Homework 1, CSCE 350, Fall 2016
Due 8 September 2016

This is an exercise in making judgements about relative costs of computation when there is more than one algorithm but the multiple algorithms do things that have different costs.

The naive GCD algorithm simply divides to get remainder repeatedly until the remainder is zero. When that happens the smaller number from the previous division is the gcd.

```
while (remainder != 0)
    remainder = larger % smaller
    larger = smaller
    smaller = remainder
return remainder
```

There are some beginning and ending conditions, and such, but the heart of the algorithm is just what's above. And of course, this is in Euclid, and is perhaps the oldest algorithm on the planet, dating to about 300 BC, and is remarkable because the naive algorithm today is exactly as written more than 2000 year ago. Not many things in science have that much staying power.

The problem with this is that integer division is expensive when compared with the other integer arithmetic operations. An old rule of thump was that if addition or subtraction cost 1 unit, then multiplication might cost 5, and division could cost 10 to 50. On more modern RISC processors, the division cost could be as high as 100.

This means that in any algorithm that involves integer division, it could be worth a great deal of effor to avoid unnecessary divisions.

Paul Levy back in the 1930s (I think) proved that for randomly chosen integers of about the same size (think of two 63-bit long integers in twos-complement notation), the quotient of larger by smaller was 1 about 42% of the time, was 2 about 15% of the time, and was 3 about 9% of the time.

That is, for about 2/3 of randomly chosen integers of the same number of bits, the quotient of larger by smaller is lessequal 3.

This suggests a hack to the naive algorithm.

Division is really just repeated subtraction.

If we subtract up to three times, and we get the remainder, then we don't need to divide.

That is, given `larger` and `smaller`:

- Subtract `smaller` from `larger` twice. If the result is negative, then the quotient of division is 1 and the remainder is `larger` minus `smaller`.
- Subtract again. If the result is negative, then the quotient of division is 2 and the remainder is `larger` minus `smaller` minus `smaller`.
- Subtract again. If the result is negative, then the quotient of division is 3 and the remainder is `larger` minus `smaller` minus `smaller` minus `smaller`.
- If the result of this subtraction is positive, punt and do the division.

Source data

You have been given three input files and corresponding output files. The inputs specify the number of pairs of integers (a, b) for which to compute the gcd and the maximum size of the values of a and b to generate.

The program then generates that many pairs of those sizes and computes the gcd using the naive algorithm and using the subtract-three-times algorithm. I keep a count of the number of divisions and the number of subtractions.

All else being equal (and of course this has to be checked), it would then be possible to cost a subtraction versus a division and determine, given the sizes of integers and the particular costs for your particular machine, what should be the best algorithm for the task at hand.

Caveat: Remember that a "comparison" is really a subtraction. That is, when you write a test

```
if ( a < b)
```

what really happens in the computer is that one computes $b - a$ and tests the sign of the result. So comparisons cost the same as subtractions. If you do your code properly, then you will probably do the same test for "larger" and for "smaller" for both gcd algorithms, and you can assume the rest of the comparison is legitimate. But if you do a comparison for one algorithm that you don't do for the other, then you are adding in the cost of a subtraction, and you need to account for that.

Assignment

You are to write code for both algorithms, count the divisions and comparisons, and write a brief summary (one or two pages) of what these results tell you about which algorithm would be better under what conditions of computer costs and sizes of operands.

RANDOM NUMBERS

The sample output comes from running my random number code on a machine in the linux lab. If you run on a different operating system, or with a different function for generating random numbers, you will get a different sequence of numbers to test. That's ok. Just remember that if you generate different numbers from what I have, you need to have a different piece of code that verifies you're getting the right answers.

You could, in fact, hack the "create" function so that instead of generating random numbers, it read in a version of the numbers from the smallest of the three output files. That way, you would have ground truth and it might make it easier to tell that you had the right answers. This would work for testing purposes, but should be changed back when submitting your assignment so I will be able to get identical values from all your programs.