Jen Shin

CMSI 402

12 March 2018

HW #3 Chapters 7 & 8

7.1 The comments are unnecessary because they don’t say much different from what the code actually shows directly. It would be better to elaborate more on why and how the code works. In addition, adding a link for more details on GCD would be nice as well.

// Use Euclid's algorithm to calculate the GCD.

// For more info, checkout https://en.wikipedia.org/wiki/Euclidean\_algorithm

provate long GCD( long a, long b )

{

a = Math.abs( a );

b = Math.abs( b );

// Repeat until b is zero

for( ; ; )

{

long remainder = a % b;

// If b can perfectly divide a, b is the GCD

If( remainder == 0 ) return b;

a = b;

b = remainder;

};

}

7.2 The programmer might have written the code after the code was finished. As a result, the programmer simply wrote (translated) each line of code into English sentences without explaining why they wrote following codes. Another reason would be if the programmer used a top-down design to write the comments first, and wrote the code after it therefore leaving only the simple comments that duplicate the code behind.

7.4 The code for 7.3 has many preexisting offensive factors. It has Debug.Assert in the beginning, checking for the validity of the arguments and checks again if the remainder of a and b is 0 before returning gcd to check for the result as well. But for extra carefulness, we can add a checking loop to see if there is a larger value that can divide both a and b without any remainders.

7.5 You can but the code would be simpler for the caller to handle the error since the caller would get an error if there is a problem any time in the code. Another way would be to wrap everything in try and catch in callee (gcd), but error handling in the gcd code is not necessary.

7.7

1. Locate the car

2. Get in the car

3. Open / Unlock phone

4. Open google maps

5. Type in the market location

6. Start Navi

7. Start the car

8. Unpark

9. Leave parking lot

10. Follow the navi’s instruction to get the market in fastest route

// assuming the driver is at LMU

// assuming driver uses google maps to navigate

// assuming the navi changes slightly depending on the traffic and time

8.1

def gcd (x, y):

while y > 0:

x, y = y, x % y

return x

def coprime (x, y):

return gcd(x, y) == 1

# print or assert to test

# make several statements of it using different x and y with the format given below

# check for a case where x or y is 0, 1, -1

# check for a case where x or y is over 1 million or less than -1 million

# check for numbers in between 1 mil and -1 mil

print(isRelativelyPrime(x, y) == coprime(x, y))

assert(isRelativelyPrime(x, y) == coprime(x, y))

8.3 Exhaustive would only be possible on faster computers because 2 million possible values per argument are too much. If the range was smaller like 100~1000 exhaustive would be plausible. Because we’re not given the content of the isRelativelyPrime method, we used black-box testing. If we had partial knowledge it of the code inside the method, we can use gray-box testing and if we have access to the whole method we can use white-box testing.

8.5 Now that we have access to the method, we can write a gray/white box test for it instead of black box. But because the range is too wide (-1 mil, 1 mil) we cannot do exhaustive testing. The edited code checks for edge cases of 0, 1, -1 more specifically and also checks for -1 in coprime function to account for negative arguments.

def gcd (x, y):

while y > 0:

x, y = y, x % y

return x

def coprime (x, y):

gcd\_value = gcd(x,y)

return gcd\_value == -1 || gcd\_value == 1

# print or assert to test

# make several statements of it using different x and y with the format given below

# check for a case where x or y is over 1 million or less than -1 million

# check for numbers in between 1 mil and -1 mil

print(isRelativelyPrime(x, y) == coprime(x, y))

assert(isRelativelyPrime(x, y) == coprime(x, y))

# edge cases

# check for a case where x or y is 0, 1, -1

Assert(isRelativelyPrime(-1, 0) == True)

Assert(isRelativelyPrime(0, 1) == True)

8.9 Exhaustive testing falls under black box testing because we don’t need to know about the code in the method to check all possible values.

8.11

\* Alice: 5, Bob: 4, Common: 2 = 5 \* 4 / 2 = 10

\* Alice: 5, Carmen: 5, Common: 2 = 5 \* 5 / 2 = 12.5

\* Bob: 4, Carmen: 5, Common: 1 = 4 \* 5 / 1 = 20

(10 + 12.5 + 20) / 3 = 14. We can estimate total of 14 bugs but for worst case it can go up to 20.

8.12 Then the divisor would be 0 and the answer would be undefined/null/cannot be determined. This means that we cannot determine how many bugs there are either. We can pretend that they have 1 bug in common to get a lower bound estimate of the number of bugs.