Senior Project ll HW3
Christina Choi & Troy Womack-Henderson
03/20/2024

Problem 7.1, Stephens page 169

The greatest common divisor (GCD) of two integers is the largest integer that evenly divides them both. For example, the GCD of 84 and 36 is 12, because 12 is the largest integer that evenly divides both 84 and 36. You can learn more about the GCD and the Euclidean algorithm, which you can find at en.wikipedia.org/wiki/Euclidean_algorithm. Knowing that background, what's wrong with the comments in the following code? Rewrite the comments so that they are more effective. (Don't worry about the code if you can't understand it. Just focus on the comments.)(Hint: It should take you only a few seconds to fix these comments. Don't make a career out of it.)

```
// Use Euclid's algorithm to calculate the GCD.
 private long GCD( long a, long b )
{
  // Get the absolute value of a and b
  a = Math.abs( a );
  b = Math.abs( b );

  //Repeat until we're done
  for( ; ; )
  {
    // Set remainder to the remainder of a / b
    long remainder = a % b;
    // If remainder is 0, we're done.  Return b.
    If( remainder == 0 ) return b;
    // Set a = b and b = remainder.
    a = b;
    b = remainder;
  };
}
```

The problem with the comments in this code is that it just states what the line is obviously doing, rather than explaining why. A simpler solution would be to put a comment of the link to the wikipedia page and delete all other comments.

Problem 7.2, Stephens page 170

Under what two conditions might you end up with the bad comments shown in the previous code?

1) The programmer could add these comments after they have written it as a way to explain what the code is doing, if they don't want the explanation of why. 2) For another person who didn't write it to easily get the idea of what the programmer who wrote it was trying to accomplish

Problem 7.4, Stephens page 170

How could you apply offensive programming to the modified code you wrote for exercise 3? [Yes, I know that problem wasn't assigned, but if you take a look at it you can still do this exercise.]

```
// Use Euclid's algorithm to calculate the GCD.
    private long GCD( long a, long b )
    {
      // Get the absolute value of a and b
      a = Math.abs( a );
      b = Math.abs( b );
        original_a = Math.abs( a );
        origiinal_b = Math.abs(b);

      //Repeat until we're done
      for( ; ; )
      {
        long remainder = a % b;
        If( remainder == 0 ) {
                Debug.assert(original_a % a) == 0;
                Debug.assert(original_b % b) == 0;
                checked{return b};
        }
        a = b;
        b = remainder;
      };
    }
```

I added offensive programming by adding Debug.assert()

Problem 7.5, Stephens page 170

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

It is unnecessary to add error handling because the Debug.assert() will throw exceptions and pass the error back up to the calling code. Any errors will be handled up there.

Problem 7.7, Stephens page 170

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.

1. Go to car
2. Turn it on
3. Back out of driveway
4. Drive straight til you reach the Stop Sign
5. Turn left at the Stop Sign
6. Turn right out of the neighborhood
7. Turn right on to the main street
8. Go straight until you reach the light
9. Turn left at the light
10. Turn right into the supermarket plaza
11. Park the car near the supermarket
12. Turn car off
13. Walk in the supermarket and purchase a Snickers bar

Assumptions:

- You know where the car is
- You have the car key
- You know how to drive
- The car has gas in it

<u>Problem 8.1, Stephens page 199</u>

Two integers are relatively prime (or coprime) if they have no common factors other than 1. For example, 21 = 3 X 7 and 35 = 5 X 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.)

```
IsRelativelyPrimeTest{
        Debug.assert(isRelativelyPrime(int 7, int 20) == true);
        checked{
                return true;
        }
        Debug.assert(isRelativelyPrime(int 21, int 35) == false);
        checked{
                return false;
```

```
        }
}
```

Problem 8.3, Stephens page 199

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.]

The testing technique used to write the test method is white-box testing. White-box testing involves examining the internal structure of the code being tested. In this case, the test method is designed based on an understanding of how the function is implemented. White-box testing allows testers to ensure that all code paths are exercised and that the function behaves as expected under different conditions.

Problem 8.5, Stephens page 199 - 200

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_algorighm
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?

After translating the provided C# code into Python and implementing both the `are_relatively_prime` and `gcd` functions, I conducted testing using a set of test cases to verify the correctness of the implementation. The testing process confirmed that the initial version of the methods was free from bugs, as the implementation correctly followed the logic outlined in the description. The testing code was essential in ensuring that the functions behaved as expected across various scenarios, including handling edge cases such as 0 and negative numbers. By systematically asserting expected outcomes for different inputs, the testing code served as a valuable tool for detecting any logical errors or misinterpretations in the translation process. Overall, the testing approach proved beneficial in validating the correctness of the implementation and enhancing confidence in the reliability of the functions for future use in larger applications.

Problem 8.9, Stephens page 200

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

Exhaustive testing falls under the category of white-box testing. This is because exhaustive testing aims to cover every possible input or code path within the system, requiring a comprehensive understanding of the internal workings of the software. White-box testing involves testing the internal structure, logic, and code paths of a software application, which aligns with the exhaustive testing approach.

My Python Code:

```
def gcd(a, b):
    """Use Euclid's algorithm to calculate the greatest common divisor (GCD)."""
    a, b = abs(a), abs(b)
    while b != 0:
        a, b = b, a % b
```

```
    return a

def are_relatively_prime(a, b):
    """Return true if a and b are relatively prime."""
    if a == 0: return b == 1 or b == -1
    if b == 0: return a == 1 or a == -1
    return gcd(a, b) == 1

# Testing code
def test():
    assert are_relatively_prime(13, 27), "13 and 27 should be relatively prime."
    assert not are_relatively_prime(14, 21), "14 and 21 should not be relatively prime."
    assert are_relatively_prime(1, 0), "1 and 0 should be relatively prime."
    assert not are_relatively_prime(0, 0), "0 and 0 should not be relatively prime."
    assert are_relatively_prime(-1, 0), "-1 and 0 should be relatively prime."
    assert not are_relatively_prime(0, 2), "0 and 2 should not be relatively prime."
    print("All tests passed!")

test()
```

Problem 8.11, Stephens page 200

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?

Using the Lincoln Index with the bug sets {1, 2, 3, 4, 5}, {2, 5, 6, 7}, and {1, 2, 8, 9, 10} found by Alice, Bob, and Carmen respectively, we can calculate their overlaps: Alice-Bob (2 bugs), Alice-Carmen (2 bugs), and Bob-Carmen (1 bug). The Lincoln Index formula estimates the total bugs as the product of the individual testers' findings divided by the intersections. Thus, the estimate would be (5 * 4 * 3) / (2 * 2 * 1) = 30 / 4 = 7.5 bugs, which can be rounded to 8 bugs. This suggests that approximately 8 bugs have been found so far. To determine the bugs still at large, we subtract the estimated bugs from the total unique bugs found, yielding 10 - 8 = 2 bugs still undiscovered.

Problem 8.12, Stephens page 200

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?

If the two testers fail to find any bugs in common, it renders the application of the Lincoln Estimate inconclusive, as its calculation relies on the intersection of bugs found by both testers. While it doesn't provide a straightforward lower bound estimate, one could tentatively consider the total number of unique bugs found by both testers combined as a provisional lower bound, acknowledging that there are likely more undiscovered bugs present.