7.1

Leave the first comment then remove the rest. All the others are unnecessary and redundant as they simply describe what the code does in the line below which a programmer should be able to understand already.

7.2

This could happen if your planning of the algorithm is poor and you go back to comment the code after numerous changes. This could also happen if adding comments simply to satisfy a project manager and trying to make them look good.

7.4

You could apply offensive programming by applying some sort of check to the numbers given to make sure that they are valid numbers such as not 0 or 1, and only run the algorithm if they meet the criteria.

7.5

Since the code is simple and easily implemented, I don’t think that error handling is required in this function. However, it would be simple to have some sort of check that verifies that the GCD returned does indeed divide both a and b.

7.7

Assuming starting right outside the car with key in hand.

```
Enter and start car
    Unlock car with key
    Get in car
    Insert key
    Fully turn to start car
    Turn off emergency brake
Steering to supermarket
    Put car in reverse
    Back out
    Put car in drive
    Steer out of garage
    Turn left onto Manchester
    Go straight to Lincoln, turn right
    Turn right into Ralph’s parking lot
```

Parking

```
    Find empty spot, park
    Turn car off, set brake
```

8.1

Write another method that compares two integers for relative primes called isAlsoRelativelyPrime

```
While (testCount < 1000)
    testNum1 = generate Random Number between -1000000 and 1000000
    testNum2 = generate Random Number between -1000000 and 1000000
    if (isRelativelyPrime(testNum1, testNum2) != isAlsoRelativelyPrime(testNum1, testNum2))
        output both numbers and results of each method
    testCount++
```

some hardcoded unit tests such as boundary cases with 0, 1, and -1

8.3

I used black-box testing. I tried a large sample of random tests that will simply alert me if there is a discrepancy between the two relatively prime methods but not where the error lies. However, just having this would be a good tool for a brief overview of the effectiveness of the methods. If it makes it past this, then we have a relatively good chance of it doing well. Then afterwards we test boundary cases such as +-1000000, +-1, and 0 which have special properties, as well as duplicate numbers, etc. If you had more of an idea of how the method worked, you could use white-box since you would be able to customize the tests more closely to the working of the function.

8.5

I'm not too sure what the definition of relatively prime provides in this case but if a and b are both 0, the program loops instead of returning true or false.

8.9

Exhaustive testing would fall into black box testing since you aren't making your tests based on what you know of the program you're testing. You're simply making a program that will test every single input possible regardless of whether or not they are trivial.

8.11

{1,2,3,4,5}, {2,5,6,7}, {1,2,8,9,10}

(P1 x B)(P2 x B)/(P1 x P2 x B) = 5x4/2 = 10

(P2 x B)(P3 x B)/(P2 x P3 x B) = 4x5/1 = 20

(P1 x B)(P3 x B)/(P1 x P3 x B) = 5x5/2 = 12.5

10+20+12.5 = 42.5/3 = 14 bugs

8.12

It means that since they didn't find any bugs in common, you must assume that they have extremely different styles of searching for bugs so the cause of no overlap isn't a fluke. Then just assume that the number of bugs is the product of how many each of them found.