Greatest Common Divisor

Lab Assignment 1 - Public Key Cryptography, UBB-CS Year 3

Popa Alex Ovidiu, group 936

Problem statement: compute the gcd of 2 numbers in 3 different ways. Arbitrarily large numbers should be supported.

Proposed solution:

Create a custom class to handle arithmetic operations for big integers, given as strings. The BigInteger class supports the following operations: +, -, *, //, %, >,>=,<=,<,==,!=

```
<<BigInteger>>=
from copy import deepcopy
from timeit import default_timer

class BigInteger:
    def __init__(self, x):
        self.__read(x)

    def __str__(self):
        cpy = deepcopy(self)
        cpy.vec.reverse()
        result = ""
        for digit in cpy.vec:
            result += str(digit)
        return result

    """
    Adding two BigInteger variables is the same as doing basic arithmetic in primary school
    i.e. there is a carry we add with each digit sum which is first initialized with 0 and
    if digit1 + digit2  > 9.
    """
    def __add__(self, other):
        result = BigInteger("0")
```

```python
    result.vec = []
    carry = 0
    i = 0
    while i < len(self.vec) and i < len(other.vec):
        digit = self.vec[i] + other.vec[i] + carry
        result.vec.append(digit % 10)
        carry = digit // 10
        i += 1
    while i < len(self.vec):
        digit = self.vec[i] + carry
        result.vec.append(digit % 10)
        carry = digit // 10
        i += 1
    while i < len(other.vec):
        digit = other.vec[i] + carry
        result.vec.append(digit % 10)
        carry = digit // 10
        i += 1
    if carry:
        result.vec.append(carry)

    return result

"""
Subtracting two BigInteger variables is (again), the same as doing basic arithmetic in
i.e. there is a carry we subtract after subtracting digit2 from digit1 which is first i
if digit1 - digit2 - carry < 0. Furthermore, if this happens, we need to add the base (
to the previously computed digit.
"""
def __sub__(self, other):
    result = deepcopy(self)
    i = 0
    carry = 0
    while i < len(other.vec) or carry:
        if i < len(other.vec):
            result.vec[i] = result.vec[i] - other.vec[i] - carry
        else:
            result.vec[i] -= carry
        if result.vec[i] < 0:
            carry = 1
            result.vec[i] += 10
        else:
            carry = 0
        i += 1

    while result.vec[-1] == 0 and len(result.vec) > 1:
```

```python
        result.vec.pop()

    return result

"""
Multiplication is a bit brute force, because we multiply each digit with each digit and
the respective result to the others computed so far for the same "position".
"""
def __mul__(self, other):
    result = BigInteger("0")
    result.vec = [0] * (len(self.vec) + len(other.vec))
    i1 = 0

    for digit1 in self.vec:
        carry = 0
        i2 = 0
        for digit2 in other.vec:
            summ = digit1 * digit2 + result.vec[i1 + i2] + carry
            carry = summ // 10
            result.vec[i1 + i2] = summ % 10
            i2 += 1

        if carry > 0:
            result.vec[i1 + i2] += carry
        i1 += 1

    while result.vec[-1] == 0 and len(result.vec) > 1:
        result.vec.pop()

    return result

"""
For integer division, subtractions are made until the current number is < the other numb
of subtractions is counted.
"""
def __floordiv__(self, other):
    cnt = BigInteger("0")
    cpy = deepcopy(self)
    while cpy > other:
        cpy = cpy - other
        cnt = cnt + BigInteger("1")
    if cpy == other:
        cnt = cnt + BigInteger("1")
    return cnt

"""
```

```
    Now, modulo is interesting because it makes use of the remainder theorem, i.e.
    D = P * Q + R, R<Q
    From where we can easily conclude that R = D - P*Q, and that is exactly how the functio
    the modulo (remainder)
    """
    def __mod__(self, other):
        divd = deepcopy(self) // deepcopy(other)
        return self - (divd * other)

    def __gt__(self, other):
        if len(self.vec) > len(other.vec):
            return True
        if len(self.vec) < len(other.vec):
            return False
        for i in range(len(self.vec) - 1, -1, -1):
            if self.vec[i] > other.vec[i]:
                return True
            elif self.vec[i] < other.vec[i]:
                return False
        return True

    def __lt__(self, other):
        return not self > other

    def __le__(self, other):
        return self == other or self < other

    def __ne__(self, other):
        return self.vec != other.vec

    def __eq__(self, other):
        return self.vec == other.vec

    """
    The big integers are passed as strings, but kept as an array of digits in reverse order
    "1234" is kept as [4,3,2,1].
    """
    def __read(self, x):
        self.vec = []
        for char in x:
            self.vec.append(ord(char) - ord('0'))
        self.vec.reverse()
```

@

Function which computes the greatest common divisor of two natural numbers x and y using the subtraction method. One may notice that the method is nothing more than a simplified version of Euclid's algorithm (see below explanation for that), the idea being that instead of using repeated divisions, each division is reduced to many subtractions, which leads to the following recursion:

$$gcdSubtract(a,b) = \begin{cases} a & a = b \\ gcdSubtract(a-b,b) & a < b \\ gcdSubtract(a,b-a) & otherwise \end{cases}$$

Proof of correctness for x=18 y=6 gcdSubtract(18,6) = 18>6 => x = 18-6 = 12 12>6 => x = 12-6 = 6 6 = 6 => result is 6

```
<<gcd_subtract>>=
def gcd_subtract(x, y):
    if x == BigInteger("0"):
        return y
    if y == BigInteger("0"):
        return x
    while x != y:
        if x > y:
            x -= y
        else:
            y -= x
    return x
@
```

Function which computes the greatest common divisor of two natural numbers x and y using the Euclidean method. The recursion can be defined as:

$$gcdEuclidean(a,b) = \begin{cases} a & b = 0 \\ gcdEuclidean(b,a\%b) & otherwise \end{cases}$$

Proof of correctness for x=18 y=12 gcdEuclidean(18,12) = 12!=0 => x = 12, y = 18%12 = 6 6!=0 => x = 6, y=12%6 = 0 0=0 => x = 6 is returned

```
<<gcd_euclidean>>=
def gcd_euclidean(x, y):
    zero = BigInteger("0")
    if x == zero:
        return y
    if y == zero:
        return x
    while y != zero:
```

```
        r = x % y
        x = y
        y = r
    return x
```

@

Function which computes the greatest common divisor of two natural numbers x and y by finding the first (biggest) number which divides both of them. Be warned, this method is highly inefficient when |x-y| amounts to a big number. For this

Proof of correctness for x=30 y=20 gcdBasic(30,20) = min (30,20) = 20 30 % 20 != 0 i = 20 // 2 = 10 30 % 10 == 0 & 20 % 10 == 0 => 10 is returned

```
<<gcd_basic>>=
def gcd_basic(x, y):
    if x > y:
        min = y
    else:
        min = x
    zero = BigInteger("0")
    if x % min == zero and y % min == zero:
        return min
    i = min // BigInteger("2")
    while i >= BigInteger("2"):
        if x % i == zero and y % i == zero:
            return i
        i = i - BigInteger("1")
    return BigInteger("1")
```

@

To change the tests, simply change the strings you want in the tests array. To change the used method, comment out the current one (gcd euclidean) and uncomment the preferred one.

```
<<*>>=
<<BigInteger>>
<<gcd_euclidean>>
<<gcd_subtract>>
<<gcd_basic>>
def main():
    tests = [("18", "12"), ("30", "11"), ("4137524", "3997244"), ("9427097152", "25719608"),
             ("737319582759902", "130030194834914"), ("2183651267535555", "85765424658761005
             ("98765423165498765423619 78", "44789548145555670001 02"),
             ("73498174143914", "13245125243635476"),
             ("98653287946511232000007777845156", "98798754621313465487654654048 0984"),
```

```
                ("1285497153615581795397428674243824621432424572421437927424242424258923463862"
                 "21312314614141414143426463464574000000789798987893213215446546549808900024")]
    for test in tests:
        print("Starting test with a={},b={}".format(test[0], test[1]))
        start = default_timer()
        x = BigInteger(test[0])
        y = BigInteger(test[1])

        gcd = gcd_euclidean(x, y)
        #gcd = gcd_subtract(x, y)
        #gcd = gcd_basic(x,y)

        end = default_timer()
        print("Time elapsed {} seconds".format(end - start))
        print("Gcd is {}\n".format(gcd))

main()
@
```

The program will output for each test its members, the elapsed time and the gcd computed with the chosen method.