Bardas Alexandru-Cristian, 321

In this file, i will demonstrate an algorithm for solving a system of congruences, along with proving the Chinese Reminder Theorem, that is going to be used to solve the system of congruences.

The algorithms will be written in Python, thus not needing a Big Integer class.Python3 handles very well very big numbers(even tens of thousands of digits), and i will demonstrate that with a short test.

```
<<BigIntTest>>=
import time
def test():
    start = time.time() # we will measure the time it takes python
    #to calculate this big number
    a = (2**(3*2659))**150
    end = time.time()
    b = str(a)
    print(len(b)) # print the number of digits in the number
    # it should have 360198 digits
    print(end - start) # print the time it took for the calculations
    # for me it was on average less than 0.1 seconds
@
```

The Chinese Reminder Theorem

Given pairwise coprime positive integers $n_1, n_2, \dots, n_k$ and arbitrary integers $a_1, a_2, \dots, a_k$, the system of simultaneous congruences

$$\begin{cases} x \equiv a_1 (mod \quad n_1) \\ x \equiv a_2 (mod \quad n_2) \\ \vdots \\ x \equiv a_k (mod \quad n_k) \end{cases}$$

has a solution, and the solution is unique modulo $N = n_1 * n_2 * \cdots * n_k$.

Construction of the solution

1. Compute $N = n_1 * n_2 * \cdots * n_k$.
2. For each $i = 1, 2, \dots, k$ compute $y_i = N/n_i = n_1 * n_2 * \cdots * n_{i-1} * n_{i+1} * \cdots * n_k$
3. For each $i = 1, 2, \dots, k$ compute $z_i \equiv (y_i)^{-1} \bmod n_i$ using Euclid's extended algorithm.(z_i exists since $n_1, n_2, \dots, n_k$ are pairwise coprime)
4. The integer $x = \sum_{i=1}^{k} a_i * y_i * z_i$ is a solution to the system of congruences, and $x \bmod N$ is the unique solution modulo N.

Proof To see why x is a solution, for each $i = 1, 2, \dots, k$, we have

$x \equiv (a_1 * y_1 * z_1 + \cdots + a_k * y_k * z_k) \pmod{n_i}$
$x \equiv (a_i * y_i * z_i) \pmod{n_i}$
$x \equiv a_i \pmod{n_i}$

where the second line follows since $y_j \equiv 0 \bmod n_i$ for each $j \neq i$, and the third line follows since $y_i * z_i \equiv 1 \bmod n_i$. Now, suppose that there are 2 solutions u and v to the system of congruences. Then $n_1|(u-v), n_2|(u-v), \ldots, n_k|(u-v)$, and since $n_1, n_2, \ldots, n_k$ are relatively prime, we have that $n_1 * n_2 * \cdots * n_k$ divides $u - v$, or $u \equiv v \pmod{n_1 * n_2 * \cdots * n_k}$. Thus, the solution is unique modulo $N = n_1 * n_2 * \cdots * n_k$.

The algorithm for solving a system of congruences

First, we will need to to build our system, that means reading the input from the user.

```
<<ReadInput>>=
def readInput():
    i = int(input("Give i(the number of congruences present in the system) ="))
    l = [] # the list where we will store pairs(a,n), where the index of the pair
    #will mean it's subscript in the system, and a is the right hand side
    #of the congruence, and n is the moduli
    for x in range(i):
        print("Give a_" + str(x), end='=')
        a = int(input())
        print("Give n_" + str(x), end='=')
        n = int(input())
        pair = (a, n)
        l.append(pair)
    return l
@
```

We have read the number of congruences present in the system, and then we have created a list of tuples that holds the reminder a_i as it's first element, and the moduli n_i as it's second element, that for each $i = 1, 2, \ldots, k$, where k is the number of congruences present.

Then, in order to be able to solve the system, we will need some functionality able to do the inverse modulus n_i.

For that, i have used an auxiliary function, a gcd function, implemented and proved in the last laboratory homework.The GCD function is used as a failsafe check to make sure the given number and the moduli are coprime.

```
<<steinGCD>>=
```

```
def steinGCD(u, v) :
        #Checking the base case u = v
        if u == v:
                return u
        #Checking the first identity
        if u == 0:
                return v
        if v == 0:
                return u
        #We are looking for factors of 2, checking the second and third identities
        if u % 2 == 0:
                if v % 2 == 0: #both even
                        return 2*steinGCD(u // 2, v // 2)
                else: #u is even, v is odd
                        return steinGCD(u // 2, v)
        if v % 2 == 0: #Checking if u is odd and v even
                return steinGCD(u , v // 2)
        #We will now reduce the larger argument
        if u > v:
                return steinGCD(u - v, v)
        return steinGCD(v - u, u)
@
```

Now we are ready to implement the inverse modulo n_i function.

```
<<inverseModN>>=
def inverseModN(a, N):
    #we check that a is different than 0 mod N or if N == 1
    if a % N == 0 or N == 1:
        return 0
    #we check if a is invertible or if N is less than 0
    if steinGCD(a, N) != 1 or N < 1:
        return -1
    n = N
    y = 0
    x = 1
    #We will attempt to use the extended euclidean algorithm
    #To obtain the Bezout coefficients such that
    #We can calculate the inverse of a mod N
    while (a > 1):
        # q is the quotient of a // N
        q = a // N
        t = N
        #N will be assigned the reminder now,so we will treat it just
        #as in the case of the extended euclidean algorithm
        N = a % N
        a = t
```

```
        t = y
        # Update x and y(Bezout coefficients)
        y = x - q * y
        x = t # this will be the bezout coefficient of a, and also it's inverse mod N
    #We have to check if x is positive
    if x < 0:
        x = n + x #we make it positive if it is not
    return x
@
```

This functionality was implemented on the basis on Bezout's identity, that being: If two integers a, b have $gcd(a, b) = 1$, then there exist two integers u, v such that $a * u + b * v = 1$.

I have based my function on the calculations used in the extended euclidean algorithm to get the Bezout coefficients, merely the u in the formulae above, that being the inverse of a moduli $n_i = b$.

Now, we are able to finally get to constructing the solution of the system.

This last functionality implements the actual computations required to solve the system using the Chinese Reminder Theorem, that being calculating $N = n_1 * n_2 * \cdots * n_k$, then it computes and stores the values $y_i = N/n_i = n_1 * n_2 * \cdots * n_{i-1} * n_{i+1} * \cdots * n_k$. We then proceed to compute the inverse modulo n_i for each of the y_i's, $i = 1, 2, ..., k$. As a final step, we compute the final solution X, namely $X = \sum_{i=1}^{k} a_i * y_i * z_i$. After that, we return the computed X modulo N, and N as a pair, to be used in later prints.

```
<<solveSystem>>=
def solveSystem():
    l = readInput() # get the list of pairs
    N = 1
    #calculate N, the product of al ni's
    for x in range(len(l)):
        N = N*l[x][1]
    lN = [] # list to store the Ni's
    #calculate each Ni
    for x in range(len(l)):
        lN.append(N // l[x][1])
    #compute x
    x = 0
    for y in range(len(l)):
        x = x + l[y][0]*lN[y]*inverseModN(lN[y], l[y][1])
    #return the computed x mod N
    return (x % N, N)
@
```

Lastly, we have the main driver of the program.

```
<<main>>=
if __name__ == "__main__":
    #test()
    #Driver program
    a = solveSystem()
    print("The solution is x = " +str(a[0]) + " mod " + str(a[1]))
#End of program
@
```

```
<<*>>=
<<BigIntTest>>
<<steinGCD>>
<<inverseModN>>
<<solveSystem>>
<<main>>

@
```