Carleen Petrosian
CMSI 402
03-07-2018

Assignment #3

1. Below is what the code should look like, after removing the unnecessary comments:

```
// Use Euclid's algorithm to calculate the GCD.

// Refer to en.wikipedia.org/wiki/Euclidean_algorithm
      private long GCD( long a, long b )
        {
           a = Math.abs( a );
           b = Math.abs( b );

           for( ; ; )
           {
              long remainder = a % b;
              if( remainder == 0 ) return b;
              a = b;
              b = remainder;
           };
        }
```

2. If the programmer took a top-down approach to the problem, which involves describing the code in heavy detail, then the code would result in a bunch of redundant comments; instead of using comments efficiently, he/she would be describing each code statement line-by-line. In addition, if the programmer inserted comments after writing the code, it can result in comments that describe what each line of code does, instead of why it is doing it.

3. The code in 7.3 validates the input and the result. In addition, Debug.Assert throws an exception if a problem arises. Therefore, this code is already offensive.

4. Error handling could be added to the code, but it's neither necessary nor wise to add it. The calling code should be able to handle any errors, which is how it is already implemented.

5. These are the steps to get to my nearest supermarket:
   (a) Walk to the car
   (b) Unlock the car
   (c) Start the car
   (d) Back out of parking spot
   (e) Drive out of parking garage
   (f) Turn right to get out of the parking garage
   (g) Continue driving straight until you reach the side of the street
   (h) Turn right

(i) Continue straight until the first stoplight
(j) Turn right
(k) Continue straight until you see the supermarket
(l) Turn right into the supermarket parking lot
(m) Find an empty parking spot and park in it
(n) Stop the car and exit
(o) Lock the car and walk to the supermarket

These are some of the assumptions made when creating the description above:
(a) The car is parked facing the wall (and therefore needs to be backed out)
(b) The car already has gas in it
(c) There's nothing behind the car when you pull out
(d) There are empty spots in the supermarket parking lot
(e) The supermarket is open

6. First off, a method to test `isRelativelyPrime` that is less efficient:

```
private bool checkIfRelativelyPrime( int a, int b )
  {
      // Use positive values
      a = Math.abs( a );
      b = Math.abs( b );


      if ((a == 1) || (b == 1)) return true;

      if ((a == 0) || (b == 0)) return false;


      // Find the smaller of a and b
      int smaller = Math.min(a, b);
      for(int i = 2; i <= smaller; i++)

      {


      if ((a % i == 0) && (b % i == 0)) return false;


      }

      return true;
  }
```

Below is pseudocode for how tests would be performed using the code above:

```
for 1,000 trials, pick a random a and b and:
    Assert isRelativelyPrime(a, b) = checkIfRelativelyPrime(a, b)
```

```
for 1,000 trials, pick a random a and:
    Assert isRelativelyPrime(a, a) = checkIfRelativelyPrime(a, a)

for 1,000 trials, pick a random a and:
    Assert checkIfRelativelyPrime(a, 1) relatively prime
    Assert checkIfRelativelyPrime(a, -1) relatively prime
    Assert checkIfRelativelyPrime(1, a) relatively prime
    Assert checkIfRelativelyPrime(-1, a) relatively prime

for 1,000 trials, pick a random a (not 1 or -1) and:
    Assert checkIfRelativelyPrime(a, 0) relatively prime
    Assert checkIfRelativelyPrime(0, a) relatively prime

for 1,000 trials, pick random a and:
    Assert isRelativelyPrime(a, -1,000,000) =
        checkIfRelativelyPrime(a, -1,000,000)
    Assert isRelativelyPrime(a, 1,000,000) =
        checkIfRelativelyPrime(a, 1,000,000)
    Assert isRelativelyPrime(-1,000,000, a) =
        checkIfRelativelyPrime(-1,000,000, a)
    Assert isRelativelyPrime(1,000,000, a) =
        checkIfRelativelyPrime(1,000,000, a)

Assert isRelativelyPrime(-1,000,000, -1,000,000) =
    checkIfRelativelyPrime(-1,000,000, -1,000,000)
Assert isRelativelyPrime(1,000,000, 1,000,000) =
    checkIfRelativelyPrime(1,000,000, 1,000,000)
Assert isRelativelyPrime(-1,000,000, 1,000,000) =
    checkIfRelativelyPrime(-1,000,000, 1,000,000)
Assert isRelativelyPrime(1,000,000, -1,000,000) =
    checkIfRelativelyPrime(1,000,000, -1,000,000)
```

7. Since 8.1 doesn't state how `isRelativelyPrime` works, it would be a black-box test. If we knew how it worked, we could write white-box or gray-box tests. Exhaustive tests wouldn't be wise because we have a large range of values, namely -1,000,000 to 1,000,000.

8. This `areRelativelyPrime` method has trouble dealing with the maximum and minimum possible integer values, so a restriction on the values of a and b needs to be implemented. Other than that, the `areRelativelyPrime` method worked fairly well.

9. Exhaustive tests are black-box tests because they don’t rely on knowing what is going on inside the method they are testing.

10. You can use each pair of testers to calculate three different Lincoln indexes:

(a) Alice/Carmen: 5 x 5 / 2 = 12.5
(b) Alice/Bob: 5 x 4 / 2 = 10
(c) Bob/Carmen: 4 x 5 / 1 = 20

By taking an average of these, you get ~ 14, which means about 14 bugs. You need to continue tracking the number of bugs you find, so that you can update your estimate.

11. If there are no bugs in common, then we have to divide by 0 in the Lincoln index equation, which doesn’t tell us sufficient information on how many bugs there are. To get a “lower bound” estimate for the number of bugs, pretend the testers found 1 bug in common and follow the Lincoln index equation.