Data Analysis and Algorithm

Practical 10

Implement Chinese reminder theorem to a constraint satisfaction problem. Analyze its complexity..

Date.: 24-10-21

Name – Yash Vasudeo Prajapati

Rollno - 022

MSc. Computer Science

Theory:-

In number theory, the Chinese remainder theorem states that if one knows the remainders of the Euclidean division of an integer n by several integers, then one can determine uniquely the remainder of the division of n by the product of these integers, under the condition that the divisors are pairwise coprime.

The following is a general construction to find a solution to a system of congruences using the Chinese remainder theorem:

1. Compute $N = n_1 \times n_2 \times \cdots \times n_k$.
2. For each $i = 1, 2, \ldots, k$, compute
$$y_i = \frac{N}{n_i} = n_1 n_2 \cdots n_{i-1} n_{i+1} \cdots n_k.$$
3. For each $i = 1, 2, \ldots, k$, compute $z_i \equiv y_i^{-1} \bmod n_i$ using Euclid's extended algorithm (z_i exists since $n_1, n_2, \ldots, n_k$ 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.

EXAMPLE

Show that there are no solutions to the system of congruences:

$$\begin{cases} x \equiv 2 \pmod{6} \\ x \equiv 5 \pmod{9} \\ x \equiv 7 \pmod{15}. \end{cases}$$

Note that each modulus is divisible by 3. The first and second congruences imply that $x \equiv 2 \pmod{3}$. However, the third congruence implies that $x \equiv 1 \pmod{3}$. Since these both cannot be true, there are no solutions to the system of congruences. $\square$

THEOREM

A system of linear congruences has solutions if and only if for every pair of congruences within the system,

$$\begin{cases} x \equiv a_i \pmod{n_i} \\ x \equiv a_j \pmod{n_j}, \end{cases} \qquad a_i \equiv a_j \pmod{\gcd(n_i, n_j)}.$$

Furthermore, if solutions exist, then they are of the form

$$x \equiv b \pmod{\operatorname{lcm}(n_1, n_2, \ldots, n_k)}$$

for some integer b.

Solve the system of congruences

$$\begin{cases} x \equiv 1 \pmod{3} \\ x \equiv 4 \pmod{5} \\ x \equiv 6 \pmod{7}. \end{cases}$$

Begin with the congruence with the largest modulus, $x \equiv 6 \pmod{7}$. Rewrite this congruence as an equivalent equation:

$$x = 7j + 6, \text{ for some integer } j.$$

Substitute this expression for x into the congruence with the next largest modulus:

$$x \equiv 4 \pmod{5} \implies 7j + 6 \equiv 4 \pmod{5}.$$

Then solve this congruence for j :

$$j \equiv 4 \pmod{5}.$$

Rewrite this congruence as an equivalent equation:

$$j = 5k + 4, \text{ for some integer } k.$$

Substitute this expression for j into the expression for x :

$$\begin{aligned} x &= 7(5k + 4) + 6 \\ x &= 35k + 34. \end{aligned}$$

Now substitute this expression for x into the final congruence, and solve the congruence for k :

$$\begin{aligned} 35k + 34 &\equiv 1 \pmod{3} \\ k &\equiv 0 \pmod{3}. \end{aligned}$$

Write this congruence as an equation, and then substitute the expression for k into the expression for x :

$$\begin{aligned} k &= 3l, \text{ for some integer } l. \\ x &= 35(3l) + 34 \\ x &= 105l + 34. \end{aligned}$$

This equation implies the congruence

$$x \equiv 34 \pmod{105}.$$

This happens to be the solution to the system of congruences. □

Program:-

Program 1

```
def reminder(a, b):
    b0 = b
    x0, x1 = 0, 1
    if b == 1: return 1
    while a > 1:
        q = a // b      #35 // 3
        a, b = b, a%b   #a = 3, b = 2
        #x0 = 1- (11 *0), x1=1
        x0, x1 = x1 - q * x0, x0
    if x1 < 0:
        x1 += b0
    return x1

if __name__ == '__main__':
    from functools import reduce

    a = [2, 3, 2]
    #a(mod n)
    n = [3, 5, 7]


    #return the N = n1*n2*n3
    t = reduce(lambda a, b: a*b, n)
    Ni = [ t//i for i in n]
    xi = [ reminder(Ni[i], n[i]) for i in range(0,len(a))]

    total = [a[i]*Ni[i]*xi[i] for i in range(0,len(a))]
    total = sum(total)
    x = total % t
    print(x)
```

Program 2

```
from functools import reduce
def chinese_remainder(n, a):
    sum = 0
    #return the N = n1*n2*n3
    prod = reduce(lambda a, b: a*b, n)
    #zip give {n1:a1,n2:a2...}
    for ni, ai in zip(n, a):
        p = prod // ni
        print(ni)
        sum += ai * mul_inv(p, ni) * p
    return sum % prod

def mul_inv(a, b):
    b0 = b
    x0, x1 = 0, 1
    #if ni = 1
    if b == 1: return 1
    #else p > 1
    while a > 1:
        q = a // b      #35 // 3
        a, b = b, a%b   #a = 3, b = 2
        x0, x1 = x1 - q * x0, x0
    if x1 < 0:
        x1 += b0
    return x1


a = [2, 3, 2]
n = [3, 5, 7]

print(chinese_remainder(n,a))
```

Output:-

```
23
>>> |
```

Complexity Analysis:-

The complexity of Chinese reminder theorem is O(n) where n is the number of input / size of array .
The program only needs one loop to calculate the product of all the n.

Conclusion:-

We implement Chinese reminder theorem and find that its complexity is O(n).