**Cryptography and Network Security Lab**

**Practical 10**

Q. Implement Chinese Remainder Theorem(CRT).

**Theory:**

The Chinese Remainder Theorem (CRT) is a fundamental theorem in number theory and modular arithmetic. It provides a method for solving systems of simultaneous modular congruences. The theorem is named the "Chinese Remainder Theorem" due to its first appearance in a book written by the Chinese mathematician Sun Tzu (Sunzi) in the 3rd century.

The CRT is particularly useful in number theory, cryptography, and computer science for efficient computations in modular arithmetic. It allows you to find a number that satisfies several modular congruences simultaneously. Here's a detailed explanation of the Chinese Remainder Theorem:

Statement of the CRT:

Suppose you have a system of congruences in the form:

```

x ≡ a₁ (mod m₁)

x ≡ a₂ (mod m₂)

x ≡ a₃ (mod m₃)

...

x ≡ aₖ (mod mₖ)

```

where `a₁, a₂, ..., aₖ` are integers and `m₁, m₂, ..., mₖ` are pairwise coprime (i.e., the greatest common divisor (GCD) of any two `mᵢ` is 1). The Chinese Remainder Theorem states that there exists a unique solution `x` modulo `M`, where `M = m₁ \* m₂ \* ... \* mₖ`.

Computing `x` using CRT:

To find the solution `x`, you can use the following procedure:

1. Compute `M = m₁ \* m₂ \* ... \* mₖ`.

2. For each `i` from 1 to `k`, compute `Mᵢ = M / mᵢ`. In other words, `Mᵢ` is the product of all `mⱼ` except `mᵢ`.

3. Find the modular multiplicative inverse of `Mᵢ` modulo `mᵢ`, denoted as `yᵢ`. This means `yᵢ` satisfies the congruence `Mᵢ \* yᵢ ≡ 1 (mod mᵢ)`.

4. Calculate `x` using the formula:

```

x = a₁ \* M₁ \* y₁ + a₂ \* M₂ \* y₂ + a₃ \* M₃ \* y₃ + ... + aₖ \* Mₖ \* yₖ

```

where `M₁, M₂, ..., Mₖ` are the precomputed values from step 2, and `y₁, y₂, ..., yₖ` are the modular inverses from step 3.

5. Finally, to ensure that `x` is the smallest non-negative solution, take `x` modulo `M`. The result is the unique solution in the range `[0, M)`.

Example:

Let's solve a simple system of congruences using the CRT:

```

x ≡ 2 (mod 3)

x ≡ 3 (mod 4)

x ≡ 2 (mod 5)

```

Here, `m₁ = 3`, `m₂ = 4`, and `m₃ = 5`. Since these numbers are pairwise coprime, we can apply the CRT.

1. Calculate `M = m₁ \* m₂ \* m₃ = 3 \* 4 \* 5 = 60`.

2. Compute `M₁ = M / m₁ = 60 / 3 = 20`, `M₂ = M / m₂ = 60 / 4 = 15`, and `M₃ = M / m₃ = 60 / 5 = 12`.

3. Find the modular inverses:

- For `m₁ = 3`, `M₁ = 20`, and `y₁` is the modular inverse of 20 modulo 3. In this case, `y₁ = 2` because `20 \* 2 ≡ 1 (mod 3)`.

- For `m₂ = 4`, `M₂ = 15`, and `y₂` is the modular inverse of 15 modulo 4. In this case, `y₂ = 3` because `15 \* 3 ≡ 1 (mod 4)`.

- For `m₃ = 5`, `M₃ = 12`, and `y₃` is the modular inverse of 12 modulo 5. In this case, `y₃ = 3` because `12 \* 3 ≡ 1 (mod 5)`.

4. Calculate `x`:

```

x = (2 \* 20 \* 2) + (3 \* 15 \* 3) + (2 \* 12 \* 3) = 40 + 135 + 72 = 247

```

5. Take `x` modulo `M`, so `x = 247 % 60 = 7`.

Therefore, the unique solution to the system of congruences is `x = 7`. This means that `x` is congruent to 7 modulo 60 and satisfies all three original congruences.

The Chinese Remainder Theorem is a powerful tool for simplifying modular arithmetic and solving systems of modular equations, especially when dealing with large integers or cryptography-related problems.

**Code:**

#include <bits/stdc++.h>

using namespace std;

*// Function for extended Euclidean Algorithm*

int ansS, ansT;

int findGcdExtended(int *r1*, int *r2*, int *s1*, int *s2*, int *t1*, int *t2*)

{

*// Base Case*

    if (*r2* == 0)

    {

        ansS = *s1*;

        ansT = *t1*;

        return *r1*;

    }

    int q = *r1* / *r2*;

    int r = *r1* % *r2*;

    int s = *s1* - q \* *s2*;

    int t = *t1* - q \* *t2*;

    cout << q << " " << *r1* << " " << *r2* << " " << r << " " << *s1* << " " << *s2* << " " << s << " " << *t1* << " " << *t2* << " " << t << endl;

    return findGcdExtended(*r2*, r, *s2*, s, *t2*, t);

}

int modInverse(int *A*, int *M*)

{

    int x, y;

    int g = findGcdExtended(*A*, *M*, 1, 0, 0, 1);

    if (g != 1) {

        cout << "Inverse doesn't exist";

        return 0;

    }

    else {

*// m is added to handle negative x*

        int res = (ansS % *M* + *M*) % *M*;

        cout << "inverse is " << res << endl;

        return res;

    }

}

int findX(vector<int> *num*, vector<int> *rem*, int *k*)

{

*// Compute product of all numbers*

    int prod = 1;

    for (int i = 0; i < k; i++)

        prod \*= num[i];

*// Initialize result*

    int result = 0;

*// Apply above formula*

    for (int i = 0; i < k; i++) {

        int pp = prod / num[i];

        result += rem[i] \* modInverse(pp, num[i]) \* pp;

    }

    return result % prod;

}

int main()

{

*// 3*

*// 3 4 5*

*// 2 3 1*

    int k;

    cin >> k;

    vector<int> num(k), rem(k);

    for (int i = 0; i < k; i++)

        cin >> num[i];

    for (int i = 0; i < k; i++)

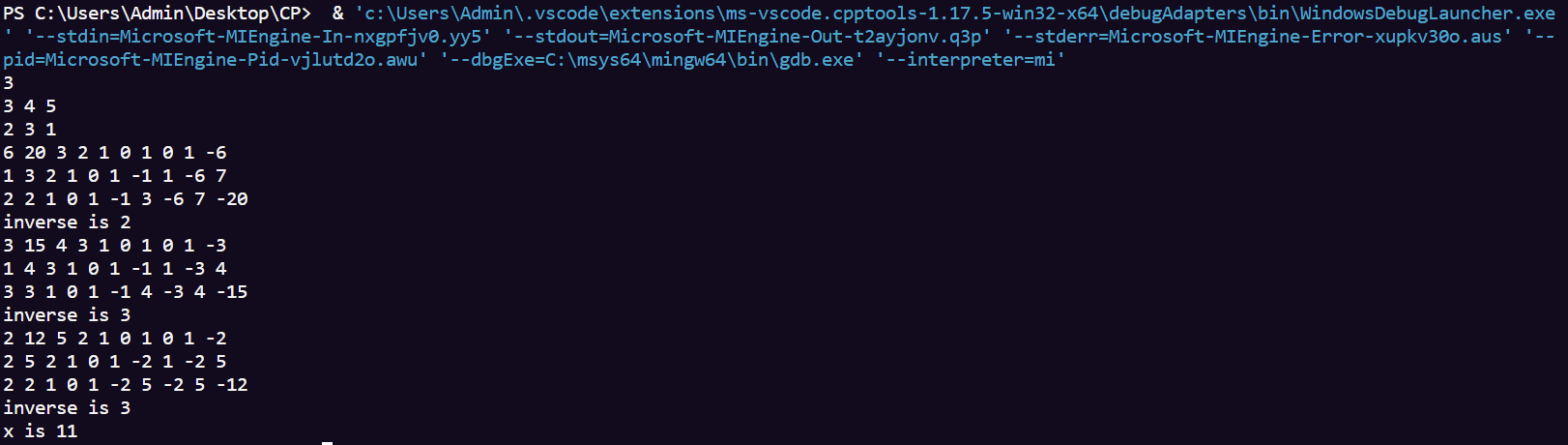
        cin >> rem[i];

    int x = findX(num, rem, k);

    cout << "x is " << x;

    return 0;

}

**Results:**