King Saud University
College of Computer and Information Sciences
Computer Science Department

CSC281
Project
Fall 2020

CSC281 Project Report

Student Name:	Mohand Alrasheed
Student ID:	439101298
Section:	21925

Objective:

The objective of this project is to find the unique solution to n linear congruencies

$$a_1 x = b_1 \ (mod\ m_1)$$
$$a_2 x = b_2 \ (mod\ m_2)$$
$$a_3 x = b_3 \ (mod\ m_3)$$
$$...$$
$$a_n x = b_n \ (mod\ m_n)$$

Given all a_i and m_i are relatively prime, and all m_i are pairwise relatively prime.

The solution to this problem is calculated in this way:

$$X = a'_1 b_1 M_1 y_1 + ... + a'_n b_n M_n y_n \ mod\ m$$

Brief algorithm description:

The algorithm performs this series of steps:

1- Get input from the user and check a_i and m_i relative primality

2- Calculate m and M_i

3- Check if m_i are pairwise relatively prime.

4- For each equation, calculate one solution term by:

- Calculate the inverse of ai in modulo mi
- Calculate the inverse of M1 in modulo mi
- Return the product of ai’ bi Mi yi

5- Sum all solution terms for each equation and mod m

Data Structures used:

The equations were represented by a datatype Equation which is a collection of a, b and m

```
type Modulo = Integer
data Equation = Equation { a :: Integer
                         , b :: Integer
                         , m :: Modulo
                         } deriving (Eq, Show)
```

Haskell’s default list (Which is implemented as a Singly Linked List) was used to operate on many equations.

Time complexity:

1- Getting the input from the user is implemented like this

```haskell
getEquation :: IO Equation
getEquation = do
            putStr "Enter an Equation (a b m): "
            hFlush stdout
            nums <- getLine
            let (a:b:m:[]) = map read $ words nums
            return (check (Equation a b m))

check :: Equation -> Equation
check eq@(Equation a _ m)
    | m <= 1 = error "m is <= 1: "
    | coprime a m = eq
    | otherwise = error $ "Coprime Error: " ++ (show a)
++ " And " ++ (show m) ++ " are not coprime"

getEquations :: IO [Equation]
getEquations = do
            putStr "Enter the number of equations: "
            hFlush stdout
            n <- getInt :: IO Int
            sequence . take n $ repeat getEquation
        where
            getInt = fmap read getLine
```

This is O(n) where n is the input from the user since the program runs the same number of tasks for each Equation

2- Calculating m and Mi is done like so:

```
calcMs :: [Modulo] -> (Modulo, [Modulo])
calcMs ms  = let bigM = product ms
                 bigMs = map (div bigM) ms
             in (bigM, bigMs)
```

Like the last step, this is O(n) since for each modulo it does 1 step in the second line (taking product), and also in the third line (dividing by big M)

3- Checking for pairwise relative primality is done in this fashion:

```
gcd' :: Integer -> Integer -> Integer
gcd' a 0 = a
gcd' a b = let (q, r) = a `divMod` b
           in gcd' b r

coprime :: Integer -> Integer -> Bool
coprime a b = gcd' a b == 1

coprimes :: [Modulo] -> Bool
coprimes [] = True
coprimes [x] = True
coprimes (x:xs) = and (map (coprime x) xs)
                  && coprimes xs
```

Since gcd' is a recursive implementation of the gcd algorithm, its complexity is O(logn), and the same thing can be said about coprime. On the other hand, coprimes is O(n^2) as the last line indicates, for each modulo we want to check for pairwise primality, we have to call it against every other modulo using the function *coprime*

4- Calculation of each solution term

```
inverse :: Integer -> Modulo -> Integer
inverse a m = let condition x = a * x `mod` m == 1
                  list = filter condition [1..m]
              in head list

solutionTerm :: Modulo -> Equation -> Integer
solutionTerm bigM (Equation a b m) =
        let a' = inverse a m
            bigMi = div bigM m
            y = inverse bigMi m
        in a' * b * bigMi * y

let summation = map (solutionTerm bigM) eqs
```

The function *inverse* has is a brute force algorithm for finding inverses of numbers in certain modulos and it's O(m). The function *solutionTerm* calls *inverse* two and is O(m). Finally, in the last line, we perform *solutionTerm* for each equation (n), so the big Oh of this step is O(mn).

5- Summing and taking modulo

```
ans = (sum summation) `mod` bigM
```

This step sums all solution terms from the last step and takes the modulo. This step is O(n)

Overall the time complexity for this algorithm is O(n^2 + mn)

Conclusions:

Computing the solution to n linear congruencies is identical to the Chinese Remainder Theorem (CRT) with the exception of the coefficient a, which we deal with by computing the inverse of a so that a * a' = 1 (mod m). We can further improve this algorithm by using the Extended Euclidean Algorithm to compute the inverse, which will result in a solution of $O(\log n)$ for that function.