Chinese Remainder Theorem

Table of Contents

The Chinese Remainder Theorem (which will be referred to as CRT in the rest of this article) was discovered by Chinese mathematician Sun Zi.

Formulation

Let $p = p_1 p_2 \cdots p_k$, where p_i are pairwise relatively prime. In addition to p_i, we are also given a set of congruence equations

$$\begin{aligned} a &\equiv a_1 \pmod{p_1} \\ a &\equiv a_2 \pmod{p_2} \\ &\dots \\ a &\equiv a_k \pmod{p_k} \end{aligned}$$

where a_i are some given constants. The original form of CRT then states that the given set of congruence equations always has *one and exactly one* solution modulo p.

Corollary

A consequence of the CRT is that the equation

$$x \equiv a \pmod{p}$$

is equivalent to the system of equations

$$\begin{aligned} x &\equiv a \pmod{p_1} \\ &\dots \\ x &\equiv a \pmod{p_k} \end{aligned}$$

(As above, assume that $p = p_1 p_2 \cdots p_k$ and p_i are pairwise relatively prime).

Garner's Algorithm

Another consequence of the CRT is that we can represent big numbers using an array of small integers. For example, let p be the product of the first 1000 primes. From calculations we can see that p has around 3000 digits.

Any number a less than p can be represented as an array $a_1, \dots, a_k$, where $a_i \equiv a \pmod{p_i}$. But to do this we obviously need to know how to get back the number a from its representation. In this section, we

discuss Garner's Algorithm, which can be used for this purpose. We seek a representation on the form

$$a = x_1 + x_2 p_1 + x_3 p_1 p_2 + \ldots + x_k p_1 \ldots p_{k-1}$$

which is called the mixed radix representation of a. Garner's algorithm computes the coefficients $x_1, \ldots, x_k$.

Let r_{ij} denote the inverse of p_i modulo p_j

$$r_{ij} = (p_i)^{-1} \pmod{p_j}$$

which can be found using the algorithm described in Modular Inverse. Substituting a from the mixed radix representation into the first congruence equation we obtain

$$a_1 \equiv x_1 \pmod{p_1}.$$

Substituting into the second equation yields

$$a_2 \equiv x_1 + x_2 p_1 \pmod{p_2}.$$

which can be rewritten by subtracting x_1 and dividing by p_1 to get

$$\begin{aligned} a_2 - x_1 &\equiv x_2 p_1 && \pmod{p_2} \\ (a_2 - x_1) r_{12} &\equiv x_2 && \pmod{p_2} \\ x_2 &\equiv (a_2 - x_1) r_{12} && \pmod{p_2} \end{aligned}$$

Similarly we get that

$$x_3 \equiv ((a_3 - x_1) r_{13} - x_2) r_{23} \pmod{p_3}.$$

Now, we can clearly see an emerging pattern, which can be expressed by the following code:

```
for (int i = 0; i < k; ++i) {
    x[i] = a[i];
    for (int j = 0; j < i; ++j) {
        x[i] = r[j][i] * (x[i] - x[j]);

        x[i] = x[i] % p[i];
        if (x[i] < 0)
            x[i] += p[i];
    }
}
```

So we learned how to calculate coefficients x_i in $O(k^2)$ time. The number a can now be calculated using the previously mentioned formula

$$a = x_1 + x_2 p_1 + x_3 p_1 p_2 + \ldots + x_k p_1 \ldots p_{k-1}$$

It is worth noting that in practice, we almost always need to compute the answer using Big Integers, but the coefficients x_i can usually be calculated using built-in types, and therefore Garner's algorithm is very efficient.

Implementation of Garner's Algorithm

It is convenient to implement this algorithm using Java, because it has built-in support for large numbers through the `BigInteger` class.

Here we show an implementation that can store big numbers in the form of a set of congruence equations. It supports addition, subtraction and multiplication. And with Garner's algorithm we can convert the set of equations into the unique integer. In this code, we take 100 prime numbers greater than 10^9, which allows numbers as large as 10^{900}.

```
final int SZ = 100;
int pr[] = new int[SZ];
int r[][] = new int[SZ][SZ];

void init() {
    for (int x = 1000 * 1000 * 1000, i = 0; i
        if (BigInteger.valueOf(x).isProbablePr
            pr[i++] = x;

    for (int i = 0; i < SZ; ++i)
        for (int j = i + 1; j < SZ; ++j)
            r[i][j] =
                BigInteger.valueOf(pr[i]).modI
}

class Number {
    int a[] = new int[SZ];

    public Number() {
    }

    public Number(int n) {
        for (int i = 0; i < SZ; ++i)
            a[i] = n % pr[i];
```

```
    }

    public Number(BigInteger n) {
        for (int i = 0; i < SZ; ++i)
            a[i] = n.mod(BigInteger.valueOf(pr
    }

    public Number add(Number n) {
        Number result = new Number();
        for (int i = 0; i < SZ; ++i)
            result.a[i] = (a[i] + n.a[i]) % pr
        return result;
    }

    public Number subtract(Number n) {
        Number result = new Number();
        for (int i = 0; i < SZ; ++i)
            result.a[i] = (a[i] - n.a[i] + pr[
        return result;
    }

    public Number multiply(Number n) {
        Number result = new Number();
        for (int i = 0; i < SZ; ++i)
            result.a[i] = (int)((a[i] * 1l * n
        return result;
    }

    public BigInteger bigIntegerValue(boolean
        BigInteger result = BigInteger.ZERO, m
        int x[] = new int[SZ];
        for (int i = 0; i < SZ; ++i) {
            x[i] = a[i];
```

```
                for (int j = 0; j < i; ++j) {
                    long cur = (x[i] - x[j]) * 1l
                    x[i] = (int)((cur % pr[i] + pr
                }
                result = result.add(mult.multiply(
                mult = mult.multiply(BigInteger.va
            }

            if (can_be_negative)
                if (result.compareTo(mult.shiftRig
                    result = result.subtract(mult)

            return result;
        }
    }
```

Note on negative numbers

- Let p be the product of all primes.
- Modular scheme itself does not allow representing negative numbers. However, it can be seen that if we know that the absolute values our numbers are smaller than $p/2$, then we know that it must be negative when the resulting number is greater than $p/2$. The flag `can_be_negative` in this code allows converting it to negative in this case.

Practice Problems:

- Hackerrank - Number of sequences

- Codeforces - Remainders Game

(c) 2014-2019 translation by http://github.com/e-maxx-eng