GUIA PARA ICPC

```cpp
#include <bits/stdc++.h>
using namespace std;

using i64 = long long;
using u64 = unsigned long long;
using i128 = __int128_t;
using u128 = __uint128_t;
using ld = long double;
template <class T> using V = vector<T>;
template <class K, class Val> using umap = unordered_map<K, Val>;
template <class K> using uset = unordered_set<K>;

#define rep(i, n) for (int i = 0; i < n; ++i)
#define per(i, n) for (int i = n - 1; i >= 0; --i)
#define reps(i, a, b) for (int i = a; i < b; ++i)
#define pers(i, a, b) for (int i = b - 1; i >= a; --i)
#define all(x) begin(x), end(x)
#define rall(x) rbegin(x), rend(x)
#define len(x) (x.size())
#define fastio                                                                 \
  ios::sync_with_stdio(false);                                                 \
  cin.tie(nullptr);                                                            \
  cout.tie(nullptr);

constexpr int INF32 = 0x3f3f3f3f; // ~1e9
constexpr i64 INF64 = (i64)4e18;
constexpr ld EPS = 1e-12L;
constexpr ld PI = 3.14159265358979323846264338327950288L;
constexpr int MOD = 1e9 + 7;

template <class T> using MinHeap = priority_queue<T, vector<T>, greater<T>>;
template <class T> using MaxHeap = priority_queue<T>;

void solve() {}

int main() {
  fastio;
  int T = 1;
  // cin >> T;
  while (T--) solve();
  return 0;
}
```

TESTER SHELL SCRIPT

```bash
#!/bin/bash
# Uso: ./tester A.cpp  -> buscará in/A.in, in/A1.in, etc.

SRC=$1
BIN="${SRC%.*}"
```

```
BASENAME=$(basename "$BIN")
mkdir -p out
g++ -std=c++17 -O2 -Wall -DLOCAL -fsanitize=address,undefined "$SRC" -o "$BIN" || exit 1
shopt -s nullglob
for IN in in/"${BASENAME}"*.in; do
    TEST_FILE=$(basename "$IN")
    TEST_NAME="${TEST_FILE%.in}"
    echo -e "\nTesting $IN..."
    /usr/bin/time -v "./$BIN" < "$IN" > "out/$TEST_NAME.out" 2>&1 || {
        time "./$BIN" < "$IN" > "out/$TEST_NAME.out"
    }
    echo "--- INPUT ($IN) ---"
    head -n 5 "$IN"
    echo -e "\n--- OUTPUT (out/$TEST_NAME.out) ---"
    head -n 10 "out/$TEST_NAME.out"
    echo "========================================"
done
```

STRINGS

1. Hashing de Cadenas (Rolling Hash)

- **Complejidad:** $O(N)$ precomputo, $O(1)$ por consulta de substring.
- **Subproblemas:**
  - Verificar si dos substrings son iguales en $O(1)$.
  - Buscar patrón P en T (Rabin-Karp).
  - Determinar número de substrings distintos (con estructuras adicionales).
  - Comprobar palíndromos (comparando hash normal con hash reverso).
- **Funcionamiento:** Se interpreta la cadena como un número en base P (primo, ej. 31, 53) módulo M (primo grande, ej. $10^9+7, 10^9+9$). $H(s) = (s[0] \cdot P^0 + s[1] \cdot P^1 + \cdots + s[n-1] \cdot P^{n-1}) \pmod M$. Para obtener el hash de un substring $s[i \dots j]$, usamos hashes prefijos: $Hash(i,j) = (h[j] - h[i-1]) \cdot P^{-i} \pmod M$. *Nota: En implementación real, solemos no multiplicar por P^{-i} sino comparar $(h[j] - h[i-1]) \cdot P^{max-1-j}$ para evitar inversos modulares si solo comparamos igualdad.*

2. Función Prefijo (KMP - Knuth-Morris-Pratt)

- **Complejidad:** $O(N)$ tiempo, $O(N)$ espacio.
- **Subproblemas:**
  - Encontrar todas las ocurrencias de P en T.
  - Contar ocurrencias de cada prefijo en la misma cadena.
  - Encontrar la subcadena palindrómica más larga que es prefijo.
  - Determinar el periodo más pequeño de una cadena.
- **Funcionamiento:** Calcula un arreglo π donde $\pi[i]$ es la longitud del prefijo propio más largo de la subcadena $s[0 \dots i]$ que también es sufijo de esta subcadena. Si buscamos P en T, computamos π para la cadena $P + \# + T$ ($\#$ es un separador que no está en P ni T). Las posiciones donde $\pi[i] == |P|$ indican una ocurrencia.

3. Manacher (Palíndromos)

- **Complejidad:** $O(N)$ tiempo, $O(N)$ espacio.
- **Subproblemas:**
  - Subcadena palindrómica más larga.
  - Número total de subcadenas palindrómicas.

- **Funcionamiento:** Calcula para cada posición el "radio" del palíndromo centrado allí. Maneja palíndromos de longitud impar y par por separado (o transforma la cadena insertando caracteres dummy `#` entre cada letra: `aba -> #a#b#a#`). Usa un intervalo $[L, R]$ del palíndromo encontrado más a la derecha para aprovechar simetrías y saltar cálculos.

4. Aho-Corasick

- **Complejidad:** $O(N + \sum |P_i| + \text{ocu})$ tiempo.
- **Subproblemas:**
  - Encontrar todas las ocurrencias de un diccionario de palabras en un texto.
  - Problemas de cadenas con programación dinámica donde el estado es un nodo del autómata.
- **Funcionamiento:**
  1. Construye un Trie con los patrones.
  2. Calcula "failure links" (enlaces de sufijo) usando BFS. Un enlace de sufijo desde un nodo u apunta al nodo que representa el sufijo propio más largo de la cadena correspondiente a u que también existe en el Trie.
  3. Para buscar, se transita por el autómata; si no hay transición, se sigue el failure link.

5. Suffix Array + LCP Array

- **Complejidad:** $O(N \log N)$ construcción, $O(N)$ para LCP.
- **Subproblemas:**
  - Contar substrings distintos ($\sum(N - SA[i] - LCP[i])$).
  - Substring repetido más largo (máximo valor en LCP).
  - Substring común más largo entre dos cadenas (concatenar $S_1 + \# + S_2$).
  - Búsqueda de patrón en $O(|P| \log N)$ usando búsqueda binaria en el SA.
- **Funcionamiento:**
  - **SA:** Arreglo de índices iniciales de todos los sufijos de S ordenados lexicográficamente.
  - **LCP:** Arreglo donde $LCP[i]$ es la longitud del prefijo común más largo entre el sufijo $SA[i]$ y $SA[i-1]$. Se construye iterativamente ordenando prefijos cíclicos de longitud 2^k.

ÁRBOLES

1. Lowest Common Ancestor (LCA) - Binary Lifting

- **Complejidad:** $O(N \log N)$ precomputo, $O(\log N)$ por consulta.
- **Subproblemas:**
  - Encontrar el LCA de dos nodos u, v.
  - Distancia entre dos nodos: $dist(u, v) = depth[u] + depth[v] - 2 \cdot depth[LCA(u, v)]$.
  - Determinar si un nodo es ancestro de otro.
- **Funcionamiento:** Precomputamos `up[u][i]`, el 2^i-ésimo ancestro de u. Para consultar $LCA(u, v)$:
  1. Asegurar $depth[u] \leq depth[v]$ (swap si necesario).
  2. Subir v hasta que $depth[v] == depth[u]$ usando saltos binarios.
  3. Si $u == v$, ese es el LCA.
  4. Si no, subir ambos simultáneamente con saltos de potencia de 2 decrecientes mientras `up[u][i] != up[v][i]`.
  5. El LCA será `up[u][0]` al finalizar.

2. Heavy-Light Decomposition (HLD)

- **Complejidad:** $O(N)$ precomputo, $O(\log^2 N)$ por consulta/actualización.
- **Subproblemas:**
  - Suma/Máximo de valores en el camino entre u y v.
  - Actualizar valores de todos los nodos/aristas en un camino $u - v$.

- **Funcionamiento:** Descompone el árbol en cadenas disjuntas. Las aristas "pesadas" conectan un nodo con su hijo de mayor tamaño de subárbol. Cualquier camino desde la raíz a un nodo cambia de cadena a lo sumo $O(\log N)$ veces. Se usa una estructura de datos (usualmente Segment Tree) para mantener los valores sobre estas cadenas linealizadas.

3. Centroid Decomposition

- **Complejidad:** $O(N \log N)$ total.
- **Subproblemas:**
  - Contar caminos con cierta longitud k.
  - Encontrar el camino con peso mínimo/máximo con cierta restricción de número de aristas.
- **Funcionamiento:** Encuentra el centroide (nodo que al eliminarse deja componentes de tamaño $\leq N/2$). Procesa caminos que pasan por el centroide. Elimina el centroide y aplica recursivamente en los subárboles restantes. Esto forma un "Árbol de Centroides" de altura $O(\log N)$.

GRAFOS

1. Dijkstra

- **Complejidad:** $O(E \log V)$ usando priority_queue.
- **Subproblemas:**
  - Ruta más corta entre dos nodos en mapas de carreteras (pesos positivos).
- **Funcionamiento:** Mantiene la distancia mínima conocida desde el origen a cada nodo. Usa una cola de prioridad para seleccionar siempre el nodo no visitado con la menor distancia tentativa, relajando sus aristas adyacentes.

2. Bellman-Ford / SPFA

Camino más corto con pesos negativos (detecta ciclos negativos).

- **Complejidad:** $O(V \cdot E)$ para Bellman-Ford, $O(k \cdot E)$ promedio para SPFA (peor caso exponencial, usar con cuidado).
- **Subproblemas:**
  - Detección de ciclos negativos (arbitraje en monedas).
  - Caminos más cortos cuando existen aristas con peso negativo.
- **Funcionamiento (Bellman-Ford):** Relaja todas las aristas $V-1$ veces. Si en una iteración adicional V alguna distancia se reduce, existe un ciclo negativo alcanzable desde la fuente.

3. Floyd-Warshall

Todos los pares de caminos más cortos.

- **Complejidad:** $O(V^3)$.
- **Subproblemas:**
  - Distancias entre todos los pares de nodos en grafos densos pequeños ($N \leq 500$).
  - Cierre transitivo (saber si i puede alcanzar a j).
- **Funcionamiento:** Programación dinámica. `d[i][j]` almacena la distancia más corta usando solo nodos intermedios del conjunto $\{0 \dots k\}$. Itera incrementando k.

4. Minimum Spanning Tree (Prim / Kruskal)

Encontrar un subgrafo que conecte todos los vértices con el peso total mínimo.

- **Complejidad:** $O(E \log V)$ o $O(E \log E)$.
- **Subproblemas:**
  - Conectar una red de ciudades con costo mínimo de cableado.

  - "Maximum Spanning Tree" para problemas de cuello de botella (minimax path).
- **Funcionamiento (Kruskal):** Ordena todas las aristas por peso. Itera y añade la arista si no forma un ciclo (usando Disjoint Set Union - DSU).

5. Topological Sort

Orden lineal de vértices en un DAG tal que para cada arista $u \to v$, u aparece antes que v.

- **Complejidad:** $O(V + E)$.
- **Subproblemas:**
  - Programación de tareas con dependencias.
  - Detección de ciclos (si no se puede completar el orden).
  - Programación dinámica sobre grafos (camino más largo en DAG).
- **Funcionamiento (DFS based):** Realiza DFS. Cuando un nodo termina de procesarse (post-orden), añádelo al frente de una lista.

6. Puentes y Puntos de Articulación

Encontrar aristas/nodos críticos cuya eliminación desconecta el grafo.

- **Complejidad:** $O(V + E)$.
- **Subproblemas:**
  - Análisis de vulnerabilidad en redes.
  - Descomposición de grafos.
- **Funcionamiento:** DFS que mantiene `tin[u]` (tiempo de entrada) y `low[u]` (el `tin` más bajo alcanzable desde u en el árbol DFS usando una back-edge).
  - Puente (u, v): si `low[v] > tin[u]`.
  - Punto de articulación u: si `low[v] >= tin[u]` para algún hijo v (caso especial para la raíz).

7. Flujo Máximo (Dinic)

El algoritmo estándar eficiente para problemas de flujo en CP.

- **Complejidad:** $O(V^2E)$ en general, $O(E\sqrt{V})$ para grafos de capacidad unitaria (bipartite matching).
- **Subproblemas:**
  - Máximo matching en grafos bipartitos.
  - Corte mínimo (Min-Cut = Max-Flow).
  - Asignación de tareas, transporte con capacidades.
- **Funcionamiento:** Construye un "Level Graph" usando BFS para encontrar caminos más cortos en el grafo residual. Luego usa DFS múltiples veces para empujar flujo por estos caminos hasta saturarlos (blocking flow). Repite hasta que no se pueda llegar del sumidero al destino en el Level Graph.

PATRONES DE DP

1. DP con Bitmask (Máscaras de Bits)

Usada cuando el estado requiere representar un subconjunto de elementos. Solo viable para N pequeño ($N \leq 20 \sim 24$).

- **Complejidad:** Típicamente $O(2^N \cdot N)$ o $O(3^N)$ (si se iteran sub-máscaras).
- **Subproblemas:**
  - Traveling Salesperson Problem (TSP): camino más corto que visita todas las ciudades.
  - Asignaciones/Matching con costos (N personas a N tareas).
  - Encontrar subconjuntos con ciertas propiedades (ej. partición en k subconjuntos iguales).
  - Juegos imparciales pequeños.
- **Funcionamiento:**

- Se usa un entero para representar un conjunto: si el bit i está encendido (1), el elemento i está en el subconjunto.
- Estado típico: `dp(mask, last_index)` donde `mask` es el conjunto de elementos ya procesados/visitados, y `last_index` es el último elemento añadido (necesario si el orden importa, como en TSP).
- Transiciones: Probar apagar (o encender) un bit a la vez y transicionar al subproblema anterior.

Implementación (TSP - Fragmento)

```
int n;
int dist[20][20];
int memo[1 << 20][20];

int tsp(int mask, int pos) {
    if (mask == (1 << n) - 1) return dist[pos][0]; // Retorno al inicio
    if (memo[mask][pos] != -1) return memo[mask][pos];

    int ans = 1e9;
    for (int nxt = 0; nxt < n; nxt++) {
        if (!((mask >> nxt) & 1)) { // Si nxt no ha sido visitado
            ans = min(ans, dist[pos][nxt] + tsp(mask | (1 << nxt), nxt));
        }
    }
    return memo[mask][pos] = ans;
}
```

2. Digit DP (DP sobre Dígitos)

Esencial para problemas de conteo en rangos gigantes (hasta 10^{18}).

- **Complejidad:** $O(\text{dígitos} \cdot 10 \cdot \text{estados\_extra})$. Para `long long`, dígitos ≈ 18. Muy rápido.
- **Subproblemas:**
  - ¿Cuántos números entre $[L, R]$ cumplen la propiedad X (ej. suma de dígitos es primo, no contienen el dígito 7)?
  - Se resuelve como `solve(R) - solve(L-1)`.
- **Funcionamiento:**
  - Construye el número dígito a dígito desde la izquierda.
  - Estado clave: `tight` (booleano). Si `tight=true`, estamos restringidos por los dígitos del número original N. Si `tight=false`, podemos colocar cualquier dígito $0-9$.
  - Estados comunes: `pos` (índice del dígito actual), `tight` (restricción), `leading_zeros` (para evitar contar 0s a la izquierda si afecta la propiedad), `val` (estado actual de la propiedad, ej. suma de dígitos hasta ahora).

Implementación (Esqueleto estándar)

```
string S; // El número límite como string
long long memo[20][2][2][200]; // Ajustar dimensiones según problema

long long dp_digit(int pos, bool tight, bool leading_zeros, int current_sum) {
    if (pos == S.size()) {
        return current_sum == target_sum; // Ejemplo de condición base
    }
    if (memo[pos][tight][leading_zeros][current_sum] != -1)
        return memo[pos][tight][leading_zeros][current_sum];
```

```
    long long ans = 0;
    int limit = tight ? (S[pos] - '0') : 9;

    for (int digit = 0; digit <= limit; digit++) {
        bool next_tight = tight && (digit == limit);
        bool next_leading = leading_zeros && (digit == 0);
        // Actualizar current_sum u otros estados según el problema
        ans += dp_digit(pos + 1, next_tight, next_leading, current_sum + digit);
    }
    return memo[pos][tight][leading_zeros][current_sum] = ans;
}
```

3. DP en Árboles (Tree DP)

Usar la estructura del árbol para definir subproblemas en subárboles.

- **Complejidad:** Generalmente $O(N)$. Algunas variantes (Knapsack en árbol) pueden ser $O(N^2)$.
- **Subproblemas:**
  - Maximum Independent Set en un árbol.
  - Minimum Vertex Cover en un árbol.
  - Diámetro del árbol (alternativa a 2-BFS).
  - Contar caminos con ciertas propiedades que pasan por un nodo u.
- **Funcionamiento:**
  - Se realiza usualmente con DFS (post-order traversal).
  - Para un nodo u, primero resolvemos recursivamente para todos los hijos v.
  - Luego, combinamos los resultados de los hijos para calcular el estado de u.
  - Estado típico: `dp[u][estado]`, donde `estado` puede ser 0/1 (si tomamos o no el nodo u), o valores más complejos.

Implementación (Max Independent Set)

```
vector<int> adj[MAXN];
int dp[MAXN][2]; // dp[u][0] = no tomamos u, dp[u][1] = tomamos u

void dfs_tree_dp(int u, int p) {
    dp[u][0] = 0;
    dp[u][1] = 1; // Peso del nodo u (aquí asumimos peso 1)
    for (int v : adj[u]) {
        if (v == p) continue;
        dfs_tree_dp(v, u);
        // Si NO tomamos u, podemos tomar o no a los hijos (lo mejor de ambos)
        dp[u][0] += max(dp[v][0], dp[v][1]);
        // Si tomamos u, NO podemos tomar a los hijos
        dp[u][1] += dp[v][0];
    }
}
```

4. DP en Grids (Tableros 2D)

Problemas de caminos y acumulaciones en matrices.

- **Complejidad:** $O(N \cdot M)$ donde N, M son dimensiones del grid.
- **Subproblemas:**
  - Número de caminos desde $(0, 0)$ a $(N-1, M-1)$ (con obstáculos).

  - Camino de suma mínima/máxima en un grid.
  - Subcuadrado más grande de solo 1s.
- **Funcionamiento:**
  - El estado es simplemente la coordenada `dp[r][c]`.
  - Las transiciones dependen de los movimientos permitidos (usualmente solo $\rightarrow$ y $\downarrow$ para problemas simples, o todas direcciones para problemas más complejos que podrían requerir Dijkstra si hay ciclos de pesos positivos, aunque eso ya no es DP pura).
  - Se puede implementar iterativamente llenando la tabla fila por fila o columna por columna.

Implementación (Camino de costo mínimo)

```
int grid[MAXN][MAXN];
long long dp[MAXN][MAXN];

// Iterativo (más común en grids para evitar stack overflow en matrices grandes)
void solve_grid() {
    // Inicializar dp[0][0], y bordes si es necesario
    dp[0][0] = grid[0][0];
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            if (i == 0 && j == 0) continue;
            dp[i][j] = INF;
            if (i > 0) dp[i][j] = min(dp[i][j], dp[i-1][j] + grid[i][j]);
            if (j > 0) dp[i][j] = min(dp[i][j], dp[i][j-1] + grid[i][j]);
        }
    }
    // Resultado en dp[N-1][M-1]
}
```

5. DP de Intervalos (Range DP)

Para problemas donde un rango se puede dividir en dos sub-rangos.

- **Complejidad:** $O(N^3)$ o $O(N^2)$.
- **Subproblemas:**
  - Matrix Chain Multiplication.
  - Optimal Binary Search Tree.
  - Palíndromo más largo (subsequencia o substring si se hace con DP).
  - Eliminar elementos de un array con ciertos costos hasta que quede vacío.
- **Funcionamiento:**
  - Estado: `dp[L][R]` = mejor valor para el intervalo desde el índice L hasta R.
  - Transición: Iterar sobre un punto de división K entre L y R, combinando `dp[L][K]` y `dp[K+1][R]`.
  - Se debe iterar por **longitud** del intervalo (de 1 a N) para asegurar que los subproblemas más pequeños estén resueltos.

Implementación (Esqueleto iterativo)

```
for (int len = 1; len <= n; len++) {     // Longitud del rango
    for (int i = 0; i <= n - len; i++) { // Inicio del rango
        int j = i + len - 1;             // Fin del rango
        if (len == 1) {
            dp[i][j] = base_case(i);
            continue;
        }
```

```
            dp[i][j] = INF;
            // Probar todos los puntos de corte k
            for (int k = i; k < j; k++) {
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + cost(i, j));
            }
        }
    }
```

ESTRUCTURAS DE DATOS

1. Disjoint Set Union (DSU / Union-Find)

Estructura fundamental para manejar conjuntos disjuntos dinámicos.

- **Complejidad:** $O(\alpha(N))$ amortizado por operación (prácticamente $O(1)$).
- **Subproblemas:**
  - Detectar componentes conexas dinámicamente.
  - Algoritmo de Kruskal para MST.
  - Problemas de conectividad dinámica.
- **Funcionamiento:** Cada conjunto tiene un representante (raíz). Se usa path compression en `find()` y union by rank/size en `unite()`.

2. Fenwick Tree (Binary Indexed Tree)

Estructura eficiente para operaciones de rango con punto de actualización.

- **Complejidad:** $O(\log N)$ por operación.
- **Subproblemas:**
  - Suma de prefijo / rango.
  - Actualización de punto.
  - Rango add + rango sum (con dos BITs).
- **Funcionamiento:** Usa representación binaria de índices para mantener sumas parciales en una estructura arbórea implícita.

3. Segment Tree

Estructura versátil para consultas y actualizaciones de rango.

- **Complejidad:** $O(\log N)$ por operación.
- **Subproblemas:**
  - Suma/mínimo/máximo de rango.
  - Rango add/set con lazy propagation.
  - Queries complejas (GCD, XOR, etc.).
- **Funcionamiento:** Árbol binario donde cada nodo representa un intervalo. Las hojas son elementos individuales.

4. Sparse Table

Para RMQ (Range Minimum Query) estático - sin actualizaciones.

- **Complejidad:** $O(N \log N)$ preprocesamiento, $O(1)$ por consulta.
- **Subproblemas:**
  - Mínimo/máximo en rango estático.
  - GCD de rango.
- **Funcionamiento:** `st[i][j]` almacena el mínimo en el rango $[i, i + 2^j - 1]$. Para consultar $[L, R]$, se usan dos rangos que cubren el intervalo completo con solapamiento permitido.

5. Bipartite Matching (Hopcroft-Karp)

Algoritmo óptimo para maximum matching en grafos bipartitos.

- **Complejidad:** $O(\sqrt{V} \cdot E)$.
- **Subproblemas:**
  - Asignación óptima.
  - Maximum independent set en grafos bipartitos.
- **Funcionamiento:** Encuentra augmenting paths en fases usando BFS y DFS. Similar a Dinic pero optimizado para grafos bipartitos.

Implementación

```
struct BipartiteGraph {
    int n, m;
    V<V<int>> adj;
    V<int> match_left, match_right;
    V<int> dist;

    BipartiteGraph(int _n, int _m)
        : n(_n), m(_m), adj(_n), match_left(_n, -1), match_right(_m, -1) {}

    void add_edge(int u, int v) { adj[u].push_back(v); }

    bool bfs_hk() {
        queue<int> q;
        dist.assign(n + 1, INF32);
        rep(u, n) {
            if (match_left[u] == -1) {
                dist[u] = 0;
                q.push(u);
            }
        }
        dist[n] = INF32;
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            if (dist[u] < dist[n]) {
                for (int v : adj[u]) {
                    int next = match_right[v];
                    if (next == -1) next = n;
                    if (dist[next] == INF32) {
                        dist[next] = dist[u] + 1;
                        if (next != n) q.push(next);
                    }
                }
            }
        }
        return dist[n] != INF32;
    }

    bool dfs_hk(int u) {
        if (u == n) return true;
        for (int v : adj[u]) {
            int next = match_right[v];
```

```
            if (next == -1) next = n;
            if (dist[next] == dist[u] + 1 && dfs_hk(next)) {
                match_left[u] = v;
                match_right[v] = u;
                return true;
            }
        }
        dist[u] = INF32;
        return false;
    }

    int max_matching() {
        int result = 0;
        while (bfs_hk()) {
            rep(u, n) {
                if (match_left[u] == -1 && dfs_hk(u)) {
                    result++;
                }
            }
        }
        return result;
    }

    V<pair<int, int>> get_matching() {
        V<pair<int, int>> result;
        rep(u, n) {
            if (match_left[u] != -1) {
                result.push_back({u, match_left[u]});
            }
        }
        return result;
    }
};
```

ARITMÉTICA MODULAR Y TEORÍA DE NÚMEROS

1. Operaciones Modulares Básicas

Fundamentos para trabajar con módulo en competitive programming.

Implementación

```
i64 string_mod(string big_int, i64 mod = MOD) {
  i64 result = 0;
  for (char c : big_int) {
    result = (result * 10 + (c - '0')) % mod;
  }
  return result;
}```

# TEORÍA DE JUEGOS (GAMES)

## 1\. Juegos Imparciales y Teorema de Sprague-Grundy
```

```
El marco estándar para resolver la mayoría de juegos competitivos donde no hay empate y la última perso

  * **Complejidad:** Varía. Nim básico es $O(N)$. Calcular números de Grundy depende del grafo de estad
  * **Subproblemas:**
      * Nim Game (pilas de piedras).
      * Juegos en grafos (mover una ficha hasta que no se pueda).
      * Juegos compuestos (varios juegos independientes jugados simultáneamente).
  * **Funcionamiento:**
      * **Nim:** Un estado con pilas de tamaños $a_1, a_2, \dots, a_n$ es perdedor si $a_1 \oplus a_2 \
      * **Sprague-Grundy:** Cualquier juego imparcial es equivalente a una pila de Nim de cierto tamaño
      * $G(estado) = \text{MEX}(\{G(\text{estado\_siguiente}) \mid \text{estado} \to \text{estado\_sigu
      * **MEX (Minimum Excluded value):** El menor entero no negativo que NO está en el conjunto.
      * Un estado es perdedor si $G(estado) = 0$, ganador si $G(estado) > 0$.
      * Para juegos compuestos: $G(\text{juego\_total}) = G(\text{subjuego}_1) \oplus G(\text{subjuego}_

### Implementación (Cálculo de Grundy con Memoización)

```cpp
int memo[MAXN];

int calculate_mex(unordered_set<int>& s) {
    int mex = 0;
    while (s.count(mex)) mex++;
    return mex;
}

int get_grundy(int state) {
    if (state == 0) return 0; // Estado perdedor base
    if (memo[state] != -1) return memo[state];

    unordered_set<int> reachable_grundy_values;
    for (int next_state : get_moves(state)) {
        reachable_grundy_values.insert(get_grundy(next_state));
    }

    return memo[state] = calculate_mex(reachable_grundy_values);
}
// En main: memset(memo, -1, sizeof(memo));
// Si get_grundy(initial_state) > 0 -> Gana el primer jugador.
```

FAST FOURIER TRANSFORM (FFT)

1. Multiplicación de Polinomios

Técnica esencial para convoluciones y problemas combinatorios avanzados.

- **Complejidad:** $O(N \log N)$ para multiplicar dos polinomios de grado N.
- **Subproblemas:**
  - Multiplicación rápida de números grandes.
  - Convolución de secuencias: $C[k] = \sum A[i] \cdot B[k-i]$.
  - Contar sumas en subconjuntos (e.g., cuántas formas de obtener suma S lanzando K dados).
  - String matching con wildcards o diferencias aproximadas.
- **Funcionamiento:**
```

1. **Evaluación:** Convierte polinomios de representación de coeficientes a representación de puntos usando raíces de la unidad. ($O(N \log N)$).
2. **Multiplicación Punto a Punto:** Multiplica los valores evaluados en $O(N)$.
3. **Interpolación (FFT Inversa):** Convierte de vuelta a representación de coeficientes.
   - *Nota:* Para problemas con módulo, usar **Number Theoretic Transform (NTT)**. Es el mismo algoritmo pero reemplazando raíces complejas con raíces primitivas módulo P.

Implementación (FFT Iterativo in-place)

```
using cd = complex<double>;
const double PI = acos(-1);

void fft(vector<cd> & a, bool invert) {
    int n = a.size();
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1) j ^= bit;
        j ^= bit;
        if (i < j) swap(a[i], a[j]);
    }
    for (int len = 2; len <= n; len <<= 1) {
        double ang = 2 * PI / len * (invert ? -1 : 1);
        cd wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            cd w(1);
            for (int j = 0; j < len / 2; j++) {
                cd u = a[i+j], v = a[i+j+len/2] * w;
                a[i+j] = u + v;
                a[i+j+len/2] = u - v;
                w *= wlen;
            }
        }
    }
    if (invert) {
        for (cd & x : a) x /= n;
    }
}

vector<int> multiply(vector<int> const& a, vector<int> const& b) {
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;
    while (n < a.size() + b.size()) n <<= 1;
    fa.resize(n); fb.resize(n);

    fft(fa, false); fft(fb, false);
    for (int i = 0; i < n; i++) fa[i] *= fb[i];
    fft(fa, true);

    vector<int> result(n);
    for (int i = 0; i < n; i++) result[i] = round(fa[i].real());
    return result;
}
```

GEOMETRÍA COMPUTACIONAL

1. Primitivas Básicas (Punto 2D)

La base de todo. Define esto correctamente o fallarás todo lo demás.

- **Complejidad:** $O(1)$ por operación.
- **Subproblemas:** Representación, distancia, rotación.
- **Funcionamiento:** Estructura con sobrecarga de operadores para facilitar la matemática vectorial.

Implementación

```
using T = ld; // O long long si las coordenadas lo permiten
const T EPS = 1e-9;

struct Pt {
    T x, y;
    Pt operator+(Pt p) const { return {x+p.x, y+p.y}; }
    Pt operator-(Pt p) const { return {x-p.x, y-p.y}; }
    Pt operator*(T d) const { return {x*d, y*d}; }
    Pt operator/(T d) const { return {x/d, y/d}; }
    bool operator<(Pt p) const { return x < p.x || (abs(x - p.x) < EPS && y < p.y); }
    bool operator==(Pt p) const { return abs(x - p.x) < EPS && abs(y - p.y) < EPS; }
};
T dot(Pt v, Pt w) { return v.x*w.x + v.y*w.y; }
T cross(Pt v, Pt w) { return v.x*w.y - v.y*w.x; }
T mag_sq(Pt v) { return dot(v, v); }
ld mag(Pt v) { return sqrt(mag_sq(v)); }
T orient(Pt a, Pt b, Pt c) { return cross(b-a, c-a); } // >0 izq, <0 der, =0 colineal
```

2. Convex Hull (Monotone Chain)

Envolvente convexa: el polígono convexo más pequeño que contiene todos los puntos.

- **Complejidad:** $O(N \log N)$ (por el ordenamiento).
- **Subproblemas:**
  - Forma de un objeto a partir de puntos.
  - Par de puntos más lejanos (diámetro del conjunto).
- **Funcionamiento:**
  1. Ordena los puntos lexicográficamente (x, luego y).
  2. Construye la parte superior ("upper hull") e inferior ("lower hull") iterando y verificando giros con producto cruz.
  3. Si se forma un giro a la derecha (no convexo), elimina el último punto añadido hasta que sea convexo.

3. Área de Polígono (Shoelace Formula)

- **Complejidad:** $O(N)$.
- **Subproblemas:** Cálculo de áreas de terrenos, regiones.
- **Funcionamiento:** Suma de productos cruz de vértices adyacentes. El resultado es $2 \times$ Área (con signo dependiendo de la orientación, usar `abs` para área real).

Implementación

```
T polygon_area_2(const vector<Pt>& p) { // Retorna 2 * Area
    T area = 0;
    int n = len(p);
    rep(i, n) {
        area += cross(p[i], p[(i+1)%n]);
    }
    return abs(area);
}
```

4. Punto en Polígono (Point in Polygon)

Determinar si un punto está dentro, fuera o en el borde de un polígono.

- **Complejidad:** $O(N)$ para polígono general, $O(\log N)$ para convexo.
- **Funcionamiento (Ray Casting - General):** Lanza un rayo horizontal desde el punto. Cuenta cuántas veces cruza las aristas del polígono. Impar = dentro, Par = fuera. Manejar casos borde cuidadosamente.

Implementación (General - Ray Casting)

```
// 1: dentro, 0: borde, -1: fuera
int point_in_polygon(const vector<Pt>& p, Pt a) {
    bool inside = false;
    int n = len(p);
    rep(i, n) {
        int j = (i + 1) % n;
        // Verificar si está en el segmento (borde)
        if (cross(p[j]-p[i], a-p[i]) == 0 &&
            dot(p[i]-a, p[j]-a) <= 0) return 0;

        // Ray casting
        if ((p[i].y > a.y) != (p[j].y > a.y)) {
            if (a.x < (p[j].x - p[i].x) * (a.y - p[i].y) / (p[j].y - p[i].y) + p[i].x) {
                inside = !inside;
            }
        }
    }
    return inside ? 1 : -1;
}
```

PROBLEMAS INTERACTIVOS

1. **FLUSH SIEMPRE:** Después de cada `cout`, usen `endl` o `cout.flush()`.
2. **NO USEN FAST I/O A CIEGAS:** Su plantilla tiene `cin.tie(nullptr)`. Esto desvincula la entrada de la salida. Si no hacen flush manual, su código esperará una entrada que nunca llegará porque la pregunta sigue en su buffer.
3. **LEAN LOS LÍMITES DE QUERIES:** 100 queries $\implies$ $O(N)$ o $O(\sqrt{N})$ es inaceptable. Necesitan $O(\log N)$ o $O(1)$.
4. **WRAPPER DE QUERY:** No repitan código. Hagan una función para consultar.

1. Búsqueda Binaria Interactiva

El estándar para encontrar un valor oculto en un rango monótono.

- **Complejidad:** $O(\log N)$ queries.
- **Subproblemas comunes:**
  - Adivinar un número oculto entre 1 y N (mayor/menor).
  - Encontrar el índice del primer elemento que cumple condición X en un array ordenado (o rotado).
  - Encontrar la raíz de una función monótona.
- **Funcionamiento:** Mantienen un rango posible $[L, R]$. Consultan el punto medio. Según la respuesta del juez, descartan la mitad inútil. Repiten hasta $L = R$.

Implementación (Basada en plantilla)

```
// Función helper para interactuar. ÚSENLA.
int query(i64 val) {
    cout << "? " << val << endl; // endl hace flush automáticamente
    int resp;
    cin >> resp;
    if (resp == -1) exit(0); // Manejo de error/trampa del juez
    return resp;
}

void solve() {
    i64 N;
    cin >> N;
    i64 L = 1, R = N, ans = -1;
    while (L <= R) {
        i64 mid = L + (R - L) / 2;
        int res = query(mid);
        if (res == 1) { // Supongamos 1: "target es menor o igual"
            ans = mid;
            R = mid - 1;
        } else {        // Supongamos 0: "target es mayor"
            L = mid + 1;
        }
    }
    cout << "! " << ans << endl;
}
```

2. Reconstrucción Bit a Bit (Bit Manipulation)

Para encontrar números ocultos cuando las queries permiten operaciones OR, AND, XOR, o sumas.

- **Complejidad:** $O(\log(\text{MAX\_VAL}))$ queries (usualmente 30 o 60).
- **Subproblemas comunes:**
  - Determinar un número oculto X.
  - Encontrar elementos de un array si el juez retorna suma o XOR de rangos.
- **Funcionamiento:** No adivinen el número completo. Determinen cada bit individualmente, del menos significativo al más significativo (o viceversa), mediante máscaras. *Si query es SUMA/OR con potencias de 2:* `? (1<<i)` revela el i-ésimo bit.

Implementación (Ejemplo encontrando X con queries OR)

```
int ask_or(int y) {
    cout << "? " << y << endl;
    int res; cin >> res; return res;
}

void solve() {
    // Supongamos que queremos encontrar X oculto.
    // El juez responde (X | Y) cuando preguntamos Y.
    int hidden_X = 0;
    rep(i, 30) { // Para enteros hasta ~10^9
        int mask = (1 << i);
        // Si preguntamos por todo EXCEPTO el bit i, y el resultado
        // tiene el bit i encendido, entonces X debe tenerlo encendido.
        // Nota: La lógica exacta depende del tipo de query disponible.
        // Este es un ejemplo genérico de iteración por bits.

        // Ejemplo más directo: si query(mask) retorna (X & mask):
        // if (query(1 << i)) hidden_X |= (1 << i);
    }
    cout << "! " << hidden_X << endl;
}
```

3. Búsqueda Ternaria Interactiva

Para encontrar el máximo/mínimo de funciones unimodales (que suben y luego bajan, o viceversa).

- **Complejidad:** $O(\log N)$ queries (base 1.5 o 2 dependiendo de implementación).
- **Subproblemas comunes:**
  - Encontrar el pico de un array bitónico oculto.
  - Optimizar una función convexa/cóncava donde evaluar la función es una query.
- **Funcionamiento:** Se necesitan dos puntos de prueba m_1 y m_2 para determinar en qué tercio (o mitad) está el óptimo. *Optimización:* En interactivos, a menudo basta consultar mid y $mid + 1$ para saber la dirección de la pendiente.

Implementación (Encontrando máximo en función unimodal)

```
int ask_val(int idx) { /* ... implementación con cout/cin ... */ return 0; }

void solve() {
    int N; cin >> N;
    int L = 1, R = N;
    while (R - L > 2) {
        int m1 = L + (R - L) / 3;
        int m2 = R - (R - L) / 3;
        if (ask_val(m1) < ask_val(m2))
            L = m1;
        else
            R = m2;
    }
    // Rango pequeño [L, R], encontrar máx manualmente con queries restantes
    int ans = L, max_v = ask_val(L);
    reps(i, L + 1, R + 1) {
```

```
        int val = ask_val(i);
        if (val > max_v) { max_v = val; ans = i; }
    }
    cout << "! " << ans << endl;
}
```

4. Detección de Grafos/Árboles (Exploración)

Problemas donde la estructura del grafo está oculta.

- **Complejidad:** Varía, usualmente $O(N)$ o $O(N \log N)$.
- **Subproblemas comunes:**
  - Encontrar el padre de cada nodo en un árbol enraizado.
  - Encontrar bordes en un grafo general.
  - Encontrar el centroide o diámetro.
- **Funcionamiento:**
  - *Árboles:* Consultar la distancia desde la raíz a todos los nodos da sus profundidades. Los nodos de profundidad D solo pueden ser hijos de nodos de profundidad $D-1$.
  - *General:* Usar búsquedas binarias sobre conjuntos de nodos para identificar a qué nodo específico está conectado un nodo U.

Implementación (Esqueleto para padres en árbol)

```
int ask_dist(int u, int v) {
    cout << "? " << u << " " << v << endl;
    int d; cin >> d; return d;
}

void solve() {
    int N; cin >> N;
    V<int> depth(N + 1);
    V<V<int>> nodes_at_depth(N + 1);
    reps(i, 2, N + 1) {
        depth[i] = ask_dist(1, i); // Asumiendo 1 es raíz
        nodes_at_depth[depth[i]].push_back(i);
    }
    V<int> parent(N + 1);
    reps(d, 1, N + 1) {
        for (int u : nodes_at_depth[d]) {
            // Lógica para encontrar cuál nodo de nodes_at_depth[d-1] es padre de u.
            // Puede requerir más queries o deducción lógica si el límite es estricto.
        }
    }
}
```

5. Randomización (Interactiva Probabilística)

Cuando un enfoque determinista excede el límite de queries, pero un enfoque aleatorio tiene alta probabilidad de éxito.

- **Complejidad:** Esperanza $O(K)$ donde K es pequeño, aunque el peor caso sea malo.
- **Subproblemas comunes:**
  - Encontrar *cualquier* elemento que cumpla una propiedad común (ej. encontrar un elemento $\geq N/2$ en un array casi ordenado).

  - Verificar si un array es igual a otro oculto con pocas queries (hashing/random index).
- **Funcionamiento:** Usar `mt19937` para generar índices aleatorios y consultarlos hasta encontrar uno que cumpla la condición deseada o agotar un presupuesto de queries fijo.

Implementación

```
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());

int ask(int idx) { /* ... */ return 0; }

void solve() {
    int N; cin >> N;
    V<int> p(N);
    iota(all(p), 1);
    shuffle(all(p), rng); // Orden de consulta aleatorio

    int queries_left = 50; // Límite seguro
    for (int idx : p) {
        if (queries_left-- == 0) break;
        if (ask(idx) == TARGET_CONDITION) {
            cout << "! " << idx << endl;
            return;
        }
    }
    // Si llegamos aquí, fallamos o se requiere otra lógica.
}
```

DIVIDE AND CONQUER

1. Binary Search

- **Complejidad:** $O(\log N)$ si el rango es $[0, N]$.
- **Subproblemas Comunes:**
  - `lower_bound` / `upper_bound` en arreglos ordenados.
  - Encontrar la respuesta mínima/máxima que satisface una condición monótona (Binary Search on Answer).
  - Minimizar el máximo valor (minimax) o maximizar el mínimo valor (maximin).
- **Receta Definitiva (Invariant-based):** Mantén un invariante: `check(L)` siempre es verdadero (o falso), y `check(R)` siempre es lo opuesto. El rango $[L, R]$ siempre contiene la frontera. Al final, L y R son adyacentes, y la respuesta es uno de ellos según lo que busques.

Implementación (Receta Universal)

Esta plantilla evita bucles infinitos y errores de off-by-one. Úsala siempre.

```
// Supongamos que queremos encontrar el MAYOR x tal que check(x) es VERDADERO.
// Invariante: check(L) == true, check(R) == false
long long L = 0, R = 1e18; // Rango seguro que cumple invariante inicial
if (!check(L)) { /* Manejar caso imposible si es necesario */ }

while (R - L > 1) {
    long long mid = L + (R - L) / 2;
    if (check(mid)) {
        L = mid; // mid sigue siendo válido, lo mantenemos en L
```

```
    } else {
        R = mid; // mid es inválido, lo mantenemos en R
    }
}
// Al final, L es el mayor valor que cumple check.
// Si buscábamos el MENOR valor que cumple check:
// Invariante: check(L) == false, check(R) == true -> R sería la respuesta.
```

2. Binary Search on Answer (Small/Large)

Cuando la respuesta tiene monotonicidad.

- **Small (Búsqueda de valor exacto/óptimo):** El rango de respuesta es pequeño (ej. 10^9, `long long`). Se usa la receta de arriba.
- **Large (Búsqueda con decimales):** El rango es continuo. Se itera un número fijo de veces para precisión deseada.
- **Complejidad:** $O(\log(\frac{R-L}{\epsilon}) \cdot F(N))$ donde $F(N)$ es el costo de `check()`.

Implementación (Large - Decimales)

```
double L = 0.0, R = 1e9;
rep(iter, 100) { // 100 iteraciones dan suficiente precisión para casi todo
    double mid = (L + R) / 2.0;
    if (check(mid)) {
        L = mid; // O R = mid dependiendo de la monotonicidad
    } else {
        R = mid;
    }
}
// Respuesta está en L (o R, son casi iguales)
```

3. Merge Sort (y conteo de inversiones)

Algoritmo de ordenamiento clásico que ilustra D&C. Vital para contar inversiones.

- **Complejidad:** $O(N \log N)$.
- **Subproblemas Comunes:**
  - Ordenar un arreglo (aunque `std::sort` es mejor para esto).
  - Contar el número de pares (i, j) tal que $i < j$ y $A[i] > A[j]$ (inversiones).
- **Funcionamiento:** Divide el arreglo en dos mitades, ordena recursivamente cada mitad, y luego fusiona las dos mitades ordenadas en $O(N)$. Durante la fusión, si tomamos un elemento de la mitad derecha mientras aún quedan elementos en la izquierda, esos son pares invertidos.

Implementación (Conteo de Inversiones)

```
long long inversions = 0;
void merge_sort(vector<int>& a, int l, int r) {
    if (l >= r) return;
    int mid = l + (r - l) / 2;
    merge_sort(a, l, mid);
    merge_sort(a, mid + 1, r);

    vector<int> tmp;
    int i = l, j = mid + 1;
```

```
    while (i <= mid && j <= r) {
        if (a[i] <= a[j]) {
            tmp.push_back(a[i++]);
        } else {
            tmp.push_back(a[j++]);
            inversions += (mid - i + 1); // Elementos restantes en izq son mayores que a[j]
        }
    }
    while (i <= mid) tmp.push_back(a[i++]);
    while (j <= r) tmp.push_back(a[j++]);
    for (int k = 0; k < len(tmp); ++k) a[l + k] = tmp[k];
}
```

4. Closest Pair of Points

Encontrar el par de puntos más cercano en un plano 2D.

- **Complejidad:** $O(N \log N)$.
- **Subproblemas Comunes:** Problemas geométricos que requieren distancias mínimas.
- **Funcionamiento:**
  1. Ordena los puntos por coordenada X.
  2. Divide el conjunto en dos mitades por una línea vertical.
  3. Resuelve recursivamente para cada mitad, obteniendo distancias mínimas d_L, d_R. Sea $d = \min(d_L, d_R)$.
  4. Combina: Revisa los puntos en la franja central de ancho $2d$. Para cada punto en la franja, solo necesita compararse con un número constante de puntos siguientes ordenados por Y.

Implementación (Esqueleto)

Se omite por extensión, pero la clave es la combinación eficiente en la franja central. Usualmente se prefiere usar un sweep-line con `std::set` para una implementación más sencilla en competencia si el límite de tiempo lo permite ($O(N \log N)$ también).

5. Quick Select (k-th element)

Encontrar el k-ésimo elemento más pequeño en un arreglo desordenado.

- **Complejidad:** $O(N)$ esperado, $O(N^2)$ peor caso (evitable con randomización o mediana de medianas).
- **Subproblemas Comunes:**
  - Encontrar la mediana en $O(N)$.
  - Encontrar los k elementos más pequeños sin ordenar todo.
- **Funcionamiento:** Elige un pivote, particiona el arreglo como en Quicksort. Si el pivote queda en la posición k, hemos terminado. Si queda en $p > k$, recursión en la izquierda. Si $p < k$, recursión en la derecha.

Implementación (Uso de STL)

En C++, usa `std::nth_element`. Es altamente optimizado y suficiente para casi todos los casos.

```
vector<int> a = {5, 2, 9, 1, 5, 6};
int k = 2; // Queremos el elemento en índice 2 (3er más pequeño) si estuviera ordenado
nth_element(a.begin(), a.begin() + k, a.end());
// Ahora a[k] contiene el valor correcto. Elementos a la izquierda son <= a[k], derecha >= a[k].
cout << a[k] << endl; // Output: 5 (arreglo podría quedar {1, 2, 5, 9, 5, 6} o similar)
```

COMBINATORIA & FUNCIONES GENERATRICES

1. Combinatoria Básica & Coeficientes Binomiales

- **Complejidad:** $O(N)$ precomputo, $O(1)$ por consulta.
- **Subproblemas Comunes:**
  - Número de formas de elegir K elementos de N: $\binom{N}{K}$.
  - Número de caminos en una grilla de $(0,0)$ a (N,M): $\binom{N+M}{N}$.
  - Distribución de objetos idénticos en cajas distintas (Stars and Bars): $\binom{N+K-1}{K-1}$.
- **Funcionamiento:**
  - $\binom{N}{K} = \frac{N!}{K!(N-K)!} \pmod M$.
  - Se precomputan factoriales y sus inversos modulares.

Implementación

```
i64 fact[MAXN], invFact[MAXN];
i64 nCr(int n, int r) {
    if (r < 0 || r > n) return 0;
    return (((fact[n] * invFact[r]) % MOD) * invFact[n - r]) % MOD;
}
void precompute() {
    fact[0] = 1;
    invFact[0] = 1;
    for (int i = 1; i < MAXN; i++) {
        fact[i] = (fact[i - 1] * i) % MOD;
    }
    invFact[MAXN - 1] = binpow(fact[MAXN - 1], MOD - 2);
    for (int i = MAXN - 2; i >= 1; i--) {
        invFact[i] = (invFact[i + 1] * (i + 1)) % MOD;
    }
}
```

2. Principio de Inclusión-Exclusión (PIE)

Para contar el tamaño de la unión de conjuntos.

- **Complejidad:** $O(2^N \cdot \text{costo\_intersección})$.
- **Subproblemas Comunes:**
  - Contar números coprimos con N en un rango.
  - Problemas de conteo con restricciones "al menos uno".
  - Desarreglos (permutaciones sin puntos fijos).
- **Funcionamiento:** $|A_1 \cup \cdots \cup A_n| = \sum |A_i| - \sum |A_i \cap A_j| + \sum |A_i \cap A_j \cap A_k| - \ldots$ Se itera sobre todas las máscaras de bits, sumando o restando según la paridad del número de bits encendidos.

Implementación (Esqueleto)

```
i64 ans = 0;
for (int mask = 1; mask < (1 << n); ++mask) {
    i64 intersection_size = calculate_intersection(mask);
    if (__builtin_popcount(mask) % 2 == 1) {
        ans = (ans + intersection_size) % MOD;
    } else {
        ans = (ans - intersection_size + MOD) % MOD;
```

```
    }
}
```

3. Números de Catalan

- **Complejidad:** $O(N)$ precomputo o $O(1)$ con factoriales.
- **Subproblemas Comunes:**
  - Número de expresiones de paréntesis válidas de longitud $2N$.
  - Número de árboles binarios completos con $N+1$ hojas.
  - Triangulaciones de un polígono convexo de $N+2$ vértices.
- **Fórmula:** $C_n = \frac{1}{n+1}\binom{2n}{n}$.

Implementación

Usa la función `nCr` precomputada.

```
i64 catalan(int n) {
    return (nCr(2 * n, n) * binpow(n + 1, MOD - 2)) % MOD;
}
```

4. Funciones Generatrices (Concepto)

Representar secuencias como coeficientes de un polinomio formal.

- **Idea:** La secuencia $a_0, a_1, a_2, \ldots$ se representa como $A(x) = a_0 + a_1 x + a_2 x^2 + \ldots$.
- **Operaciones Clave:**
  - **Suma:** $A(x) + B(x)$ corresponde a la suma término a término de las secuencias.
  - **Producto (Convolución):** $A(x)B(x) = C(x)$ donde $c_k = \sum_{i=0}^{k} a_i b_{k-i}$. Esto modela "combinar" dos estructuras de tamaño i y $k-i$.
- **Uso Competitivo:**
  - Transformar problemas de conteo complejos en multiplicaciones de polinomios.
  - Resolver recurrencias lineales.
  - Problemas de particiones (ej. cambio de monedas).
- **Herramienta:** FFT/NTT para multiplicar polinomios rápidamente ($O(N \log N)$).

5. Burnside's Lemma / Polya Enumeration Theorem

Contar objetos distintos bajo simetrías (rotaciones, reflexiones).

- **Complejidad:** $O(|G| \cdot \text{costo\_fijo})$, donde G es el grupo de simetrías.
- **Subproblemas Comunes:**
  - Número de formas de colorear un collar con N cuentas usando K colores, considerando rotaciones iguales.
- **Funcionamiento:** El número de órbitas (objetos distintos) es el promedio del número de elementos fijados por cada simetría en G. $|X/G| = \frac{1}{|G|}\sum_{g \in G} |X^g|$, donde X^g son los elementos que no cambian al aplicar la simetría g.

Implementación (Collar, solo rotaciones)

Para un collar de N cuentas y K colores, las simetrías son rotaciones de $0, 1, \ldots, N-1$ posiciones. Una rotación de i posiciones tiene $\gcd(i, N)$ ciclos. Cada ciclo debe tener el mismo color. Número de coloraciones fijas por rotación i: $K^{\gcd(i,N)}$.

```
i64 count_necklaces(int n, int k) {
    i64 ans = 0;
    for (int i = 0; i < n; i++) {
```

```
        ans = (ans + binpow(k, gcd(i, n))) % MOD;
    }
    ans = (ans * binpow(n, MOD - 2)) % MOD;
    return ans;
}
```