        print(f"{a} and {b} are NOT relatively prime.")
```

When I implemented this version and ran my testing code, I discovered a small bug in my earlier version that the special case handling for zero wasn't strict enough, only 1 and -1 should be considered relatively prime to 0, but my initial version simply returned False when either number was 0, even if the other was 1 or -1. The testing code helped me uncover this oversight, ensuring that edge cases such as (0,1) or (0,-1) are correctly handled. This experience

demonstrated the significant benefit of systematic testing in catching and correcting subtle errors.

Problem 8.9, Stephens page 200

Exhaustive testing falls under black-box testing because it focuses solely on verifying that every possible input produces the expected output based on the program's specifications, without any regard for the internal workings or the structure of the code. In other words, exhaustive testing treats the system as a "black box" where only the inputs and outputs matter, not how the software achieves its results internally.

Problem 8.11, Stephens page 200

Alice and Bob
N = (5*4)/2 = 20/2 = 10
Bob and Carmen
N = (4*5)/1 = 20/1 = 20
Alice and Carmen
N = (5*5)/2 = 25/2 = 12.5

(10+20+12.5)/3 = 14.1667 = 14 estimated total bugs
Bugs still at large = 14-10 = 4 bugs

Problem 8.12, Stephens page 200

If the two testers don't find any bugs in common, the Lincoln index formula becomes undefined due to division by zero. This suggests that the actual number of bugs is likely much higher since testers are uncovering completely separate issues. However, a lower bound estimate can still be determined by summing the unique bugs found.