Last update: August 25, 2024 Translated From: e-maxx.ru

Factorial modulo p

In some cases it is necessary to consider complex formulas modulo some prime p, containing factorials in both numerator and denominator, like such that you encounter in the formula for Binomial coefficients. We consider the case when p is relatively small. This problem makes only sense when the factorials appear in both numerator and denominator of fractions. Otherwise $p!$ and subsequent terms will reduce to zero. But in fractions the factors of p can cancel, and the resulting expression will be non-zero modulo p.

Thus, formally the task is: You want to calculate $n! \bmod p$, without taking all the multiple factors of p into account that appear in the factorial. Imagine you write down the prime factorization of $n!$, remove all factors p, and compute the product modulo p. We will denote this *modified* factorial with $n!_{\%p}$. For instance $7!_{\%p} \equiv 1 \cdot 2 \cdot \underbrace{1}_{3} \cdot 4 \cdot 5 \underbrace{2}_{6} \cdot 7 \equiv 2 \bmod 3$.

Learning how to effectively calculate this modified factorial allows us to quickly calculate the value of the various combinatorial formulas (for example, Binomial coefficients).

Algorithm

Let's write this modified factorial explicitly.

$$\begin{aligned} n!_{\%p} &= 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot \underbrace{1}_{p} \cdot (p+1) \cdot (p+2) \cdot \ldots \cdot (2p-1) \cdot \underbrace{2}_{2p} \\ &\quad \cdot (2p+1) \cdot \ldots \cdot (p^2-1) \cdot \underbrace{1}_{p^2} \cdot (p^2+1) \cdot \ldots \cdot n \pmod{p} \\ &= 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot \underbrace{1}_{p} \cdot 1 \cdot 2 \cdot \ldots \cdot (p-1) \cdot \underbrace{2}_{2p} \cdot 1 \cdot 2 \\ &\quad \cdot \ldots \cdot (p-1) \cdot \underbrace{1}_{p^2} \cdot 1 \cdot 2 \cdot \ldots \cdot (n \bmod p) \pmod{p} \end{aligned}$$

It can be clearly seen that factorial is divided into several blocks of same length except for the last one.

$$\begin{aligned} n!_{\%p} = &\underbrace{1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot 1}_{1\text{st}} \cdot \underbrace{1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot 2}_{2\text{nd}} \cdot \ldots \\ &\cdot \underbrace{1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot 1}_{p\text{th}} \cdot \ldots \cdot \quad \underbrace{1 \cdot 2 \cdot \cdot \ldots \cdot (n \bmod p)}_{\text{tail}} \pmod{p}. \end{aligned}$$

The main part of the blocks it is easy to count — it's just $(p-1)! \bmod p$. We can compute that programmatically or just apply Wilson theorem which states that $(p-1)! \bmod p = -1$ for any prime p.

We have exactly $\lfloor \frac{n}{p} \rfloor$ such blocks, therefore we need to raise -1 to the power of $\lfloor \frac{n}{p} \rfloor$. This can be done in logarithmic time using Binary Exponentiation; however you can also notice that the result will switch between -1 and 1, so we only need to look at the parity of the exponent and multiply by -1 if the parity is odd. And instead of a multiplication, we can also just subtract the current result from p.

The value of the last partial block can be calculated separately in $O(p)$.

This leaves only the last element of each block. If we hide the already handled elements, we can see the following pattern:

$$n!_{\%p} = \underbrace{\ldots \cdot 1}_{} \cdot \underbrace{\ldots \cdot 2}_{} \cdot \ldots \cdot \underbrace{\ldots \cdot (p-1)}_{} \cdot \underbrace{\ldots \cdot 1}_{} \cdot \underbrace{\ldots \cdot 1}_{} \cdot \underbrace{\ldots \cdot 2}_{} \cdots$$

This again is a *modified* factorial, only with a much smaller dimension. It's $\lfloor n/p \rfloor !_{\%p}$.

Thus, during the calculation of the *modified* factorial $n!_{\%p}$ we did $O(p)$ operations and are left with the calculation of $\lfloor n/p \rfloor !_{\%p}$. We have a recursive formula. The recursion depth is $O(\log_p n)$, and therefore the complete asymptotic behavior of the algorithm is $O(p \log_p n)$.

Notice, if you precompute the factorials $0!,\ 1!,\ 2!,\ \dots,\ (p-1)!$ modulo p, then the complexity will just be $O(\log_p n)$.

Implementation

We don't need recursion because this is a case of tail recursion and thus can be easily implemented using iteration. In the following implementation we precompute the factorials $0!,\ 1!,\ \dots,\ (p-1)!$, and thus have the runtime $O(p + \log_p n)$. If you need to call the function multiple times, then you can do the precomputation outside of the function and do the computation of $n!_{\%p}$ in $O(\log_p n)$ time.

```
int factmod(int n, int p) {
    vector<int> f(p);
    f[0] = 1;
    for (int i = 1; i < p; i++)
        f[i] = f[i-1] * i % p;

    int res = 1;
    while (n > 1) {
        if ((n/p) % 2)
            res = p - res;
        res = res * f[n%p] % p;
        n /= p;
    }
    return res;
}
```

Alternative, if you only have limit memory and can't afford storing all factorials, you can also just remember the factorials that you need, sort them, and then compute them in one sweep by computing the factorials $0!,\ 1!,\ 2!,\ \dots,\ (p-1)!$ in a loop without storing them explicitly.

Multiplicity of p

If we want to compute a Binomial coefficient modulo p, then we additionally need the multiplicity of the p in n, i.e. the number of times p occurs in the prime factorization of n, or number of times we erased p during the computation of the *modified* factorial.

Legendre's formula gives us a way to compute this in $O(\log_p n)$ time. The formula gives the multiplicity ν_p as:

$$\nu_p(n!) = \sum_{i=1}^{\infty} \left\lfloor \frac{n}{p^i} \right\rfloor$$

Thus we get the implementation:

```
int multiplicity_factorial(int n, int p) {
    int count = 0;
    do {
        n /= p;
        count += n;
    } while (n);
    return count;
}
```

This formula can be proven very easily using the same ideas that we did in the previous sections. Remove all elements that don't contain the factor p. This leaves $\lfloor n/p \rfloor$ element remaining. If we remove the factor p from each of them, we get the product $1 \cdot 2 \cdots \lfloor n/p \rfloor = \lfloor n/p \rfloor!$, and again we have a recursion.