MAXimal

The Chinese remainder theorem

The wording

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 relatively prime numbers.

We associate with an 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, can be corresponding element of the tuple - by the independent performance of 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 AD Namely, he showed in the particula of the system of modular equations and solutions of one of the 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

The consequence is a connection between the system of modular equations and a 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- an arbitrary integer)

Algorithm Garner

Of the Chinese remainder theorem, it follows that it is possible to replace operations on the number of operations on tuples. Recall, each numbe where:

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

It can be widely used in practice (in addition to the direct application for the restoration of its residues on the different modules), as we thus can r operations with an array of "short" numbers. Say, an array of 1000elements of "enough" to number around 3000characters (if selected as $p_i$the as a $p_i$simple -s about a billion, then enough already by with about 9000signs. But, of course, then you need to learn how to **restore** by $a$this tup a recovery is possible and, moreover, the only (provided $0 \le a < p_1 \cdot p_2 \cdot \ldots \cdot p_k$). **Garner algorithm** is an algorithm that allows to perform

We seek a solution in the form of:

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

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

We denote by r_{ij}($i = 1 \ldots k-1$, $j = i+1 \ldots k$) the number of which is the inverse $p_i$modulo p_j(finding the inverse elements in the ring m

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

Substituting the expression $a$in a mixed value 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 on 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, we obtain a similar manner:

$$a_3 \equiv x_1 + x_2 \cdot p_1 + x_3 \cdot p_1 \cdot p_2 \pmod{p_3};$$
$$(a_3 - x_1) \cdot r_{13} \equiv x_2 + x_3 \cdot p_2 \pmod{p_3};$$
$$((a_3 - x_1) \cdot r_{13} - x_2) \cdot r_{23} \equiv x_3 \pmod{p_3};$$
$$x_3 \equiv ((a_3 - x_1) \cdot r_{13} - x_2) \cdot r_{23} \pmod{p_3}.$$

Already clearly visible pattern that is easier 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 how to calculate the coefficients $x_i$of the time $O(k^2)$, the very same answer - 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 s
because the whole algorithm Garner is very effective.

Implementation of the algorithm Garner

Most convenient to implement this algorithm in Java, and because it contains a standard length arithmetic, and therefore there are no problems
modular system in an ordinary number (a standard class BigInteger).

The below implementation of the algorithm Garner support addition, subtraction and multiplication, with support work with negative numbers (se
code). Implemented transfer of conventional desyatichkogo represent a modular system and vice versa.

In this example, taken 100after the 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 the **negative** numbers deserves mention (flag $\mathrm{can_be_negative}$function $\mathrm{bigIntegerValue}()$). Modular scheme itself does not im
negative numbers. However, it can be seen that if a particular problem the answer modulo does not exceed half of the product of all primes, the
the negative that the positive numbers turn out less than this mid-, and negative - more. Therefore, we are after classical algorithm Garner compa
is, then we derive a minus, and invert the result (ie, subtract it from the product of all primes, and print it already).

2 Комментариев e-maxx

Лучшее вначале

Присоединиться к обсуждению...

Богдан • 2 года назад

"Преобразуем это выражение, отняв от обеих частей x1 и разделив на n1"
Может разделив на p1?

5 • Ответить • Поделиться ›

e_maxx Модератор Богдан • 2 года назад

Вы правы, исправил опечатку.

4 • Ответить • Поделиться ›

Подписаться Добавь Disqus на свой сайт