The Chinese remainder theorem

Formulation

In its modern formulation of the theorem is as follows:

Let $p = p_1 \cdot p_2 \cdot \ldots \cdot p_k$where p_i- pairwise coprime.

We assign the arbitrary number of tuple where :a $(0 \le a < p)$$(a_1, \ldots, a_k)$
$a_i \equiv a \pmod{p_i}$

$$a \Longleftrightarrow (a_1, \ldots, a_k).$$

Then the correspondence (between numbers and tuples) will be **one to one** . And, moreover, the operations performed on the number a, you can perform the equivalent of the corresponding elements of tuples - by independent operations on each component.

That is, if

$$a \Longleftrightarrow \Big(a_1, \ldots, a_k\Big),$$

$$b \Longleftrightarrow \Big(b_1, \ldots, b_k\Big),$$

then we have:

$$(a+b) \pmod p \Longleftrightarrow \Big((a_1+b_1) \pmod{p_1}, \ldots, (a_k+b_k) \pmod{p_k}\Big),$$

$$(a-b) \pmod p \Longleftrightarrow \Big((a_1-b_1) \pmod{p_1}, \ldots, (a_k-b_k) \pmod{p_k}\Big),$$

$$(a \cdot b) \pmod p \Longleftrightarrow \Big((a_1 \cdot b_1) \pmod{p_1}, \ldots, (a_k \cdot b_k) \pmod{p_k}\Big).$$

In its original formulation of this theorem was proved by the Chinese mathematician Sun Tzu around 100 BC Namely, he has shown in the special case of modular equivalence of solutions of equations and solving a modular equation (see Corollary 2 below).

Corollary 1

Modular system of equations:

$$\begin{cases} x \equiv a_1 \pmod{p_1}, \\ \ldots, \\ x \equiv a_k \pmod{p_k} \end{cases}$$

has a unique solution modulo p.

(As above $p = p_1 \cdot \ldots \cdot p_k$, the numbers $p_i$are relatively prime, and the set $a_1, \ldots, a_k$- an arbitrary set of integers)

Corollary 2

Consequence is the connection between the system of modular equations and one corresponding modular equation:

The equation:

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

equivalent to the system of equations:

$$\begin{cases} x \equiv a \pmod{p_1}, \\ \ldots, \\ x \equiv a \pmod{p_k} \end{cases}$$

(As above, it is assumed that $p = p_1 \cdot \ldots \cdot p_k$, the number $p_i$of pairwise relatively prime, and a- is an arbitrary integer)

Garner's algorithm

Of the Chinese remainder theorem that can replace operations on the number of operations on tuples. Recall, each number $a$is assigned a tuple $(a_1, \ldots, a_k)$where:

$$a_i \equiv a \pmod{p_i}.$$

It can be widely used in practice (in addition to direct application to restore the number he balances of the various modules), as we thus can replace surgery in a long arithmetic operations with an array of "short" numbers. Say, an array of 1000 elements "enough" to number around with 3000signs (if selected as a $p_i$first-'s 1000 simple), and if selected as$p_i$simple's about a billion, then by enough already with some 9000signs. But, of course, then you need to learn how to **restore** by $a$this tuple. Corollary 1 shows that such a recovery is possible and, moreover, the only (provided $0 \leq a < p_1 \cdot p_2 \cdot \ldots \cdot p_k$). **Garner algorithm** and an algorithm that allows to perform this restoration, and quite effectively.

We seek a solution in the form:

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

ie mixed radix with digit weights $p_1, p_2, \ldots, p_k$.

Denoted by r_{ij}($i = 1 \ldots k-1$, $j = i+1 \ldots k$) number is the inverse $p_i$modulo p_j (finding the inverse elements in the ring modulo described here :

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

We substitute $a$in the mixed radix in the first equation, we get:

$$a_1 \equiv x_1.$$

We now substitute in the second equation:

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

Transform this expression, subtracting from both sides $x_1$and dividing by p_1:

$$a_2 - x_1 \equiv x_2 \cdot p_1 \pmod{p_2};$$

$$(a_2 - x_1) \cdot r_{12} \equiv x_2 \pmod{p_2};$$

$$x_2 \equiv (a_2 - x_1) \cdot r_{12} \pmod{p_2}.$$

Substituting into the third equation, the same way we obtain:

$$a_3 \equiv x_1 + x_2 \cdot p_1 + x_3 \cdot p_1 \cdot p_2 \pmod{p_3};$$

$$(a_3 - x_1) \cdot r_{13} \equiv x_2 + x_3 \cdot p_2 \pmod{p_3};$$

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

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

Already quite clearly visible pattern that is best to express the 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 to calculate the coefficients x_i for the time $O(k^2)$, he's the answer - the number a - can be restored by the formula:

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

It should be noted that in practice almost always need to calculate the answer using the Long arithmetic , but the coefficients themselves x_i are still calculated on the built-in types, and therefore the whole Garner algorithm is very effective.

Implementation of the algorithm Garner

Preferred to implement this algorithm in Java, because it contains a long arithmetic standard, but because there are no problems with the transfer of the number of the modular system in the usual number (using a standard class BigInteger).

The following implementation of the algorithm Garner supports addition, subtraction and multiplication, and supports the work with negative numbers (see notes about this after the code). Implemented a number of customary translation desyatichkogo submission to a modular system and vice versa.

In this example, taken 100 after a simple 10^9, allowing you to work with numbers up to about 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<SZ; ++x)
        if (BigInteger.valueOf(x).isProbablePrime(100))
            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]
).modInverse(
                    BigInteger.valueOf( pr[j] )
).intValue();
}


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[i] )
).intValue();
    }

    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[i];
        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[i]) %
pr[i];
        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.a[i])
% pr[i] );
        return result;
    }

    public BigInteger bigIntegerValue (boolean
can_be_negative) {
        BigInteger result = BigInteger.ZERO,
            mult = BigInteger.ONE;
        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 * r[j]
[i];
                x[i] = (int)( (cur % pr[i] + pr[i]) %
pr[i] );
            }
            result = result.add( mult.multiply(
BigInteger.valueOf( x[i] ) ) );
            mult = mult.multiply( BigInteger.valueOf(
pr[i] ) );
        }

        if (can_be_negative)
            if (result.compareTo( mult.shiftRight(1) )
>= 0)
                result = result.subtract( mult );

        return result;
    }
}
```

Support for **negative** numbers deserves mention (flag can_be_negativefunction $\text{bigIntegerValue}()$). Modular scheme itself does not imply differences between positive and negative numbers. However, it can be seen that if the answer to a specific problem modulo does not exceed half of the product of all primes, the positive numbers will be different from the negative that positive numbers are obtained than this mid and negative - more. Therefore we are after classical algorithm Garner compare the result with the middle, and if it is, then we deduce minus, and invert the result (ie, subtract it from the product of all prime, and print it already).