Homework Assignment #3

7.1 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.)

We would probably get rid of most of the comments since they're not really needed because the code is relatively self-explanatory. The only comment we would keep is the first one that says to use Euclid's algorithm to calculate the GCD. Since the algorithm itself isn't explained in the code but including this explanation would be overkill for a comment, we can include something that says "Read more about the algorithm here en.wikipedia.org/wiki/Euclidean_algorithm."

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

One reason could be that the programmer wrote the steps first and then added the code which could lead to redundant comments. Another reason may be that the programmer added the comments after they wrote the code so they know what the code does already but didn't think about why since they were implementing an already existing algorithm.

7.4 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.]

We can add Debug.Assert statements to check for the inputs (a > 0 and b > 0). We can also save the original values for later in case we need them. Lastly, we can use another Debug.Assert to make sure our output evenly divides the original values (og_a % b == 0 and og_b % b == 0). These additions are already offensive, we validate

the inputs and output; in addition, the assert method should throw an error if the conditions aren't met.

<u>7.5</u> Should you add error handling to the modified code you wrote for Exercise 4?

We don't really need to because the Debug asserts will handle the errors for us anyway. Any errors or exceptions will be passed to the asserts and they will be handled there.

<u>7.7</u> 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.

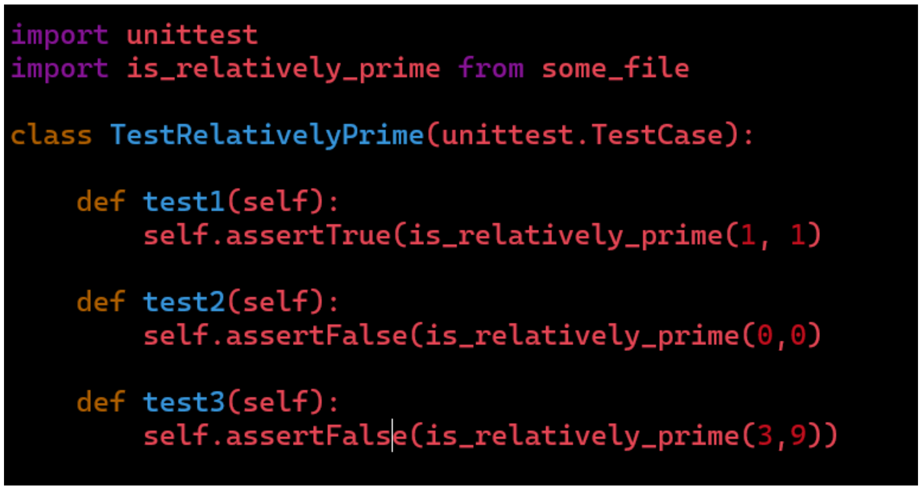
a. **Take the elevator down to Lower Garage.**
b. **Unlock the car and get in.**
c. **Start the car.**
d. **Back out of the parking space.**
e. **Turn left to go to the Upper Garage.**
f. **Drive through the gates out of the parking garage.**
g. **Turn left, drive until the stop sign.**
h. **Turn right, drive until the first traffic light.**
i. **Turn left, drive until the street ends.**
j. **Turn right, drive until the street ends.**
k. **Turn left, drive until you see the Ralph's parking lot to your right.**
l. **Turn right into the parking lot.**
m. **Find a parking space and park the car.**
n. **Stop the car, get out, and lock it.**
o. **Walk into Ralph's and buy some Malibu.**
p. **Get carded and show your ID.**

We make the following assumptions:

a. **The car is parked in its usual location.**
b. **The car is parked straight ahead.**
c. **You know how to adjust the seat and mirrors or you are my height (5'4).**

d. **There's enough gas in the car to make it there (not always the case).**
e. **You know how to drive an automatic transmission or know how to drive in general.**
f. **There is no one standing behind the car when you are backing out.**
g. **There is nothing blocking the street when you are driving and you know how to read traffic lights and stop if necessary.**
h. **There are empty parking spaces in the front parking lot of Ralph's, if not you will have to go behind the building.**
i. **Ralph's is open at that time (it should be if you go before midnight).**

8.1 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.)


```
import unittest
import is_relatively_prime from some_file

class TestRelativelyPrime(unittest.TestCase):

    def test1(self):
        self.assertTrue(is_relatively_prime(1, 1)

    def test2(self):
        self.assertFalse(is_relatively_prime(0,0)

    def test3(self):
        self.assertFalse(is_relatively_prime(3,9))
```

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.]

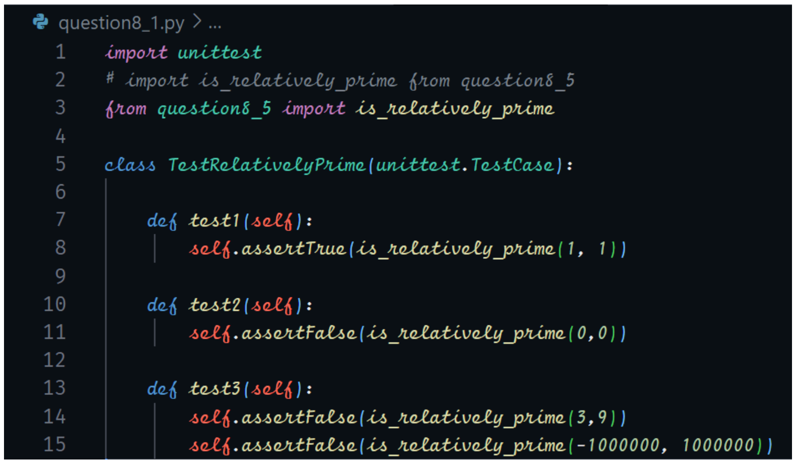
We used the black-box technique to test the method in Exercise 1 because we don't know how this method works since we were not given the exact code. If we were given the code, we could've used white-box or gray-box testing.

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

We did not find any bugs in the initial version of the method. Our testing code needed to be modified a little because Diego confused Python import statements with JavaScript's (oops). We did benefit from testing the code because we were able to fix the import statements. The fixed test code is below:


question8_1.py > ...

```python
import unittest
# import is_relatively_prime from question8_5
from question8_5 import is_relatively_prime

class TestRelativelyPrime(unittest.TestCase):

    def test1(self):
        self.assertTrue(is_relatively_prime(1, 1))

    def test2(self):
        self.assertFalse(is_relatively_prime(0,0))

    def test3(self):
        self.assertFalse(is_relatively_prime(3,9))
        self.assertFalse(is_relatively_prime(-1000000, 1000000))
```

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

Exhaustive tests, where you test a method with every possible input, fall into black-box category since we don't need to know how the method works exactly to test it.

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?

- **Alice/ Carmen = 5 x 5 / 2 = 25 / 2 = 12.5**
- **Bob/ Carmen = 4 x 5 / 1 = 20 / 1 = 20**
- **Alice / Bob = 5 x 4 / 2 = 20 / 2 = 10**
- **Take average to get the total estimate: (12.5 + 20 + 10) / 3 = 42.5 / 3 = 14.17**
- **Approximately 14 bugs.**

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?

Since the Lincoln index formula divides the multiplication of the number of bugs the testers found(E1 x E2) by the common found bugs(S) : (E1 x E2) / S, if S = 0 in this case that means the result of this equation would be undefined. This would mean that we have no idea how many bugs there are. If we were to assume they found at least 1 bug in common (S = 1), we can take E1 x E2 as our lower bound estimate, where E1 is the number of bugs tester 1 found and E2 is the number of bugs tester 2 found.