Matemática para programación competitiva

¿Cómo representamos números muy grandes?

En c++, el tipo de dato int funciona con números de 32 bits, pero el primer bit se reserva para el signo, entonces este tipo de dato sólo puede representar enteros hasta $2^{31} - 1 = 2147483647$.

Si uno quisiera representar números más grandes, se puede utilizar el tipo de dato long long, que lo representa con 64 bits, osea el número representable más grande es $2^{63} - 1 = 9223372036854775807$.

Pero, ¿Y si quisieramos representar números aún más grandes?.

Aritmética modular

En varios problemas de matemática se pide calcular un número en un módulo específico. Esto es debido a que los estos valores crecen rápidamente. ¿Qué es un módulo?

Se dice que un número a es congruente a r módulo p, si es que a deja resto r al dividirse por p. O en lenguaje matemático, $a = r \bmod p \Leftrightarrow pk + r = a$.

En c++ podemos realizar esta operación utilizando el operador %. Y si queremos sumar, restar o multiplicar números en este formato:

```cpp
// (a+b)%mod
long long ans = ((a%mod)+(b%mod))%mod;
// (a-b)%mod
long long ans = ((a%mod)-(b%mod))%mod;
// (a*b)%mod
long long ans = ((a%mod)*(b%mod))%mod;
```

Exponenciación binaria

Podemos calcular $a^b \mod p$ de multiplicando a el resultado b veces, pero esto tomaría $O(b)$ operaciones. ¿Existe una forma más rápida de calcularlo?

La respuesta es sí, se puede realizar exponenciación binaria en $O(log(b))$, y el algoritmo consiste en lo siguiente:

$$a^b = \begin{cases} 1 & \text{if } b = 0 \\ \left(a^{b/2}\right)^2 & \text{if } b \text{ is even} \\ a \cdot \left(a^{b-1}\right) & \text{if } b \text{ is odd} \end{cases}$$

Exponenciación binaria

Aquí una implementación iterativa del código de exponenciación binaria:

```
using ll = long long;
ll bin_pow(ll a, ll b, ll mod){
    a %= mod;
    ll res = 1;
    while (b != 0){
        if (b%2 != 0)
            res = (res*a)%mod;
        a = (a*a)%mod;
        b /= 2ll;
    }
    return res;
}
```

Inverso modular

Hemos visto como hacer sumas, multiplicaciones, potencias y restas. Ahora, ¿cómo realizamos división?

El inverso modular de a se define como $a^{-1} \bmod p$, y se cumple que $a \cdot a^{-1} = 1 \bmod p$. Para calcular, este valor nos vamos a aprovechar del pequeño teorema de fermat, que dice, dado un número primo p se cumple que:

$$a^{p-1} = 1 \bmod p$$

Ahora si dividimos por a en ambos lados:

$$a^{p-2} = a^{-1} \bmod p$$

Que se puede calcular usando exponenciación binaria :).

Combinatoria básica

En combinatoria, los dos principios fundamentales para contar el número de formas de elegir o construir objetos son el **Principio de la Suma** y el **Principio de la Multiplicación**.

Definición

Si tenemos dos (o más) conjuntos disjuntos de opciones, y queremos contar todas las formas de elegir exactamente una opción de entre todos ellos, entonces el número total de formas es la **suma** de las cantidades de cada conjunto.

> **Formalmente:**
> Si un experimento puede realizarse de n_1 formas o, alternativamente, de n_2 formas (pero no ambas al mismo tiempo), entonces en total hay $n_1 + n_2$ formas de realizar el experimento.

Principio de la Suma

Ejemplo 1

Tienes 3 camisas rojas y 5 camisas azules. ¿Cuántas camisas puedes elegir si solo puedes escoger una?

- Opciones para camisas rojas: 3
- Opciones para camisas azules: 5
- **Total**: $3 + 5 = 8$ camisas.

Ejemplo 2

Dispones de dos rutas para ir al trabajo:

- Ruta A: 4 variaciones posibles (por calles distintas)
- Ruta B: 3 variaciones posibles
  En total puedes elegir tu camino de $4 + 3 = 7$ maneras distintas.

Principio de la Multiplicación

Definición

Si un proceso se descompone en una secuencia de etapas, donde la etapa 1 ofrece n_1 posibilidades y, para cada opción de la etapa 1, la etapa 2 ofrece n_2 posibilidades (y así sucesivamente), entonces el número total de resultados posibles es el **producto** de las cantidades de cada etapa.

> **Formalmente:**
>
> Si un experimento consta de dos subexperimentos independientes, el primero con n_1 resultados y, para cada uno de esos resultados, el segundo con n_2 resultados, entonces el número total de formas de realizar ambos experimentos en secuencia es $n_1 \times n_2$.

Principio de la Multiplicación

Ejemplo 1

Formar un código de 2 dígitos donde el primer dígito puede ser del 0 al 9 (10 opciones) y el segundo dígito también puede ser del 0 al 9 (10 opciones).

- Etapa 1 (primer dígito): 10 opciones
- Etapa 2 (segundo dígito): 10 opciones
- **Total**: $10 \times 10 = 100$ códigos posibles.

Ejemplo 2

Seleccionar un entrante y un plato principal de un menú:

- Entrantes: 4 opciones
- Platos principales: 6 opciones
- **Total**: $4 \times 6 = 24$ menús posibles.

Problema: contrucción de caminos en grafo

Enunciado: Tenemos que construir un grafo dirigido con a lo más 200 nodos, de forma que la cantidad de caminos desde el nodo 1 hasta el nodo n son exactamente k.

Permutaciones

Aplicando el **Principio de la Multiplicación** para ordenar n objetos distintos:

1. Para la primera posición hay n opciones.
2. Una vez elegida, para la segunda quedan $n-1$ opciones.
3. Así sucesivamente hasta la última posición, con 1 opción.

Por tanto,

$$n! \;=\; n \times (n-1) \times (n-2) \times \cdots \times 1.$$

Queremos **elegir** un subconjunto de k objetos de un total de n, sin importar el orden:

1. Primero, contamos las permutaciones de elegir un subconjunto de tamaño k:
$$P(n,k) \;=\; n \times (n-1) \times \cdots \times (n-k+1).$$
2. Luego, como el orden dentro del grupo de k no importa, dividimos por las $k!$ maneras de ordenar esos k:

$$\binom{n}{k} \;=\; \frac{P(n,k)}{k!} \;=\; \frac{n \times (n-1) \times \cdots \times (n-k+1)}{k!} \;=\; \frac{n!}{k!\,(n-k)!}.$$

CSES - Distributing Apples

Enunciado: Tenemos que contar de cuantas formas podemos repartir m manzanas entre n niños.

Descomposición en Factores Primos

La **Descomposición Prima** (o factorización prima) de un entero $n > 1$ consiste en expresar n como el producto de números primos, únicos salvo el orden:

$$n = p_1^{e_1} \times p_2^{e_2} \times \cdots \times p_k^{e_k},$$

donde cada p_i es un número primo y cada e_i es un entero positivo.

Ejemplo:

$$360 = 2^3 \times 3^2 \times 5^1.$$

Factorización por División

El método más sencillo es probar divisores crecientes hasta $\sqrt{n}$:

```
vector<ll> primeFactors(ll n) {
    vector<ll> factors;
    for (ll i = 2; (i*i) <= n; i++) {
        while (n%i == 0) {
            factors.push_back(i);
            n /= i;
        }
    }
    if (n > 1) factors.push_back(n);
    return factors;
}
```

Funciones de divisores

Para un entero positivo

$$n = p_1^{e_1} \times p_2^{e_2} \times \cdots \times p_k^{e_k},$$

donde los p_i son primos y los e_i exponentes:

- **Número de divisores** ($d(n)$ o $\tau(n)$):

$$d(n) = \prod_{i=1}^{k} (e_i + 1).$$

- **Suma de divisores** ($\sigma(n)$):

$$\sigma(n) = \sum_{d|n} d = \prod_{i=1}^{k} \frac{p_i^{e_i+1} - 1}{p_i - 1}.$$

Criba de eratóstenes

Si quisieramos obtener todos los primos hasta n con el método anterior, nos tomaría tiempo $O(n\sqrt{n})$. Para esto existe la criba de eratóstenes, que nos permite hacer lo mismo, pero en $O(nlog(log(n)))$.

```
struct eratosthenes_sieve {
    vector<ll> primes;
    vector<bool> isPrime;
    eratosthenes_sieve(ll n) {
        isPrime.resize(n + 1, true);
        isPrime[0] = isPrime[1] = false;
        for (ll i = 2; i <= n; i++) {
            if (isPrime[i]) {
                primes.push_back(i);
                for (ll j = i*i; j <= n; j += i)
                isPrime[j] = false;
            }
        }
    }
};
```

Gcd y lcm en C++

El **máximo común divisor** de dos enteros a y b es el número más grande que divide a ambos sin dejar resto.

```
int a = 48, b = 18;
cout << "gcd(" << a << ", " << b << ") = " << __gcd(a,b) << "\n";
```

El **mínimo común múltiplo** de dos enteros a y b es el número más pequeño que es múltiplo de ambos.

```
int a = 48, b = 18;
cout << "lcm(" << a << ", " << b << ") = " << lcm(a,b) << "\n";
```