Katie Nguyen

Sofia Ruiz

Moriah Tolliver

**Homework 03**

Problem 7.1, Stephens page 169

|  |
| --- |
| // Use Euclid’s algorithm to calculate the greatest common divisor  // Algorithm: <https://mathworld.wolfram.com/EuclideanAlgorithm.html>  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

One way to end up with bad comments like those in the previous exercise is the developer adding comments after the code is complete. Most likely, the programmer went through and restated what each line does instead of providing an overarching picture of what the code should do. Another way to end up with these sorts of comments is when the top-level design goes into so much detail that it describes each line of code, and the programmer uses these descriptions as comments.

Problem 7.4, Stephens page 170

Assuming that validation code has been added to the code from problem 1, the code would already be offensive, because it would throw some kind of error when b is 0.

Problem 7.5, Stephens page 170

Error handling should not be added to the code for Exercise 4 because it is a single function that throws an error. Instead, error handling should be added to the function that calls gcd to respond to any errors thrown by the gcd function.

Problem 7.7, Stephens page 170

a. Open garage door

b. Walk to car

c. Open the car

d. Start the car

e. Back out of the garage

f. Drive in reverse

g. Turn wheel to the left

h. Change gears to drive. Drive until the street ends.

i. Turn left. Drive until the first stoplight

j. Turn left. Drive until the first stoplight

k. Turn right. Drive until the first light

l. Turn left. Drive until you see super market

m. Turn into the supermarket parking lot

n. Find an empty parking spot and park in it

o. Stop the car

p. Turn off the car and get out

Problem 8.1, Stephens page 199

In order to test the relatively prime method, we can write another method that also tests if two integers are relatively prime, but with a different algorithm.

def relativelyPrimeTwo(int a, int b):

a = abs(a)

b = abs(b)

if a is 1 or b is 1: return True

if a is 0 or b is 0: return False

min = min(a, b)

factor = 2

for factor in range min:

if a % factor is 0 and b % factor is 0:

return False

return True

Now, with this second method, we can compare the results of both to test the original one. In pseudocode, the tests would look something like:

Pick random a and b:

Assert AreRelativelyPrime (a, b) = relativelyPrimeTwo (a, b)

Problem 8.3, Stephens page 199

We used a black-box test, because we didn’t explain how the method actually works. If we wanted to do white-box or gray-box tests, we would have had to first explain how the method works. The only way to test this would be to perform an exhaustive test, but there would be approximately one million pairs to test, which is not feasible.

Problem 8.5, Stephens page 199 - 200

After implementing the method and testing code, we did not find any bugs in the initial version of the method or the testing code. We did however run into some edge cases that were tricky to think about in the cases that the numbers being passed in were -1, 0, or 1. In this sense, these edge case values from the testing code provided us more insight in the method being tested and the values being passed in that we needed to consider special outcomes for.

Problem 8.9, Stephens page 200

Exhaustive testing falls into the black-box category. This is because these tests do not know or take into consideration what is happening within the method being tested.

Problem 8.11, Stephens page 200

Three different Lincoln indexes can be used to estimate the total number of bugs. These can be calculated by combining each pair of testers. For example: Alice/Bob - 5 x 4 / 2 = 10, Alice/Carmen - 5 x 5 / 2 = 12.5, Bob/Carmen - 4 x 5 / 1 = 20. By taking the average of these three numbers, the average is about 14 bugs. At large, the worst-case scenario would be 20 bugs.

Problem 8.12, Stephens page 200

If two testers don’t find any bugs in common, the Lincoln estimate is calculated by dividing by zero. Essentially, this means that the answer will produce an infinite result, which does not provide any insight as to how many bugs there are. A “lower bound” estimate of the number of bugs is achievable by simulating that the users found a common bug, using one as the denominator in the equation.