${}_nC_k \mod p$ を高速に求める

BV21013　堀毛 晴輝

令和 3 年 11 月 4 日

目次

1　研究背景

競技プログラミングというプログラミングコンテストがある。参加者全員に同一の問題が出題され、より速く与えられた要求を満足するプログラムを正確に記述することを競うコンテストのことである。その中で、${}_nC_k \mod p$ を求める必要のある問題が出題されることがある。この時、問題の制約にもよるのだが、たいていの場合愚直に計算していてはオーバーフローしてしまったり、問題の制限時間を超えても処理が完了しなくなってしまう。今回は、高速かつオーバーフローしない、正確に ${}_nC_k \mod p$ を求めるにはどうすべきか考察していく。

以下、p は 10^9+7 とする。これは競技プログラミングではよく出てくる制約である。素数であり 10^9+7 未満の数同士を足しても 32bit 整数に収まり、また 10^9+7 未満の数同士を掛けても 64bit 整数に収まるからである。

2　ビッグオー記法

処理の実行にかかる時間を計算量という客観的な尺度を用いてアルゴリズムを評価することがある。計算量は O という記号を用いて表される。データ数を n として、処理回数を n の式で示したものを係数・定数を無視して最高次数のみを $O()$ の中に入れて表す。例えば、データ数 n に対し $3n+5$ 回の処理をするときは $O(n)$ と表す。

3　愚直に計算する

$$ {}_nC_k = \frac{n!}{k!(n-k)!} $$

であるため、n に対して $n!$ を返す関数を作ることにより、$n, k, (n-k)$ それぞれの階乗を計算したうえで ${}_nC_k$ を求め、 $\mod p$ をとれば $O(n)$ で ${}_nC_k \mod p$ を求めることができる。しかし、$21! = 51,090,942,171,709,440,000$ となり、64bit 符号なし整数の上限である $18,446,744,073,709,551,615$ を超えてしまいオーバーフローが起きてしまう。

4　逆元を用いる

$$ {}_nC_k = \frac{n!}{k!(n-k)!} = n! \times (k!)^{-1} \times ((n-k)!)^{-1} $$

であるから、 $\mod p$ 下で $k!, (n-k)!$ の逆元を求めればよい。

Fermat の小定理

p を素数とする。整数 a が p と互いに素なら、次が成り立つ

$$ a^{p-1} \equiv 1 \pmod{p} $$

より、$a \times a^{p-2} \equiv 1 \pmod{p}$　であるから a の逆元は a^{p-2} である。すなわち $k!, (n-k)!$ それぞれを $(p-2)$ 乗することで $k!, (n-k)!$ の逆元を求めることができる。ここで、愚直に $(p-2)$ 乗をしようとすると $p = 10^9 + 7$ のとき $10^9 + 5$ 回の掛け算を行うこととなる。競技プログラミングでは 1 秒当たり 10^8 程度の計算が行えるが、これだと 10 秒程度かかってしまい低速である。しかし、繰り返し 2 乗法を用いることによって $\log_2(10^9 + 5) \fallingdotseq 30$ 回の処理で答を出すことができる。

ソースコード 1　繰り返し 2 乗法

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
#define MOD 1000000007
ll modpow(ll a, ll n){
    ll ans = 1;
    while (n < 0){
        if (n & 1) ans = ans * a % MOD;
        a = a * a % MOD;
        n >>= 1;
    }
    return ans;
}
```

これらを使って、

ソースコード 2　Fermat の小定理を用いて ${}_nC_k \mod p$ を求める

```
ll factorial(ll n){
    ll ans = 1;
    for (ll i = 1;i <= n;i++){
        ans = ans * i % MOD;
    }
    return ans;
}

ll nCk(ll n, ll k){
    ll fan = factorial(n); //n!
    ll fak = factorial(k); //k!
    ll fank = factorial(n-k); //(n-k)!
    ll invfak = modpow(fak,MOD-2); //(k!)^-1
    ll invfank = modpow(fank,MOD-2); //((n-k)!)^-1
    ll ans = fan * invfak % MOD * invfank % MOD;
    return ans;
}
```

このようにして ${}_nC_k \mod p$ が求められる。15 行目で %MOD を 2 回行っているのはオーバーフロー対策である。以上より、${}_nC_k \mod p$ が $O(n)$ で計算できた。

5　前処理をあらかじめ行う

$1 \leq n, k \leq 10^7$ 程度のとき、あらかじめ 10^7 以下のすべての自然数について階乗とその逆元を求めておけば、それ以降の ${}_nC_k \mod p$ の計算をするときにそれらを用いることで、前処理以降は $O(1)$ で計算することができる。

ソースコード 3　前処理をあらかじめ行う

```
1    const ll nCkMax = 1e7;
2    vector <ll> fac(1,0), finv(1,0), inv(1,0);
3
4    void COMinit(){
5        fac.resize(nCkMax+1); //n!
6        finv.resize(nCkMax+1); //n^-1
7        inv.resize(nCkMax+1); //(n!)^-1
8        fac[0] = fac[1] = 1;
9        finv[0] = finv[1] = 1;
10       inv[1] = 1;
11       for(ll i=2;i<=nCkMax;i++){
12       fac[i] = fac[i-1] * i % MOD;
13       inv[i] = modpow(i,MOD-2);
14       finv[i] = finv[i - 1] * inv[i] % MOD;
15   }
16
17   ll nCk(ll n, ll k){
18       if(fac[0] != 1) COMinit();
19       if(n<k) return 0;
20       if(n<0 || k<0) return 0;
21       return fac[n] * (finv[k] * finv[n - k] % MOD) % MOD;
22   }
```

${}_nC_k \mod p$ が呼ばれたとき、前処理 (COMinit()) がなされていなかった場合、18 行目で前処理を行っている。コメントにあるように fac,inv,finv にそれぞれ $n!, n^{-1}, (n!)^{-1}$ を格納していく。inv 計算に $O(\log_2 p)$ かかるため、前処理が $O(N \log_2 p)$ かかる。それ以降の ${}_nC_k \mod p$ は fac,finv を用いることによって $O(1)$ で計算できる。

6　前処理をより高速に

p を a で割ると、

$$p = \left\lfloor \frac{p}{a} \right\rfloor \times a + (p\%a)$$

($p\%a$ は p を a で割った余り) となり、

$$\begin{aligned}
&\left\lfloor \frac{p}{a} \right\rfloor \times a + (p\%a) \equiv 0 \pmod{p} \\
\Leftrightarrow &\left\lfloor \frac{p}{a} \right\rfloor + (p\%a) \times a^{-1} \equiv 0 \\
\Leftrightarrow &(p\%a) \times a^{-1} \equiv -\left\lfloor \frac{p}{a} \right\rfloor \\
\Leftrightarrow &a^{-1} \equiv -(p\%a)^{-1} \times \left\lfloor \frac{p}{a} \right\rfloor
\end{aligned}$$

$p\%a$ は a より小さいため、inv[a] の計算を行うときには inv[$p\%a$] は計算済である。よってこの結果を使い、ソースコード 3 の 13 行目を、

```
1        inv[a] = MOD - inv[MOD % a] * (MOD / i) % MOD;
```

に書き換えることにより、inv 計算が $O(1)$ でできるようになり、結果 $O(N)$ で前処理が完了する。

最終的に完成したコードが以下の通りである。

ソースコード 4　完成コード

```
1        #include <bits/stdc++.h>
2        using namespace std;
3        using ll = long long;
4        #define MOD 1000000007
5        const ll nCkMax = 1e7;
6        vector <ll> fac(1,0), finv(1,0), inv(1,0);
7
8        void COMinit(){
9            fac.resize(nCkMax+1); //n!
10           finv.resize(nCkMax+1); //n^-1
11           inv.resize(nCkMax+1); //(n!)^-1
12           fac[0] = fac[1] = 1;
13           finv[0] = finv[1] = 1;
14           inv[1] = 1;
15           for(ll i = 2;i <= nCkMax;i++){
16           fac[i] = fac[i-1] * i % MOD;
17           inv[a] = MOD - inv[MOD % a] * (MOD / i) % MOD;
18           finv[i] = finv[i - 1] * inv[i] % MOD;
19       }
20
21       ll nCk(ll n, ll k){
22           if(fac[0] != 1) COMinit();
23           if(n<k) return 0;
24           if(n<0 || k<0) return 0;
25           return fac[n] * (finv[k] * finv[n - k] % MOD) % MOD;
26       }
```

7　終わりに

だいぶ解説を端折ってしまったが、これらを踏まえることで ${}_nC_k \mod p$ を競技プログラミングで使える程度高速に求めることができた。「あらかじめ前処理をしておくことでそれ以降の処理を高速に行う」ことは競技プログラミングでよく出される題材であり、十分理解することが必要である。また、今回詳しく解説しなかったが繰り返し 2 乗法はシミュレーションの高速化などに使われるダブリングにつながるとても重要な部分である。理解をした上で今後も競技プログラミングの精進に励んでいきたいと思う。

参考文献

[1] けんちょんの競プロ精進記録 よくやる二項係数 (${}_nC_k \mod p$)、逆元 ($a^{-1} \mod p$) の求め方 2018/6/8 https://drken1215.hatenablog.com/entry/2018/06/08/210000