Ben Kern

CMSI 402

Dr. Johnson

HW #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. You can learn more about the GCD and the Euclidean algorithm, which you can find at en.wikipedia.org/wiki/Euclidean\_algorithm. (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.)
   1. All comments are unnecessary and clutter up the code by describing exactly what the paraphrased version of the next line does. It would be better if there were no comments in the code at all and just a link to the wikipedia page for further reference.
2. Under what two conditions might you end up with the bad comments shown in the previous code?
   1. First, the programmer might have started by describing the program from a top-down approach in comments and then incrementally breaking it down into code and comments until it reaches code that works, leaving a lot of comments there
   2. The alternative is that the programmer wrote the comments after writing the code, deciding to be excruciatingly detailed with every line.
3. 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.]
   1. You could write something that asserts that both a and b are positive before running, since if b is 0 it will break.
4. Should you add error handling to the modified code you wrote for Exercise 4?
   1. You don’t need to. The code that calls GCD should do all error handling.
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.
   1. Assume they are already in the car and the car is at my house
   2. Turn on the car, make a u turn since the car is on the street.
   3. Turn left at the stop light.
   4. Turn right at the 4th stop light.
   5. Find a parking spot.
   6. Turn off the car
   7. Get out of the car
   8. Go in to the store.
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. (Hint: You may find it useful to write another method that also tests two integers to see if they are relatively prime.)

Assume I have slowIsRelativelyPrime(a,b) that should always respond the same as IsRelativelyPrime, though with a slower algorithm

1. For \_ in range(10000) choose a and b from -1,000,000 to 1,000,000:
   1. Assert IsRelativelyPrime(a, b) == slowIsRelativelyPrime(a, b)
2. Report percentage correct
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.]
   1. Since I am assuming I dont know how either method works, it is a blackbox test.
4. 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?

1. Add to the code cases for when a or b is 1 and -1 since that is a special case you want to test for.
2. Exhaustive testing actually falls into one ot the categoris black-box, white-box, or gray-box. Which one is it and why?
   1. Black-box. THey dont rely on knowledge of the program.
3. 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?
   1. AB = (5 bugs) \* (4 bugs) / (2 shared bugs) = 10 bugs
   2. BC = (5 bugs) \* (4 bugs) / (1 shared bug) = 20 bugs
   3. AC = (5 bugs) \* (5 bugs) / (2 shared bugs) = 12.5 bugs
   4. Averaging these gives 14 bugs left
4. 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?
   1. The equation doesnt work, but you can always assume that every tester has 1 bug in common to get a lower bound.