Aritmética Modular
Professor Wladimir

Aritmética modular

Na programação competitiva, é comum lidar com problemas cujos resultados excedem os limites dos tipos de dados disponíveis. Nesses casos, geralmente solicita-se a resposta **resto da divisão** por algum número primo grande.

O valor mais utilizado é:

$$10^9 + 7 = 1\,000\,000\,007,$$

escolhido por diversos motivos:

- Encaixa em um inteiro de 32 bits sem causar *overflow*.
- É suficientemente grande para reduzir o risco de colisões em cálculos.
- Por ser primo, garante propriedades matemáticas úteis, como a existência de inverso multiplicativo para qualquer número que seja coprimo a ele.

Coeficente Binomial

O coeficiente binomial, também chamado de *número binomial*, é definido como:

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

Além dessa forma fechada, o coeficiente pode ser calculado de maneira recursiva, utilizando a seguinte relação:

$$\binom{n}{k} = \begin{cases} 1, & \text{se } k = 0 \text{ ou } k = n, \\ \binom{n-1}{k-1} + \binom{n-1}{k}, & \text{caso contrário.} \end{cases}$$

Essa recorrência pode ser interpretada da seguinte maneira: dado um elemento do conjunto, temos duas opções:

- **Escolher o elemento**: nesse caso, ainda precisamos selecionar $k-1$ elementos dentre os $n-1$ restantes.
- **Não escolher o elemento**: nesse caso, ainda precisamos selecionar k elementos dentre os $n-1$ restantes.

É importante observar que os números binomiais crescem muito rapidamente. A tabela a seguir ilustra alguns valores e os respectivos limites de variáveis inteiras em linguagens de programação:

Coeficiente Binomial	Valor	Observação
$\binom{33}{16}$	1.166.803.110	limite do `int`
$\binom{34}{17}$	2.333.606.220	limite do `unsigned int`
$\binom{66}{33}$	7.219.428.434.016.265.740	limite do `long long`
$\binom{67}{33}$	14.226.520.737.620.288.370	limite do `unsigned long long`

Tempo e Memória quadrático

Uma forma direta de calcular todos os coeficientes binomiais é construir a tabela completa do Triângulo de Pascal. Essa abordagem requer tempo $O(n^2)$ e memória $O(n^2)$, pois armazenamos todos os valores $\binom{i}{j}$ para $0 \leq j \leq i \leq n$.

```
    long long int C[n+1][n+1];
const long long int MOD = 1e9 + 7;

C[0][0] = 1;
for(int i = 1; i <= n; i++){
    C[i][0] = 1;
    for(int j = 1; j < i; j++){
        C[i][j] = (C[i-1][j-1] + C[i-1][j])%MOD;
    }
    C[i][i] = 1;
}
```

Tempo quadrático e Memória linear

Observando a recorrência:

$$\binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j},$$

percebemos que o cálculo da i-ésima linha depende apenas da linha anterior. Assim, não é necessário manter toda a tabela na memória, mas apenas duas linhas de cada vez.

Essa otimização reduz o uso de memória de $O(n^2)$ para $O(n)$, mantendo o mesmo tempo de execução $O(n^2)$.

```
    long long int C[n+1][n+1];
const long long int MOD = 1e9 + 7;

C[0][0] = 1;
for(int i = 1; i <= n; i++){
        C[i%2][0] = 1;
        for(int j = 1; j < i; j++){
            C[i%2][j] = (C[(i+1)%2][j-1] + C[(i+1)%2][j])%MOD;
        }
        C[i%2][i] = 1;
}
```

Tempo e Memória linear

A ideia é utilizar a fórmula fechada para o coeficiente binomial, da seguinte maneira:

$$\binom{n}{k} \mod MOD = n! \times (k!)^{-1} \times ((n-k)!)^{-1} \mod MOD.$$

Para isso, vamos pré-computar os fatoriais módulo MOD e seus inversos multiplicativos para todos os valores $0 \leq i \leq MAX$.

Inverso Modular via Teorema de Fermat

Quando MOD é primo, o Teorema de Fermat garante que:

$$a^{MOD-1} \equiv 1 \pmod{MOD} \quad \Rightarrow \quad a^{-1} \equiv a^{MOD-2} \pmod{MOD},$$

para todo a tal que $\gcd(a, MOD) = 1$.

Além disso, podemos explorar a seguinte relação de recorrência entre inversos fatoriais:

$$n! = n \times (n-1)! \pmod{MOD} \quad \Rightarrow \quad ((n-1)!)^{-1} \equiv n \times (n!)^{-1} \pmod{MOD}.$$

Exponenciação Modular Rápida

Para calcular o inverso modular de $invfact[MAXN]$, utilizamos *exponenciação modular rápida*:

```
/* Exponenciação modular rápida: base^exp % MOD */
int64_t mod_pow(int64_t base, int64_t exp) {
    int64_t res = 1 % MOD;
    base %= MOD;
    if (base < 0) base += MOD;
    while (exp > 0) {
        if (exp & 1) res = (res * base) % MOD;
        base = (base * base) % MOD;
        exp >>= 1;
    }
    return res;
}
```

Pré-cálculo dos Fatoriais e Inversos

Combinando os resultados, podemos pré-computar todos os valores necessários:

```
fact[0] = 1;
for (int i = 1; i <= MAXN; ++i)
    fact[i] = (fact[i-1] * i) % MOD;

/* Inverso do maior fatorial via Teorema de Fermat */
invfact[MAXN] = mod_pow(fact[MAXN], MOD-2);

/* Relação recorrente para os demais inversos */
for (int i = MAXN; i >= 1; --i)
    invfact[i-1] = (invfact[i] * i) % MOD;
```

Programa Completo

```
ll fact[MAXN+1];
ll invfact[MAXN+1];

/* Pré-cálculo */
fact[0] = 1;
for (int i = 1; i <= MAXN; ++i)
    fact[i] = (fact[i-1] * i) % MOD;

invfact[MAXN] = mod_pow(fact[MAXN], MOD-2);
for (int i = MAXN; i >= 1; --i)
    invfact[i-1] = (invfact[i] * i) % MOD;

/* Leitura e cálculo do binomial */
int n, k;
scanf("%d %d", &n, &k);
ll res;
if (k < 0 || k > n) res = 0;
else res = (((fact[n] * invfact[k]) % MOD) * invfact[n-k]) % MOD;
printf("%lld\n", res);
```

Calculando o Inverso Multiplicativo com o Algoritmo de Euclides Estendido

O *algoritmo de Euclides estendido* permite encontrar, para dois inteiros a e b, números inteiros x e y tais que:

$$\gcd(a, b) = a \cdot x + b \cdot y.$$

Se $\gcd(a, b) = 1$, então a e b são coprimos, e o valor de y na equação acima é o **inverso multiplicativo** de b módulo a.

Exemplo: Seja $a = 7$ e $b = 5$. O algoritmo de Euclides estendido encontra inteiros x e y tais que:

$$7(-2) + 5(3) = 1.$$

Portanto, o inverso multiplicativo de $5 \mod 7$ é 3, pois $5 \cdot 3 \equiv 1 \pmod 7$.

```
int gcd(int a, int b, int &x, int &y)
{
    int q, t;
    int x1, y1;
    x = 1; y = 0;
    x1 = 0; y1 = 1;
    while (b)
    {
        q = a / b;
        t = b;
        b = a - q * b;
        a = t;

        t = x1;
        x1 = x - q * x1;
        x = t;

        t = y1;
        y1 = y - q * y1;
        y = t;
    }
    return a;
}
```

Observação: O valor de y obtido pelo algoritmo geralmente satisfaz $|y| < a$, portanto ele não será arbitrariamente grande.

Para garantir que o inverso esteja no intervalo $[0, a-1]$, podemos calcular:

$$b^{-1} \equiv (y \bmod a + a) \bmod a.$$

Dessa forma, $b \cdot b^{-1} \equiv 1 \pmod a$ e o resultado é sempre positivo.

Link da Aula: Aritmética - Aula 23 - O algoritmo de Euclides estendido https://www.youtube.com/watch?v=oRwuQrm3gqE

Problemas de Programação Competitiva

1. **DIOFANTO – Equações diofantinas**
   Plataforma: SPOJ
   Descrição: Resolver a equação diofantina $x_1 + x_2 + \cdots + x_N = C$ com soluções inteiras não-negativas, onde $0 \leq x_i \leq C$ para todo i. Calcular o número de soluções módulo $10^9 + 7$.
   Dica: Use combinatória clássica: o número de soluções é $\binom{C+N-1}{N-1}$. Pré-calcule fatoriais e inversos modulares para eficiência.

2. **SMALL – Smallest Number**
   Plataforma: SPOJ
   Descrição: Dado um número N, encontrar o menor número que é múltiplo de todos os inteiros de 1 a N; saída deve ser $\mod 10^9 + 7$.
   Dica: Utilize LCM incremental ou fatoração de primos até N. Como N pode ser grande (até 10000), pré-calcule os menores expoentes de cada primo. Não tente calcular LCM diretamente de todos inteiros (porque explode). Use módulo em cada multiplicação.

3. **FIBOSUM – Fibonacci Sum**
   Plataforma: SPOJ
   Descrição: Dado N, M, calcular $\big(F(N) + F(N+1) + \cdots + F(M)\big) \mod 10^9 + 7$, para casos grandes (ex: M pode ser 10^9).
   Dica: Use *exponenciação de matriz* para calcular $F(n)$ rápido $O(\log n)$. Use a identidade de

soma de Fibonacci: $\sum_{i=N}^{M} F(i) = F(M+2) - F(N+1)$. Lidar bem com módulo e resultados negativos.

4. **SIMPLEPATH – Simple Path**
   Plataforma: SPOJ
   Descrição: Dada uma árvore com pesos nas arestas, para cada subárvore, computar soma dos comprimentos de todos os paths simples; imprimir resultado módulo $10^9 + 7$.
   Dica: Use DFS para acumular contribuições de caminhos. Saber que pares de vértices em subárvore implicam contar quantas vezes cada aresta participa. Use prefixos / soma / contagem de nós. Cuidado com overflow: peso pode ser grande, número de pares também. Apply mod em cada operação.

5. **466D – Equalize the Array**
   Plataforma: Codeforces
   Descrição: Dada uma sequência de inteiros $a_1, a_2, \ldots, a_n$, calcular o número de formas distintas de torná-los todos iguais, utilizando operações de incremento em segmentos; resultado $\bmod 10^9 + 7$.
   Dica: Utilize técnicas de programação dinâmica ou combinatória para contar as formas possíveis de realizar as operações, aplicando módulo em cada cálculo.

6. **100570A – LCM Queries**
   Plataforma: Codeforces
   Descrição: Dada uma sequência $a_1, a_2, \ldots, a_n$ e m consultas, para cada consulta x, calcular o LCM dos x primeiros elementos da sequência; resultado $\bmod 10^9 + 7$.
   Dica: Utilize técnicas eficientes para calcular o LCM de subarrays, aplicando módulo em cada operação para evitar overflow.

7. **A – Internship Assignment**
   Plataforma: Codeforces
   Descrição: Dada uma matriz $N \times N$ representando as preferências de estagiários por disciplinas, calcular o número de formas distintas de atribuir as disciplinas aos estagiários, considerando as preferências; resultado $\bmod 10^9 + 7$.
   Dica: Utilize técnicas de contagem combinatória para calcular o número de formas possíveis de atribuição, aplicando módulo em cada cálculo.