Factorial calculation modulo

In some cases it is necessary to consider some simple module for $p$complex formulas, including that may contain factorials. Here we consider the case when the module p is relatively small. It is clear that this problem only makes sense if the factorial and included in the numerator and the denominator of the fractions. Indeed, factorial $p!$ and all subsequent vanish modulo p, but all the factors fractions containing p, may be reduced, and the resulting expression has to be different from zero modulo p.

Thus, formally **challenge** this. Required to calculate the $n!$modulo a prime p, thus not taking into account all the multiple $p$factors included in the factorial. By learning to effectively compute a factorial, we can quickly calculate the value of various combinatorial formulas (eg, Binomial coefficients).

Algorithm

Let us write this "modified" factorial explicitly:

$$n!_{\%p} =$$

$$= 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot \underbrace{1}_{p} \cdot (p+1) \cdot (p+2) \cdot \ldots \cdot (2p-1) \cdot \underbrace{2}_{2p} \cdot (2p+1) \cdot \ldots \cdot$$

$$\cdot (p^2-1) \cdot \underbrace{1}_{p^2} \cdot (p^2+1) \cdot \ldots \cdot n =$$

$$= 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot \cdot \underbrace{1}_{p} \cdot 1 \cdot 2 \cdot \ldots \cdot (p-1) \cdot \underbrace{2}_{2p} \cdot 1 \cdot 2 \cdot \ldots \cdot (p-1) \cdot \underbrace{1}_{p^2} \cdot$$

$$\cdot 1 \cdot 2 \cdot \ldots \cdot (n\%p) \pmod p.$$

When such a record shows that the "modified" factorial divided into several blocks of length p(the last block may be shorter), which are all identical, except for the last element:

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

$$\cdot \underbrace{1 \cdot 2 \cdot \ldots \cdot (n\%p)}_{tail} \pmod p.$$

Common part count blocks easily - it's just $(p-1)! \bmod p$ that you can count programmatically or by Theorem Wilson (Wilson) immediately find $(p-1)! \bmod p = p-1$. To multiply the common parts of blocks, it is necessary to build a value found in the exponentiation p that can be done in $O(\log n)$ time (see binary exponentiation , however, you will notice that we actually erect minus one in some degree, and therefore the result will always be either 1 or $p-1$, depending on the parity index. value in the last, incomplete block also can be calculated separately for $O(p)$. Only the last elements of the blocks, a closer look at them:

$$n!_{\%p} = \underbrace{\ldots \cdot 1} \cdot \underbrace{\ldots \cdot 2} \cdot \underbrace{\ldots \cdot 3} \cdot \ldots \cdot \underbrace{\ldots \cdot (p-1)} \cdot \underbrace{\ldots \cdot 1} \cdot \underbrace{\ldots \cdot 1} \cdot \underbrace{\ldots \cdot 2} \ldots$$

And again we come to the "modified" factorial, but smaller dimension (as much as was complete blocks, and they were $\lfloor n/p \rfloor$). Thus, the calculation of the "modified" factorial $n!_{\%p}$ we reduced for $O(p)$ operations to the calculation already $(n/p)!_{\%p}$. Expanding this recurrence relation, we find that the depth of recursion is $O(\log_p n)$, total **asymptotic behavior of** the algorithm is obtained $O(p \log_p n)$.

Implementation

It is clear that the implementation is not necessary to use recursion explicitly: as tail recursion, it is easy to deploy in the cycle.

```
int factmod (int n, int p) {
	int res = 1;
	while (n > 1) {
		res = (res * ((n/p) % 2 ? p-1 : 1)) % p;
		for (int i=2; i<=n%p; ++i)
			res = (res * i) % p;
		n /= p;
	}
	return res % p;
}
```

This implementation works for $O(p \log_p n)$.