Cherrell Finister
Professor BJ
CMSI 402: Senior Project Lab
20 March 2019

Homework #3

1. 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.
    a. The comments just say what it does and not why it does it. The comments are obvious about what it does. A good way to include a comment would be to provide a link about the GCD algorithm that is a much better description of it.
2. Under what two conditions might you end up with the bad comments shown in the previous code?
    a. The first condition as to why this is seen as bad comments is because it is taken from a top down design to where the code is described in lots of detail. The comments should be more why the code is happening and not what it does. Secondly, another condition could be that the comments were written after the code was written so it is more repetitive than going in depth of what is happening.
3. How could you apply offensive programming to the modified code you wrote for exercise 3?
    a. I could apply offensive programming to that code would be that the code validates the inputs and the result, and the debug.assert methods throws an exception if there is a problem.
4. Should you add error handling to the modified code you wrote for Exercise 4?
    a. You do not need to add error handling code here because usually you want the calling code to handle those errors.
5. 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.
    a. Find my car.
    b. Open my car door.
    c. Press start button to start the car.
    d. Back out of parking space.
    e. Take a left out of the parking lot.
    f. Drive until you reach the roundabout.
    g. Drive around the roundabout and drive down the street until you reach the gate.
    h. Take a left down the street. Drive until you see the Vons on the left hand side.
    i. Take a left into the parking lot of Vons.

j. Find an empty space and park.
k. Stop the car and get out.
    i. This list makes these assumptions:
        1. That my car is parked head first.
        2. There is gas in the car.
        3. You look both ways before pulling out the parking space.
        4. You drive safely.
        5. That you look around your surroundings to make sure you get to the store.
        6. That there is parking at the store.
        7. That the store is open.

6. 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.

```
a. In pseudocode this would be the test for IsRelativelyPrime:
    i.   For 1,000 trials, pick random a and b and:

         Assert AreRelativelyPrime(a, b) =

               Validate_AreRelativelyPrime(a, b)

         For 1,000 trials, pick random a and:

               Assert AreRelativelyPrime (a, a) =

                     Validate_AreRelativelyPrime(a, a)

         For 1,000 trials, pick random a and:

               Assert AreRelativelyPrime(a, 1) relatively prime

               Assert AreRelativelyPrime(a, -1) relatively prime

               Assert AreRelativelyPrime(1, a) relatively prime

               Assert AreRelativelyPrime(-1, a) relatively prime

         For 1,000 trials, pick random a (not 1 or -1) and:

               Assert AreRelativelyPrime(a, 0) relatively prime

               Assert AreRelativelyPrime(0, a) relatively prime

         For 1,000 trials, pick random a and:
```

```
Assert AreRelativelyPrime (a, -1,000,000) =
      Validate_AreRelativelyPrime(a, -1,000,000)
Assert AreRelativelyPrime (a, 1,000,000) =
      Validate_AreRelativelyPrime(a, 1,000,000)
Assert AreRelativelyPrime (-1,000,000, a) =
      Validate_AreRelativelyPrime(-1,000,000, a)
Assert AreRelativelyPrime (1,000,000, a) =
      Validate_AreRelativelyPrime(1,000,000, a)
Assert AreRelativelyPrime (-1,000,000, -1,000,000) =
      Validate_AreRelativelyPrime(-1,000,000, -1,000,000)
Assert AreRelativelyPrime (1,000,000, 1,000,000) =
      Validate_AreRelativelyPrime(1,000,000, 1,000,000)
Assert AreRelativelyPrime (-1,000,000, 1,000,000) =
      Validate_AreRelativelyPrime(-1,000,000, 1,000,000)
Assert AreRelativelyPrime (1,000,000,  -1,000,000) =
      Validate_AreRelativelyPrime(1,000,000, -1,000,000)
```

7. 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.]

A. In the first problem it doesn't say exactly how AreRelativelyPrime method works, so this is a black-box test. You could use the white-box and gray-box tests if told how the method works. If the values were changed and ranged from -1,000 and 1,000 then you could use exhaustive tests.

8. 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?

A. This problem reminded me of a previous GCD assignment that was given in algorithms, I had done this before in javascript recursively. Doing it for the requirements of AreRelativelyPrime was a bit hard, as there is no restrictions for a and b. It was quite hard for me to think of methods of how to get out of being stuck on the tests as the code

struggled handling max and min values. The benefit of testing the code is being able to think outside of the box for solutions and how to code this method better.

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

A. Exhaustive testing falls into black-box tests because it does not rely on knowledge of what is happening inside the method that it is testing.

10. 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?

A. You can use each pair of testers to calculate three different lincoln indexes.
    a. alice/bob: 5 x 4/2 = 10
    b. alice/carmen: 5 x 5/2 = 12.5
    c. bob/carmen: 4 x 5/1 = 20
B. Taking the average and get a rough estimate of 14 bugs. Making sure to keep track of your information to plan on the worse outcome.

11. 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?

A. If two testers don’t have any bugs in common then using the Lincoln equation, that means that you have no idea how many bugs there actually is because the equation calls for dividing by the number of common bugs which is zero which means infinity. You may be able to get a lower bound estimate if the testers had at least one bug in common then you would be able to divide by that bug count and get a rough estimate of bugs.