Geoffrey Colman

CMSI 402

20 March 2019

HW #3

7.1 The greatest common divisor (GCD) of two integers is the largest integer that evenly divides them both. Knowing that background, what's wrong with the comments in the following code? Rewrite the comments so that they are more effective.

```
/* Use Euclid's algorithm to calculate the GCD, or greatest common divisor
of two integers (the largest integer that evenly divides them both). You can
learn more about the GCD and the Euclidean algorithm at
https://en.wikipedia.org/wiki/Euclidean_algorithm. */
      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;
         };
      }
```

7.2 Why might you end up with the bad comments shown in the previous code?

Two great ways to end up with bad comments (as in the previous exercise) are to write the comments as one writes the code (and potentially neglecting to revise them after something in the code changes) and to go back and add comments only after all of the code has already been written. In this specific case, most of the comments simply redundantly rephrase the actions the code performs (as opposed to what it *should* do) in “natural” English, so the latter is likely the bigger culprit here.

7.4 How could you apply offensive programming to the modified code you wrote for Exercise 3?

The general idea behind offensive programming is to write code so that it throws an exception any time it encounters a potential problem, rather than defensively obscuring those errors with the sorts of clever tricks the author cites in the factorial example.
There are a variety of ways in which the method could receive bad inputs: namely, anything other than a positive integer (at least according to my reading on the Wikipedia article cited in the text, the algorithmic procedure seems only to be specified for positive integers). Taking the absolute value of a true garbage input in the first step of the algorithm should raise a type error, but that won’t catch negative integers (or zero) or numerical inputs with fractional components (decimals/floating points), and the absolute value validation procedure masks problems arising from negative or zero inputs. So this method should likely start with some assertions to check that the inputs are, in fact, positive integers: I’m not especially familiar with C#, but I assume something of the form `Debug.Assert(a >= 1 && a % 1 == 0);` , repeated for `b` should do. As additional benefits of this, if the inputs pass these offensive assertions, not only do the 2 absolute value steps become extraneous, but we don’t need to explicitly check for a division by zero exception at the `a % b` step.

7.5 Should you add error handling to the modified code you wrote for Exercise 4?

Beyond the fact that one should always write code that handles errors gracefully, “offensive programming” inherently involves raising exceptions that must then be handled somehow.

The author specifically encourages the use of exceptions, rather than error codes (e.g., `return -1;` in the event of an error).

7.7 Using top-down design, write the highest level of instructions that you would use to tell someone how to drive your car to the nearest supermarket. (Keep it at a very high level.) List any assumptions you make.

For starters, I would tend to assume that anyone to whom I would be comfortable lending my car would already possess a valid driver's license (ideally also auto insurance), and therefore be sufficiently competent not to require precise, explicit enumeration of all the fundamentals of how to operate a car (doors, seat belt, ignition, transmission, steering wheel, pedals, etc.); respect traffic laws; or drive in general. I very pointedly *do not* assume, however, that he or she is familiar with the idiosyncrasies of my particular car's user interface.

Also, for simplicity with respect to the specified task at-hand (the supermarket), I'm making the assumption that my car is parked in front of my own house: I don't recall off the top of my head the names of the various residential streets in the Westchester neighborhood around LMU, or the distances between all of them and between them and the major streets, heh.

(*Italic text* in parentheses below denotes editorial commentary explaining what might seem to be an unusual instruction to someone unfamiliar with my car.)

- Unlock the car with the fob. *(The alarm will go off if you only use the key in the door.)*
- Release the parking brake by pulling on the L-shaped pin below the dashboard just in front of your left knee. (*The handle broke off several years ago, and the design of my car necessitates replacing the entire assembly rather than just the handle. I was given a quote of over $400, so it functions just fine with my hack.*)

- Drive straight until you reach the T-intersection with Highland Ave. (approximately 2 blocks).
- Turn right onto Highland Ave.
- Drive straight until you reach a traffic signal at Manhattan Beach Blvd. (approximately 9 blocks).
- Turn right onto Manhattan Beach Blvd.
- Drive straight until you reach the entrance driveway to the Vons parking lot (approximately 1.5 blocks).
- Turn right into the Vons parking lot.
- Park the car. (*It might be best to avoid the parking brake...*)
- Open the door by pulling up on the front of the shiny silver cylindrical door handle at the front of the armrest attached to the door. *(This is an enormous, ongoing source of confusion for anyone who gets in my car, as all 4 inner door handles are of a tremendously unorthodox design.)*
- Lock the car using the fob.

8.1 Two integers are relatively prime (or coprime) if they have no common factors other than 1. For example, 21 = 3 × 7 and 35 = 5 × 7 are not relatively prime because they are both divisible by 7. By definition –1 and 1 are relatively prime to every integer, and they are the only numbers relatively prime to 0.

Suppose you've written an efficient `IsRelativelyPrime` method that takes two integers between -1 million and 1 million as parameters and returns `true` if they are relatively prime. Use either your favorite programming language or pseudocode (English that sort of looks like code) to write a method that tests the

`IsRelativelyPrime` method. (Hint: You may find it useful to write another method that also tests two integers to see if they are relatively prime.)

I assume from the hint that I can use the author's `GCD` method defined in the first exercise; as implied in the above problem statement, the GCD of any 2 numbers that are relatively prime is 1, and that fact can be leveraged here in order to use the `GCD` method as an independent test. The "meat and potatoes" of the test work would thus be the single method defined below which internally compares the outcomes of the 2 functions, repeatedly invoked (not shown here for the sake of brevity) with many different sets of inputs. In addition to the typical integer inputs, it would be important to test for many different edge cases and undesirable inputs, such as 0 and garbage data of incompatible types.

The `outcome` input parameter allows us to deliberately specify test cases in which the expected `IsRelativelyPrime` output would be false, as in the example in the problem statement. The `try-catch` should catch any overtly bad inputs (such as trying to determine if the string "cheese" is relatively prime to a `PacketReader` object), which encompasses a large set of edge cases.

I'm not clear on the proper C# syntax/semantics/best practices for tests (I've already disclaimed that I'm not particularly well-versed in it, but am using it here to be consistent with the author), so I've implemented this simply as a boolean function; some higher-order test code or framework is presumed to handle the overhead,, including argument passing. While there may be some syntax and semantic errors, at least the *intent* of the code should be readily apparent to readers familiar with any of the C-like languages or Java.

```
// Tests the "IsRelativelyPrime" method for accuracy.
private bool testIsRelativelyPrime(bool outcome, int a, int b) {
   try {
         int resultOfGCD = GCD(a, b);
         bool resultOfIsRelativelyPrime = IsRelativelyPrime(a, b);
         bool IsRelativelyPrimeByGCD = resultOfGCD == 1;
         bool testResult = outcome ? resultOfIsRelativelyPrime &&
               IsRelativelyPrimeByGCD : !(resultOfIsRelativelyPrime
                  || IsRelativelyPrimeByGCD);
         return testResult;
   } catch(Exception e) {
         return false;
   }
}
```

8.3 What testing techniques did you use to write the test method in Exercise 1? (Exhaustive, black-box, white-box, or gray-box?) Which ones *could* you use and under what circumstances? [Please justify your answer with a short paragraph to explain.]

It's certainly not *practical* to exhaustively test every single pair of integers in the ±1 million range every time the test suite is run, but (unlike the numerical example quoted in the text with a runtime of over 500 years) it should at least be *possible* to do so within a human time scale. I believe this input space would result in $(2 \times 10^6)^2$, or about 4×10^{12} combinations; assuming we can automate the process to actually *specify* all 4 trillion test cases, if we assume each individual pair constituted a single operation (a highly unrealistic assumption), a computer capable of performing 2×10^9 operations per second (a very crude approximation for a garden variety 2 GHz processor) would therefore take approximately 2000 seconds, or 33 minutes, to check every single pair in the input domain. Each pairwise comparison, however, involves many constituent operations (the GCD algorithm, for example, has a worst-case performance of approximately 7 divisions on numbers near 1 million), so that estimate should be multiplied by a minimum of 7, for an estimate of closer to

4 hours. Add a healthy dose of tests for various forms of bad inputs, and a 4.5 hour estimate seems reasonable. It's certainly possible to run these tests overnight after the development team has gone home, but no team would want to do that every single time a test needed to be run during the day. Some form of representative and comprehensive test regime (including as many adversarial inputs as possible) would strike a better performance compromise for "live" code.

I wrote my test code with absolutely no knowledge of the underlying implementation of the `IsRelativelyPrime` method (only the function signature), so that aspect of it could certainly qualify as a "black-box" technique. Even though I have full knowledge of the `GCD` method's implementation, that knowledge did not appreciably factor into my test procedure (though it did very much in the aforementioned performance estimate). The function signature and the semantic point that `GCD(a, b) == 1` implies a and b to be relatively prime (which is related to the underlying reality the algorithm reflects) were specifically relevant to my test code, so that could be said to be "black-box" as well. In retrospect, it occurs to me that I may have been subconsciously aware that `GCD` uses the absolute values of its inputs, but I can also chalk that one up to the definition of the algorithm itself.

Under these specific circumstances, true "white-" or "grey-box" testing for `IsRelativelyPrime` are not possible: the implementation is not known, so no knowledge of the method's inner workings can be brought to bear on potentially problematic inputs/outputs. Only with some (or full, in the case of "white-box") knowledge of the code can either of these techniques be utilized. As mentioned (exhaustively, heh) above, continuous exhaustive testing is not practical in this case due to performance concerns. It can be automated to be done overnight, but likely no more than once per business day.

8.5 The following code shows a C# version of the `AreRelativelyPrime` method and the `GCD` method it calls.

```
// Return true if a and b are relatively prime.
private bool AreRelativelyPrime( int a, int b )
{
   // Only 1 and -1 are relatively prime to 0.
   if( a == 0 ) return ((b == 1) || (b == -1));
   if( b == 0 ) return ((a == 1) || (a == -1));

   int gcd = GCD( a, b );
   return ((gcd == 1) || (gcd == -1));
}

// Use Euclid's algorithm to calculate the
// greatest common divisor (GCD) of two numbers.
// See https://en.wikipedia.org/wiki/Euclidean_algorithm
private int GCD( int a, int b )
{
   a = Math.abs( a );
   b = Math.abs( b );

   // if a or b is 0, return the other value.
   if( a == 0 ) return b;
   if( b == 0 ) return a;

   for( ; ; )
   {
      int remainder = a % b;
      if( remainder == 0 ) return b;
      a = b;
      b = remainder;
   };
}
```

The `AreRelativelyPrime` method checks whether either value is 0. Only -1 and 1 are relatively prime to 0, so if a or b is 0, the method returns `true` only if the other value is -1 or 1.

The code then calls the `GCD` method to get the greatest common divisor of a and b. If the greatest common divisor is -1 or 1, the values are relatively prime, so the method returns `true`. Otherwise, the method returns `false`.

Now that you know how the method works, implement it and your testing code in your favorite programming language. Did you find any bugs in your initial version of the method or in the testing code? Did you get any benefit from the testing code?

One major problem here is that this implementation and my testing code are not at all independent from one another: my tests that were intended to "independently verify" the algorithm effectively rely upon a tremendous amount of the same code (the GCD method). Another major problem arose when I first copied both functions into my interpreter straight from the text: I got nothing but falsy outputs from `AreRelativelyPrime` regardless of what numbers I entered, so I assumed there must have been some sort of bug in either that method or his GCD implementation. In order to mitigate this issue, I reimplemented both my test code and the function in Python - my (deeply sarcastic) "favorite" language for quick and dirty stuff like this - using its standard library function for GCD instead of the (potentially flawed) version provided by the author. No disrespect to Mr. Stephens, but he *did* explicitly suggest in an earlier chapter that we use the extensively-tested and -benchmarked standard library functions wherever possible to maximize performance and avoid bugs in our own algorithms; I have to assume there's a C# standard library function for GCD, as well.

For the `AreRelativelyPrime` method (I've taken some artistic license with names here for Pythonicness and brevity):

```
from math import gcd
def relatively_prime(a, b):
   if a == 0:
        return ((b == 1) or (b == -1))
   if b == 0:
        return ((a == 1) or (a == -1))
   result = gcd(a, b)
   return ((result == 1) or (result == -1))
```

And my test code (running on the same session, so no need to import math again):

```
def test_prime(outcome, a, b):
   result_gcd = gcd(a, b)
```

```
    result_rel_prime = relatively_prime(a, b)
    rel_prime_by_gcd = (result_gcd == 1)
    test_result = (result_rel_prime and rel_prime_by_gcd) if
outcome else not (result_rel_prime or rel_prime_by_gcd)
    # I’m well aware the above line is terrible Python style,
    # but these variable names are killing me with their verbosity.
    return test_result
```

This code ran without incident, passing the roughly 2 dozen integer test cases I provided to it (which included every possibility involving 0, 1, and -1). Clearly neither that nor this code would be sufficiently robust for “serious” production due to the utter lack of any error handling and associated edge cases involving garbage inputs (like my aforementioned example of seeing if “cheese” was relatively prime to an object), but I believe it’s acceptable for this purpose.

I suppose you could say I derived benefit from the test code in the sense that it was a worthwhile exercise in TDD and validated that the algorithm behaved more or less as advertised. I uncovered an issue in the original C# code without even having to resort to the actual test code, however. Indeed, to a large extent, it would have been feasible to test this particular method simply by inspection of a few dozen representative values - which is effectively what my simplistic test code actually *does* without a better framework.

8.9 Exhaustive testing actually falls into one of the categories: black-box, white-box, or gray-box. Which one is it and why?

It is at least theoretically possible to exhaustively test methods without any inside knowledge as long as the domains of the expected inputs and outputs are well-defined. For example, the author’s suggestion to use explicitly adversarial test-cases with methods that you know rely on quicksort (e.g., the list is already sorted, or all list items have the same value) could just as easily apply to fully black-box testing as a reasonable edge case. However, this sort of “wing-and-a-prayer” paradigm of test design (“I *guess* that’s all of the edge cases…”) doesn’t seem practical in the real world: one could not reasonably expect such an edge case to exist for any particular method without at least *some* knowledge of its internals. It therefore intuitively stands to reason that, in order to perform *truly* exhaustive testing, one must have at least some knowledge of the underlying implementation - and, thus, have some idea of where issues could potentially arise. Exhaustive testing should therefore fall at least into grey-box, if not white-box.

8.11 Suppose you have three testers: Alice, Bob, and Carmen. You assign numbers to the bugs so the testers find the sets of bugs {1, 2, 3, 4, 5}, {2, 5, 6, 7}, and {1, 2, 8, 9, 10}. How can you use the Lincoln index to estimate the total number of bugs? How many bugs are still at large?

The Lincoln index, given in the text by the formula $L = \frac{E_1 \times E_2}{S}$ (though I believe $L = \frac{\prod_{i=1}^{n} E_i}{\bigcap_{i=1}^{n} E_i}$ describes it more generally), estimates the total number of bugs by taking the product of the cardinality of each set of bugs, then the quotient of that result with the cardinality of the intersection of all the sets. In this specific case, the sets have cardinality $5 \times 4 \times 5 = 100$, but bug #2 is the only bug in common across all 3 sets; the cardinality of the intersection is therefore 1, and the total index therefore stands at $100 \div 1 = 100$. With a total of 10 bugs found, we would therefore estimate that $100 - 10 = 90$ bugs *remain* at-large, though - as the text also issues the important caveat that this method likely underestimates the true number of total bugs - the actual number yet outstanding is almost certainly higher.

8.12 What happens to the Lincoln estimate if the two testers don't find any bugs in common? What does it mean? Can you get a "lower bound" estimate of the number of bugs?

According to the Wikipedia article on the Lincoln index which the author cites, in the situation in which observers find 0 bugs in common (the article suggests this arises primarily due to systemic flaws in the sampling procedure, such as not looking hard/long enough or using search techniques that are not statistically independent of one another), “the Lincoln index is formally undefined.” Neither the text nor the Wikipedia article immediately discuss any specific remedies for this situation, which would tend to suggest that one ostensibly cannot get a lower bound estimate for the number of bugs. I reason, however, that treating the denominator as though it were 1 (the smallest possible mathematically valid value) can act as an *exceedingly* rough estimate of the lower bound. The Wikipedia article indicates that any number of shared observations lower than 5 is “extremely rough” anyway, so the simple product of their respective counts “divided by 1” may as well do.