CS 2003 – Fall 2018
Lab 02

The lab assignment:

Create a text file named lab02data.txt with the following contents in your project
(Use spaces, not tabs; do not worry about exact spacing)

```
#this is file lab02data.txt
3   8   1   1
-3  8   1   1
-3  4   1   4
-4  11  3  11
-4  11  8 -22
-7  12  5  -8
4   -9 -3  -7
-5   8  3  -4
-6 -13  5  13
7   -9 -5   8
16  -5  3   1
9  -13  7 -12
```

Write, modify, and/or update a Python program that will do the following:

Do nothing with the first header record in the data file. Subsequently, for each record (line) in the file, treat the first two integers as the numerator and denominator, respectively, of a first fraction and treat the latter two integers as the numerator and denominator, respectively, of a second fraction. Then, print the following:

```
"The first fraction is:"...
"The second fraction is:"...
"The sum of the two fractions is: " ...
"The difference of the two fractions is: " ...
"The product of the two fractions is: " ...
"The first fraction divided by the second fractons is: " ...
"The first fraction is ≤ the second fraction: "<True or False>
```

The Background:

Recall that a rational number (fraction) is an expression of the form a/b where a and b are integers and $b \neq 0$.

For the time being, assume that we are not asked to deal with an expression of the form "a/b" where b = 0. (Of course a/0 is not a fraction regardless of the integer a)

Also recall that a/b = c/d in case a•d = b•c

Hence, **any rational has an easy representation of the form a/b where b > 0**
(e.g. 5/(- 7) is the same rational as (- 1)(5)/(- 1)(- 7) = (- 5)/(7) or, similarly,
-10/-13 is the same rational as (-1)/(-10)/(-1)(-13) = 10/13

So, any time we construct a rational where the denominator argument is negative, we can, in the constructor, obtain an equivalent rational that has a denominator that is positive. We do this by multiplying both the numerator argument and the denominator argument by (-1). Life is better (i.e. the case analysis associated with other objectives is diminished) if we always construct a rational in a way so that the denominator is positive.

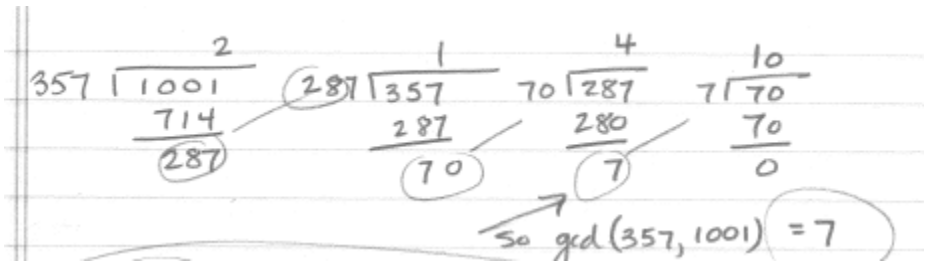
If both the numerator and denominator of a/b where b > 0 are divided by the greatest common divisor (gcd) of a and b, then we have an equivalent expression for a/b that is **reduced to lowest terms**.

The greatest common divisor of two integers when the second of the two is positive (and regardless of whether the first is positive or negative) is given by

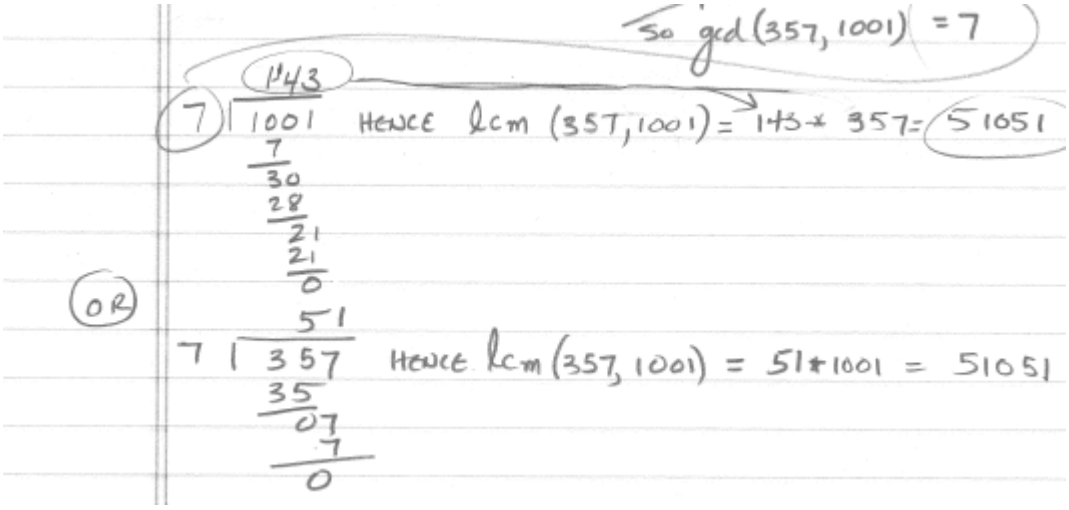
```
def gcd(m,n):
    while m%n != 0:
        oldm = m
        oldn = n

        m = oldn
        n = oldm%oldn
    return n
```

An illustration to conclude that the gcd of 1001 and 357 is 7 follows:



Although, we do not need the least common multiple in our definition of the rational class, while we are here we observe that the least common multiple of 1001 and 357 is 51051 as follows:



Also recall:

a/b == c/d in case a • d = b • c

a/b + c/d = (a • d + b • c)/(b • d),which should be reduced to an equivalent expression in lowest terms

a/b - c/d = (a • d - b • c)/(b • d), which should be reduced to an equivalent expression in lowest terms

a/b • c/d = (a • c)/(b • d), which should be reduced to an equivalent expression in lowest terms

a/b ÷ c/d = (a • d)/(b • c) if $c \neq 0$ since, otherwise c/d is 0 and one cannot divide by 0. Again, the result should be reduced to an equivalent expression in lowest terms.

Finally, for example,

a/b <= c/d in case a • d <= b • c where of course the a • d and b • c comparison is a comparison of integers.

(Recall that we are assuming that b > 0 and d > 0 ; otherwise, this would not always work correctly. For example, when comparing (1)/(- 2) with (-2)/(3), we would test that
(1)(3) <= (- 2)(- 2) and incorrectly conclude that (1/(-2) <= (-2)/(3) which is not correct;
(-2)/(3) is to the left of (1)/(- 2) on a real number line.)

etc.

Correspondence between operators and function names for operator overloading in Python:

OPERATOR	FUNCTION	METHOD DESCRIPTION
`+`	`__add__(self, other)`	Addition
`*`	`__mul__(self, other)`	Multiplication
`-`	`__sub__(self, other)`	Subtraction
`%`	`__mod__(self, other)`	Remainder
`/`	`__truediv__(self, other)`	Division
`<`	`__lt__(self, other)`	Less than
`<=`	`__le__(self, other)`	Less than or equal to
`==`	`__eq__(self, other)`	Equal to
`!=`	`__ne__(self, other)`	Not equal to
`>`	`__gt__(self, other)`	Greater than
`>=`	`__ge__(self, other)`	Greater than or equal to
`[index]`	`__getitem__(self, index)`	Index operator
`in`	`__contains__(self, value)`	Check membership
`len`	`__len__(self)`	The number of elements
`str`	`__str__(self)`	The string representation