Last update: August 20, 2023 Translated From: e-maxx.ru

Modular Multiplicative Inverse

Definition

A modular multiplicative inverse of an integer a is an integer x such that $a \cdot x$ is congruent to 1 modular some modulus m. To write it in a formal way: we want to find an integer x so that

$$a \cdot x \equiv 1 \mod m.$$

We will also denote x simply with a^{-1}.

We should note that the modular inverse does not always exist. For example, let $m = 4$, $a = 2$. By checking all possible values modulo m, it should become clear that we cannot find a^{-1} satisfying the above equation. It can be proven that the modular inverse exists if and only if a and m are relatively prime (i.e. $\gcd(a, m) = 1$).

In this article, we present two methods for finding the modular inverse in case it exists, and one method for finding the modular inverse for all numbers in linear time.

Finding the Modular Inverse using Extended Euclidean algorithm

Consider the following equation (with unknown x and y):

$$a \cdot x + m \cdot y = 1$$

This is a Linear Diophantine equation in two variables. As shown in the linked article, when $\gcd(a, m) = 1$, the equation has a solution which can be found using the extended Euclidean algorithm. Note that $\gcd(a, m) = 1$ is also the condition for the modular inverse to exist.

Now, if we take modulo m of both sides, we can get rid of $m \cdot y$, and the equation becomes:

$$a \cdot x \equiv 1 \mod m$$

Thus, the modular inverse of a is x.

The implementation is as follows:

```
int x, y;
int g = extended_euclidean(a, m, x, y);
if (g != 1) {
    cout << "No solution!";
}
else {
    x = (x % m + m) % m;
    cout << x << endl;
}
```

Notice that the way we modify `x`. The resulting `x` from the extended Euclidean algorithm may be negative, so `x % m` might also be negative, and we first have to add `m` to make it positive.

Finding the Modular Inverse using Binary Exponentiation

Another method for finding modular inverse is to use Euler's theorem, which states that the following congruence is true if a and m are relatively prime:

$$a^{\phi(m)} \equiv 1 \mod m$$

ϕ is Euler's Totient function. Again, note that a and m being relative prime was also the condition for the modular inverse to exist.

If m is a prime number, this simplifies to Fermat's little theorem:

$$a^{m-1} \equiv 1 \mod m$$

Multiply both sides of the above equations by a^{-1}, and we get:

- For an arbitrary (but coprime) modulus m: $a^{\phi(m)-1} \equiv a^{-1} \mod m$
- For a prime modulus m: $a^{m-2} \equiv a^{-1} \mod m$

From these results, we can easily find the modular inverse using the binary exponentiation algorithm, which works in $O(\log m)$ time.

Even though this method is easier to understand than the method described in previous paragraph, in the case when m is not a prime number, we need to calculate Euler phi function, which involves factorization of m, which might be very hard. If the prime factorization of m is known, then the complexity of this method is $O(\log m)$.

Finding the modular inverse for prime moduli using Euclidean Division

Given a prime modulus $m > a$ (or we can apply modulo to make it smaller in 1 step), according to Euclidean Division

$$m = k \cdot a + r$$

where $k = \left\lfloor \frac{m}{a} \right\rfloor$ and $r = m \bmod a$, then

$$\begin{aligned}
\implies & & 0 \equiv k \cdot a + r & \mod m \\
\iff & & r \equiv -k \cdot a & \mod m \\
\iff & & r \cdot a^{-1} \equiv -k & \mod m \\
\iff & & a^{-1} \equiv -k \cdot r^{-1} & \mod m
\end{aligned}$$

Note that this reasoning does not hold if m is not prime, since the existence of a^{-1} does not imply the existence of r^{-1} in the general case. To see this, lets try to calculate 5^{-1} modulo 12 with the above formula. We would like to arrive at 5, since $5 \cdot 5 \equiv 1 \bmod 12$. However, $12 = 2 \cdot 5 + 2$, and we have $k = 2$ and $r = 2$, with 2 being not invertible modulo 12.

If the modulus is prime however, all a with $0 < a < m$ are invertible modulo m, and we can have the following recursive function (in C++) for computing the modular inverse for number a with respect to m

```
int inv(int a) {
  return a <= 1 ? a : m - (long long)(m/a) * inv(m % a) % m;
}
```

The exact time complexity of the this recursion is not known. It's is somewhere between $O(\frac{\log m}{\log \log m})$ and $O(m^{\frac{1}{3} - \frac{2}{177} + \epsilon})$. See On the length of Pierce expansions. In practice this implementation is fast, e.g. for the modulus $10^9 + 7$ it will always finish in less than 50 iterations.

Applying this formula, we can also precompute the modular inverse for every number in the range $[1, m-1]$ in $O(m)$.

```
inv[1] = 1;
for(int a = 2; a < m; ++a)
    inv[a] = m - (long long)(m/a) * inv[m%a] % m;
```

Finding the modular inverse for array of numbers modulo m

Suppose we are given an array and we want to find modular inverse for all numbers in it (all of them are invertible). Instead of computing the inverse for every number, we can expand the fraction by the prefix product (excluding itself) and suffix product (excluding itself), and end up only computing a single inverse instead.

$$\begin{aligned} x_i^{-1} &= \frac{1}{x_i} = \frac{\overbrace{x_1 \cdot x_2 \cdots x_{i-1}}^{\text{prefix}_{i-1}} \cdot ~1~ \cdot \overbrace{x_{i+1} \cdot x_{i+2} \cdots x_n}^{\text{suffix}_{i+1}}}{x_1 \cdot x_2 \cdots x_{i-1} \cdot x_i \cdot x_{i+1} \cdot x_{i+2} \cdots x_n} \\ &= \text{prefix}_{i-1} \cdot \text{suffix}_{i+1} \cdot \left(x_1 \cdot x_2 \cdots x_n\right)^{-1} \end{aligned}$$

In the code we can just make a prefix product array (exclude itself, start from the identity element), compute the modular inverse for the product of all numbers and than multiply it by the prefix product and suffix product (exclude itself). The suffix product is computed by iterating from the back to the front.

```
std::vector<int> invs(const std::vector<int> &a, int m) {
    int n = a.size();
    if (n == 0) return {};
    std::vector<int> b(n);
    int v = 1;
    for (int i = 0; i != n; ++i) {
        b[i] = v;
        v = static_cast<long long>(v) * a[i] % m;
    }
    int x, y;
    extended_euclidean(v, m, x, y);
    x = (x % m + m) % m;
    for (int i = n - 1; i >= 0; --i) {
        b[i] = static_cast<long long>(x) * b[i] % m;
        x = static_cast<long long>(x) * a[i] % m;
    }
    return b;
}
```

Practice Problems

- UVa 11904 - One Unit Machine
- Hackerrank - Longest Increasing Subsequence Arrays
- Codeforces 300C - Beautiful Numbers
- Codeforces 622F - The Sum of the k-th Powers
- Codeforces 717A - Festival Organization
- Codeforces 896D - Nephren Runs a Cinema

Contributors:

jakobkogler (39.76%) thanhtnguyen (27.11%) hly1204 (13.25%) duckladydinh (5.42%) adamant-pwn (4.22%) prprprpony (3.61%) tcNickolas (3.01%) mlangc (1.81%) iAngkur (0.6%) jxu (0.6%) panwar2001 (0.6%)