2023香农修仙班第二次课（一）

取模相关

引入

常常遇到题目要求计算一个特别大的数字，然后只要求输出这个答案对某个数取模的结果，例如：

> 给定 $1 \le n \le 10^5$ ，求 $n!$ 的从个位开始的后 8 位数字。$\Rightarrow$ 求 $n! \bmod 10^9$

> 由于答案可能会很大，所以请你将答案模以 998244353 后输出

> .....

为什么要进行取模操作

已知 C/C++ 常用的数据范围：

- `int` 通常 $[-2^{31}, 2^{31}-1]$ ， 4 字节，其中 $2^{31} \approx 2.1 \times 10^9$
- `long long` $[-2^{63}, 2^{63}-1]$ ， 8 字节，其中 $2^{63} \approx 9.2 \times 10^{18}$
- `unsigned long long` ， $[0, 2^{64}-1]$ ， 8 字节
- `__int128` $[-2^{127}, 2^{127}-1]$ ， 16 字节，其中 $2^{127} \approx 8.5 \times 10^{37}$
- `double` 有效位数约 15 位， 8 字节，范围约为 $[-1.79 \times 10^{308}, 1.79 \times 10^{308}]$
- `long double` 通常有效位数约 20 位，16 字节，范围约为 $[-1.2 \times 10^{4932}, 1.2 \times 10^{4932}]$

> `__int128` 一般不支持标准输入输出，只用于中间运算，如确实需要输入输出，要么转 `long long`，要么用快读快写
>
> 部分编译器无法编译，但是大部分 OJ 确实能用 `__int128`。

对于很多题目，答案远超于上述数据范围所能表示的。即使不考虑 TLE ，假设使用上述数据类型，要输出答案对某个数的取模，如果仅仅在全部计算结束后才进行取模，在全部计算完毕前已经溢出了。

所以需要一种能够避免溢出的做法， So here comes 取模公式。

> **高精度**在比赛里考得非常少，所以不专门讲，有兴趣者自学。更推荐使用 Python 自带的 `int` 或 `Decimal` 库，也可以用 Java 。

取模运算的直观展示

取模说简单点就是计算两个数相除的余数

所以我们可以用小学二年级的知识[😄] 来求余数

我们来演示以下 $100 \bmod 6$:

> $100 \div 6 = 16 \dots 4$
>
> 所以 $100 \bmod 6 = 4$

不过在 C 语言中与专门的取模操作符： `%` , 所以上述过程可以写为 `100 % 6` ， 结果为 4

加减乘的取模公式

取模 符号是 $\bmod$ ，如 $a \bmod b$ 代表 a 除 b 的余数。且有 $a \in Z, b \in Z^*$

对 $a, b \in Z^+$ ，显然有： $a \bmod b \in [0, b-1]$

> 下文规定所有出现在取模两边的数都是整数，且模数不为零

基本公式：

$$(a+b) \bmod p = (a \bmod p + b \bmod p) \mod p$$

$$(a-b) \bmod p = (a \bmod p - b \bmod p + p) \mod p$$

$$(a \times b) \bmod p = (a \bmod p \times b \bmod p) \mod p$$

应用：基本公式的运用避免了在计算过程中造成溢出，且减少了运算量(运算的一方总是不超过 p)，避免了开高精度。

使用结合律，可以推导出：

$$\begin{aligned} &(x_1 + x_2 + \cdots + x_n) \bmod p \\ =&((\cdots((x_1 + x_2) + x_3) + \cdots) + x_n) \bmod p \\ =&((\cdots((x_1 \bmod p + x_2 \bmod p) \bmod p + x_3 \bmod p) \bmod p + \cdots) \bmod p + x_n \bmod p) \bmod p \end{aligned}$$

同理：

$$\begin{aligned} &(x_1 - x_2 - \cdots - x_n) \bmod p \\ =&((\cdots((x_1 - x_2) - x_3) - \cdots) - x_n) \bmod p \\ =&((\cdots((x_1 \bmod p - x_2 \bmod p + p) \bmod p - x_3 \bmod p + p) \bmod p - \cdots) \bmod p - x_n \bmod + p) \bmod p \end{aligned}$$

$$\begin{aligned} &(x_1 \times x_2 \times \cdots \times x_n) \bmod p \\ =&((\cdots((x_1 \times x_2) \times x_3) \times \cdots) \times x_n) \bmod p \\ =&((\cdots((x_1 \bmod p \times x_2 \bmod p) \bmod p \times x_3 \bmod p) \bmod p \times \cdots) \bmod p \times x_n \bmod p) \bmod p \end{aligned}$$

除法的取模方法

我们在之前的加减乘法都是直接先进行一般性的运算，之后直接取余就好了

但加减乘除中除法并不遵循以上类似的规则，即：

$$\frac{a}{b} \bmod p \neq (\frac{a \bmod p}{b \bmod p}) \mod p$$

既然除法没办法使用这种规律，那么我们想办法把除法转化为乘法

假设：

$$\frac{a}{b} \bmod p = a \times b_{inv} \bmod p$$

那么我们现在就需要求出b_{inv}

具体的推导过程如下：

> 令 $\frac{a}{b} \equiv x \mod p$, 则有 $a \cdot b_{inv} \equiv x \mod p$ 。

> 两边同时乘上 b， 则有：
>
> $a \equiv x \cdot b \mod p$ 和 $a \cdot b_{inv} \cdot b \equiv x \cdot b \mod p$
>
> 之后根据模的减法得： $a \cdot (b \cdot b_{inv} - 1) \equiv 0 \mod p$, 因为我们在正整数范围内讨论逆元，所以 $a \neq 0$
>
> 所以： $b \cdot b_{inv} - 1 \equiv 0 \mod p$, 即： $b \cdot b_{inv} \equiv 1 \mod p$

所以我们得出了一下结论：

$$b \cdot b_{inv} \equiv 1 \mod p$$

考虑一个问题：上述式子在任何情况下都能有b_{inv} 使式子成立吗：

我们不妨将上面得式子改写一下：

> 因为 $b \cdot b_{inv}$ 除以 p 的余数为 1 , 所以我们可以将式子改为： $b \cdot b_{inv} = k \cdot p + 1$
>
> 再令g 为p , b 的最大公约数， 即： $g = gcd(p, b)$, 我们约定： $g \cdot b_1 = b$, $g \cdot p_1 = p$
>
> 则下列式子成立： $g \cdot b_1 \cdot b_{inv} = k \cdot g \cdot p_1 + 1$
>
> 我们可以将g 提出来： $g \cdot (b_1 \cdot b_{inv} - k \cdot p_1) = 1$

因为我们是在整数范围内考虑这些数， 所以可以得出这样两个结论：

> - $g = 1$
> - $b_1 \cdot b_{inv} - k \cdot p_1$ = 1

所以逆元存在的条件是： 当且仅当b 与 p 互质。

求乘法逆元

首先我们再次将逆元的结论抄下来：

$$b \cdot b_{inv} \equiv 1 \mod p$$

那我们将如何求出这个b_{inv} 呢：

我们可以想到**费马小定理**

> 若 p 为质数， 则 $a^{p-1} \equiv 1 \mod p$

所以我们可以将这个定理套到我们之前得出的结论里：

如果 p 为质数：

> $b^{p-1} \equiv 1 \mod p$
>
> $b \cdot b^{p-2} \equiv 1 \mod p$ == $b \cdot b_{inv} \equiv 1 \mod p$
>
> $b_{inv} = b^{p-2}$

所以我们就获得了一个逆元：

$$\text{如果}p\text{为质数}: b_{inv} = b^{p-2}$$

至于代码如何实现：我们可以使用快速幂来减少时间复杂度：

```
#include<bits/stdc++.h>
using namespace std;
typedef long long int lli;
lli P = 998244353;
lli mod(lli number) {return number % P;}
```

```
lli fast_pow(lli number, lli power){
    lli ans = 1LL; number %= P;
    while(power){
        if(power & 1) ans  = mod(ans * number);
        number = mod(number * number);
        power >>= 1;
    }
    return ans;
}
lli inv(lli number, lli p){
    return fast_pow(number, p - 2);
}
signed main()
{
    lli K; cin >> K;
    lli K_inv = inv(K, P);
    cout << ((K_inv * K) % P == 1);
}
```

费马小定理有一个弊端，那么就是要求 p 必须为质数：

为了可以适配更好的情况，我们可以使用**拓展欧几里得定理**， 它不要求p 为质数

> **拓展欧几里得定理 exgcd**
>
> exgcd 主要是求 $a \cdot x + b \cdot y = gcd(a, b)$ 中的一组 x 和 y

我们简单说明以下为什么拓展欧几里得定理可以求逆元：

我们根据**裴蜀定理**的逆定理可以得到：

> 设 a, b 是不全为零的整数，若 $d > 0$ 是 a, b 的公因数，且存在整数 $x, y,$ 使得 $ax + by = d$，则 $d = \gcd(a, b)$。
>
> 特殊地，设 a, b 是不全为零的整数，若存在整数 $x, y,$ 使得 $ax + by = 1$，则 a, b互质

精简一下上面这个定理： 存在一组解x, y使得$ax + by = \gcd(a, b)$

由于逆元存在，则有 $\gcd(a, b) = 1$，

我们将这个式子单独拿出来： $ax + by = \gcd(a, b)$

等式两边同时对 b 取余，由于 by 必然是 b 的倍数，所以 $by \bmod b = 0$，所以：

$$\begin{aligned} ax + by &\equiv \gcd(a, b) &\pmod b \\ ax + 0 &\equiv 1 &\pmod b \\ ax &\equiv 1 &\pmod b \end{aligned}$$

还记得我们之前提到的逆元的定义吗：

$$b \cdot b_{inv} \equiv 1 \mod p$$

所以上面的一个解 x 就是 a 在 p 下的逆元

如何利用拓展欧几里得定理来求逆元

为了让大家更好的理解**拓展欧几里得算法**的求逆元方法，我们需要通过 **欧几里得算法** 来证明 $\gcd(a,b) = \gcd(b, a \bmod b)$

> 我们假设 d 是 a, b 的任意一个公约数， 则满足 $a \bmod d = 0$ ，$b \bmod d = 0$
>
> 我们令 $a = kb + r$， 那么 $k = \lfloor \frac{a}{b} \rfloor$, $r = a \bmod b$， 并且满足 $0 \le r < b$
>
> 我们发现 d 是 b 和 r 的公因数
>
> > *为什么 d 是 r 的公因数*
> >
> > 由于 a, b 都可以被 d 整除， 所以我们将 a, b 表示为 $a = md, b = nd$
> >
> > 带入 $a = kb + r \Rightarrow md = knd + r \Rightarrow r = (m - kn)d$
> >
> > 所以 d 是 r 的公因数
>
> 因为 d 是 b 和 r 的公因数 也是 a 和 b 的公因数，所以 a, b 的最大公因数也等于 b, r 的最大公因数
>
> 即 $\gcd(a,b) = \gcd(b, a \bmod b)$

懂了上面的道理， 我们来思考以下如何求解拓展欧几里得定理：

- 当 $b = 0$ 时 , $\gcd(a,b) = a$， 这个时候 $x = 1, y = 0$
- 当 $b \ne 0$ 时 :

> 我们可以得到这个式子： $ax + by = \gcd(a,b) = \gcd(b, a \bmod b) = bx_2 + (a \bmod b)y_2$
> 因为 $a \bmod b = a - \lfloor \frac{a}{b} \rfloor \times b$, 所以我们就可以代入式子：
> $ax + by = bx_2 + (a - \lfloor \frac{a}{b} \rfloor \times b)y_2 \Rightarrow ax + by = bx_2 + ay_2 - b \times \lfloor \frac{a}{b} \rfloor y_2$
> $\Rightarrow ax + by = ay_2 + b(x_2 - b \times \lfloor \frac{a}{b} \rfloor y_2)$
> 所以我们就得到了： $x = y_2, y = x_2 - b \times \lfloor \frac{a}{b} \rfloor y_2$
> 至于 x_2, y_2 的值， 我们同样是使用上面的方法递归计算，直到 $b = 0$

代码

```
#include<bits/stdc++.h>
using namespace std;
typedef long long int lli;
lli P = 998244353;
lli mod(lli number) {return number % P;}
void exgcd(lli a, lli b, lli& x, lli& y){
    if(b == 0) {
        x = 1; y = 0;
        return;
    }
    exgcd(b, a % b, y, x);
    y -= (a / b) * x;
}
lli inv(lli number, lli p){
    lli x = 1, y = 0;
    exgcd(number, p, x, y);
    return (x % p + p) % p;
}
signed main()
{
```

```
    lli K; cin >> K;
    lli K_inv = inv(K, P);
    cout << ((K_inv * K) % P == 1);
}
```