Andrea Carver

CMSI 402

B.J. Johnson

12 March 2018

*Homework 3*

*Problem 7.1*

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

provate long GCD( long a, long b )

{

// Make a and b positive

a = Math.abs( a );

b = Math.abs( b );

// Repeat until the GCD is found.

for( ; ; )

{

// Calculate the remainder of a / b

long remainder = a % b;

// If a is divisible by b, then b is the GCD.

If ( remainder == 0 ) return b;

// Try again, but divide b by the remainder.

a = b;

b = remainder;

};

}

*Problem 7.2*

The first condition could be writing comments as the programmer codes. Another condition is to write the comments after the code is finished. Both conditions lead to the programmer writing what the code does, not what it should do. The comments should not change when the particular implementation of that specified purpose changes.

*Problem 7.4*

Instead of taking the absolute value of a and b, you could assert that both a and b need to be greater than zero. In the style of the book:

Debug.Assert(a > 0);  
Debug.Assert(b > 0);

*Problem 7.5*

It could be helpful, as it could potentially be not immediately clear to the end-user that the intent of checking if a number is greater than zero is that to determine its sign. Error handling could help the end user grasp the meaning of the exception message.

*Problem 7.7*

Possible high level instructions could include:

* Optimize seat and mirror orientation
* Navigate along path
* Change lane
* Maintain speed
* Detect pedestrians
* Respond to hazard

*Problem 8.1*

anyTestsFailed = false;

assert(found, expected, codeString) {

if (found != expected) {  
anyTestsFailed = true;

print(“Test failed: expected “ + codeString + “ to evaluate to “ + expected + “, but got “ + found)

}

tests = [

{

found: IsRelativelyPrime(21, 35),

expected: false

codeString: “IsRelativelyPrime(21, 35)”

},

{

found: IsRelativelyPrime(-1, 0),

expected: true

codeString: “IsRelativelyPrime(-1, 0)”

},

{

found: IsRelativelyPrime(1, 0),

expected: true

codeString: “IsRelativelyPrime(1, 0)”

},

{

found: IsRelativelyPrime(1, 4),

expected: false

codeString: “IsRelativelyPrime(1, 4)”

}

];

for (test in tests) {

assert(test[‘found’], test[‘expected’], test[‘codeString’]);

}

if (!anyTestsFailed) {

print(“All tests passed”);

}

*Problem 8.3*

I used black-box testing. I knew what the function was supposed to do, but not how it worked. I could only use black-box testing because no implementation of the function was provided. Thus, white-box and grey-box testing methods are not possible. Exhaustive testing is impractical, as there are 2,000,000 x 2,000,000 = 4,000,000,000,000 possible pairs of inputs that could be tested.

*Problem 8.5*

function GCD(a, b) {

a = Math.abs(a);

b = Math.abs(b);

if (a == 0) { return b; }

if (b == 0) { return a; }

while (true) {

var remainder = a % b;

if (remainder == 0) { return b; }

a = b;

b = remainder;

}

}

function IsRelativelyPrime(a, b) {

if (a == 0) { return ((b == 1) || (b == -1)); }

if (b == 0) { return ((a == 1) || (a == -1)); }

var gcd = GCD(a, b);

return ((gcd == 1) || (gcd == -1));

}

function runTests() {

var anyTestsFailed = false;

function runTest(found, expected, codeString) {

if (found != expected) {

anyTestsFailed = true;

console.log('Test failed: expected ' + codeString + ' to evaluate to ' + expected + ', but got ' + found);

}

}

var tests = [

{

found: IsRelativelyPrime(21, 35),

expected: false,

codeString: 'IsRelativelyPrime(21, 35)'

},

{

found: IsRelativelyPrime(-1, 0),

expected: true,

codeString: 'IsRelativelyPrime(-1, 0)'

},

{

found: IsRelativelyPrime(1, 0),

expected: true,

codeString: 'IsRelativelyPrime(1, 0)'

},

{

found: IsRelativelyPrime(1, 4),

expected: true,

codeString: 'IsRelativelyPrime(1, 4)'

}

];

for (var test in tests) {

runTest(tests[test]['found'], tests[test]['expected'], tests[test]['codeString']);

}

if (!anyTestsFailed) {

console.log('All tests passed');

}

}

runTests();

From coding and running the testing, I found one of my tests was actually wrong! I was also missing a closing semicolon in the runTests method.

*Problem 8.9*

Exhaustive testing falls into the category of black-box testing because the tester is not strategically testing values predicted to trick the method. *All* values are tested, so the tester might as well not know anything about the method.

*Problem 8.11*

I cannot find any information in the book or online about using more than two people with the Lincoln Index, so I will take the average of using the three possible pairs.

Alice and Bob: 5 \* 4 / 2 = 10

Alice and Carmen: 5 \* 5 / 2 = 13

Bob and Carmen: 4 \* 5 / 1 = 20

Average = 14

At an average of 14 total bugs with 10 discovered bugs, around 4 bugs remain undiscovered.

*Problem 8.12*

If *S*, the number of common bugs, is zero, then the number of bugs is undefined as you cannot divide by zero. As S becomes smaller, then the number of total bugs approaches infinity. This means that the more that the two testers are finding unique bugs, the higher probability that there are more undiscovered bugs.