

Índice

1. Setup	2
2. STL	2
2.1. Vector	2
2.2. Sets	3
2.3. Estructuras policy based (set indexado + prefix trie)	5
2.4. Misc.	5
3. Range queries	5
3.1. Prefix y diff array	5
3.2. Sparse table	6
3.3. Segment tree	6
3.4. Fenwick tree	7
3.5. MO	7
4. Grafos	8
4.1. Toposort de un DAG	8
4.2. DAG condensado	8
4.3. Bipartite check	8
4.4. Encontrar puentes y articulaciones	8
4.5. Matching maximo bipartito (Kuhn)	9
4.6. Menores caminos	9
5. Programacion Dinamica	10
5.1. LIS (Longest Increasing Subsequence)	10
5.2. Menor ciclo hamiltoniano	11
6. Matemática	11
6.1. Aritmética	11
6.2. Teoria de numeros	12
7. Geometria	12
7.1. Template: floats, punto	12
7.2. Recta: Interseccion, Comparacion	13
7.3. Circulo: Interseccion	13
7.4. Misc: Triangulo, Poligono, Convex Hull	13
8. Estructuras locas	14
8.1. Disjoint set union	14
8.2. Binary trie	14
9. Sin categorizar	15
9.1. Búsqueda binaria sobre un predicado	15
9.2. Enumerar subconjuntos de un conjuto con bitmask	15
9.3. Hashing Rabin Karp	15

9.4. Lowest common ancestor (LCA)	15
9.5. Euler tour	16
10.Brainstorming	16

1. Setup

Template corto

```
#include <bits/stdc++.h>
using namespace std;
#define forr(i,a,b) for(int i = int(a); i < int(b); i++)
#define forn(i,n) forr(i,0,n)
#define all(v) begin(v), end(v)
#define mt(...) make_tuple(__VA_ARGS__)
#define sz(v) int(v.size())

int main (int argc, char** argv) { if (argc == 2) freopen(argv[1], "r
    ↪ ", stdin);

    return 0;
}
```

run.sh: Compilar y ejecutar \$1 con archivo input opcional \$2

```
clear
make -s $1 && ./ $1 $2
```

Makefile

```
CC = g++
CPPFLAGS = -Wall -g \
-fsanitize=undefined -fsanitize=bounds \
-std=c++17 -O0
```

compilar.sh: Compilar \$1 y mostrar primeras \$2 líneas de error

```
clear
make -s $1 2>&1 | head -$2
```

Template completo

```
#pragma region // (vscode; en otros editores usar #if 1 / #endif)
#include <bits/stdc++.h>
using namespace std;
#define forr(i,a,b) for(int i = int(a); i < int(b); i++)
#define forn(i,n) forr(i,0,n)
#define all(v) begin(v), end(v)
#define mt(...) make_tuple(__VA_ARGS__)
#define sz(v) int(v.size())
#define pb push_back
#define fst first
#define snd second
#define endl '\n'
#define dprint(x) cerr << #x << " = " << (x) << endl
#define raya cerr << "===== " << endl
#define templT template <class T>
```

```
#define templAB template <class A, class B>
using ll = long long;
using Grafo = vector<vector<int>>>;
#pragma endregion

int main (int argc, char** argv) {
    ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
    if (argc == 2) freopen(argv[1], "r", stdin);

    return 0;
}
```

2. STL

2.1. Vector

Búsqueda binaria - Primer igual

```
templT int primer_igual (vector<T>& arr, T x) {
    auto it = lower_bound(all(arr), x);
    if (it == arr.end() || *it != x) return -1;
    return it - arr.begin();
}
```

Búsqueda binaria - Último igual

```
templT int ultimo_igual (vector<T>& arr, T x) {
    if (arr.begin() == arr.end()) return -1;
    auto it = prev(upper_bound(all(arr), x));
    if (*it != x) return -1;
    return it - arr.begin();
}
```

Búsqueda binaria - Primer mayor

```
templT int primer_mayor (vector<T>& arr, T x) {
    auto it = upper_bound(all(arr), x);
    if (it == arr.end()) return -1;
    return it - arr.begin();
}
```

Búsqueda binaria - Último menor

```
templT int ultimo_menor (vector<T>& arr, T x) {
    if (arr.begin() == arr.end()) return -1;
    auto it = prev(lower_bound(all(arr), x));
    if (*it >=) return -1;
    return it - arr.begin();
}
```

Funciones que modifican rangos

Función	Params	Ejemplo
copy	first last result	B.resize(A.size()); copy(all(A), B)
fill	first last val	memo.resize(MAXN); fill(all(memo), -1)
rotate	first middle last	rotate(begin(A), begin(A) + 3, end(A));

Operaciones de conjuntos con vectors ordenados (lineal)

// Siempre hacer resize al final asi:

```
vector<int> A = { 5, 10, 15, 20, 25};
vector<int> B = {10, 20, 30, 40, 50};
```

```
vector<int> U(A.size() + B.size());
```

```
auto it = set_union(all(A), all(B), begin(U));
```

```
U.resize(it - U.begin());
```

Función	Descripción
set_union	Unión
set_intersection	Intersección
set_difference	Elementos que están en el primero y no en el segundo
set_symmetric_difference	Elementos que están en uno pero no los dos (como el xor)

2.2. Sets

Multiset

```
struct Multiset {
    map<int, int> ocurrencias;
    void insertar (int x) { ocurrencias[x]++; }
    void eliminar (int x) {
        auto it = ocurrencias.find(x);
        assert(it != ocurrencias.end());
        if (--it->snd == 0) ocurrencias.erase(it);
    }
    int count (int x) {
        auto it = ocurrencias.find(x);
        return it == ocurrencias.end() ? 0 : it->snd;
    }
    int min (void) {
        assert(ocurrencias.size());
        return ocurrencias.begin()->fst;
    }
    int max (void) {
        assert(ocurrencias.size());
        return ocurrencias.rbegin()->fst;
    }
}
```

```
int primer_mayor (int x) {
    assert(x < max());
    return ocurrencias.upper_bound(x)->fst;
}
int primer_menor (int x) {
    assert(min() < x);
    return prev(ocurrencias.lower_bound(x))->fst;
}
};
```

Intervalos consecutivos (simular cortes en un palito)

```
struct IntervalosConsecutivos {
    set<int> I;
    map<int, int> L;
    IntervalosConsecutivos (int i, int j) {
        I.insert(i);
        I.insert(j);
        L[j - i]++;
    }
    void cortar (int k) {
        int i = *prev(I.lower_bound(k));
        int j = *(I.lower_bound(k));
        L[j - i]--;
        if (L[j - i] == 0) L.erase(j - i);
        L[k - i]++;
        L[j - k]++;
        I.insert(k);
    }
    int max_intervalo () {
        return (*L.rbegin()).fst;
    }
    int min_intervalo () {
        return (*L.begin()).fst;
    }
};
```

Multiset de diferencias

```
struct MultisetDiferencias {
    Multiset elementos, diferencias;
    void insertar (int x) {
        if (elementos.count(x)) {
            elementos.insertar(x);
            diferencias.insertar(0);
            return;
        }
        elementos.insertar(x);
        int m = elementos.min(), M = elementos.max();
        int a, b;
        if (x < M) {
            b = elementos.primer_mayor(x);
```

```

        diferencias.insertar(b - x);
    }
    if (m < x) {
        a = elementos.primer_menor(x);
        diferencias.insertar(x - a);
    }
    if (m < x && x < M) diferencias.eliminar(b - a);
}
void eliminar (int x) {
    assert(elementos.count(x));
    if (elementos.count(x) > 1) {
        elementos.eliminar(x);
        diferencias.eliminar(0);
        return;
    }
    int m = elementos.min(), M = elementos.max();
    elementos.eliminar(x);
    int a, b;
    if (x < M) {
        b = elementos.primer_mayor(x);
        diferencias.eliminar(b - x);
    }
    if (m < x) {
        a = elementos.primer_menor(x);
        diferencias.eliminar(x - a);
    }
    if (m < x && x < M) diferencias.insertar(b - a);
}
};

```

Array enlazado

```

#pragma region // ArrayEnlazado
struct ArrayEnlazado {
    // ocurrencias[x] := { i | arr[i] = x }
    // prevA[i] := max j tq. j < i && arr[j] = arr[i] (def. -1)
    // nextA[i] := min j tq. i < j && arr[i] = arr[j] (def. sz(arr))
    map<int, set<int>> ocurrencias;
    vector<int> arr;
    vector<int> prevA;
    vector<int> nextA;
    void quitar (int i) {
        int x = arr[i];
        int ia = prevA[i], ib = nextA[i];
        if (ia != -1) nextA[ia] = ib;
        if (ib != sz(arr)) prevA[ib] = ia;
        ocurrencias[x].erase(i);
        prevA[i] = -1, nextA[i] = sz(arr);
    }
    void agregar (int i, int x) {
        arr[i] = x;

```

```

        if (ocurrencias[x].empty()) {
            ocurrencias[x].insert(i);
            return;
        }
        int m = *ocurrencias[x].begin();
        int M = *ocurrencias[x].rbegin();
        ocurrencias[x].insert(i);
        if (m < i) {
            int ia = *prev(ocurrencias[x].lower_bound(i));
            nextA[ia] = i;
            prevA[i] = ia;
        }
        if (i < M) {
            int ib = *ocurrencias[x].upper_bound(i);
            prevA[ib] = i;
            nextA[i] = ib;
        }
    }
    void update (int i, int x) { quitar(i); agregar(i, x); }
};

auto ArrayEnlazado_crear (vector<int>& A) {
    auto arr = A;
    vector<int> prevA(sz(A), -1);
    vector<int> nextA(sz(A), sz(A));
    ArrayEnlazado AE = {{}}, arr, prevA, nextA;
    forn(i, sz(A)) AE.agregar(i, A[i]);
    return AE;
}
#pragma endregion

```

Suma K nums mas grandes

```

struct SumaKNumsMasGrandes {
    ll k = 0 / 0, sum = 0 / 0;
    priority_queue<ll, vector<ll>, greater<ll>> minheap;
    void insertar (ll x) {
        if (sz(minheap) < k) {
            __insertar(x);
            return;
        }
        ll m = minheap.top();
        if (m < x) {
            minheap.pop();
            sum -= m;
            __insertar(x);
        }
    }
    void __insertar (ll x) {
        minheap.push(x);
        sum += x;

```

```
    }
};
```

2.3. Estructuras policy based (set indexado + prefix trie)

Set indexado

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

templT struct SetIndexado {
    tree<
        T, null_type, less<T>,
        rb_tree_tag, tree_order_statistics_node_update
    > s;
    void add (T x) { s.insert(x); }
    int idx (T x) { return s.order_of_key(x); }
    bool has (T x) { return s.find(x) != ms.end(); }
    T ith (int i) { return *s.find_by_order(i); }
};
```

Multiset indexado

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

templT struct MultisetIndexado {
    int t = 0; tree<
        pair<T, int>, null_type, less<pair<T, int>>,
        rb_tree_tag, tree_order_statistics_node_update
    > ms;
    void add (T x) { ms.insert(mp(x, t++)); }
    int nle (T x) { return ms.order_of_key(mp(x, -1)); }
    int nleq (T x) { return ms.order_of_key(mp(x, INT_MAX)); }
    int cnt (T x) { return nleq(x) - nle(x); }
    T ith (int i) { return (*ms.find_by_order(i)).fst; }
};
```

Prefix trie

```
using Trie = trie<string, null_type, trie_string_access_traits<>,
    ↪ pat_trie_tag, trie_prefix_search_node_update>;

bool string_existe (Trie& T, string& s) {
    auto [from, to] = T.prefix_range(x);
    return from != to && *from == x;
}

void print_prefix_matches (Trie& T, string& prefix) {
    auto [from, to] = T.prefix_range(prefix);
```

```
    for (auto it = from; it != to; it++) cout << *it << ' ';
    cout << endl;
}
```

2.4. Misc.

Compresión de coordenadas para números enteros (lower_bound)

```
// Obtener valor original con D[A[i]]
vector<ll> CompCoordenadas (vector<ll>& A) {
    vector<ll> D = A;
    sort(all(D));
    D.resize(unique(all(D)) - D.begin());
    forn(i, sz(A)) A[i] = lower_bound(all(D), A[i]) - D.begin();
    return D;
}
```

Compresión de coordenadas genérica (map)

```
templT map<T, int> CompCoordenadas (vector<T>& A) {
    map<T, int> ord;
    int n = 0;
    for (auto v : A) ord[v];
    for (auto& e : ord) e.snd = n++;
    return ord;
}
```

3. Range queries

3.1. Prefix y diff array

Prefix array

```
#pragma region // prefix_array
templT vector<T> prefix_array_crear (vector<T>& A) {
    vector<T> P(sz(A));
    P[0] = A[0];
    forn(i, sz(P) - 1) P[i+1] = P[i] + A[i+1];
    return P;
}

templT T prefix_array_query (vector<T>& P, int i, int j) {
    T res = P[j];
    if (i > 0) res -= P[i-1];
    return res;
}
#pragma endregion
```

Diff array

```
#pragma region // diff_array
templT vector<T> diff_array_crear (vector<T>& A) {
    vector<T> D(sz(A));
    D[0] = A[0];
    forn(i, sz(D) - 1) D[i+1] = A[i+1] - A[i];
    return D;
}

// Aplica +x en A[i] ... A[j]
templT void diff_array_incrementar (vector<T>& D, int i, int j, T x)
    ↪ {
    D[i] += x;
    if (j + 1 < sz(D)) D[j+1] -= x;
}
#pragma endregion
```

3.2. Sparse table

RMQ 1D

```
#pragma region // SparseTable
using SparseTable = vector<vector<int>>>;
SparseTable SparseTable_crear (vector<int>& A) {
    SparseTable res(20, vector<int>(sz(A)));
    res[0] = A;
    forr(w, 1, 20) forn(i, sz(A) - (1 << w) + 1)
        res[w][i] = min(res[w-1][i], res[w-1][i + (1 << (w - 1))]);
    return res;
}
int SparseTable_query (SparseTable& S, int i, int j) {
    int w = 63 - __builtin_clzll(j - i + 1);
    return min(S[w][i], S[w][j - (1 << w) + 1]);
}
#pragma endregion
```

3.3. Segment tree

Range max query

```
#pragma region // Segtree max query
struct SegtreeRMQ {
    vector<int> tree;
    int get (int i) { return tree[i + sz(tree) / 2]; }
    void update (int i, int x) {
        for (tree[i += sz(tree) / 2] = x; i > 1; i /= 2)
            tree[i / 2] = max(tree[i], tree[i ^ 1]);
    }
    int query (int i, int j) {
        int res = INT_MIN;
        int l = i + sz(tree) / 2;
```

```
        int r = j + sz(tree) / 2 + 1;
        for (; l < r; l /= 2, r /= 2) {
            if (l & 1) res = max(res, tree[l++]);
            if (r & 1) res = max(res, tree[--r]);
        } return res;
    }
};
```

```
auto SegtreeRMQ_crear (vector<int>& A) {
    vector<int> tree(2 * sz(A), INT_MIN);
    forn(i, sz(A)) tree[i + sz(A)] = A[i];
    for (int i = sz(A) - 1; i; i--)
        tree[i] = max(tree[2*i], tree[2*i + 1]);
    return SegtreeRMQ { tree };
}
#pragma endregion
```

Segment tree generico

```
struct STNode {
    // Completar
};

STNode operator * (STNode a, STNode b) {
    // Completar
}

const STNode ST_ID = {
    // Completar
}

using STree = vector<STNode>;
STree segtree_build (STree& hojas) {
    int N = hojas.size();
    STree S(N << 1);
    forn(i, N) S[i + N] = hojas[i];
    for (int i = N - 1; i; i--) S[i] = S[i << 1] * S[i << 1 | 1];
    return S;
}

STNode segtree_query (STree& S, int i, int j) {
    int N = S.size() >> 1;
    STNode res = ST_ID;
    int l = i + N;
    int r = j + N + 1;
    for (; l < r; l >>= 1, r >>= 1) {
        if (l & 1) res = res * S[l++];
        if (r & 1) res = res * S[--r];
    }
    return res;
}
```

```
void segtree_update (STree& S, int i, STNode x) {
    int N = S.size() >> 1;
    S[i += N] = x;
    for (; i > 1; i >>= 1) S[i >> 1] = S[i] * S[i ^ 1];
}
```

3.4. Fenwick tree

Suma en rango, actualizar en punto

```
#pragma region // Fenwick
using Fenwick = unordered_map<int, ll>;
ll FT_prefix (Fenwick& A, int i) {
    ll res = 0;
    for (int j = i; j >= 0; j = j & (j + 1), j--) res += A[j];
    return res;
}
void FT_add (Fenwick& A, int N, int i, ll x) {
    for (; i < N; i = i | (i + 1)) A[i] += x;
}
ll FT_sum (Fenwick& A, int i, int j) {
    return FT_prefix(A, j) - FT_prefix(A, i - 1);
}
void FT_set (Fenwick& A, int N, int i, ll x) {
    FT_add(A, N, i, - FT_sum(A, i, i));
    FT_add(A, N, i, + x);
}
#pragma endregion
```

Incrementar en rango, leer en punto

```
#pragma region // Fenwick
using Fenwick = unordered_map<int, ll>;
ll FT_prefix (Fenwick& A, int i) {
    ll res = 0;
    for (int j = i; j >= 0; j = j & (j + 1), j--) res += A[j];
    return res;
}
void FT_update (Fenwick& A, int N, int i, ll x) {
    for (; i < N; i = i | (i + 1)) A[i] += x;
}
ll FT_get (Fenwick& A, int i) {
    return FT_prefix(A, i);
}
void FT_add (Fenwick& A, int N, int i, int j, ll x) {
    FT_update(A, N, i, x);
    FT_update(A, N, j+1, -x);
}
#pragma endregion
```

3.5. MO

MO comun

```
#pragma region // template Mo
struct Query { int idx, i, j; };

const int MOSIZE = 0 / 0; // ~sqrt(n) (entre 150 y 800)

bool mosort (Query const& p, Query const& q) {
    int bp = p.i / MOSIZE, bq = q.i / MOSIZE;
    if (bp == bq) return bq % 2 ? p.j > q.j : p.j < q.j;
    return bp < bq;
}
#pragma endregion

vector<int> mosolve (vector<Query>& queries) {
    sort(all(queries), mosort);
    vector<int> res(sz(queries));
    // Inicializar estructura
    Query a = { -1, 0, -1 };
    for (auto q : queries) {
        while (a.i > q.i) {
            a.i--;
            assert(0); // Estructura.add(a.i)
        }
        while (a.j < q.j) {
            a.j++;
            assert(0); // Estructura.add(a.j)
        }
        while (a.i < q.i) {
            assert(0); // Estructura.erase(a.i)
            a.i++;
        }
        while (a.j > q.j) {
            assert(0); // Estructura.erase(a.j)
            a.j--;
        }
        res[q.idx] = 0 / 0; // Resolver query
    }
    return res;
}
```

Sort con curva de Hilbert

??? (LMAO)

Con rollback

??? (LMAO)

4. Grafos

4.1. Toposort de un DAG

```
vector<int> Toposort (Grafo& G) {
    vector<int> indegree(sz(G)), res;
    forn(u, sz(G)) for (int v : G[u]) indegree[v]++;
    // Elegir criterio de priorizacion cambiando el orden en el que se
    // sacan (por defecto el menor)
    using Bag = priority_queue<int, vector<int>, greater<int>>;
    Bag bag;
    forn(u, sz(G)) if(indegree[u] == 0) bag.push(u);
    while (sz(bag)) {
        int u = bag.top();
        bag.pop();
        res.push_back(u);
        for (int v : G[u]) {
            indegree[v]--;
            if (indegree[v] == 0) bag.push(v);
        }
    }
    return res;
}
```

4.2. DAG condensado

```
Grafo DAGCondensado (Grafo& G) {
    vector<bool> visitado(sz(G));

    vector<int> orden;
    auto get_orden = [&](int u) -> void {
        visitado[u] = true;
        for (int v : G[u]) if (!visitado[v]) get_orden(v);
        orden.pb(u);
    };
    forn(u, sz(G)) if (!visitado[u]) get_orden(u);
    reverse(all(orden));

    Grafo T(sz(G));
    forn(u, sz(G)) for (int v : G[u]) T[v].pb(u);

    vector<int> comp, raiz(sz(G)), raices;
    auto extraer_comp = [&](int u) -> void {
        visitado[u] = true;
        comp.pb(u);
        for (int v : T[u]) if (!visitado[v]) extraer_comp(v);
    };

    visitado.assign(sz(G), false);
    for (int u : orden) if (!visitado[u]) {
```

```
        extraer_comp(u);
        int r = comp.front();
        for (int v : comp) raiz[v] = r;
        raices.pb(r);
        comp.clear();
    }

    // Opcion 1: hacer compresion de coordenadas: O(nlogn) lento
    int c = 0;
    map<int, int> coords;
    for (int r : raices) coords[r];
    for (auto& e : coords) e.snd = c++;
    Grafo SCC(raices.size());
    forn(u, sz(G)) for (int v : G[u]) {
        int ru = coords[raiz[u]], rv = coords[raiz[v]];
        if (ru != rv) SCC[ru].pb(rv);
    }

    return SCC;

    // Opcion 2: no hacer compresion y devolver raices (rapido)
    // AdjList SCC(sz(G));
    // forn(u, sz(G)) for (auto [v, w] : G[u]) {
    //     int ru = raiz[u], rv = raiz[v];
    //     if (ru != rv) SCC[ru].pb({rv, w});
    //     // else RC[ru] += R(w);
    // }
}
```

4.3. Bipartite check

```
bool EsBipartito (Grafo& G) {
    vector<int> color(sz(G), -1);
    color[0] = 0;
    queue<int> bag;
    for (bag.push(0); sz(bag);) {
        int u = bag.front();
        bag.pop();
        for (int v : G[u]) {
            if (color[u] == color[v]) return false;
            if (color[v] == -1) {
                color[v] = 1 - color[u];
                bag.push(v);
            }
        }
    }
    return true;
}
```

4.4. Encontrar puentes y articulaciones


```

auto GetPuentesYArticulaciones (Grafo& G) {
    int time = 0;
    vector<bool> visitado(sz(G));
    vector<int> tin(sz(G), -1), tlow(sz(G), -1), articulaciones;
    vector<pair<int, int>> puentes;
    auto dfs = [&](int u, int p) -> void {
        visitado[u] = true;
        tin[u] = tlow[u] = time++;
        int hijos = 0;
        for (int v : G[u]) {
            if (v == p) continue;
            if (visitado[v]) tlow[u] = min(tlow[u], tin[v]);
            else {
                dfs(v, u);
                hijos++;
                tlow[u] = min(tlow[u], tlow[v]);
                if (tlow[v] > tin[u])
                    puentes.pb({u,v});
                if (tlow[v] >= tin[u] && p != -1)
                    articulaciones.pb(u);
            }
        }
        if (p == -1 && hijos > 1) articulaciones.pb(u);
    };
    forn(r, sz(G)) if (!visitado[r]) dfs(r, -1);
    return mt(puentes, articulaciones);
}

```

4.5. Matching maximo bipartito (Kuhn)

```

vector<int> kuhn_matching (AdjList& grafo) {
    vector<int> match(sz(grafo), -1);
    vector<bool> visitado(sz(grafo));

    auto kuhn_dfs = [&](int u) -> bool {
        if (visitado[u]) return false;
        visitado[u] = true;
        for (int v : grafo[u]) if (match[v] == -1 || kuhn_dfs(match[v]
            ↪ )) {
            match[v] = u;
            return true;
        } return false;
    };

    forn(u, sz(grafo)) {
        visitado.assign(sz(grafo), false);
        kuhn_dfs(u);
    }

    return match;
}

```

```

int res = 0;
forn(u, sz(grafo)) if (match[u] != -1) res++;
cout << res / 2 << endl;

```

4.6. Menores caminos

Dijkstra

```

#pragma region // Dijkstra
using GrafoPond = vector<vector<pair<ll, int>>>;
auto Dijkstra (GrafoPond& grafo, int origen) {
    vector<ll> dist(sz(grafo), LLONG_MAX);
    vector<int> padre(sz(grafo), -1);
    dist[origen] = 0;
    padre[origen] = origen;
    priority_queue<pair<ll, int>, vector<pair<ll, int>>, greater<pair
        ↪ <ll, int>>> visitados;
    for (visitados.push({0, origen}); visitados.size();) {
        auto [du, u] = visitados.top();
        visitados.pop();
        if (du > dist[u]) continue; // (du,u) es un valor viejo
        for (auto [w, v] : grafo[u]) {
            ll nueva_dist = du + w;
            if (nueva_dist >= dist[v]) continue; // no mejora
            dist[v] = nueva_dist;
            padre[v] = u;
            visitados.push({nueva_dist, v});
        }
    }
    return mt(dist, padre);
}
#pragma endregion

```

Floyd-Warshall

```

templT using Matriz = vector<vector<T>>;
const ll INF = LLONG_MAX / 4;

void FloydWarshall (Matriz<ll>& D) {
    int N = D.size();
    forn(u, N) D[u][u] = 0;
    forn(k, N) forn(u, N) forn(v, N) if (D[u][k] < INF) if (D[k][v] <
        ↪ INF)
        D[u][v] = min(D[u][v], D[u][k] + D[k][v]);
    // Opcional: chequear ciclos negativos
    forn(u, N) forn(v, N) forn(k, N) if (D[u][k] < INF && D[k][k] < 0
        ↪ && D[k][v] < INF)
        D[u][v] = -INF;
}

```

Menores dos distancias (dijkstra)

```
#pragma region // Menores dos distancias (dijkstra)
using GrafoPond = vector<vector<pair<ll, int>>>;
auto MenoresDosDistancias (GrafoPond& grafo, int origen) {
    vector<ll> dist1(sz(grafo), LLONG_MAX), dist2(sz(grafo),
        ↪ LLONG_MAX);
    dist1[origen] = 0; dist2[origen] = LLONG_MAX;
    auto update_dist = [&](ll nueva_dist, int v) -> bool {
        if (nueva_dist < dist1[v]) {
            dist2[v] = dist1[v];
            dist1[v] = nueva_dist;
            return true;
        }
        else if (nueva_dist < dist2[v]) { // && dist1[v] < nueva_dist
            ↪ para tomar dists. distintas
            dist2[v] = nueva_dist;
            return true;
        }
        return false;
    };
    priority_queue<pair<ll, int>, vector<pair<ll, int>>, greater<pair
        ↪ <ll, int>>> visitados;
    for (visitados.push({0, origen}); visitados.size();) {
        auto [du, u] = visitados.top();
        visitados.pop();
        if (du > dist2[u]) continue; // (du,u) es un valor viejo
        for (auto [w, v] : grafo[u]) {
            ll nueva_dist = du + w;
            bool mejora = update_dist(nueva_dist, v);
            if (mejora) visitados.push({nueva_dist, v});
        }
    }
    return mt(dist1, dist2);
}
#pragma endregion
```

Reconstruir camino

```
auto reconstuir_camino (vector<int>& padre, int destino) {
    vector<int> res = { destino };
    int u = destino;
    do { u = padre[u]; res.pb(u); } while (u != padre[u]);
    reverse(all(res));
    return res;
}
```

5. Programacion Dinamica

5.1. LIS (Longest Increasing Subsequence)

Obtener largo del LIS

```
// Usa compresion de coordenadas y segtree point set RMQ (tomar el
    ↪ maximo)
int LIS (vector<int>& A) {
    int N = A.size();
    auto C = Compress(A);

    vector<STNode> hojas(N, {0});
    STree dp = segtree_build(hojas);

    segtree_update(dp, C[A[0]], {1});
    forr(i, 1, N) {
        int x = C[A[i]];
        int subres = 0;
        if (x-1 >= 0) subres = segtree_query(dp, 0, x-1).val;
        segtree_update(dp, x, {1 + subres});
    }

    return segtree_query(dp, 0, N - 1).val;
}
```

Construir LIS lexicograficamente menor

```
struct STNode { int len, idx, val, parent; };
bool operator < (STNode a, STNode b) {
    if (a.len != b.len) return a.len < b.len;
    return a.val > b.val;
}
STNode operator * (STNode a, STNode b) { return max(a,b); }
const STNode ST_ID = { -INT_MAX, -1, INT_MAX, -1 };
```

```
vector<int> LIS (vector<int>& A) {
    int N = A.size();
    auto C = Compress(A);

    STNode def = {0, -1, INT_MAX, -1};
    vector<STNode> hojas(N, def);
    STree dp = segtree_build(hojas);

    vector<STNode> res(N);
    res[0] = {1, 0, A[0], -1};
    segtree_update(dp, C[A[0]], {1, 0, A[0], -1});
    forr(i, 1, N) {
        int x = C[A[i]];
        STNode subres = def;
        if (x-1 >= 0) subres = segtree_query(dp, 0, x-1);
        STNode r = {1 + subres.len, i, A[i], subres.idx};
        segtree_update(dp, x, r);
        res[i] = r;
    }

    vector<int> path;
```

```

    STNode best = *max_element(all(res));
    STNode x;
    for (x = best; x.parent != -1; x = res[x.parent]) path.pb(x.idx);
    path.pb(x.idx);
    reverse(all(path));

    return path;
}

```

LIS en arbol (largo del LIS de raiz a cada nodo)

```

// Usa compresion de coordenadas y segtree RMQ (tomar el maximo)
vector<int> LIS (Grafo& G, vector<int>& valor_nodo, int root = 0) {
    auto C = Compress(valor_nodo);

    STNode def = { 0 };
    vector<STNode> hojas(sz(valor_nodo), def);
    STree dp = segtree_build(hojas);

    vector<int> res(sz(valor_nodo));

    segtree_update(dp, C[valor_nodo[root]], {1});
    auto dfs = [&](int u) {
        int x = C[valor_nodo[u]];
        int old = segtree_query(dp, x, x).val;
        int subres = {0};
        if (x-1 >= 0) subres = segtree_query(dp, 0, x-1).val;
        segtree_update(dp, x, {1 + subres});
        res[u] = segtree_query(dp, 0, sz(valor_nodo) - 1).val;
        for (int v : G[u]) dfs(v);
        segtree_update(dp, x, {old});
    };
    dfs(root);

    return res;
}

```

5.2. Menor ciclo hamiltoniano

```

ll MinimoCicloHamiltoniano (vector<vector<int>>& costo, int inicio) {
    vector<vector<ll>> memo(1 << sz(costo), vector<ll>(sz(costo), -1)
        ↪ );

    auto dfs = [&](auto self, int mask, int actual) -> ll {
        if (mask == (1 << (sz(costo))) - 1) return (actual == inicio)
            ↪ ? 0 : LLONG_MAX / 2;

        if (memo[mask][actual] != -1) return memo[mask][actual];

        ll res = LLONG_MAX / 2;

```

```

        forn(proximo, sz(costo)) if (proximo != actual) {
            if (mask & (1 << proximo)) continue; // ya fue visitado
            ll costo_visitar = self(self, mask | (1 << proximo),
                ↪ proximo) + costo[actual][proximo];
            res = min(res, costo_visitar);
        }
        return memo[mask][actual] = res;
    };
    return dfs(dfs, 0, inicio);
}

```

6. Matemática

6.1. Aritmética

Techo de la división

```
#define ceildiv(a,b) ((a + b - 1) / b)
```

Piso de la raiz cuadrada

```
using ll = long long;
```

```

ll isqrt (ll x) {
    ll s = 0;
    for (ll k = 1 << 30; k; k >>= 1)
        if ((s+k) * (s+k) <= x) s += k;
    return s;
}

```

Piso del log2

```
#define log2fl(x) (x ? 63 - __builtin_clzll(x) : -1)
```

Aritmética en \mathbb{Z}_p

```
const ll mod = 1e9 + 7;
```

```
ll resta_mod (ll a, ll b) { return (a - b + mod) % mod; }
```

```

ll pow_mod (ll x, ll n) {
    ll res = 0;
    while (n) {
        if (n % 2) res = res * x % mod;
        n /= 2;
        x = x * x % mod;
    } return res;
}

```

```
ll div_mod (ll a, ll b) { return a * pow_mod(b, mod - 2) % mod; }
```

6.2. Teoria de numeros

Criba

```
struct Criba {
    bool c[1000001]; vector<int> p;
    Criba () {
        p.reserve(1<<16);
        for (int i = 2; i <= 1000000; i++) if (!c[i]) {
            p.pb(i);
            for (int j = 2; i*j <= 1000000; j++) c[i*j] = 1;
        }
    }
    bool isprime (int x) {
        for (int i = 0, d = p[i]; d*d <= x; d = p[++i])
            if (!(x % d)) return false;
        return x >= 2;
    }
};
```

Phollards Rho

```
ll gcd(ll a, ll b){return a?gcd(b %a, a):b;}
```

//COMPILAR CON G++20

```
ll mulmod(ll a, ll b, ll m) {
    return ll(__int128(a) * b % m);
}
```

```
ll expmod (ll b, ll e, ll m){//0(log b)
    if(!e) return 1;
    ll q= expmod(b,e/2,m); q=mulmod(q,q,m);
    return e%2? mulmod(b,q,m) : q;
}
```

```
bool es_primo_prob (ll n, int a)
{
    if (n == a) return true;
    ll s = 0,d = n-1;
    while (d % 2 == 0) s++,d/=2;

    ll x = expmod(a,d,n);
    if ((x == 1) || (x+1 == n)) return true;

    forn (i, s-1){
        x = mulmod(x, x, n);
        if (x == 1) return false;
        if (x+1 == n) return true;
    }
    return false;
}
```

```
bool rabin (ll n){ //devuelve true si n es primo
    if (n == 1) return false;
    const int ar[] = {2,3,5,7,11,13,17,19,23};
    forn (j,9)
        if (!es_primo_prob(n,ar[j]))
            return false;

    return true;
}
```

```
ll rho(ll n){
    if( (n & 1) == 0 ) return 2;
    ll x = 2 , y = 2 , d = 1;
    ll c = rand() % n + 1;
    while( d == 1 ){
        x = (mulmod( x , x , n ) + c)%n;
        y = (mulmod( y , y , n ) + c)%n;
        y = (mulmod( y , y , n ) + c)%n;
        if( x - y >= 0 ) d = gcd( x - y , n );
        else d = gcd( y - x , n );
    }
    return d==n? rho(n):d;
}
```

```
map<ll,ll> prim;
void factRho (ll n){ //0 (lg n)^3. un solo numero
    if (n == 1) return;
    if (rabin(n)){
        prim[n]++;
        return;
    }
    ll factor = rho(n);
    factRho(factor);
    factRho(n/factor);
}
```

7. Geometria

7.1. Template: floats, punto

Punto flotante

```
#pragma region // template geometria
using flt = long double;
const flt EPS = 1e-9;
bool flt_le (flt a, flt b) { return a + EPS < b; }
bool flt_eq (flt a, flt b) { return -EPS <= a - b && a - b <= EPS;
    ↪ }
bool flt_leq (flt a, flt b) { return flt_le(a, b) || flt_eq(a, b); }
#pragma endregion
```

Punto

```
#pragma region // template geometria
// const bool USO_FLOAT = false; using Sca = ll;
bool sca_le (Sca a, Sca b) { return USO_FLOAT ? flt_le(a, b) : a <
    ↪ b; }
bool sca_eq (Sca a, Sca b) { return USO_FLOAT ? flt_eq(a, b) : a ==
    ↪ b; }
bool sca_leq (Sca a, Sca b) { return sca_le(a, b) || sca_eq(a, b); }
struct Punto { Sca x, y; };
Punto operator * (Punto p, Sca a) { return {p.x * a, p.y * a}; }
Punto operator + (Punto p, Punto q) { return {p.x + q.x, p.y + q.y};
    ↪ }
Sca operator * (Punto p, Punto q) { return p.x * q.x + p.y * q.y;
    ↪ } // |p| |q| cos theta
Sca operator ^ (Punto p, Punto q) { return p.x * q.y - p.y * q.x;
    ↪ } // |p| |q| sin theta
Punto operator - (Punto p, Punto q) { return p + (q * -1); }
Punto operator / (Punto p, Sca a) { return {p.x / a, p.y / a}; }
Sca proj (Punto p, Punto q) { return p * q / sqrtl(q * q); }
Sca angulo (Punto p, Punto q) { return acos(p * q / sqrtl(p*p)
    ↪ / sqrtl(q*q)); }
Sca dist2 (Punto p, Punto q) { return (p - q) * (p - q); }
Sca dist (Punto p, Punto q) { return sqrtl(dist2(p, q)); }
bool operator == (Punto p, Punto q) { return sca_eq(p.x, q.x) &&
    ↪ sca_eq(p.y, q.y); }
bool operator < (Punto p, Punto q) { return sca_eq(p.y, q.y) ?
    ↪ sca_le(p.x, q.x) : sca_le(p.y, q.y); }
ostream& operator << (ostream& o, Punto& p) { return o << "(" << p.x
    ↪ << ", " << p.y << ")"; }
#pragma endregion
```

7.2. Recta: Interseccion, Comparacion

Interseccion de recta

```
Punto intersect_recta (Punto p1, Punto d1, Punto p2, Punto d2) {
    // d1 ^ d2 == 0 ?
    return p1 + d1 * ((p2 - p1) ^ d1) / (d1 ^ d2);
}
```

Ecuacion de la recta (comparar por igualdad)

```
vector<Sca> ecuacion_recta (Punto p, Punto q) {
    Sca a = p.y - q.y;
    Sca b = q.x - p.x;
    Sca c = -a*p.x - b*p.y;
    Sca z = USO_FLOAT ? sqrtl(a*a + b*b) : __gcd(abs(a), __gcd(abs(b)
    ↪ , abs(c)));
    a /= z, b /= z, c /= z;
    if (sca_le(a, 0) || (sca_eq(a, 0) && sca_le(b, 0))) a *= (-1), b
    ↪ *= (-1), c *= (-1);
```

```
    return {a, b, c};
}
```

7.3. Circulo: Interseccion

Template

```
#pragma region // circulo
struct Circulo { Punto p; Sca r; };
bool operator == (Circulo o1, Circulo o2) { return o1.p == o2.p &&
    ↪ sca_eq(o1.r, o2.r); }
bool contiene (Circulo& c, Punto& punto) { return sca_leq(dist2(c.p,
    ↪ punto), c.r*c.r); }
#pragma endregion
```

Interseccion circulo-recta

```
vector<Punto> intersect_circulo_recta (Sca radio, Sca a, Sca b, Sca c
    ↪ ) {
    // a, b, c := parametros recta general
    Sca z = a*a + b*b;
    Punto punto_mas_cercano_origen = {-a*c / z, -b*c / z};
    if (flt_le(radio*radio*z, c*c)) return {}; // dist_origen > radio
    if (flt_eq(radio*radio*z, c*c)) return {punto_mas_cercano_origen
    ↪ }; // dist_origen = radio
    Sca m = sqrtl((radio*radio - c*c / z) / z);
    Punto d1 = { b*m, -a*m }, d2 = { -b*m, a*m };
    return { punto_mas_cercano_origen + d1, punto_mas_cercano_origen
    ↪ + d2 };
}
```

Interseccion circulo-circulo

```
vector<Punto> intersect_circulo_circulo (Circulo o1, Circulo o2) {
    // Mismo origen, distinto radio -> cero puntos
    // Mismo origen, mismo radio -> infinitos puntos
    Circulo o = {{o2.p - o1.p}, o2.r};
    Sca x = o.p.x, y = o.p.y;
    auto res = intersect_circulo_recta(o1.r, -2*x, -2*y, x*x + y*y +
    ↪ o1.r*o1.r - o.r*o.r);
    forn(i, res.size()) res[i] = res[i] + o1.p;
    return res;
}
```

7.4. Misc: Triangulo, Poligono, Convex Hull

Triangulo: area, isosceles

```
Sca area_triang (vector<Punto> T) { return abs((T[2] - T[1]) ^ (T[1]
    ↪ - T[0])) / 2; }
```

```
bool es_isosceles (vector<Punto> T) {
```

```

    forn(i, 3) {
        Punto a = T[i], b = T[(i+1)%3], c = T[(i+2)%3];
        if (sca_eq(dist2(a,b), dist2(b,c))) return true;
    } return false;
}

```

Poligono: area

```

Sca area_poli (vector<Punto> P) {
    Sca res = 0;
    forn(i, P.size()) {
        Punto p = i ? P[i - 1] : P.back();
        res += (p.x - P[i].x) * (p.y + P[i].y);
    } return abs(res) / 2;
}

```

Convex Hull

```

vector<Punto> convex_hull (vector<Punto>& puntos, bool
    ↪ incluir_colineales = false) {
    vector<Punto> P = puntos;

    Punto p0 = *min_element(all(P), [](Punto a, Punto b) { return mt(
        ↪ a.y, a.x) < mt(b.y, b.x); });

    auto orientation = [&](const Punto& a, const Punto& b, const
        ↪ Punto& c) {
        Sca area = (b - a) ^ (c - b);
        if (sca_le(area, 0)) return -1;
        if (sca_le(0, area)) return +1;
        return 0;
    };

    sort(all(P), [&](const Punto& a, const Punto& b) {
        int o = orientation(p0, a, b);
        if (o == 0) return sca_le(dist2(p0, a), dist2(p0, b));
        return o < 0;
    });

    if (incluir_colineales) {
        int i = sz(P) - 1;
        while (i >= 0 && orientacion(p0, P[i], P.back()) == 0) i--;
        reverse(P.begin() + i + 1, P.end());
    }

    vector<Punto> res;
    forn(i, sz(P)) {
        while (sz(res) > 1) {
            int o = orientacion(res[sz(res) - 2], res.back(), P[i]);
            if (o < 0 || (incluir_colineales && o == 0)) break;
            res.pop_back();
        }
    }
}

```

```

        res.pb(P[i]);
    }
    if (!incluir_colineales && sz(res) == 2 && res[0] == res[1]) res.
        ↪ pop_back();

    return res;
}

```

8. Estructuras locas

8.1. Disjoint set union

```

struct DSU {
    vector<int> p, w; int nc;
    DSU (int n) {
        nc = n, p.resize(n), w.resize(n);
        forn(i,n) p[i] = i, w[i] = 1;
    }
    int get (int x) { return p[x] == x ? x : p[x] = get(p[x]); }
    void join (int x, int y) {
        x = get(x), y = get(y);
        if (x == y) return;
        if (w[x] > w[y]) swap(x,y);
        p[x] = y, w[y] += w[x];
    }
    bool existe_camino (int x, int y) { return get(x) == get(y); }
};

```

8.2. Binary trie

```

struct BinaryTrieVertex { vector<int> next = {-1, -1}; };

using BinaryTrie = vector<BinaryTrieVertex>;

void binary_trie_add (BinaryTrie& trie, int x) {
    int v = 0;
    for (int i = 31; i >= 0; i--) {
        bool b = (x & (1 << i)) > 0;
        if (trie[v].next[b] == -1) {
            trie[v].next[b] = trie.size();
            trie.emplace_back();
        }
        v = trie[v].next[b];
    }
}

int binary_trie_max_xor (BinaryTrie& trie, int x) {
    int v = 0, res = 0;
    for (int i = 31; i >= 0; i--) {

```

```

        bool b = (x & (1 << i)) > 0;
        if (trie[v].next[!b] != -1) {
            v = trie[v].next[!b];
            if (!b) res |= (1 << i);
        }
        else {
            v = trie[v].next[ b];
            if ( b) res |= (1 << i);
        }
    } return res;
}

```

```

// Inicializar asi:
BinaryTrie trie(1);

```

9. Sin categorizar

9.1. Búsqueda binaria sobre un predicado

```

int primer_true (int i, int j, function<bool(int)> P, int def) {
    while (j - i > 1) {
        int m = i + ((j - i) >> 1);
        P(m) ? j = m : i = m;
    }
    if (P(i)) return i;
    if (P(j)) return j;
    return def;
}

int ultimo_false (int i, int j, function<bool(int)> P, int def) {
    while (j - i > 1) {
        int m = i + ((j - i) >> 1);
        P(m) ? j = m : i = m;
    }
    if (!P(j)) return j;
    if (!P(i)) return i;
    return def;
}

```

9.2. Enumerar subconjuntos de un conjunto con bitmask

```

// Imprimir representaciones en binario de todos los numeros "[0,
    ↪ ..., 2^N-1]"
forn(mask, (1 << N)) {
    forn(i, N) cout << "01"[(mask & (1 << i)) > 0] << "\0\n"[i == N
        ↪ -1];
}

// Iterar por los bits de cada subconjunto
forn(mask, (1 << N)) {

```

```

    forn(i, N) {
        bool on = (mask & (1 << i)) > 0;
        if (on) { ... }
        else { ... }
    }
}

```

9.3. Hashing Rabin Karp

```

using ll = long long;

const ll primo = 27, MAX_PRIME_POW = 1e6;

ll prime_pow[MAX_PRIME_POW];
void get_prime_pow () {
    prime_pow[0] = 1;
    forn(i, MAX_PRIME_POW) prime_pow[i+1] = prime_pow[i] * primo %
        ↪ mod;
}

vector<ll> get_rolling_hash (string& s) {
    vector<ll> rh(s.size() + 1);
    rh[0] = 0;
    // Ojo: es 'A' o 'a' ???
    forn(i, s.size()) rh[i+1] = (rh[i] * primo % mod + s[i] - 'A') %
        ↪ mod;
    return rh;
}

ll hash_range_query (vector<ll>& rh, int i, int j) {
    j++;
    return (rh[j] - (rh[i] * prime_pow[j - i] % mod) + mod) % mod;
}

#define log2fl(x) (x ? 63 - __builtin_clzll(x) : -1)
using AdjList = vector<vector<int>>;

struct LCA {
    AdjList& G; int N, R; // Grafo (ROOTEADO), #vertices y raiz
    int M; vector<int> e, f, d; AdjList st;
    void dfs (int u, int de = 0) {
        d[u] = de, f[u] = e.size(), e.pb(u);
        for (int v : G[u]) dfs(v, de+1), e.pb(u);
    }
    int op (int a, int b) {
        if (a == -1) return b;
        if (b == -1) return a;
        return d[a] < d[b] ? a : b;
    }
}

```

```

void make () {
    f.resize(N), d.resize(N), dfs(R), M = e.size();
    st.resize(20, vector<int>(M));
    st[0] = e; scn(w,1,19) scn(i,0,M - (1 << w))
        st[w][i] = op(st[w-1][i], st[w-1][i + (1 << (w-1))]);
}
int q (int u, int v) {
    int i = f[u], j = f[v];
    if (i > j) swap(i,j);
    int w = log2fl(j - i + 1);
    return op(st[w][i], st[w][j - (1 << w) + 1]);
}
int di (int u, int v) {
    int c = q(u,v);
    return d[u] + d[v] - 2*d[c];
}
};

bool visited[500001]; void rootear (int u) {
    visited[u] = 1;
    for (int v : grafo_original[u]) if (!visited[v]) {
        grafo_rooteado[u].pb(v);
        rootear(v);
    }
}

// Usar asi:
rootear(r);
LCA lca = {grafo_rooteado, N, r}; lca.make();

```

9.5. Euler tour

```

typedef vector<vector<int>>> adj;
typedef vector<vector<pair<int,i64>>>> wadj;

struct ETour {
    adj& G; int N, R;
    vector<int> t, f, d;
    void dfs (int u, int de = 0) {
        d[u] = de, f[u] = t.size(), t.pb(u);
        for (int v : G[u]) { dfs(v,de+1); t.pb(u); }
    }
    void make () { f.resize(N), d.resize(N), dfs(R); }
};

int main (void) {
    ios::sync_with_stdio(0); cin.tie(0);

    adj G; int N; cin >> N; G.resize(N);
    scn(u,1,N-1) {

```

```

        int p; cin >> p; p--;
        G[p].pb(u);
    }

    ETour et = {G, N, 0}; et.make();
    forall(v,et.t) { cout << *v + 1 << " "; } cout << endl;
    forall(v,et.t) { cout << et.d[*v] << " "; } cout << endl;
    forall(v,et.t) { cout << et.f[*v] << " "; } cout << endl;

    return 0;
}
/*
1 3 2 3 5 3 1 4 1
0 1 2 1 2 1 0 1 0
0 1 2 1 4 1 0 7 0
*/

```

10. Brainstorming

- Graficar como puntos/grafos
- Usar geometria
- ¿Que propiedades debe cumplir una solución?
- ¿Existen varias soluciones? ¿Hay una forma canónica?
- ¿Hay elecciones independientes?
- Pensarlo al revez
- ¿El proceso es parecido a un algoritmo conocido?
- Si se busca calcular $f(x)$ para todo x , calcular cuánto contribuye x a $f(y)$ para los otros y
- Definiciones e identidades: ¿que significa que un array sea palindromo? (ejemplo)