1. Algoritmia

1.1. Búsqueda exhaustiva, fuerza bruta o backtracking, como lo quieras llamar

Buscar todo/casi todo el espacio de soluciones, y hasta encontrar una solución. Es todo lo lento que puede ser, pero no es mala idea intentarlo si no se ocurre otra idea.

```
// Búsqueda exhaustiva incluye, entre otras cosas:
//  Probar permutaciones
void permutaciones() {
    int n = 8, p[8] = {0, 1, 2, 3, 4, 5, 6, 7};
    do {
        // Comprobar si la permutación p es válida
    } while (
        next_permutation(p, p + n)); // next_permutation genera la siguiente
                                     // permutación y devuelve true si existe
}

// Backtracking
bool se_puede_colocar(int c, const vector<int> &reinas);

void backtrack(int c, vector<int> &reinas) {
    // Ejemplo de backtracking para el problema de las 8 reinas
    if (c == 8) {
        // Solución encontrada, imprimir reinas o devolver o lo que sea
        // Para salir del todo estaría bien devolver un booleano, o algo así
        // ...
        return;
    }

    // Buscamos todos los estados hijos
    for (int i = 0; i < 8; i++) {
        if (se_puede_colocar(c, reinas)) {
            reinas[c] = i; // Colocamos la reina en la columna c y fila i
            backtrack(c + 1, reinas); // Llamamos al siguiente nivel
            reinas[c] = -1;           // Deshacemos el movimiento (backtracking)
        }
    }

    // Si no se puede colocar ninguna reina en la columna c, no hacemos nada
    return;
}
```

1.2. Divide y vencerás

Consiste en seguir tres sencillos pasos: dividir el problema en sub-problemas (como por la mitad más menos), encontrar soluciones a esos problemas más pequeños (que será más fácil, porque son más pequeños) y combinar las soluciones de los subproblemas. Normalmente consiste en utilizar una estructura que haga esto (montículos, bBST, etc.) o en hacer **búsqueda binaria de la solución**.

```
// Pongamos una función que nos dice si un valor x cumple y
bool can(int x);

// El problema quiere encontrar el mínimo valor x que cumpla y
// Para ello, podemos hacer una búsqueda binaria sobre el espacio de soluciones
int bsta(int low, int high) {
    while (low < high) {
        int mid = (low + high) / 2; // Punto medio
        if (can(mid))               // Lo probamos
            high = mid;             // Si cumple, buscamos valores más pequeños
        else                        // Si no,
            low = mid + 1;          // buscamos valores más grandes
    }

    // Al final, low y high valen lo mismo, y es el valor mínimo que cumple la
    // condición
    return low;
}

// Se puede hacer con flontantes también, estableciendo un epsilon y sin
// necesidad de incrementar low en 1
```

1.3. Voraz, greedy

Para que un problema se pueda resolver con un algoritmo voraz hace falta que: tenga sub-estructuras óptimas (aka. que la solución óptima para el problema completo contenga las soluciones de los sub-problemas) y cumple la *propiedad voraz*, escoger la mejor opción en algún momento siempre lleva al mejor resultado, nunca hay que hacer backtrack (esta segunda no merece la pena intentar demostrarla durante el concurso, mejor implementarlo y probar). Tirar una cola de prioridad a las cosas acostumbra a ser greedy.

1.4. Programación dinámica

Backtracking pero mejor. Se utiliza principalmente en problemas de optimización o contar cosas. Para que un problema se pueda resolver con DP hacen falta dos cosas: que tenga sub-estructuras óptimas (como voraz) y que sus sub-problemas se superpongan (es decir, hay varias maneras de llegar a un mismo estado de búsqueda). DP consiste en cachear los resultados de los subproblemas para usos posteriores. Se puede hacer "desde arriba" o "desde abajo".

```
int fib_tonto(int n) {
    if (n <= 1) return n;                       // Caso base
    return fib_tonto(n - 1) + fib_tonto(n - 2); // Llamadas recursivas
}

// Programación dinámica, top-down
// Utiliza la tabla de memoización para evitar cálculos repetidos
// Basicamente, "cachea" los resultados de las llamadas recursivas
vector<int> memo_td(100, -1); // Tabla de memoización, inicializada a -1
int fib_dp_td(int n) {
    if (n <= 1) return n;      // Caso base
    auto &res = memo_td[n];    // Referencia a la posición n de memo
    if (res != -1) return res; // Si ya está calculado, lo devolvemos
    res = fib_dp_td(n - 1) +
          fib_dp_td(n - 2); // Si no, lo calculamos y lo guardamos
    return res;             // Devolvemos el resultado
}

// Programación dinámica, bottom-up
// Rellena en la tabla de memoización los casos base. A partir de ahí, calcula
// los siguientes casos utilizando los anteriores hasta llegar al caso n.
vector<int> memo_bu(100, -1); // Tabla de memoización, inicializada a -1
int fib_dp_bu(int n) {
    memo_bu[0] = 0; // Caso base
    memo_bu[1] = 1; // Caso base

    for (int i = 2; i <= n; i++) {
        memo_bu[i] = memo_bu[i - 1] + memo_bu[i - 2]; // Calculamos el valor
    }
    return memo_bu[n]; // Devolvemos el resultado
}
```

LIS

```
typedef vector<int> vi;

/*
 * Si solo se quiere la longitud del LIS, se puede obviar todo lo relacionado
 * con el array de predecesores (p, lis_end, L_id) y la función print_lis.
 */

int n; // Longitud del array A
vi A;  // Array de entrada
vi p;  // Array de predecesores

void print_lis(int i) {
    if (p[i] == -1) {
        cout << A[i];
        return;
    }

    print_lis(p[i]);     // Imprimir el LIS recursivamente
    cout << "␣" << A[i]; // Imprimir el elemento actual
}

void lis() {
```

```
23      int k = 0;        // Longitud del LIS
24      int lis_end = 0; // Índice del último elemento del LIS
25      vi L(n, 0), L_id(n, 0);
26      p.assign(n, -1); // Inicializar el array de predecesores
27
28      for (int i = 0; i < n; i++) {
29          // Búsqueda binaria para encontrar la posición de A[i] en L
30          int pos = lower_bound(L.begin(), L.begin() + k, A[i]) - L.begin();
31          L[pos] = A[i]; // Actualizar L con el nuevo valor
32          L_id[pos] = i; // Guardar el índice del elemento en L_id
33          p[i] = pos ? L_id[pos - 1] : -1; // Actualizar el predecesor
34          if (pos == k) {
35              k++;          // Incrementar la longitud del LIS
36              lis_end = i; // Actualizar el índice del último elemento del LIS
37          }
38      }
39
40      // La longitud del LIS es k
41      print_lis(lis_end); // Imprimir el LIS
42  }
```

2. Estructuras

2.1. Listas, Pilas, colas, sets, hashmaps, heaps

Todas las estructuras de datos básicas que se pueden encontrar en la STL de C++.

```
vector<int> lista;                  // Suele venir bien inicializada
stack<int> pila;                    // LIFO
queue<int> cola;                    // FIFO
priority_queue<int> cola_prioridad; // La cabeza es el mayor elemento
unordered_set<int> conjunto;        // Conjunto de elementos únicos
unordered_map<string, int> hashmap; // Mapa de clave-valor, sin orden
multimap<string, int> multimap;     // Mapa de clave-valor, con claves repetidas
                                    // un bBST

// No hay clase para el heap, pero se puede usar un vector
vector<int> H;
is_heap(H.begin(), H.end());   // Comprueba si es un heap
make_heap(H.begin(), H.end()); // Crea un heap a partir de un vector
push_heap(H.begin(), H.end()); // Incluye el último elemento en el heap
pop_heap(H.begin(), H.end());  // Mueve el mayor elemento al final
sort_heap(H.begin(), H.end()); // Convierte el heap en un vector ordenado
```

2.2. Conjuntos disjuntos

Modela conjuntos disjuntos. Útil para encontrar componentes conexos en grafos.

```
typedef vector<int> vi;

class UFDS {
  private:
    vi p, rank;

  public:
    // Crea un UFDS con n elementos
    UFDS(int n) : p(n, 0), rank(n, 0) {
        for (int i = 0; i < n; i++)
            p[i] = i;
    }

    // Encuentra el representante del conjunto al que pertenece i
    int findRep(int i) {
        if (p[i] != i) {
            p[i] = findRep(p[i]); // Path compression
        }
        return p[i];
    }

    // Comprueba si i y j pertenecen al mismo conjunto
    bool sameSet(int i, int j) { return findRep(i) == findRep(j); }

    // Hacer que i y j pertenezcan al mismo conjunto
    void unionSet(int i, int j) {
        if (sameSet(i, j))
            return;
        int repI = findRep(i);
        int repJ = findRep(j);
        if (rank[repI] > rank[repJ])
            swap(repI, repJ);
        p[repI] = repJ; // Unir los conjuntos
        if (rank[repI] == rank[repJ])
            rank[repJ]++;
    }
};
```

2.3. Grafos

Los grafos se pueden representar de varias formas, pero la más común es mediante una lista de adyacencia (es la que se usa en todos). También hay veces que no hace falta una estructura específica para los grafos. Ordenados de más a menos uso:

```
int n; // Número de nodos

/*
 * Grafo como lista de adyacencia:
 *   - El elemento AL[i] es un vector que contiene los nodos adyacentes a i.
```

```
 *   - Si el grafo es ponderado, AL[i] contiene pares (nodo, peso).
 *   - Puede representar multigrafos.
 */
vector<vector<pair<int, int>>> AL(n);

/*
 * Grafo como matriz de adyacencia:
 *   - El elemento AM[i][j] es -1 si no hay arista entre i y j,
 *     o el peso de la arista si existe.
 *   - Si el grafo es no dirigido, AM es igual a su transpuesta.
 *   - No puede representar multigrafos
 *   - No debería usarse para grafos donde n > 5000
 */
vector<vector<int>> AM(n, vector<int>(n, -1));

/*
 * Grafo como lista de aristas:
 *   - El elemento E[i] es una trupla (n1, n2, peso)
 *   - Puede representar multigrafos.
 *   - No es eficiente para consultas de adyacencia.
 */
vector<tuple<int, int, int>> EL;
```

BFS y DFS

Las dos formas más comunes de recorrer un grafo. BFS (Breadth-First Search) recorre el grafo primero por niveles mientras que DFS (Depth-First Search) recorre el grafo en profundidad. La primera se suele implementar con una cola y la segunda de manera recursiva (o con una pila).

Se pueden usar para: encontrar componentes conexos, detectar ciclos (al intentar volver a un nodo ya visitado), contar el orden topológico de un grafo dirigido (DFS), comprobar si un grafo es bipartito (BFS, si vamos guardando distancias nos aseguramos de que los nodos pares solo están conectados a impares y viceversa), etc.

```
typedef vector<int> vi;
typedef vector<vi> vvi;

int n;  // Número de nodos
vvi AL; // Lista de adyacencia del grafo

void bfs(int start) {
    vi dist(n, -1);  // Distancias desde el nodo inicial
    dist[start] = 0; // Distancia al nodo inicial es 0
    queue<int> q;    // Cola para BFS
    q.push(start);

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        // Se puede hacer algo con el nodo aquí

        for (auto &[v, w] : AL[u]) {
            if (dist[v] != -1)
                continue;          // Nodo visitado
            dist[v] = dist[u] + w; // Actualizar la distancia
            q.push(v);             // Añadir el nodo a la cola
        }
    }
}

vector<bool> visited(n, false); // Vector para DFS
void dfs(int u) {
    visited[u] = true; // Marcar el nodo como visitado

    // Se puede hacer algo con el nodo aquí

    for (auto &[v, w] : AL[u]) {
        if (!visited[v])
            dfs(v); // Llamada recursiva a DFS
    }
}
```

Componentes conexos

```
typedef vector<int> vi;

int n;                  // Vértices del grafo
vector<vi> AL;          // Lista de adyacencia
vector<bool> visitados; // -1: no visitado, 0: visitado

void dfs(int u); // Prototipo de la función DFS

// Componentes conexas
int cc() {
    visitados.assign(AL.size(), -1);

    int numCC = 0;
    for (int u = 0; u < n; u++) // Por cada vértice
        if (!visitados[u]) {    // Si no ha sido visitado
            numCC++;            // Incrementa el contador de componentes conexas
            dfs(u); // Llama a DFS para visitar todos los vértices de esta
                    // componente
        }

    return numCC; // Retorna el número de componentes conexas
}
```

Orden topológico

```
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;

int n;                  // Vértices del grafo
vector<vii> AL;         // Lista de adyacencia
vector<bool> visitados; // -1: no visitado, 0: visitado
vi ts;                  // Vector para almacenar el orden topológico

void toposort(int u) {
    visitados[u] = true;
    for (auto &[v, w] : AL[u])
        if (visitados[v] == false) toposort(v);
    ts.push_back(u); // this is the only change
}

// Componentes conexas
void calc_toposort() {
    visitados.assign(AL.size(), -1);
    ts.clear(); // Limpiar el vector de orden topológico

    for (int u = 0; u < n; u++) // Por cada vértice
        if (!visitados[u])      // Si no ha sido visitado
            toposort(u);

    reverse(ts.begin(),
            ts.end()); // Invertir el orden para obtener el topológico

    // Ahora `ts` contiene el orden topológico de los vértices
}
```

MST

Construcción de un árbol con coste mínimo que conecta todos los nodos de un grafo.

```
int n;                  // Número de nodos
vector<vii> AL;         // Lista de adyacencia del grafo
vi taken;               // Vector para marcar nodos visitados
priority_queue<ii> pq; // Para escoger los edges (es un max-heap por defecto)
                        // así que usamos el peso negativo

void process(int u) {
    taken[u] = 1; // Marcar el nodo como visitado
    for (auto &[v, w] : AL[u])
        if (!taken[v])
            pq.push({-w, -v}); // Añadir los edges al priority queue
}
```

```
13
14 void prim() {
15     taken.assign(n, 0); // Inicializar el vector de nodos visitados
16     process(0);         // Empezar desde el nodo 0
17
18     int mst_cost = 0, num_taken = 0; // Coste del MST y número de nodos tomados
19     while (!pq.empty()) {
20         auto [w, u] = pq.top(); // Obtener el edge con menor peso
21         pq.pop();
22         w = -w; // Convertir el peso y el nodo a su valor original
23         u = -u;
24         if (taken[u])
25             continue; // Si el nodo ya está visitado, saltar
26
27         mst_cost += w; // Añadir el peso al coste del MST
28
29         process(u);  // Procesar el nodo
30         num_taken++; // Incrementar el número de nodos tomados
31         if (num_taken == n - 1) {
32             break; // Si ya hemos tomado n-1 nodos, el MST está completo
33         }
34     }
35 }
```

Dijkstra

Camino mínimo desde un nodo a todos los demás en un grafo ponderado (con pesos **NO** negativos).

```
1  int n;  // Número de nodos
2  vvi AL; // Lista de adyacencia del grafo
3
4  // s es el nodo de inicio
5  void dijkstra(int s) {
6      vi dist(n, 1e9);
7      dist[s] = 0;
8      priority_queue<ii, vector<ii>, greater<ii>> pq;
9      pq.emplace(0, s);
10
11     while (!pq.empty()) {
12         auto [d, u] = pq.top(); // Nodo con distancia mínima
13         pq.pop();
14         if (d > dist[u]) continue; // Si la distancia no es la mínima, saltamos
15
16         // Si quisiesemos salir al llegar a un nodo destino, podríamos añadir
17         // una condición aquí (sin no hay negativos):
18         // if (u == destino) return;
19
20         for (auto &[v, w] : AL[u]) {     // Recorremos los vecinos del nodo u
21             if (dist[u] + w < dist[v]) { // Si encontramos una distancia menor
22                 dist[v] = dist[u] + w;   // Actualizamos la distancia
23                 pq.emplace(dist[v], v);  // Añadimos el nodo a la cola
24             }
25         }
26     }
27 }
```

Bipartitos

Grafos en los que los nodos se pueden dividir en dos conjuntos disjuntos de tal forma que no hay aristas entre nodos del mismo conjunto. Se utilizan para modelar relaciones entre dos tipos de entidades. MCBM (Maximum Cardinality Bipartite Matching) es un algoritmo que encuentra el emparejamiento máximo en un grafo bipartito.

```
1  typedef vector<int> vi;
2  typedef vector<vi> vvi;
3  typedef pair<int, int> ii;
4  typedef vector<ii> vii;
5
6  vi match, vis;
7  vvi AL;
8
9  int try_kuhn(int v) {
10     if (vis[v]) return 0;    // Si ya hemos visitado este nodo, no hacemos nada
11     vis[v] = 1;              // Marcamos el nodo como visitado
```

```
    for (auto &to : AL[v]) { // Recorremos los nodos derechos conectados
        if (match[to] == -1 ||     // Si el nodo derecho no está emparejado
            try_kuhn(match[to])) { // o podemos emparejarlo recursivamente
            match[to] = v; // Emparejamos el nodo derecho con el izquierdo
            return 1;      // Retornamos que hemos encontrado un emparejamiento
        }
    }

    return 0; // Si no hemos encontrado un emparejamiento, retornamos 0
}

int n, m; // n = nodos izquierdos, m = nodos derechos
int mcbm() {
    match.assign(m, -1); // Inicializamos el emparejamiento con -1
    int MCVM = 0;
    for (int v = 0; v < n; v++) {
        vis.assign(n, 0);    // Reiniciamos el vector de visitados
        MCVM += try_kuhn(v); // Intentamos emparejar el nodo izquierdo v
    }

    // En este punto, match contiene el emparejamiento máximo
    // match[i] = j significa que el nodo i de la parte izquierda está
    // emparejado con el nodo j de la parte derecha.
    return MCVM; // Retornamos el número máximo de emparejamientos
}
```

Max flow

Problema clásico de flujo máximo en un grafo. El algoritmo de Edmonds-Karp utiliza BFS para encontrar caminos aumentantes y calcular el flujo máximo. Si las capacidades de las aristas son enteros, el algoritmo termina en un tiempo polinómico, si son racionales la complejidad no está boundeada y con cantidades irracionales el algoritmo no funciona.

```
#define INF 1e9

int n;        // Nodos del grafo
vvi capacity; // capacity[i][j] = capacidad de la arista i -> j
vvi adj;      // adj[i] = lista de nodos adyacentes a i (grafo NO dirigido)

int bfs(int s, int t, vector<int> &parent) {
    fill(parent.begin(), parent.end(), -1);
    parent[s] = -2;
    queue<pair<int, int>> q;
    q.push({s, INF});

    while (!q.empty()) {
        int cur = q.front().first;
        int flow = q.front().second;
        q.pop();

        for (int next : adj[cur]) {
            if (parent[next] == -1 && capacity[cur][next]) {
                parent[next] = cur;
                int new_flow = min(flow, capacity[cur][next]);
                if (next == t) return new_flow;
                q.push({next, new_flow});
            }
        }
    }

    return 0;
}

// Flujo máximo usando el algoritmo de Edmonds-Karp
// desde el nodo source al nodo drain
int maxflow(int source, int drain) {
    int flow = 0; // Flujo total encontrado
    vi parent(n); // Vector para almacenar el camino encontrado por BFS
    int new_flow; // Flujo encontrado en cada iteración

    // Mientras haya un camino aumentante desde source a drain
    // se actualiza el flujo total y las capacidades de las aristas
    // en el camino encontrado
    while ((new_flow = bfs(source, drain, parent))) {
```

```
        flow += new_flow; // Aumenta el flujo total con el flujo encontrado
        int cur = drain;  // Recorremos el camino encontrado hacia atrás
        while (cur != source) { // actualizando las capacidades residuales
            int prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }

    return flow;
}
```

Árboles de segmentos

Árboles que permiten realizar consultas y actualizaciones en un rango de un array de forma eficiente. Útiles para problemas que piden realizar operaciones en rangos de un array, como sumar o encontrar el mínimo **y** además es necesario actualizar el array de manera frecuente.

```
#define SEGTREE_EMPTY INT_MIN // cuidao con esto

class SegmentTree {
private:
    int n;
    vi A, st;

    int l(int p) { return p << 1; }
    int r(int p) { return (p << 1) + 1; }

    int conquer(int a, int b) {
        if (a == SEGTREE_EMPTY) return b; // childs missing
        if (b == SEGTREE_EMPTY) return a;

        // Aquí va tu operación, la función que toma dos nodos consecutivos y
        // devuelve un valor. En este caso es multiplicar
        return a * b;
    }

    void build(int p, int L, int R) {
        if (L == R)
            st[p] = A[L]; // this is a leaf node
        else {
            int m = (L + R) / 2;
            build(l(p), L, m);                  // left
            build(r(p), m + 1, R);              // right
            st[p] = conquer(st[l(p)], st[r(p)]); // and conquer
        }
    }

    int rmq(int p, int L, int R, int i, int j) {
        if (i > j) return SEGTREE_EMPTY; // invalid range

        if ((L >= i) && (R <= j)) return st[p]; // this is the node we want

        int m = (L + R) / 2;
        return conquer(rmq(l(p), L, m, i, min(m, j)),          // conquer left
                       rmq(r(p), m + 1, R, max(i, m + 1), j)); // and right
    }

    void update(int p, int L, int R, int i, int val) {
        if (L == R) {
            st[p] = val; // this is a leaf node, update it
            return;
        }

        int m = (L + R) / 2;
        if (i <= m)
            update(l(p), L, m, i, val); // go left
        else
            update(r(p), m + 1, R, i, val); // go right

        st[p] = conquer(st[l(p)], st[r(p)]); // conquer to update the value
    }

```

```
public:
    SegmentTree(int size) : n(size), st(4 * n) {}

    SegmentTree(const vi &A) : SegmentTree(A.size()) {
        // A TIENE que tener tamaño potencia de 2
        assert(fmod(log2(A.size()), 1) == 0); // check útil para debuggear

        this->A = A;
        build(1, 0, n - 1);
    }

    // actualiza el valor en la posición i con el valor val
    void update(int i, int val) { update(1, 0, n - 1, i, val); }

    // devuelve el valor del rango [i, j]
    int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); }
};
```

Árbol de Fenwick

Hace algo

```
// https://cp-algorithms.com/data_structures/fenwick.html

struct FenwickTree {
    vector<int> bit; // binary indexed tree
    int n;

    FenwickTree(int n) {
        this->n = n;
        bit.assign(n, 0);
    }

    FenwickTree(vector<int> const &a) : FenwickTree(a.size()) {
        for (size_t i = 0; i < a.size(); i++)
            add(i, a[i]);
    }

    int sum(int r) {
        int ret = 0;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret += bit[r];
        return ret;
    }

    int sum(int l, int r) { return sum(r) - sum(l - 1); }

    void add(int idx, int delta) {
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] += delta;
    }
};
```

3. Mates

3.1. Primos

Unos cuantos primos muy grandes: $1e9+7$, $1e9+9$, $1e9+21$.

Eratóstenes

```cpp
typedef long long ll;
typedef vector<ll> vll;

ll _sieve_size;
bitset<10000010> bs; // 1e7 (más o menos el tope de lo que es viable)
vll p;               // Vector de primos

void sieve(ll upperbound) {
    _sieve_size = upperbound + 1; // +1 porque el límite es inclusivo
    bs.set();                     // Inicializar el bitset a 1
    bs[0] = bs[1] = 0;            // 0 y 1 no son primos
    for (ll i = 2; i <= _sieve_size; i++) {
        if (bs[i]) { // Si es primo
            for (ll j = i * i; j <= _sieve_size; j += i) {
                bs[j] = 0; // Marcar múltiplos de i como no primos
            }
            p.push_back(i); // Añadir i a la lista de primos
        }
    }
}

// Solo funciona para números menores que p.back()^2
bool is_prime(ll n) {
    if (n < _sieve_size) return bs[n]; // O(1) para números pequeños
    for (ll i = 0; i < p.size() && p[i] * p[i] <= n; i++) {
        if (n % p[i] == 0) return false; // Si es divisible por algún primo
    }
    return true; // Si no es divisible por ningún primo, es primo
}
```

Cosas con factores

```cpp
vll p; // Vector de primos

vll fac_primos(ll n) {
    vll res;
    for (ll i = 0; (i < (int)p.size()) && (p[i] * p[i] <= n); i++) {
        while (n % p[i] == 0) {
            res.push_back(p[i]);
            n /= p[i];
        }
    }

    if (n > 1) {
        res.push_back(n); // Si queda un factor primo mayor que sqrt(n)
    }
    return res;
}

// Número de primos de n
int numPF(ll n) {
    int ans = 0;
    for (int i = 0; (i < (int)p.size()) && (p[i] * p[i] <= n); i++) {
        while (n % p[i] == 0) {
            n /= p[i];
            ans++;
        }
    }
    return ans + (n > 1); // Si queda un factor primo mayor que sqrt(n)
}

// Número de factores primos distintos de n
int numDiffPF(ll n) {
    int ans = 0;
    for (int i = 0; (i < (int)p.size()) && (p[i] * p[i] <= n); i++) {
        if (n % p[i] == 0) ans++;
        while (n % p[i] == 0)
            n /= p[i];
```

```
37      }
38
39      return ans + (n > 1); // Si queda un factor primo mayor que sqrt(n)
40  }
41
42  // Número de divisores de n
43  int numDiv(ll n) {
44      int ans = 1;
45      for (int i = 0; (i < (int)p.size()) && (p[i] * p[i] <= n); i++) {
46          int cnt = 0;
47          while (n % p[i] == 0) {
48              n /= p[i];
49              cnt++;
50          }
51          ans *= (cnt + 1);
52      }
53
54      return n > 1 ? ans * 2 : ans; // Si queda un factor primo mayor que sqrt(n)
55  }
56
57  // Suma de divisores de n
58  ll sumDiv(ll n) {
59      ll ans = 1;
60      for (int i = 0; (i < (int)p.size()) && (p[i] * p[i] <= n); i++) {
61          ll multiplier = p[i], total = 1;
62          while (n % p[i] == 0) {
63              n /= p[i];
64              total += multiplier;
65              multiplier *= p[i];
66          }
67          ans *= total;
68      }
69
70      return (n > 1) ? ans * (n + 1)
71                     : ans; // Si queda un factor primo mayor que sqrt(n)
72  }
73
74  ll phi(ll n) {
75      ll ans = n;
76      for (int i = 0; (i < (int)p.size()) && (p[i] * p[i] <= n); i++) {
77          if (n % p[i] == 0) ans -= ans / p[i];
78          while (n % p[i] == 0)
79              n /= p[i];
80      }
81      return (n > 1) ? ans - ans / n
82                     : ans; // Si queda un factor primo mayor que sqrt(n)
83  }
```

3.2. Aritmética modular

EGCD

```
1   // Modulo positivo siempre
2   int mod(int a, int m) { return ((a % m) + m) % m; }
3
4   // Exponente modular: b^p (mod m)
5   // 0 <= b < m
6   int modPow(int b, int p, int m) {
7       if (p == 0) return 1;
8       int ans = modPow(b, p / 2, m);
9       ans = mod(ans * ans, m);
10      if (p & 1) ans = mod(ans * b, m);
11      return ans;
12  }
13
14  // Euclides extendido (devuelve gdc(a, b) y x, y tal que a*x + b*y == gdc(a, b))
15  int extEuclid(int a, int b, int &x, int &y) {
16      int xx = y = 0;
17      int yy = x = 1;
18      while (b) {
19          int q = a / b;
20          tie(a, b) = make_tuple(b, a % b);
21          tie(x, xx) = make_tuple(xx, x - q * xx);
22          tie(y, yy) = make_tuple(yy, y - q * yy);
```

```
    }
    return a;
}

// Inverso modular: b^(-1) (mod m)
int modInverse(int b, int m) {
    int x, y;
    int d = extEuclid(b, m, x, y);
    if (d != 1) return -1;
    return mod(x, m);
}
```

3.3. Combinatoria

Números catalanes

Los números catalanes cuentan un montón de cosas, como:

- $Cat(n)$ es el número de árboles binarios con n nodos.
- $Cat(n)$ el número de expresiones de paréntesis bien formadas con n pares de paréntesis.
- $Cat(n)$ el número de formas de dividir un polígono convexo de $n+2$ lados en triángulos.
- $Cat(n)$ el número de caminos de n pasos que no suben por encima de la diagonal en un plano.

Números catalanes, fibonacci y coeficientes binomiales

```
// Fibonacci
int fact_pow(int n, int k) {
    int res = 0;
    while (n) {
        n /= k;
        res += n;
    }
    return res;
}

// Binomial coefficient
int C(int n, int k) {
    int res = 1;
    for (int i = n - k + 1; i <= n; ++i)
        res *= i;
    for (int i = 2; i <= k; ++i)
        res /= i;
    return res;
}

// Binomial coefficient (puede que el resultado sea ligeramente menor que el
// real)
int C_rara(int n, int k) {
    double res = 1;
    for (int i = 1; i <= k; ++i)
        res = res * (n - k + i) / i;
    return (int)(res + 0.01);
}

// Catalan
const int MOD = 1000000007;
const int MAX = 100000;
int catalan[MAX];

void init_cat(int n) {
    catalan[0] = catalan[1] = 1;
    for (int i = 2; i <= n; i++) {
        catalan[i] = 0;
        for (int j = 0; j < i; j++) {
            catalan[i] += (catalan[j] * catalan[i - j - 1]) % MOD;
            if (catalan[i] >= MOD) {
                catalan[i] -= MOD;
            }
        }
    }
}
```

Nth Permutation con Factoradic

El orden de las permutaciones se puede calcular con un sistema factorádico, que es una representación de números enteros en base factorial. Cada dígito del número factorádico representa la cantidad de veces que se usa cada factorial en la descomposición del número. (Por ejemplo, el número 1010 en factorádico es 1*3! + 0*2! + 1*1! + 0*0! = 6 + 0 + 1 + 0 = 7). Se puede utilizar para encontrar la k-ésima permutación de un conjunto de elementos sin necesidad de generar todas las permutaciones (muy rápido con un árbol de fenwick).

```
1
2  typedef vector<int> vi;
3
4  vi factoradic(int n) {
5      vi factors;
6      n--;
7      for (int i = 1; i <= n; i++) {
8          factors.push_back(n % i);
9          n /= i;
10     }
11
12     reverse(factors.begin(), factors.end());
13     return factors;
14 }
15
16 vi nth_permutation(vi &factors) {
17     vi res;
18     res.reserve(factors.size());
19
20     vi nums(factors.size());
21     for (int i = 0; i < factors.size(); i++)
22         nums[i] = i + 1;
23
24     for (auto f : factors) {
25         int n = nums[f];
26         nums.erase(nums.begin() + f); // esto es muy lento
27         res.push_back(n);
28     }
29
30     return res;
31 }
```

3.4. Matrices

Ecuaciones

Se puede resolver un sistema de ecuaciones lineales utilizando el método de Gauss a partir de la matriz aumentada del sistema.

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1m}x_m = b_1 \tag{3.1}$$
$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2m}x_m = b_2 \tag{3.2}$$
$$\vdots \tag{3.3}$$
$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nm}x_m = b_n \tag{3.4}$$

```
1  // Resolver un sistema de ecuaciones lineales con el método de Gauss
2
3  const double EPS = 1e-9;
4  const int INF = 2; // it doesn't actually have to be infinity or a big number
5
6  // La entrada es la matriz aumentada del sistema de ecuaciones
7  int gauss(vector<vector<double>> a, vector<double> &ans) {
8      int n = (int)a.size();
9      int m = (int)a[0].size() - 1;
10
11     vector<int> where(m, -1);
12     for (int col = 0, row = 0; col < m && row < n; ++col) {
13         int sel = row;
14         for (int i = row; i < n; ++i)
15             if (abs(a[i][col]) > abs(a[sel][col])) sel = i;
16         if (abs(a[sel][col]) < EPS) continue;
17         for (int i = col; i <= m; ++i)
18             swap(a[sel][i], a[row][i]);
```

```
        where[col] = row;

        for (int i = 0; i < n; ++i)
            if (i != row) {
                double c = a[i][col] / a[row][col];
                for (int j = col; j <= m; ++j)
                    a[i][j] -= a[row][j] * c;
            }
        ++row;
    }

    ans.assign(m, 0);
    for (int i = 0; i < m; ++i)
        if (where[i] != -1) ans[i] = a[where[i]][m] / a[where[i]][i];
    for (int i = 0; i < n; ++i) {
        double sum = 0;
        for (int j = 0; j < m; ++j)
            sum += ans[j] * a[i][j];
        if (abs(sum - a[i][m]) > EPS) return 0;
    }

    for (int i = 0; i < m; ++i)
        if (where[i] == -1) return INF;
    return 1;
}
```

Rank

El rango de una matriz es el número máximo de columnas linealmente independientes. Se puede calcular utilizando el método de eliminación de Gauss.

```
const double EPS = 1E-9;

int compute_rank(vector<vector<double>> A) {
    int n = A.size();
    int m = A[0].size();

    int rank = 0;
    vector<bool> row_selected(n, false);
    for (int i = 0; i < m; ++i) {
        int j;
        for (j = 0; j < n; ++j) {
            if (!row_selected[j] && abs(A[j][i]) > EPS) break;
        }

        if (j != n) {
            ++rank;
            row_selected[j] = true;
            for (int p = i + 1; p < m; ++p)
                A[j][p] /= A[j][i];
            for (int k = 0; k < n; ++k) {
                if (k != j && abs(A[k][i]) > EPS) {
                    for (int p = i + 1; p < m; ++p)
                        A[k][p] -= A[j][p] * A[k][i];
                }
            }
        }
    }
    return rank;
}
```

Determinante

No te lo vas a creer, pero el determinante de una matriz se puede calcular utilizando un método que tiene un nombre que se parece a Mouse.

```
const double EPS = 1E-9;
int n;
vector<vector<double>> a(n, vector<double>(n));

double get_det() {
    double det = 1;
    for (int i = 0; i < n; ++i) {
        int k = i;
```

```
        for (int j = i + 1; j < n; ++j)
            if (abs(a[j][i]) > abs(a[k][i])) k = j;
        if (abs(a[k][i]) < EPS) {
            det = 0;
            break;
        }
        swap(a[i], a[k]);
        if (i != k) det = -det;
        det *= a[i][i];
        for (int j = i + 1; j < n; ++j)
            a[i][j] /= a[i][i];
        for (int j = 0; j < n; ++j)
            if (j != i && abs(a[j][i]) > EPS)
                for (int k = i + 1; k < n; ++k)
                    a[j][k] -= a[i][k] * a[j][i];
    }

    return det;
}
```

Multiplicación

```
typedef long long ll;

const ll MOD = 1e9 + 7; // Módulo para las operaciones (primo grande)
const int MAXN = 2;     // Tamaño máximo de la matriz
struct Matrix {
    ll mat[MAXN][MAXN];
};

// Asegura que el resultado sea positivo
ll mod(ll a, ll b) { return (a % b + b) % b; }

// Multiplica dos matrices
Matrix mat_mul(const Matrix &a, const Matrix &b) {
    Matrix ans;

    // Inicializa la matriz resultado a cero
    for (int i = 0; i < MAXN; i++)
        for (int j = 0; j < MAXN; j++)
            ans.mat[i][j] = 0;

    // Realiza la multiplicación de matrices
    for (int i = 0; i < MAXN; i++) {
        for (int k = 0; k < MAXN; k++) {
            if (a.mat[i][k] == 0) continue;
            for (int j = 0; j < MAXN; j++) {
                ans.mat[i][j] += mod(a.mat[i][k], MOD) * mod(b.mat[k][j], MOD);
                ans.mat[i][j] = mod(ans.mat[i][j], MOD);
            }
        }
    }

    return ans;
}
```

4. Geometría (no mates)

Evitamos los flotantes a toda costa en esta casa

Punto genérico

```
struct point {
    int x, y;
    point() : x(0), y(0) {}
    point(int x, int y) : x(x), y(y) {}

    bool operator==(const point &p) const { return x == p.x && y == p.y; }

    // bool operator==(const point &p) const { return (fabs(x - p.x) < EPS) &&
    // (fabs(y - p.y) < EPS); }

    point operator+(const point &p) const { return point(x + p.x, y + p.y); }
    point operator-(const point &p) const { return point(x - p.x, y - p.y); }
    point operator*(int k) const { return point(x * k, y * k); }
    point operator/(int k) const { return point(x / k, y / k); }

    bool operator<(const point &p) const {
        if (x != p.x) return x < p.x;
        return y < p.y;
    }

    double dist(const point &p) const { return hypot(x - p.x, y - p.y); }
};

int dot(point a, point b) { return a.x * b.x + a.y * b.y; }
int norm(point a) { return dot(a, a); }
double abs(point a) { return sqrt(norm(a)); }
double proj(point a, point b) { return dot(a, b) / abs(b); }
double angle(point a, point b) { return acos(dot(a, b) / abs(a) / abs(b)); }
int cross(point a, point b) { return a.x * b.y - a.y * b.x; }

point lines_intersect(point a1, point d1, point a2, point d2) {
    return a1 + d1 * (cross(a2 - a1, d2) / cross(d1, d2));
}
```

Líneas

```
// https://cp-algorithms.com/geometry/lines-intersection.html
struct pt {
    double x, y;
};

struct line {
    double a, b, c;
};

const double EPS = 1e-9;

double det(double a, double b, double c, double d) { return a * d - b * c; }

bool intersect(line m, line n, pt &res) {
    double zn = det(m.a, m.b, n.a, n.b);
    if (abs(zn) < EPS) return false;
    res.x = -det(m.c, m.b, n.c, n.b) / zn;
    res.y = -det(m.a, m.c, n.a, n.c) / zn;
    return true;
}

bool parallel(line m, line n) { return abs(det(m.a, m.b, n.a, n.b)) < EPS; }

bool equivalent(line m, line n) {
    return abs(det(m.a, m.b, n.a, n.b)) < EPS &&
           abs(det(m.a, m.c, n.a, n.c)) < EPS &&
           abs(det(m.b, m.c, n.b, n.c)) < EPS;
}
```

```
// https://cp-algorithms.com/geometry/check-segments-intersection.html
struct pt {
    long long x, y;
    pt() {}
```

```
    pt(long long _x, long long _y) : x(_x), y(_y) {}
    pt operator-(const pt &p) const { return pt(x - p.x, y - p.y); }
    long long cross(const pt &p) const { return x * p.y - y * p.x; }
    long long cross(const pt &a, const pt &b) const {
        return (a - *this).cross(b - *this);
    }
};

int sgn(const long long &x) { return x >= 0 ? x ? 1 : 0 : -1; }

bool inter1(long long a, long long b, long long c, long long d) {
    if (a > b) swap(a, b);
    if (c > d) swap(c, d);
    return max(a, c) <= min(b, d);
}

bool check_inter(const pt &a, const pt &b, const pt &c, const pt &d) {
    if (c.cross(a, d) == 0 && c.cross(b, d) == 0)
        return inter1(a.x, b.x, c.x, d.x) && inter1(a.y, b.y, c.y, d.y);
    return sgn(a.cross(b, c)) != sgn(a.cross(b, d)) &&
           sgn(c.cross(d, a)) != sgn(c.cross(d, b));
}
```

```
// https://cp-algorithms.com/geometry/segments-intersection.html
const double EPS = 1E-9;

struct pt {
    double x, y;

    bool operator<(const pt &p) const {
        return x < p.x - EPS || (abs(x - p.x) < EPS && y < p.y - EPS);
    }
};

struct line {
    double a, b, c;

    line() {}
    line(pt p, pt q) {
        a = p.y - q.y;
        b = q.x - p.x;
        c = -a * p.x - b * p.y;
        norm();
    }

    void norm() {
        double z = sqrt(a * a + b * b);
        if (abs(z) > EPS) a /= z, b /= z, c /= z;
    }

    double dist(pt p) const { return a * p.x + b * p.y + c; }
};

double det(double a, double b, double c, double d) { return a * d - b * c; }

inline bool betw(double l, double r, double x) {
    return min(l, r) <= x + EPS && x <= max(l, r) + EPS;
}

inline bool intersect_1d(double a, double b, double c, double d) {
    if (a > b) swap(a, b);
    if (c > d) swap(c, d);
    return max(a, c) <= min(b, d) + EPS;
}

bool intersect(pt a, pt b, pt c, pt d, pt &left, pt &right) {
    if (!intersect_1d(a.x, b.x, c.x, d.x) || !intersect_1d(a.y, b.y, c.y, d.y))
        return false;
    line m(a, b);
    line n(c, d);
    double zn = det(m.a, m.b, n.a, n.b);
    if (abs(zn) < EPS) {
        if (abs(m.dist(c)) > EPS || abs(n.dist(a)) > EPS) return false;
```

```
51          if (b < a) swap(a, b);
52          if (d < c) swap(c, d);
53          left = max(a, c);
54          right = min(b, d);
55          return true;
56      } else {
57          left.x = right.x = -det(m.c, m.b, n.c, n.b) / zn;
58          left.y = right.y = -det(m.a, m.c, n.a, n.c) / zn;
59          return betw(a.x, b.x, left.x) && betw(a.y, b.y, left.y) &&
60                 betw(c.x, d.x, left.x) && betw(c.y, d.y, left.y);
61      }
62  }
```

4.1. Polígonos simples

Area de triángulos y polígonos

```
1  int signed_area_parallelogram(point p1, point p2, point p3) {
2      return cross(p2 - p1, p3 - p2);
3  }
4
5  double triangle_area(point p1, point p2, point p3) {
6      return abs(signed_area_parallelogram(p1, p2, p3)) / 2.0;
7  }
8
9  bool clockwise(point p1, point p2, point p3) {
10     return signed_area_parallelogram(p1, p2, p3) < 0;
11 }
12
13 bool counter_clockwise(point p1, point p2, point p3) {
14     return signed_area_parallelogram(p1, p2, p3) > 0;
15 }
```

```
1  double area(const vector<point> &fig) {
2      double res = 0;
3      for (unsigned i = 0; i < fig.size(); i++) {
4          point p = i ? fig[i - 1] : fig.back();
5          point q = fig[i];
6          res += (p.x - q.x) * (p.y + q.y);
7      }
8      return fabs(res) / 2;
9  }
```

Comprobar si un punto está en un polígono

```
1  struct pt {
2      long long x, y;
3      pt() {}
4      pt(long long _x, long long _y) : x(_x), y(_y) {}
5      pt operator+(const pt &p) const { return pt(x + p.x, y + p.y); }
6      pt operator-(const pt &p) const { return pt(x - p.x, y - p.y); }
7      long long cross(const pt &p) const { return x * p.y - y * p.x; }
8      long long dot(const pt &p) const { return x * p.x + y * p.y; }
9      long long cross(const pt &a, const pt &b) const {
10         return (a - *this).cross(b - *this);
11     }
12     long long dot(const pt &a, const pt &b) const {
13         return (a - *this).dot(b - *this);
14     }
15     long long sqrLen() const { return this->dot(*this); }
16 };
17
18 bool lexComp(const pt &l, const pt &r) {
19     return l.x < r.x || (l.x == r.x && l.y < r.y);
20 }
21
22 int sgn(long long val) { return val > 0 ? 1 : (val == 0 ? 0 : -1); }
23
24 vector<pt> seq;
25 pt translation;
26 int n;
27
28 bool pointInTriangle(pt a, pt b, pt c, pt point) {
```

```
    long long s1 = abs(a.cross(b, c));
    long long s2 = abs(point.cross(a, b)) + abs(point.cross(b, c)) +
                   abs(point.cross(c, a));
    return s1 == s2;
}

void prepare(vector<pt> &points) {
    n = points.size();
    int pos = 0;
    for (int i = 1; i < n; i++) {
        if (lexComp(points[i], points[pos])) pos = i;
    }
    rotate(points.begin(), points.begin() + pos, points.end());

    n--;
    seq.resize(n);
    for (int i = 0; i < n; i++)
        seq[i] = points[i + 1] - points[0];
    translation = points[0];
}

bool pointInConvexPolygon(pt point) {
    point = point - translation;
    if (seq[0].cross(point) != 0 &&
        sgn(seq[0].cross(point)) != sgn(seq[0].cross(seq[n - 1])))
        return false;
    if (seq[n - 1].cross(point) != 0 &&
        sgn(seq[n - 1].cross(point)) != sgn(seq[n - 1].cross(seq[0])))
        return false;

    if (seq[0].cross(point) == 0) return seq[0].sqrLen() >= point.sqrLen();

    int l = 0, r = n - 1;
    while (r - l > 1) {
        int mid = (l + r) / 2;
        int pos = mid;
        if (seq[pos].cross(point) >= 0)
            l = mid;
        else
            r = mid;
    }
    int pos = l;
    return pointInTriangle(seq[pos], seq[pos + 1], pt(0, 0), point);
}
```

4.2. Convex hull

```
// https://cp-algorithms.com/geometry/convex-hull.html

struct pt {
    double x, y;
    bool operator==(pt const &t) const { return x == t.x && y == t.y; }
};

int orientation(pt a, pt b, pt c) {
    double v = a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y - b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}
bool collinear(pt a, pt b, pt c) { return orientation(a, b, c) == 0; }

void convex_hull(vector<pt> &a, bool include_collinear = false) {
    pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    sort(a.begin(), a.end(), [&p0](const pt &a, const pt &b) {
        int o = orientation(p0, a, b);
        if (o == 0)
```

```
            return (p0.x - a.x) * (p0.x - a.x) + (p0.y - a.y) * (p0.y - a.y) <
                   (p0.x - b.x) * (p0.x - b.x) + (p0.y - b.y) * (p0.y - b.y);
        return o < 0;
    });
    if (include_collinear) {
        int i = (int)a.size() - 1;
        while (i >= 0 && collinear(p0, a[i], a.back()))
            i--;
        reverse(a.begin() + i + 1, a.end());
    }

    vector<pt> st;
    for (int i = 0; i < (int)a.size(); i++) {
        while (st.size() > 1 &&
               !cw(st[st.size() - 2], st.back(), a[i], include_collinear))
            st.pop_back();
        st.push_back(a[i]);
    }

    if (include_collinear == false && st.size() == 2 && st[0] == st[1])
        st.pop_back();

    a = st;
}
```

4.3. Sweep line

```
// https://cp-algorithms.com/geometry/intersecting_segments.html

const double EPS = 1E-9;

struct pt {
    double x, y;
};

struct seg {
    pt p, q;
    int id;

    double get_y(double x) const {
        if (abs(p.x - q.x) < EPS) return p.y;
        return p.y + (q.y - p.y) * (x - p.x) / (q.x - p.x);
    }
};

bool intersect1d(double l1, double r1, double l2, double r2) {
    if (l1 > r1) swap(l1, r1);
    if (l2 > r2) swap(l2, r2);
    return max(l1, l2) <= min(r1, r2) + EPS;
}

int vec(const pt &a, const pt &b, const pt &c) {
    double s = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
    return abs(s) < EPS ? 0 : s > 0 ? +1 : -1;
}

bool intersect(const seg &a, const seg &b) {
    return intersect1d(a.p.x, a.q.x, b.p.x, b.q.x) &&
           intersect1d(a.p.y, a.q.y, b.p.y, b.q.y) &&
           vec(a.p, a.q, b.p) * vec(a.p, a.q, b.q) <= 0 &&
           vec(b.p, b.q, a.p) * vec(b.p, b.q, a.q) <= 0;
}

bool operator<(const seg &a, const seg &b) {
    double x = max(min(a.p.x, a.q.x), min(b.p.x, b.q.x));
    return a.get_y(x) < b.get_y(x) - EPS;
}

struct event {
    double x;
    int tp, id;

    event() {}
    event(double x, int tp, int id) : x(x), tp(tp), id(id) {}
```

```
48
49      bool operator<(const event &e) const {
50          if (abs(x - e.x) > EPS) return x < e.x;
51          return tp > e.tp;
52      }
53  };
54
55  set<seg> s;
56  vector<set<seg>::iterator> where;
57
58  set<seg>::iterator prev(set<seg>::iterator it) {
59      return it == s.begin() ? s.end() : --it;
60  }
61
62  set<seg>::iterator next(set<seg>::iterator it) { return ++it; }
63
64  pair<int, int> solve(const vector<seg> &a) {
65      int n = (int)a.size();
66      vector<event> e;
67      for (int i = 0; i < n; ++i) {
68          e.push_back(event(min(a[i].p.x, a[i].q.x), +1, i));
69          e.push_back(event(max(a[i].p.x, a[i].q.x), -1, i));
70      }
71      sort(e.begin(), e.end());
72
73      s.clear();
74      where.resize(a.size());
75      for (size_t i = 0; i < e.size(); ++i) {
76          int id = e[i].id;
77          if (e[i].tp == +1) {
78              set<seg>::iterator nxt = s.lower_bound(a[id]), prv = prev(nxt);
79              if (nxt != s.end() && intersect(*nxt, a[id]))
80                  return make_pair(nxt->id, id);
81              if (prv != s.end() && intersect(*prv, a[id]))
82                  return make_pair(prv->id, id);
83              where[id] = s.insert(nxt, a[id]);
84          } else {
85              set<seg>::iterator nxt = next(where[id]), prv = prev(where[id]);
86              if (nxt != s.end() && prv != s.end() && intersect(*nxt, *prv))
87                  return make_pair(prv->id, nxt->id);
88              s.erase(where[id]);
89          }
90      }
91
92      return make_pair(-1, -1);
93  }
```

5. Otras cosas

5.1. Plantillas

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

/*
%Descripcion%
*/

typedef long long ll;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;

void casoDePrueba()
{
    // Aquí va todo
}

int main()
{
    ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);

    unsigned int numCasos;

    cin >> numCasos;
    for (unsigned int i = 0; i < numCasos; ++i)
    {
        casoDePrueba();
    }

    return 0;
}
```

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

/*
%Descripcion%
*/

typedef long long ll;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;

bool casoDePrueba()
{
    /*
     * leer caso de prueba
     * si es el final -> devolver false
     * hacer todo
     * devolver true
     */

    return true;
}

int main()
{
    ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);

    while(casoDePrueba());

    return 0;
}
```

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

```
4
5  /*
6  %Descripcion%
7  */
8
9  typedef long long ll;
10 typedef pair<int, int> ii;
11 typedef vector<int> vi;
12 typedef vector<ii> vii;
13
14 bool casoDePrueba()
15 {
16     // leer el inicio del caso de prueba (cin)
17     if (!cin) return false;
18
19     // CÓDIGO PRINCIPAL AQUÍ (incluyendo el resto de la lectura)
20
21     return true;
22 }
23
24 int main()
25 {
26     ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);
27
28     while(casoDePrueba());
29
30     return 0;
31 }
```