Combinatoria

Brian Morris Esquivel

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

Training Camp Argentina Virtual 2020

Guía de la clase

- **Problemas Clásicos**
  - Permutaciones
  - K-Permutaciones
  - Combinaciones
  - Permutaciones con elementos iguales
  - Distribuciones de pelotas en cajas
- **Implementaciones**
  - Álgebra modular
  - Factoriales
  - Coeficientes binomiales
- **Principio de adición**
- **Principio de Inclusión-Exclusión**
  - Álgebra de conjuntos
  - Principio de Inclusión-Exclusión
  - Ejemplos
- **Números de Catalán**
- **Números de Stirling y Bell**
- **Recurrencias lineales**
  - Recurrencias Lineales
  - Exponenciación de Matrices
  - Yapa

Problemas Clásicos

Introducción

Lo primero que vamos a ver en esta clase son algunos problemas clásicos, muy populares, en los que se pide contar posibilidades.

Los primeros tres son los más difundidos y tienen su propio nombre: *permutaciones*, *k-permutaciones* y *combinaciones.*

Los siguientes dos son problemas especiales que se resuelven a partir de los anteriores:

- permutaciones cuando hay elementos iguales
- cantidad de formas de distribuir n de pelotas iguales en k cajas distintas.

Permutaciones – Problema ejemplo

Juancito se compró n pares de medias de distintos colores y los va a guardar en un cajón. Va a guardar los pares de medias uno a lado del otro de izquierda a derecha en el cajón, y de esa forma puede verse un patrón de colores al abrir el cajón.

¿Cuántos patrones distintos puede armar Juancito?

Si Juancito tiene 3 pares de medias, el total de patrones distintos que se pueden ver es 6.

Éste problema es equivalente al siguiente:

"Si tenemos n pelotitas, todas distintas, ¿Cuántas formas hay de ordenarlas?"

Solución al problema

La solución a este problema se conoce como *"cantidad de permutaciones de n elementos"* y se obtiene con la siguiente fórmula:

$$\boldsymbol{Permutaciones(n) = n!}$$

¿Por qué es esa la respuesta?

Hay n formas de elegir el primer elemento, para cada una de esas hay $n - 1$ formas de elegir el segundo elemento, y así sucesivamente…

La respuesta es: $n \cdot (n-1) \cdot \cdots \cdot 1 = n!$

k-Permutaciones – Problema ejemplo

María es dueña de un supermercado que está en crecimiento.
En el supermercado trabajan n personas y María decidió ascender a 3 de ellas a nuevos puestos de trabajo, todos distintos.

¿De cuántas formas distintas puede asignar María los nuevos puestos de trabajo?

Si María tiene 5 empleados, puede asignar los puestos de trabajo de 60 formas.

Éste problema es equivalente al siguiente:

"Si tenemos n pelotitas distintas,
¿Cuántas formas hay de elegir 3 de ellas, en un orden específico?"

Solución al problema

La solución a este problema se conoce como *"cantidad de 3-permutaciones de n elementos"* y se obtiene con la siguiente fórmula:

$$\boldsymbol{3 - Permutaciones(n) = \frac{n!}{(n-3)!}}$$

Y se puede generalizar para el caso en el que quiero elegir una cantidad k de elementos distinta de 3.

Calculamos la "*cantidad de k-permutaciones de n elementos*" con la siguiente fórmula:

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

Solución al problema (cont.)

¿Por qué es esa la respuesta?

Hay n formas de elegir el primer elemento, para cada una de esas, hay $n-1$ formas de elegir el segundo elemento, y así sucesivamente, hasta haber elegido a los k elementos.

La respuesta es: $n \cdot (n-1) \cdot \cdots \cdot \big(n-(k-1)\big)$

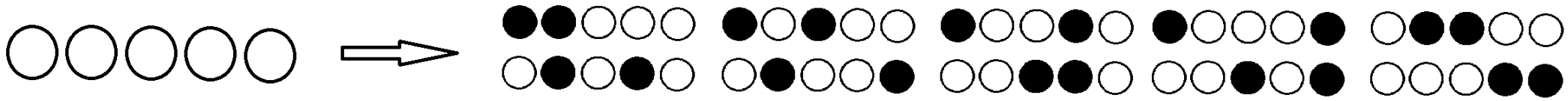
$$\frac{n \cdot (n-1) \cdot \cdots \cdot \big(n-(k-1)\big) \cdot (n-k) \cdot \cdots \cdot 1}{(n-k) \cdot \cdots \cdot 1} = \frac{n!}{(n-k)!}$$

Combinaciones – Problema ejemplo

Rogelio es un mal perdedor, nunca juega sin estar seguro de que va a ganar.
Decidió jugar a la n-k-iniela, una lotería en la que para ganar debe acertar, en algún orden, k números sorteados entre 1 y n.

¿Cuántas apuestas debe hacer Rogelio para asegurarse de ganar?

Si fuera que $n = 5$ y $k = 2$, existen exactamente 10 posibles resultados del sorteo.



Éste problema es equivalente al siguiente:

"Si tenemos n pelotitas, todas distintas,
¿Cuántas formas hay de elegir k de ellas, sin importar el orden?"

Solución al problema

La solución a este problema se conoce como *"cantidad de combinaciones de n elementos en k"* y se obtiene con la siguiente fórmula:

$$\boldsymbol{Combinaciones(n,k) = nCk = \binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}}$$

¿Por qué es esa la respuesta?

Es la cantidad de formas de elegir k elementos en orden (k-permutaciones de n) dividido la cantidad de formas de ordenar esos k elementos.

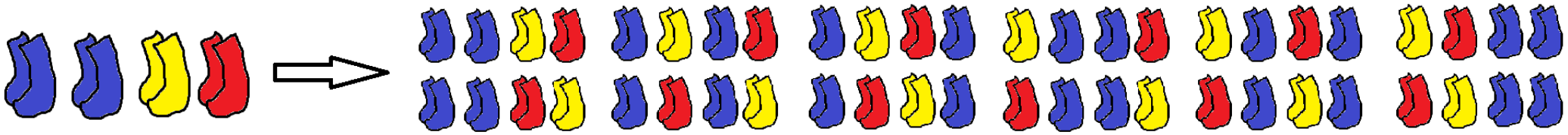
Por eso la respuesta es: $\frac{k\text{-}Permutaciones(n)}{Permutaciones(n)} = \frac{n!}{(n-k)! \cdot k!}$

Permutaciones con elementos iguales – Problema ejemplo

Juancito se compró n pares de medias y los va a guardar en un cajón, los va a guardar de izquierda a derecha y de esa forma puede verse un patrón de colores al abrir el cajón. Sabemos cuantas medias compró de cada color.

¿Cuántos patrones distintos puede armar Juancito?

Si tuviera 2 pares de medias azules, 1 par amarillas y 1 par rojas, la respuesta sería 12.



Éste problema es equivalente al siguiente:

"Si tenemos n pelotitas, entre las cuales pueden haber algunas iguales, ¿Cuántas formas hay de ordenarlas?"

Solución al problema

Este problema se resuelve partiendo de calcular las permutaciones y eliminando casos repetidos.

Sean $\{a_1, a_2, \ldots, a_k\}$ las cantidades de colores, la respuesta es:

$$Solución = \frac{(a_1 + a_2 + \cdots + a_k)!}{a_1! \cdot a_2! \cdot \cdots \cdot a_k!} = \frac{n!}{a_1! \cdot a_2! \cdot \cdots \cdot a_k!}$$

¿Por qué es esa la respuesta?

Si los pares de medias fueran todos distintos hay un total de n! casos.

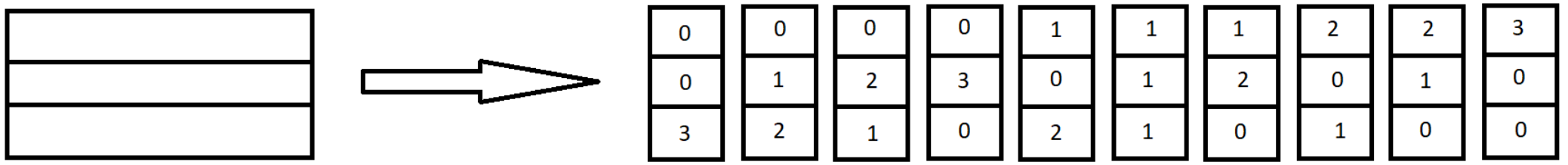
Para cada color i que se repite, los casos se repiten $a_i!$ veces.

Pelotas iguales en cajas distintas – Problema ejemplo

A Micaela le gusta mucho leer, acaba de comprar su primer biblioteca, que tiene k estantes, y en ella piensa acomodar todos sus n libros dentro de la misma.
No importa el orden de los libros dentro de un mismo estante.

¿De cuántas formas puede hacerlo?

Si tuviera 3 estanterías y 3 libros, la respuesta sería 10.



Éste problema es equivalente al siguiente:

"Si tenemos n pelotitas, todas iguales,
¿Cuántas formas hay de distribuirlas en k cajas distintas?"

Solución al problema

La respuesta a este problema es:

$$Solución = \frac{(n+k-1)!}{n! \cdot (k-1)!} = \binom{n+k-1}{n} = \binom{n+k-1}{k-1}$$

¿Por qué es esa la respuesta?

Pensamos a las cajas alineadas de izquierda a derecha.

Una forma de distribuir las n pelotas en k cajas es igual a una forma de poner $k-1$ separadores entre las n pelotas.

Debemos hallar la cantidad de formas de ordenar $n+k-1$ objetos, entre los cuales hay n iguales (pelotas) y otros $k-1$ iguales (cajas).

Implementación

Introducción

No es suficiente entender las ecuaciones matemáticas para resolver problemas, también necesitamos saber implementarlos de forma eficiente.

En las soluciones que acabamos de ver encontramos factoriales, los cuales no suelen poder computarse velozmente.

Las respuestas a problemas de conteo suelen tener valores gigantescos.
Se suele pedir que se de la respuesta "módulo cierto primo p". Se necesita saber hacer cálculos en álgebra modular para estos casos.

Aritmética Modular

Para hacer los cálculos es necesario tener claros algunos conceptos básicos de aritmética modular.

No vamos a desarrollar ese tema en esta clase, pero sí voy a enfatizar algunas funciones que son clave para la implementación de soluciones.

Lo primero y principal que tiene que quedar claro es la siguiente relación:

$$\frac{P}{Q} = S \in Z \quad \Rightarrow \quad S(\text{mod } M) = P \cdot Q^{-1}(\text{mod } M)$$

Exponenciación Binaria Modular

Con el método de exponenciación binaria podés calcular potencias enteras de un número rápidamente.

El chiste del método es aprovechar el cálculo de $a^k \pmod M$ para calcular $a^{2k} \pmod M$ utilizando una única operación.

```
const int M = 1e9+7;
typedef long long ll;
ll expmod(ll b, ll e){
    ll ans = 1;
    while(e){
        if(e&1) ans = ans*b %M;
        b = b*b %M; e >>= 1;
    }
    return ans;
}
```

Inverso Modular

Hay muchas formas rápidas (complejidad logarítmica) de calcular el inverso modular de un entero a módulo otro entero M.

Algunos métodos involucran teoremas como: Pequeño Teorema de Fermat, Teorema de Fermat-Euler, Método de Euclides Extendido, etc.

El Pequeño Teorema de Fermat dice que sea p primo y a coprimo a p:

$$a^{p-2} \equiv a^{-1} \pmod{p}$$

```
ll invmod(ll a){ return expmod(a, M-2); }
```

Inverso Modular (cont.)

Además del cálculo del inverso modular en tiempo logarítmico existe otra práctica interesante.

Cuando el módulo M es un número primo se satisface la siguiente fórmula:

$$a^{-1} (\mathrm{mod}\ M) = -\left\lfloor \frac{M}{a} \right\rfloor \cdot \left(M (\mathrm{mod}\ a) \right)^{-1} (\mathrm{mod}\ M)$$

Podemos precalcular los inversos modulares de M para valores de $n \leq 10^6$.

```
typedef long long ll;
const int MAXN = 1e7, M = 1e9+7;
int INV[MAXN];
// ...
INV[1] = 1;
for(ll a = 2; a < MAXN; a++) INV[a] = M - (ll)(M/a)*INV[M%a]%M;
```

Cómputo de factoriales

Calcular factoriales es una tarea que no se puede hacer rápido.

Los problemas que requieren el cálculo de factoriales $n!$ suelen tener cotas en el orden de 10^6 para la variable n.

Lo que se hace para calcular factoriales cuando $n \leq 10^6$ es **precalcular** los factoriales y guardarlos en un arreglo.

```
ll F[MAXN];
// ...
F[0] = 1;
for(ll i = 1; i < MAXN; i++) F[i] = F[i-1]*i %M;
```

Cómputo de factoriales inversos

Le llamamos "factorial inverso" de n módulo M, al inverso modular del *factorial de* n módulo M.

Se pueden calcular los factoriales inversos usando la función *invmod*.

Pero cuando $n \leq 10^6$, es más eficiente tener en cuenta que:

$$(n!)^{-1} \pmod M = (1 \cdot 2 \cdot \cdots \cdot n)^{-1} \pmod M = 1^{-1} \cdot 2^{-1} \cdot \cdots \cdot n^{-1} \pmod M$$

Tras lo cual podemos precalcular los factoriales inversos en tiempo lineal.

```
ll F[MAXN], INV[MAXN], FI[MAXN];
// ...
F[0] = 1; forr(i, 1, MAXN) F[i] = F[i-1]*i %M;
INV[1] = 1; forr(i, 2, MAXN) INV[a] = M - (ll)(M/a)*INV[M%a]%M;
FI[0] = 1; forr(i, 1, MAXN) FI[i] = FI[i-1]*INV[i] %M;
```

Factorial para números grandes y módulos primos pequeños

En los casos que analizaremos $n \leq 10^{18}$ y $M \leq 10^7$, M es primo.

Es sencillo ver que si $n \geq M$, entonces $n! \pmod M = 0$

Pero quizás quieras calcular $S = \frac{A!}{B!} \pmod M$, para valores grandes de A y B.

Sean $n! = s \cdot M^k, d = \left\lfloor \frac{n}{M} \right\rfloor$, calculamos los valores de k y de $s \pmod M$.

$$n! \pmod M = (1 \cdot 2 \cdot \cdots \cdot M) \cdot \big((M+1) \cdot (M+2) \cdot \cdots \cdot 2M\big) \ldots \cdot n \pmod M$$

$$n! \pmod M = \big((M-1)!\big)^d \cdot \big(M^d \cdot d!\big) \cdot (n \bmod M)!$$

$$n! \pmod M = (n \bmod M)! \cdot (-1)^d \cdot M^d \cdot d!$$

Teorema de Wilson

Implementación

Implementamos una estructura para almacenar los factoriales calculados módulo M.

```
ll F[MAXN];
struct Fact{
    ll s, k;
    Fact(ll n){
        if(n < M){ s = F[n]; k = 0; return; }
        ll d = n / M;
        *this = Fact(d);
        s = s*F[n%M] %M  * (d%2 ? M-1 : 1) %M;
        k += d;
    }
};
```

Factoriales – Resumen

Según las cotas que especifique el problema, en términos generales, conviene utilizar un método u otro para calcular factoriales y factoriales inversos.

Cuando $n \leq 10^7$:	Tablas precalculadas.
Cuando $n \leq 10^{18}, M \leq 10^7$:	Tabla precalculada + método con Wilson

Combinaciones – A partir de factoriales

Cuando ya sabés precomputar factoriales y factoriales inversos es sencillo calcular coeficientes binomiales para valores de n y k menores que 10^7.

Por comodidad y prolijidad se suele definir una función que la calcule.

```
ll Comb(ll n, ll k){
    if(n < k) return 0;
    return F[n]*FI[k] %M *FI[n-k] %M;
}
```

Combinaciones - Precálculo

Una propiedad muy útil de coeficientes binomiales es la siguiente:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

(Nótese que los coeficientes binomiales conforman un triángulo de Pascal)

Y por definición: $\binom{n}{0} = \binom{n}{n} = \frac{n!}{0! \cdot n!} = 1$

Precalculamos una tabla de coeficientes binomiales cuando $n \cdot k \leq 10^7$.

```
ll C[MAXN][MAXK];
// ...
forn(i, MAXN){
     C[i][0] = 1; if(i < MAXK) C[i][i] = 1;
    forr(j, 1, min(i, MAXK))
        C[i][j] = (C[i-1][j-1] + C[i-1][j])%M;
}
```

Combinaciones – Precálculo (cont.)

Aunque los valores de n y k no tengan cotas individuales adecuadas, en algunas ocasiones sucede que el valor de $n \cdot k$ está acotado.

Si existe una cota $n \cdot k \leq 10^7$ también podés crear la tabla de coeficientes binomiales para cada entrada n, k.

```
vector<vector<ll>> C;
// ...
scanf("%d %d", &N, &K); C.resize(N, vector<ll>(K));
forn(i, N){
    C[i][0] = 1; if(i < K) C[i][i] = 1;
    forr(j, 1, min(i, K))
        C[i][j] = (C[i-1][j-1] + C[i-1][j])%M;
}
```

Combinaciones – Lucas

Cuando los valores de n y k pueden ser muy grandes ($\leq 10^{18}$), pero el valor M en el que modulamos es un primo pequeño (≤ 3000), existe un teorema que nos permite calcular las combinaciones.

El Teorema de Lucas dice que, sean:

$$n = \sum_{i=0}^{k} n_i p^i \,, \qquad m = \sum_{i=0}^{k} m_i p^i$$

Se verifica:

$$\binom{n}{m} \pmod p = \prod_{i=0}^{k} \binom{n_i}{m_i} \pmod p$$

Combinaciones – Lucas (cont.)

Sabiendo eso hacemos la siguiente implementación.

```
const int M = 3005;
int C[M][M];
// ...
ll lucas(ll n, ll k, int p){
    ll ans = 1;
    while(n + k){
        ans = (ans * C[n%M][k%M]) % M;
        n /= M; k /= M;
    }
    return ans;
}
```

Combinaciones – Resumen

Según las cotas que especifique el problema, en términos generales, conviene utilizar un método u otro para calcular coeficientes binomiales.

Cuando $MAXN \cdot MAXK \leq 10^7$:	Tabla precalculada.
Cuando $n \cdot k \leq 10^7$:	Tabla calculada en cada input.
Cuando $n, k \leq 10^7$:	A partir de factoriales.
Cuando $n, k \leq 10^{18}, M \leq 3000$:	Algoritmo de Lucas.

Principio de adición

En general no van a aparecer problemas cuya solución sea aplicar una fórmula de las conocidas.

Una estrategia clave en muchos problemas es hallar la solución en función de cosas que sabemos calcular.

Veamos un par de ejemplos.

Problema 1

https://codeforces.com/contest/584/problem/B

En este problema tenemos $3n$ duendes en una ronda y tenemos que asignarle un número entre 0 y 2 a cada uno.

Queremos que haya al menos una terna $\{k, k+n, k+2n\}$ tal que los números de esos duendes no sumen un múltiplo de 6.

¿Cuántas formas hay?

Solución 1

La idea es calcular todas las posibles combinaciones de duendes.

Y restamos todas las combinaciones en las cuales todas las ternas suman múltiplo de 6.

La solución resulta:

$$S = 3^{3n} - 7^n$$

Código: https://codeforces.com/contest/584/submission/41540323

Problema 2

https://codeforces.com/contest/420/problem/C

En este problema tenemos un grafo general de n nodos y tenemos que elegir dos nodos que cubran al menos p aristas. $p \leq n \leq 3 \cdot 10^5$.

¿Cuántas formas hay de elegir un par de nodos?

INTERLUDIO

Tomémonos 5’

Solución 2

Primero calculamos la cantidad de pares de nodos cuya suma de grados sea al menos p.

Luego descontamos la cantidad de pares de nodos vecinos cuya suma de grados sea al menos p, pero no sea un par válido.

Código: https://codeforces.com/contest/420/submission/86750082

Principio de Inclusión-Exclusión

Introducción

En el curso de Viviana hay N chicas que practican deportes, todas ellas toman clases de fútbol o vóley.
Sabemos que F de las chicas toman clases de fútbol y V de las chicas toman clases de vóley.

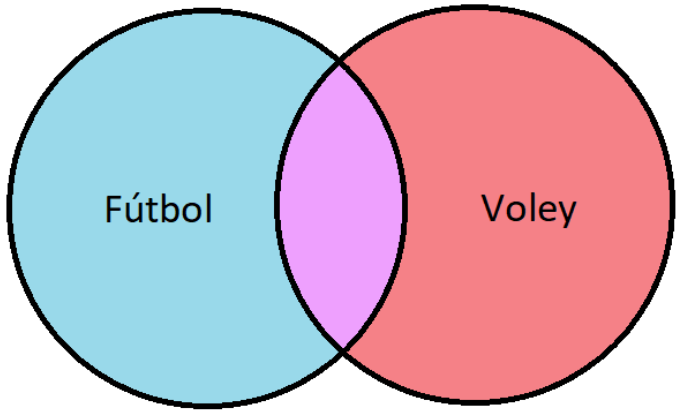
¿Cuántas de ellas toman clases de ambos deportes?

En ocasiones se nos presentarán 2 (o más) propiedades y tendremos que calcular la cantidad de elementos que cumplen ambas, o al menos una de ellas, en estos casos aplicamos lo que se conoce como “teoría de conjuntos”.

Dentro de la teoría de conjuntos se encuentran propiedades y ecuaciones que nos facilitan el cálculo de tamaños de conjuntos.

Diagrama de Venn

Para modelar el problema anterior podemos hacer un gráfico que lleva el nombre de Diagrama de Venn.



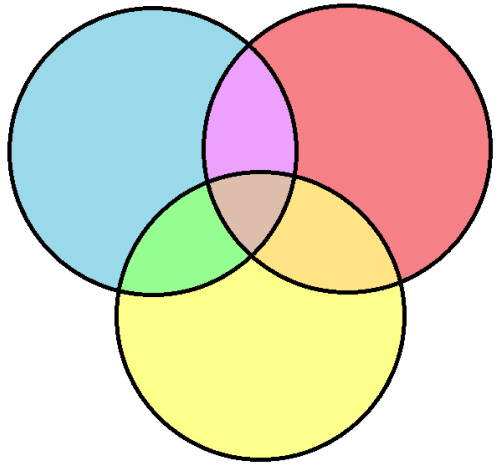
En este diagrama podemos visualizar la siguiente ecuación:

$$F + V - CantFV = N$$

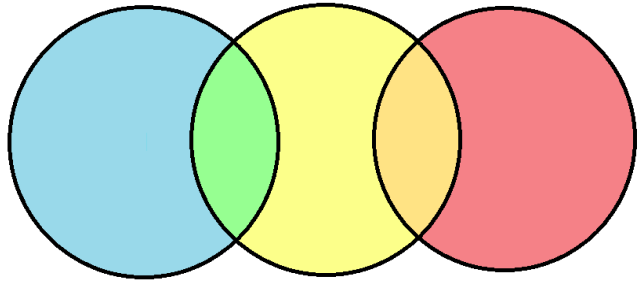
Diagrama de Venn (cont.)

Otros diagramas de Venn populares:

Tres conjuntos



Tres conjuntos, dos de ellos disjuntos



En ambos casos, en los que se presentan tres conjuntos, se verifica lo siguiente:

$$|A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C| = |A \cup B \cup C| = N$$

Propiedades en el álgebra de conjuntos

Estas ecuaciones pertenecen a un área de la *Teoría de Conjuntos* que llamamos "*Álgebra de Conjuntos*".

Hay muchas ecuaciones y propiedades que relacionan tamaños de conjuntos, algunas útiles son:

Propiedad distributiva: $$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

Leyes de DeMorgan: $$(A \cup B)' = A' \cap B', \quad (A \cap B)' = A' \cup B'$$

Aunque la más útil es la que ya exploramos y se puede generalizar para más conjuntos.

Generalización

Es sencillo ver en el diagrama de Venn de dos conjuntos que se verificaba:

$$|A| + |B| - |A \cap B| = |A \cup B|$$

Es menos sencillo ver en el diagrama de tres conjuntos que se verificaba:

$$|A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C| = |A \cup B \cup C|$$

Pero se puede demostrar que para cualquier cantidad de conjuntos se verifica:

$$\sum_{S \subseteq \{1,2,\ldots,n\}} \left((-1)^{|S|+1} \cdot \left| \bigcap_{i \in S} A_i \right| \right) = \left| \bigcup_{i=1}^{n} A_i \right|$$

Esta propiedad lleva el nombre de **principio de inclusión-exclusión**.

Principio de Inclusión-Exclusión

Este principio se hace muy útil cuando contamos la cantidad de elementos que cumplen al menos una propiedad dentro de un conjunto de propiedades.

Suele ser sencillo hallar una expresión matemática o algoritmo sencillo para contar la cantidad de objetos que cumplen simultáneamente un conjunto de propiedades.

En estos casos el principio de inclusión-exclusión nos permite hallar la respuesta.

Problema ejemplo

En cierta galería existe un pasillo estrecho con una única hilera de N baldosas. Fernanda y Bianca tenían que pintar las baldosas, Fernanda pintó las baldosas en posiciones múltiplos de 2 y Bianca las baldosas en posiciones múltiplos de 3.

Al final del trabajo ¿Cuántas baldosas fueron pintadas en total?

No hallamos una ecuación inmediata que resuelva el problema.
Pero si podemos ver ecuaciones inmediatas para contar:

- Cuantas baldosas pintó Fernanda, que son $\frac{N}{2}$.
- Cuantas baldosas pintó Bianca, que son $\frac{N}{3}$.
- Cuantas baldosas pintaron ambas, que son $\frac{N}{6}$.

El total de baldosas pintadas resulta $S = \frac{N}{2} + \frac{N}{3} - \frac{N}{6}$

Problema ejemplo 2

Hay que pintar otro pasillo de la galería, de tamaño L, pero necesitamos que esta vez sea más rápido el trabajo, así que lo va a hacer un equipo de n personas, a cada una se la asignará un número a_i y deberá pintar los múltiplos de ese número.

Al final del trabajo ¿Cuántas baldosas fueron pintadas en total?

Vemos que para un conjunto de personas, es sencillo calcular cuántas baldosas fueron pintadas por todas esas personas.

Al igual que en el caso anterior podemos aplicar el principio de inclusión-exclusión.

Aunque no parezca la implementación de esto es linda.

Implementación

En este tipo de problemas, una solución inclusión-exclusión requiere hacer una cuenta para todos los posibles subconjuntos de personas.

Por eso para hallar la solución iteramos todos los posibles conjuntos.

```
ll ans = 0;
forr(bitmask, 1, (1<<n)){
    bool resta = __builtin_popcount(bitmask)%2;
    ans = (ans + (resta ? 1 : M-1)*cuenta(bitmask) %M) %M;
}
```

Para terminar de resolver el problema hay que implementar la función ***cuenta***.

Números de Catalán

Definición

Una sucesión popular en problemas de combinatoria es la sucesión de Catalán.

Esta sucesión queda definida para todo n con una expresión matemática:

$$Catalán(n) = \frac{1}{n+1} \cdot \binom{2n}{n} = \frac{2n!}{(n+1)! \cdot n!}$$

Y también puede ser definida recursivamente:

$$Catalán(0) = Catalán(1) = 1$$

$$Catalán(n) = \sum_{k=0}^{n-1} Catalán(k) \cdot Catalán(n-1-k)$$

Implementación

Al igual que con las combinaciones tenemos dos opciones para calcular un número de Catalán.

Podemos calcularlo a partir de tener precalculados factoriales y factoriales inversos, o podemos precalcular una tabla a partir de la definición recursiva.

Ambos casos sirven solo para valores de n menores que 10^7 aproximadamente.

Con la expresión:

```
ll F[MAXN], INV[MAXN], FI[MAXN];
ll Cat(int n){
    return F[2*n] *FI[n+1]%M *FI[n]%M;
}
```

Con la recursión:

```
ll CAT[MAXN];
// ...
CAT[0] = CAT[1] = 1;
forr(i, 2, MAXN){
    CAT[i] = 0;
    forn(k, i)
        CAT[i]=(CAT[i]+CAT[k]*CAT[i-1-k]%M)%M;
}
```

Problemas que resuelve

Los números de catalán son solución de varios problemas clásicos, no los vamos a nombrar todos, pero sí algunos que destacan:

- **Cantidad de formas de ubicar correctamente n pares de paréntesis.**

Es muy común encontrarse con problemas equivalentes a este.

- **Número de árboles binarios con n nodos.**
- **Número de árboles binarios completos con $n + 1$ hojas.**
- **Cantidad de triangulaciones de un polígono convexo de $n + 2$ lados.**

Pueden encontrar una lista más larga en el siguiente artículo:
https://cp-algorithms.com/combinatorics/catalan-numbers.html

Problema ejemplo

Mauro está parado en la esquina superior izquierda de un tablero de $n \times n$ y quiere llegar a la esquina inferior derecha moviéndose hacia abajo y hacia la derecha.
En ningún momento puede cruzar la diagonal del tablero.

¿Cuántas caminos distintos puede tomar?

El chiste es que Mauro tiene que dar $n - 1$ pasos hacia abajo y $n - 1$ pasos hacia la derecha, en algún orden.
Si comienza hacia la derecha en ningún momento puede haber hacho más pasos hacia abajo que hacia la derecha.

El problema se reduce a contar las formas de ubicar correctamente $n - 1$ pares de paréntesis.

La respuesta es dos veces "catalán de $n - 1$": $2 \cdot Catalán(n-1)$

Números de Stirling y Bell

Introducción

Un problema popular en combinatoria es el siguiente:

"¿Cuántas formas distintas hay de particionar un conjunto de n elementos?"

No es un problema tan popular como los que resuelve Catalán, pero aparece y es útil saber resolverlo.

La solución a este problema no tiene una ecuación directa conocida.

En esas soluciones introducimos dos nuevas secuencias que son los números de Stirling y de Bell.

Los números de Stirling

Vamos a resolver el problema anterior introduciendo un subproblema:

"¿Cuántas formas hay de partir un conjunto de n elementos en k conjuntos?"

La solución de este problema está definida por los **números de Stirling**, los cuales toman dos parámetros n y k, éstos se pueden hallar recursivamente.

Inicialmente podemos ver que:

$$\forall n \geq 1, Stirling(n, 1) = 1, \qquad \forall k > 1, Stirling(1, k) = 0$$

Y luego $\forall n, k > 1$ se puede hallar la siguiente expresión:

$$Stirling(n, k) = k * Stirling(n - 1, k) + Stirling(n - 1, k - 1)$$

Implementación

Al igual que las soluciones recursivas que vimos hasta ahora, la forma linda de implementarlo en tu solución es precalculando una tabla con los valores.

```
ll Stirling[MAXN][MAXN];
// ...
forr(i, 1, MAXN)Stirling[i][1] = 1;
forr(i, 2, MAXN)Stirling[1][i] = 0;
forr(i, 2, MAXN)forr(j, 2, MAXN){
    Stirling[i][j] =
        (Stirling[i-1][j-1] + j*Stirling[i-1][j]%MOD) %MOD;
}
```

Números de Bell

Tras haber hallado los números de Stirling es tarea sencilla resolver el problema planteado inicialmente.

La solución a ese problema esta definida por la sucesión conocida como los **números de Bell** y se calcula con un único parámetro que es el tamaño n del conjunto.

Por como se definen cada problema, vemos que se verifica la siguiente ecuación:

$$Bell(n) = \sum_{k=1}^{n} Stirling(n, k)$$

Implementación

Como la solución se obtiene a partir de los números de Stirling, en la implementación primero calculamos los números de Stirling.

Al igual que antes, la forma linda de implementar es precalculando una tabla.

```
ll Stirling[MAXN][MAXN], Bell[MAXN];
// ...
forn(i, MAXN){
    Bell[i] = 0;
    forn(j, MAXN)
        Bell[i] = (Bell[i] + Stirling[i][j]) %MOD;
}
```

Recurrencias Lineales

Introducción

Existen muchas sucesiones en las cuales cada elemento se define en función de elementos anteriores.

En muchas de ellas cada elemento se calcula como una combinación lineal de una cantidad finita de elementos anteriores.

Por ejemplo:

- Sucesión de potencias de 2: $\{1, 2, 4, 8, 16, 32, \dots\}$
- Sucesión de Fibonacci: $\{1, 1, 2, 3, 5, 8, \dots\}$

Estas sucesiones reciben el nombre de *"recurrencias lineales"*.

Sucesión de Fibonacci

La sucesión de Fibonacci es la recurrencia lineal más popular.

$$Fibo(0) = Fibo(1) = 1, \qquad Fibo(n) = Fibo(n-1) + Fibo(n-2)$$

Es común encontrar problemas cuya solución es simplemente “el n-ésimo número de Fibonacci”.

En un estacionamiento hay exactamente n lugares para estacionar.

Puede estacionar una moto y ocupar exactamente 1 lugar, o estacionar un auto y ocupar exactamente 2 lugares consecutivos.

¿De cuántas formas se puede llenar el estacionamiento?

Problemas parecidos a Fibonacci

Si en lugar de tener motos y autos, tuviéramos camionetas grandes que ocupan 3 lugares y colectivos que ocupan 5 lugares, tendríamos otra recurrencia:

$$Rec(1) = Rec(2) = Rec(4) = 0, \qquad Rec(0) = 1, \qquad Rec(3) = 1$$

$$n \geq 5: \;\; Rec(n) = Rec(n-3) + Rec(n-5)$$

¿Y si pudiera ubicar los colectivos en dos posiciones diferentes?

$$n \geq 5: \;\; Rec(n) = Rec(n-3) + 2 \cdot Rec(n-5)$$

Implementación sencilla

La parte difícil del problema es hallar la recurrencia.

En muchos casos el valor máximo de n está acotado en 10^6 o menos, y suele ser suficiente implementar un precálculo donde llenamos una tabla.

```
ll Garage[MAXN];
// ...
Garage[0] = Garage[3] = 1;
forr(i, 5, MAXN){
    Garage[i] = (Garage[i-3] + 2*Garage[i-5]) %MOD;
}
```

Casos en los que el número es grande

En ocasiones vamos a tener una cota en el orden de $n \leq 10^{18}$.

Hay dos valores importantes en las recurrencias lineales, uno es el valor n para el cual queremos calcular y otro es un valor k, que indica cuantas condiciones iniciales necesitamos.

En el caso de Fibonacci tenemos $k = 2$, y en el problema de camiones y colectivos tenemos $k = 5$.

El algoritmo anterior tiene complejidad $O(n \cdot k)$.

Vectores de la sucesión

Definimos un "*vector inicial*" con los primeros k valores de la sucesión.

Sea la sucesión a_i, con $k = 3$, definimos el vector inicial V_0

$$V_0 = \{a_0, a_1, a_2\}$$

A partir del vector inicial podemos calcular el "vector siguiente"

$$V_1 = \{a_1, a_2, a_3\}$$

El proceso de "calcular el vector siguiente" lo podemos repetir suficientes veces para calcular cualquier valor de la sucesión.

Matriz de Transición

Para calcular "el vector siguiente" definimos la matriz de transición T.

Esta matriz verifica: $V_k \cdot T = V_{k+1}$

Para el problema de camiones y colectivos definimos los siguientes vector inicial y matriz de transición:

$$V_0 = \{1, 0, 0, 1, 0\}, \qquad T = \begin{bmatrix} 0 & 0 & 0 & 0 & 2 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Recurrencia

Exponenciación de Matrices

A partir del *vector inicial* y la *matriz de transición* podés calcular cualquier valor de la sucesión, multiplicando muchas veces la matriz.

Si quiero calcular el n-ésimo elemento de la sucesión, calculo:

$$V_n = V_0 \cdot T^n \quad \rightarrow \quad V_n = \{a_n, a_{n+1}, a_{n+2}, \dots, a_{n+k-1}\}$$

La multiplicación de dos matrices requiere k^3 operaciones.

Aplicando exponenciación binaria se consigue una complejidad de $O(k^3 \cdot \log n)$

Esta solución funciona para valores $k \leq 200$ aproximadamente.

Implementación

Es útil tener una estructura con la cual representar matrices.

```
struct Mat {
  vector<vector<ll>> vec;
  Mat(): vec(1, vector<ll>(1, 0)) {}
  Mat(int n): vec(n, vector<ll>(n) ) {}
  Mat(int n, int m): vec(n, vector<ll>(m) ) {}
  vector<ll> &operator[](int f){ return vec[f]; }
  const vector<ll> &operator[](int f) const { return vec[f]; }
  int size() const { return vec.size(); }
};
```

Implementación (cont.)

Es también útil definir las operaciones entre matrices.
La implementación importante es la multiplicación.

```
Mat operator *(Mat A, Mat B) {
  int n = A.size(), m = A[0].size(), t = B[0].size();
  Mat ans(n, t);
  forn(i, n) forn(j, t) forn(k, m)
    ans[i][j] = (ans[i][j] + A[i][k] * B[k][j] % MOD) % MOD;
  return ans;
}
```

Implementación (cont.)

Y por último necesitás una exponenciación binaria con matrices.

```
Mat expmat(Mat A, ll e){
    int n = A.size();
    Mat Ans(n); forn(i, n) Ans[i][i] = 1;
    while(e){
        if(e&1) Ans = Ans*A;
        A = A*A; e >>= 1;
    }
    return Ans;
}
```

Ejemplo 1

Con el código anterior, podemos implementar una función que calcule rápido el n-ésimo número de Fibonacci.

```
ll Fibo(ll n) {
    Mat V0(1, 2), T(2);
    V0[0] = {1, 1};
    T[0] = {0, 1}; T[1] = {1, 1};
    Mat V = V0*expmat(T, n);
    return V[0][0];
}
```

Ejemplo 2

Para el problema de las camionetas y colectivos podemos escribir la siguiente función:

```
ll solve(ll n) {
    Mat V0(1, 5), T(5);
    V0[0] = {1, 0, 0, 1, 0}; // Vector inicial
    T[0] = {0, 0, 0, 0, 2};
    T[1] = {1, 0, 0, 0, 0};
    T[2] = {0, 1, 0, 0, 1};
    T[3] = {0, 0, 1, 0, 0};
    T[4] = {0, 0, 0, 1, 0};
    Mat V = V0*expmat(T, n);
    return V[0][0];
}
```

Hay un algoritmo más...

Una vez en mi vida me topé con el siguiente algoritmo, el cual no sé explicar por qué funciona ni qué hace.

Pero parece que calcula valores en recurrencias lineales rápido.

El código

```
struct LRec{
  int n;
  vector<int> In, T;
  vector<vector<int>> B;
  vector<int> add(vector<int> &a, vector<int> &b){
    vector<int> ans(2*n+1, 0);
    forn(i, n+1)forn(j, n+1)
      ans[i+j] = (ans[i+j] + (ll)a[i]*b[j]%MOD + MOD)%MOD;
    for(int i = 2*n; i > n; i--)forn(j, n)
      ans[i-1-j] = (ans[i-1-j] + (ll)ans[i]*T[j]%MOD + MOD)%MOD;
    ans.resize(n+1); return ans;
  }
...
```

```
  LRec(vector<int> V, vector<int> T): In(V), T(T){
    n = sz(V);
    vector<int> a(n+1, 0);
    a[1] = 1; B.pb(a);
    forr(i, 1, LOG) B.pb(add(B[i-1], B[i-1]));
  }
  int calc(ll k){
    vector<int> a(n+1, 0); a[0] = 1;
    forn(i, LOG)if(k>>i&1)a = add(a, B[i]);
    int ret = 0;
    forn(i, n)ret = (ret + (ll)a[i+1]*In[i]%MOD + MOD)%MOD;
    return ret;
  }
};
```

¿Preguntas?

Fin
¡Gracias por venir!