Last update: July 26, 2023

Garner's algorithm

A consequence of the Chinese Remainder Theorem 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. p has around 3000 digits.

Any number a less than p can be represented as an array $a_1, \ldots, a_k$, 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. One way is discussed in the article about the Chinese Remainder Theorem.

In this article we discuss an alternative, Garner's Algorithm, which can also be used for this purpose.

Mixed Radix Representation

We can represent the number a in the **mixed radix** representation:

$$a = x_1 + x_2 p_1 + x_3 p_1 p_2 + \ldots + x_k p_1 \cdots p_{k-1} \text{ with } x_i \in [0, p_i)$$

A mixed radix representation is a positional numeral system, that's a generalization of the typical number systems, like the binary numeral system or the decimal numeral system. For instance the decimal numeral system is a positional numeral system with the radix (or base) 10. Every a number is represented as a string of digits $d_1 d_2 d_3 \ldots d_n$ between 0 and 9, and E.g. the string 415 represents the number $4 \cdot 10^2 + 1 \cdot 10^1 + 5 \cdot 10^0$. In general the string of digits $d_1 d_2 d_3 \ldots d_n$ represents the number $d_1 b^{n-1} + d_2 b^{n-2} + \cdots + d_n b^0$ in the positional numeral system with radix b.

In a mixed radix system, we don't have one radix any more. The base varies from position to position.

Garner's algorithm

Garner's algorithm computes the digits $x_1, \ldots, x_k$. Notice, that the digits are relatively small. The digit x_i is an integer between 0 and $p_i - 1$.

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 digits x_i in $O(k^2)$ time. The number a can now be calculated using the previously mentioned formula

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

It is worth noting that in practice, we almost probably need to compute the answer a using Arbitrary-Precision Arithmetic, but the digits x_i (because they are small) 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 representing 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 < 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;
    }
}
```