Sebastian Cruz

CMSI 4072 Assignment #3

7.1

* The comments within the method are not really needed because what the comments say could be inferred from the code itself (since it’s explaining what the code DOES)
* If we wanted to explain why the Euclid’s algorithm works to calculate the GCD, we could explain it in the methods docstring
  + If we wanted to keep things clean, we could comment in the link to the Wikipedia page explaining the algorithm, since they have a much more detailed explanation of it already
* Ex:
* // Use Euclid's algorithm to calculate the GCD
* // See [en.wikipedia.org/wiki/Euclidean\_algorithm](http://en.wikipedia.org/wiki/Euclidean_algorithm) for explanation
* // Rest of the code…

7.2

* Two conditions where bad comments might have shown up from this GCD method are:
  + Either the comments were added after writing the code, which leads to comments not having too much thought aside from explaining what each line of code does
  + Or the programmer designing the method may have taken a top-down approach where the code is described with a lot of detail rather than focusing on creating comments for the concepts that come right before the code

7.4

* Our validation code would look to validate both the original input and also the result, using Debug.Assert
  + As such, the validation code already applies offensive programming because the Debug.Assert method will throw an exception if there is a problem AND both the input and output are already being validated.

7.5

* Error handling could be added but isn’t needed because if the code throws any exceptions, they will be handled by the calling code.

7.7

* Look for car
* Open the car door
* Start the car
* Turn to the left and drive out of the parking spot
* Drive until the stop sign then turn left
* Drive until the stop sign then turn left
* Drive until the stop sign then turn right
* Drive until the stop light then turn right
* Drive until you see the entrance to the supermarket parking lot then turn left
* Find an empty parking spot and park in it.
* Stop the car
* Open the car door
* Lock the car
* Enter the supermarket
* Assumptions
  + The car is parked backed in
  + There is gas in the car
  + You know how to drive an automatic vehicle
  + All of the seats and mirrors are correctly adjusted for visibility and comfort
  + There are no cars when you pull out of the driveway
  + No objects are obstructing your way as you drive
  + You know when to turn at a stoplight
  + There are spots available to park at the lot
  + The supermarket is open
  + You know how to lock your car and start it

8.1

* // A test method to validate the IsRelativelyPrime method
* // Returns true if x, y are relatively prime
* public bool TestIsRelativelyPrime (int x, int y)
  + x = Abs(x)
  + y = Abs(y)
  + If x == 1 or y == 1
    - Return true
  + If x == 0 or y == 0
    - Return false
  + int minimum = Min(x,y)
  + for (int factor = 2; factor <= minimum; factor++)
    - if a % factor == 0 && b % factor == 0
      * Return false
  + return true

8.3

* Black-box test because we don’t exactly know how the IsRelativelyPrime method works, we're only making tests to check its proper functionality
* White-box or gray-box tests could work if we knew how the IsRelativelyPrime method worked
* Exhaustive could technically be performed but there are so many value pairs to test
  + Perhaps if the range of values allowed were shortened, this could be possible

8.5

* Worked well for the most part!
  + Couple edge case values had a bit of trouble

8.9

* Exhaustive testing falls into black-box testing because we don’t exactly require knowledge about what the method is actually doing when testing it

8.11

* You can have 3 different pairs between the testers to calculate the Lincoln indexes
* Ex:
  + Alice, Bob = 5\*4/2= 10
  + Alice, Carmen = 5\*5/2 = 12.5
  + Bob, Carmen = 4\*5/1 = 20
* Taking the average of the 3 pairs, you would get about 14 bugs on average and 20 at most

8.12

* If testers find no bugs in common, the formula to calculate the Lincoln index will divide by zero and thus give an infinite result
  + This means that we will have no idea about how many bugs there are
* A lower bound estimate can be made by “assuming” that there will be 1 bug in common among the testers
  + Since there will always be at least one common bug, the formula will divide by one and give an estimate