

Team notebook

CodeRats al fallo

October 21, 2023

Contents

1 Codes	2		
1.1 Algebra	2		
1.1.1 FFT	2		
1.1.2 Linear Recurrence	3		
1.1.3 Matrix Multiplication	3		
1.2 Data Structures	4		
1.2.1 Fenwick Tree	4		
1.2.2 Ordered Set	4		
1.2.3 Segment Tree(Lazy)	4		
1.2.4 Segment Tree	6		
1.2.5 Sparse Table	7		
1.3 Flows	7		
1.3.1 Blossom	7		
1.3.2 Dinic	9		
1.3.3 Hungarian	10		
1.3.4 Maximum Bipartite Matching	10		
1.3.5 MinCost MaxFlow	12		
1.3.6 Weighted Matching	14		
1.4 Graphs	15		
1.4.1 2SAT	15		
		1.4.2 Articulation Points	16
		1.4.3 Bellman Ford	17
		1.4.4 Bridges	17
		1.4.5 Dijkstra	18
		1.4.6 Floyd Warshall	19
		1.4.7 Heavy Light Decomposition	19
		1.4.8 LCA	20
		1.4.9 Prim	21
		1.4.10 SCC	22
		1.4.11 Topological Sort	23
		1.5 Number theory	23
		1.5.1 Chinese Remainder Theorem	23
		1.5.2 Extended Euclidean	24
		1.5.3 Lineal Sieve	24
		1.5.4 Miller Rabin	25
		1.5.5 Mobius	25
		1.5.6 Modular Multiplication	25
		1.5.7 Natural Sieve	26
		1.5.8 Pollard Rho	26
		1.6 Strings	27
		1.6.1 Aho Corasick	27
		1.6.2 Hashing	28

1.6.3	Manacher	28
1.6.4	Prefix Function	29
1.6.5	Suffix Array	29
1.6.6	Suffix Automaton	30
1.6.7	Trie	31
1.6.8	Z Function	32

Codes

1.1 Algebra

1.1.1 FFT

```
// Multiplicacion de polinomios en  $O(n \log n)$ 
```

```
const double PI = acos(-1.0);
```

```
namespace fft {
    struct pt {
        double r, i;
        pt(double r = 0.0, double i = 0.0) : r(r), i(i) {}
        pt operator + (const pt &b) { return pt(r+b.r, i+b.i); }
        pt operator - (const pt &b) { return pt(r-b.r, i-b.i); }
        pt operator * (const pt &b) { return pt(r*b.r - i*b.i,
            r*b.i + i*b.r); }
    };
    vector<int> rev;

    void fft(vector<pt> &y, int on) {
        int n = y.size();
        for (int i = 1; i < n; i++)
            if (i < rev[i]) swap(y[i], y[rev[i]]);
        for (int m = 2; m <= n; m <= 1) {
            double ang = -on * 2 * PI / m;
```

```
        pt wm(cos(ang), sin(ang));
        for (int k = 0; k < n; k += m) {
            pt w(1, 0);
            for (int j = 0; j < m / 2; j++) {
                pt u = y[k + j];
                pt t = w * y[k + j + m / 2];
                y[k + j] = u + t;
                y[k + j + m / 2] = u - t;
                w = w * wm;
            }
        }
    }
    if (on == -1) for (int i = 0; i < n; i++) y[i].r /= n;
}

vector<ll> mul(vector<ll> &a, vector<ll> &b) {
    int n = 1, t = 0, la = a.size(), lb = b.size();
    for (; n <= (la+lb+1); n <= 1, t++); t = 1<<(t-1);
    vector<pt> x1(n), x2(n);
    rev.assign(n, 0);
    for (int i = 0; i < n; i++) rev[i] = rev[i >> 1] >> 1 |
        (i & 1 ? t : 0);
    for (int i = 0; i < la; i++) x1[i] = pt(a[i], 0);
    for (int i = 0; i < lb; i++) x2[i] = pt(b[i], 0);
    fft(x1, 1); fft(x2, 1);
    for (int i = 0; i < n; i++) x1[i] = x1[i] * x2[i];
    fft(x1, -1);
    vector<ll> ans(n);
    for (int i = 0; i < n; i++) ans[i] = x1[i].r + 0.5;
    return ans;
}
}
```

1.1.2 Linear Recurrence

```

/*
Calcula el n-esimo termino de una recurrencia lineal (que
depende de los k terminos anteriores).
* Llamar init(k) en el main una unica vez si no es necesario
inicializar las matrices multiples veces.
Este ejemplo calcula el fibonacci de n como la suma de los k
terminos anteriores de la secuencia (En la secuencia comun k
es 2).
Agregar Matrix Multiplication con un constructor vacio.
*/
matrix F, T;

void init(int k) {
    F = {k, 1}; // primeros k terminos
    F[k-1][0] = 1;
    T = {k, k}; // fila k-1 = coeficientes: [c_k, c_{k-1}, ...,
        c_1]
    for (int i = 0; i < k-1; i++) T[i][i+1] = 1;
    for (int i = 0; i < k; i++) T[k-1][i] = 1;
}
/// O(k^3 log(n))
int fib(ll n, int k = 2) {
    init(k);
    matrix ans = pow(T, n+k-1) * F;
    return ans[0][0];
}

```

1.1.3 Matrix Multiplication

```

// Estructura para realizar operaciones de multiplicacion y
exponenciacion modular sobre matrices.

```

```

const int mod = 1e9+7;

struct matrix {
    vector<vector<int>> v;
    int n, m;

    matrix(int n, int m, bool o = false) : n(n), m(m), v(n,
        vector<int>(m)) {
        if (o) while (n--) v[n][n] = 1;
    }

    matrix operator * (const matrix &o) {
        matrix ans(n, o.m);
        for (int i = 0; i < n; i++)
            for (int k = 0; k < m; k++) if (v[i][k])
                for (int j = 0; j < o.m; j++)
                    ans[i][j] = (1ll*v[i][k]*o.v[k][j] +
                        ans[i][j]) % mod;
        return ans;
    }

    vector<int>& operator[] (int i) { return v[i]; }
};

matrix pow(matrix b, ll e) {
    matrix ans(b.n, b.m, true);
    while (e) {
        if (e&1) ans = ans*b;
        b = b*b;
        e /= 2;
    }
    return ans;
}

```

1.2 Data Structures

1.2.1 Fenwick Tree

```
int lso(int n) {return (n & (-n));}

// las consultas estn indexadas en 1
struct FenwickTree{
    vector<int> ft;
    FenwickTree(int m) {ft.assign(m + 1, 0);};
    int rsq(int j) {
        int sum = 0;
        for(; j; j -= lso(j)) sum += ft[j];
        return sum;
    }
    void upd(int i, int v) {
        for(; i < ft.size(); i += lso(i)) ft[i] += v;
    }
};
```

1.2.2 Ordered Set

```
/*
Estructura de datos basada en politicas. Funciona como un set<>
pero es internamente indexado, cuenta con dos funciones
adicionales.
.find_by_order(k) -> Retorna un iterador al k-esimo elemento, si
k >= size() retona .end()
.order_of_key(x) -> Retorna cuantos elementos hay menores (<)
que x
*/

#include <ext/pb_ds/assoc_container.hpp>
```

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

typedef tree<int, null_type, less<int>, rb_tree_tag,
tree_order_statistics_node_update> ordered_set;
```

1.2.3 Segment Tree(Lazy)

```
const int MOD = 1e9 + 7;
// modificar los struct
// operadores * y +
// en node + = merge
// en operation * = merge
// node * operation = como afecta la operacion al nodo
struct operation{
    bool swapped;
    operation() : swapped(false) {}
    operation(bool s) : swapped(s) {}

    operation operator * (const operation &rhs) const {
        operation res;
        res.swapped = (this -> swapped) ^ rhs.swapped;
        return res;
    }

    bool id() {
        return !swapped;
    }

    void clear() {
        swapped = false;
    }
};
```

```

struct node {
    int imx, imn, mx, mn;
    node() { imx = imn = mx = mn = 0; }
    node(int imx, int imn, int mx, int mn) : imx(imx), imn(imn),
        mx(mx), mn(mn) { }

    node operator + (const node &rhs) const {
        node res;
        res.mx = max(this -> mx, rhs.mx);
        res.mn = min(this -> mn, rhs.mn);
        if(this -> mx >= rhs.mx) res.imx = (this -> imx);
        else res.imx = rhs.imx;
        if(this -> mn <= rhs.mn) res.imn = (this -> imn);
        else res.imn = rhs.imn;
        return res;
    }

    node operator * (const operation &rhs) const {
        node res;
        if(rhs.swapped) {
            res.imx = imn;
            res.imn = imx;
            res.mx = MOD - mn;
            res.mn = MOD - mx;
        } else {
            res.imx = imx;
            res.imn = imn;
            res.mx = mx;
            res.mn = mn;
        }
        return res;
    }
};

```

```

const int N = 4e6 + 20;
node tree[N * 4];
operation lazy[N * 4];
pii arr[N];

void build(int v, int tl, int tr) {
    lazy[v].clear();
    if(tl == tr) {
        tree[v] = node(arr[tl].second, arr[tl].second,
            arr[tl].first, arr[tl].first);
        return;
    }
    int tm = (tl + tr) / 2;
    build(v * 2, tl, tm);
    build(v * 2 + 1, tm + 1, tr);
    tree[v] = tree[v * 2] + tree[v * 2 + 1];
}

void push(int v) {
    if(lazy[v].id()) return;
    tree[v * 2] = tree[v * 2] * lazy[v];
    tree[v * 2 + 1] = tree[v * 2 + 1] * lazy[v];
    lazy[v * 2] = lazy[v] * lazy[v * 2];
    lazy[v * 2 + 1] = lazy[v] * lazy[v * 2 + 1];
    lazy[v].clear();
}

void update(int v, int tl, int tr, int l, int r, operation &op) {
    if(l > r) return;
    if(l == tl && r == tr) {
        tree[v] = tree[v] * op;
        lazy[v] = op * lazy[v];
        return;
    }
}

```

```

    push(v);
    int tm = (tl + tr) / 2;
    update(v * 2, tl, tm, l, min(r, tm), op);
    update(v * 2 + 1, tm + 1, tr, max(l, tm + 1), r, op);
    tree[v] = tree[v * 2] + tree[v * 2 + 1];
}

node query(int v, int tl, int tr, int l, int r) {
    if(l > r) return node();
    if(l == tl && r == tr) return tree[v];
    push(v);
    int tm = (tl + tr) / 2;
    return query(v * 2, tl, tm, l, min(r, tm))
        + query(v * 2 + 1, tm + 1, tr, max(l, tm + 1), r);
}

```

1.2.4 Segment Tree

```

const int N = 100000 + 5;
const long long inf = 1e18 + 10;

struct node {
    long long sum;
    long long maxi;
    node(){
        sum = 0;
        maxi = -inf;
    }

    node(long long x) {
        sum = maxi = x;
    }

    node operator + (const node &rhs) const {

```

```

        node q;
        q.sum = sum + rhs.sum;
        q.maxi = max(maxi, rhs.maxi);
        return q;
    }
};

int n;
int q;
node NIL;
long long a[N];
node st[4 * N];

void build(int pos = 1, int l = 1, int r = n) {
    if(l == r) {
        st[pos] = node(a[l]);
        return;
    }
    int mi = (l + r) / 2;
    build(2 * pos, l, mi);
    build(2 * pos + 1, mi + 1, r);
    st[pos] = st[2 * pos] + st[2 * pos + 1];
}

void update(int x, int y, int pos = 1, int l = 1, int r = n) {
    if(st[pos].maxi <= 1) return; // Funcion Potencial
    sqrt(1) = 1
    if(y < l or r < x) return;
    if(l == r) {
        // to change
        st[pos].sum = sqrt(st[pos].sum);
        st[pos].maxi = st[pos].sum;
        return;
    }
    int mi = (l + r) / 2;

```

```

    update(x, y, 2 * pos, l, mi);
    update(x, y, 2 * pos + 1, mi + 1, r);
    st[pos] = st[2 * pos] + st[2 * pos + 1];
}

node query(int x, int y, int pos = 1, int l = 1, int r = n) {
    if(y < l or r < x) return NIL;
    if(x <= l and r <= y) return st[pos];
    int mi = (l + r) / 2;
    return query(x, y, 2 * pos, l, mi) + query(x, y, 2 * pos
        + 1, mi + 1, r);
}

int main() {
    build();
    update(1, r);
    query(1, r).sum;
    query(1, r).maxi;
}

```

1.2.5 Sparse Table

Estructura de datos que permite procesar consultas por rangos.

```

const int MX = 1e5+5;
const int LG = log2(MX)+1;
int spt[LG][MX];
int arr[MX];
int n;

void build() {
    for (int i = 0; i < n; i++) spt[0][i] = arr[i];
    for (int j = 0; j < LG-1; j++)
        for (int i = 0; i+(1<<(j+1)) <= n; i++)

```

```

        spt[j+1][i] = min(spt[j][i], spt[j][i+(1<<j)]);
}

int rmq(int i, int j) {
    int k = 31-__builtin_clz(j-i+1);
    return min(spt[k][i], spt[k][j-(1<<k)+1]);
}

```

1.3 Flows

1.3.1 Blossom

// Halla el mximo match en un grafo general $O(E * v^2)$

```

struct network {
    struct struct_edge {
        int v; struct_edge * n;
    };

    typedef struct_edge* edge;

    int n;
    struct_edge pool[MAXE]; ///2*n*n;
    edge top;
    vector<edge> adj;
    queue<int> q;
    vector<int> f, base, inq, inb, inp, match;
    vector<vector<int>> ed;

    network(int n) : n(n), match(n, -1), adj(n), top(pool),
        f(n), base(n),
            inq(n), inb(n), inp(n), ed(n, vector<int>(n))
        {}
}

```

```

void add_edge(int u, int v) {
    if(ed[u][v]) return;
    ed[u][v] = 1;
    top->v = v, top->n = adj[u], adj[u] = top++;
    top->v = u, top->n = adj[v], adj[v] = top++;
}

int get_lca(int root, int u, int v) {
    fill(inp.begin(), inp.end(), 0);
    while(1) {
        inp[u = base[u]] = 1;
        if(u == root) break;
        u = f[ match[u] ];
    }
    while(1) {
        if(inp[v = base[v]]) return v;
        else v = f[ match[v] ];
    }
}

void mark(int lca, int u) {
    while(base[u] != lca) {
        int v = match[u];
        inb[ base[u] ] = 1;
        inb[ base[v] ] = 1;
        u = f[v];
        if(base[u] != lca) f[u] = v;
    }
}

void blossom_contraction(int s, int u, int v) {
    int lca = get_lca(s, u, v);
    fill(inb.begin(), inb.end(), 0);
    mark(lca, u); mark(lca, v);
    if(base[u] != lca) f[u] = v;
}

```

```

    if(base[v] != lca) f[v] = u;
    for(int u = 0; u < n; u++){
        if(inb[base[u]]) {
            base[u] = lca;
            if(!inq[u]) {
                inq[u] = 1;
                q.push(u);
            }
        }
    }
}

int bfs(int s) {
    fill(inq.begin(), inq.end(), 0);
    fill(f.begin(), f.end(), -1);
    for(int i = 0; i < n; i++) base[i] = i;
    q = queue<int>();
    q.push(s);
    inq[s] = 1;
    while(q.size()) {
        int u = q.front(); q.pop();
        for(edge e = adj[u]; e; e = e->n) {
            int v = e->v;
            if(base[u] != base[v] && match[u] != v) {
                if((v == s) || (match[v] != -1 && f[match[v]]
                    != -1)){
                    blossom_contraction(s, u, v);
                }else if(f[v] == -1) {
                    f[v] = u;
                    if(match[v] == -1) return v;
                    else if(!inq[match[v]]) {
                        inq[match[v]] = 1;
                        q.push(match[v]);
                    }
                }
            }
        }
    }
}

```



```

    }
    }
}
return -1;
}

int doit(int u) {
    if(u == -1) return 0;
    int v = f[u];
    doit(match[v]);
    match[v] = u; match[u] = v;
    return u != -1;
}

/// (i < net.match[i]) => means match
int maximum_matching() {
    int ans = 0;
    for(int u = 0; u < n; u++)
        ans += (match[u] == -1) && doit(bfs(u));
    return ans;
}
};

```

1.3.2 Dinic

Halla el flujo mximo $O(E * V^2)$

```
struct edge { int v, cap, inv, flow; };
```

```
struct network {
    int n, s, t;
    vector<int> lvl;
    vector<vector<edge>> g;
```

```

network(int n) : n(n), lvl(n), g(n) {}

void add_edge(int u, int v, int c) {
    g[u].push_back({v, c, (int)g[v].size(), 0});
    g[v].push_back({u, 0, (int)g[u].size()-1, c});
}

bool bfs() {
    fill(lvl.begin(), lvl.end(), -1);
    queue<int> q;
    lvl[s] = 0;
    for(q.push(s); q.size(); q.pop()) {
        int u = q.front();
        for(auto &e : g[u]) {
            if(e.cap > 0 && lvl[e.v] == -1) {
                lvl[e.v] = lvl[u]+1;
                q.push(e.v);
            }
        }
    }
    return lvl[t] != -1;
}

int dfs(int u, int nf) {
    if(u == t) return nf;
    int res = 0;
    for(auto &e : g[u]) {
        if(e.cap > 0 && lvl[e.v] == lvl[u]+1) {
            int tf = dfs(e.v, min(nf, e.cap));
            res += tf; nf -= tf; e.cap -= tf;
            g[e.v][e.inv].cap += tf;
            g[e.v][e.inv].flow -= tf;
            e.flow += tf;
            if(nf == 0) return res;
        }
    }
}

```

```

    }
    if(!res) lvl[u] = -1;
    return res;
}

int max_flow(int so, int si, int res = 0) {
    s = so; t = si;
    while(bfs()) res += dfs(s, INT_MAX);
    return res;
}
};

```

1.3.3 Hungarian

```

typedef ll T;
const T inf = 1e18;

struct hung {
    int n, m;
    vector<T> u, v; vector<int> p, way;
    vector<vector<T>> g;

    hung(int n, int m):
        n(n), m(m), g(n+1, vector<T>(m+1, inf-1)),
        u(n+1), v(m+1), p(m+1), way(m+1) {}

    void set(int u, int v, T w) { g[u+1][v+1] = w; }

    T assign() {
        for (int i = 1; i <= n; ++i) {
            int j0 = 0; p[0] = i;
            vector<T> minv(m+1, inf);
            vector<char> used(m+1, false);
            do {

```

```

                used[j0] = true;
                int i0 = p[j0], j1; T delta = inf;
                for (int j = 1; j <= m; ++j) if (!used[j]) {
                    T cur = g[i0][j] - u[i0] - v[j];
                    if (cur < minv[j]) minv[j] = cur, way[j] = j0;
                    if (minv[j] < delta) delta = minv[j], j1 = j;
                }
                for (int j = 0; j <= m; ++j)
                    if (used[j]) u[p[j]] += delta, v[j] -= delta;
                    else minv[j] -= delta;
                j0 = j1;
            } while (p[j0]);
            do {
                int j1 = way[j0]; p[j0] = p[j1]; j0 = j1;
            } while (j0);
        }
        return -v[0];
    }
};

```

1.3.4 Maximum Bipartite Matching

```

// Halla el maximo match en un grafo bipartito O(|E|*|V|)

struct mbm {
    int l, r;
    vector<vector<int>> g;
    vector<int> match, vis;

    mbm(int l, int r) : l(l), r(r), g(l) {}

    void add_edge(int l, int r) {
        g[l].push_back(r);
    }
}

```

```

bool dfs(int u) {
    for (auto &v : g[u]) {
        if (vis[v]++) continue;
        if (match[v] == -1 || dfs(match[v])) {
            match[v] = u;
            return true;
        }
    }
    return false;
}

int max_matching() {
    int ans = 0;
    match.assign(r, -1);
    for (int u = 0; u < l; ++u) {
        vis.assign(r, 0);
        ans += dfs(u);
    }
    return ans;
}

};

// Hopcroft Karp:  $O(E * \sqrt{V})$ 

const int INF = INT_MAX;

struct mbm {
    vector<vector<int>> g;
    vector<int> d, match;
    int nil, l, r;
    /// u -> 0 to l, v -> 0 to r
    mbm(int l, int r) : l(l), r(r), nil(l+r), g(l+r),
                      d(1+l+r, INF), match(1+r, l+r) {}

```

```

void add_edge(int a, int b) {
    g[a].push_back(1+b);
    g[1+b].push_back(a);
}

bool bfs() {
    queue<int> q;
    for(int u = 0; u < l; u++) {
        if(match[u] == nil) {
            d[u] = 0;
            q.push(u);
        } else {
            d[u] = INF;
        }
    }
    d[nil] = INF;
    while(q.size()) {
        int u = q.front(); q.pop();
        if(u == nil) continue;
        for(auto v : g[u]) {
            if(d[ match[v] ] == INF) {
                d[ match[v] ] = d[u]+1;
                q.push(match[v]);
            }
        }
    }
    return d[nil] != INF;
}

bool dfs(int u) {
    if(u == nil) return true;
    for(int v : g[u]) {
        if(d[ match[v] ] == d[u]+1 && dfs(match[v])) {
            match[v] = u; match[u] = v;
            return true;
        }
    }
    return false;
}

```

```

    }
}
d[u] = INF;
return false;
}

int max_matching() {
    int ans = 0;
    while(bfs()) {
        for(int u = 0; u < l; u++) {
            ans += (match[u] == nil && dfs(u));
        }
    }
    return ans;
}
};

```

1.3.5 MinCost MaxFlow

Dado un grafo, halla el flujo maximo y el costo minimo entre el source s y el sink t.

```

struct edge {
    int u, v, cap, flow, cost;
    int rem() { return cap - flow; }
};

const int inf = 1e9;
const int MX = 405; //Cantidad maxima TOTAL de nodos
vector<int> g[MX]; //Lista de adyacencia
vector<edge> e; //Lista de aristas
vector<bool> in_queue; //Marca los nodos que estan en cola
vector<int> pre, dist, cap; //Almacena el nodo anterior, la
    distancia y el flujo de cada nodo

```

```

int mxflow, mncost; //Flujo maximo y costo minimo
int N; //Cantidad TOTAL de nodos

void add_edge(int u, int v, int cap, int cost) {
    g[u].push_back(e.size());
    e.push_back({u, v, cap, 0, cost});
    g[v].push_back(e.size());
    e.push_back({v, u, 0, 0, -cost});
}

void flow(int s, int t) {
    mxflow = mncost = 0;
    in_queue.assign(N, false);
    while (true) {
        dist.assign(N, inf); dist[s] = 0;
        cap.assign(N, 0); cap[s] = inf;
        pre.assign(N, -1); pre[s] = 0;
        queue<int> q; q.push(s);
        in_queue[s] = true;

        while (q.size()) {
            int u = q.front(); q.pop();
            in_queue[u] = false;
            for (int &id : g[u]) {
                edge &ed = e[id];
                int v = ed.v;
                if (ed.rem() && dist[v] > dist[u]+ed.cost) {
                    dist[v] = dist[u]+ed.cost;
                    cap[v] = min(cap[u], ed.rem());
                    pre[v] = id;
                    if (!in_queue[v]) {
                        q.push(v);
                        in_queue[v] = true;
                    }
                }
            }
        }
    }
}

```

```

    }
}
if (pre[t] == -1) break;
mxflow += cap[t];
mncost += cap[t] * dist[t];
for (int v = t; v != s; v = e[pre[v]].u) {
    e[pre[v]].flow += cap[t];
    e[pre[v]^1].flow -= cap[t];
}
}
}

void init() {
    e.clear();
    for (int i = 0; i <= N; i++) {
        g[i].clear();
    }
}

// O(V * E * 2 * log(E))
template <class type>
struct mcmf {
    struct edge { int u, v, cap, flow; type cost; };

    int n;
    vector<edge> ed;
    vector<vector<int>> g;
    vector<int> p;
    vector<type> d, phi;

    mcmf(int n) : n(n), g(n), p(n), d(n), phi(n) {}

    void add_edge(int u, int v, int cap, type cost) {
        g[u].push_back(ed.size());
        ed.push_back({u, v, cap, 0, cost});
    }
}

```

```

        g[v].push_back(ed.size());
        ed.push_back({v, u, 0, 0, -cost});
    }

bool dijkstra(int s, int t) {
    fill(d.begin(), d.end(), INF);
    fill(p.begin(), p.end(), -1);
    set<pair<type, int>> q;
    d[s] = 0;
    for(q.insert({d[s], s}); q.size();) {
        int u = (*q.begin()).second; q.erase(q.begin());
        for(auto v : g[u]) {
            auto &e = ed[v];
            type nd = d[e.u] + e.cost + phi[e.u] - phi[e.v];
            if(0 < (e.cap - e.flow) && nd < d[e.v]) {
                q.erase({d[e.v], e.v});
                d[e.v] = nd; p[e.v] = v;
                q.insert({d[e.v], e.v});
            }
        }
    }
    for(int i = 0; i < n; i++) phi[i] = min(INF, phi[i] + d[i]);
    return d[t] != INF;
}

pair<int, type> max_flow(int s, int t) {
    type mc = 0;
    int mf = 0;
    fill(phi.begin(), phi.end(), 0);
    while(dijkstra(s, t)) {
        int flow = INF;
        for(int v = p[t]; v != -1; v = p[ed[v].u])
            flow = min(flow, ed[v].cap - ed[v].flow);
        for(int v = p[t]; v != -1; v = p[ed[v].u]) {
            edge &e1 = ed[v];
            edge &e2 = ed[v^1];

```

```

        mc += e1.cost*flow;
        e1.flow += flow;
        e2.flow -= flow;
    }
    mf += flow;
}
return {mf, mc};
}
};

```

1.3.6 Weighted Matching

Halla el mximo match con pesos $O(V^3)$

```

typedef int type;
struct matching_weighted {
    int l, r;
    vector<vector<type>> c;
    matching_weighted(int l, int r) : l(l), r(r), c(l,
        vector<type>(r)) {
        assert(l <= r);
    }

    void add_edge(int a, int b, type cost) { c[a][b] = cost; }

    type matching() {
        vector<type> v(r), d(r); // v: potential
        vector<int> ml(l, -1), mr(r, -1); // matching pairs
        vector<int> idx(r), prev(r);
        iota(idx.begin(), idx.end(), 0);
        auto residue = [&](int i, int j) { return c[i][j]-v[j]; };
        for(int f = 0; f < l; ++f) {
            for(int j = 0; j < r; ++j) {
                d[j] = residue(f, j);
            }
        }
    }
};

```

```

        prev[j] = f;
    }
    type w;
    int j, l;
    for (int s = 0, t = 0;;) {
        if(s == t) {
            l = s;
            w = d[ idx[t++] ];
            for(int k = t; k < r; ++k) {
                j = idx[k];
                type h = d[j];
                if (h <= w) {
                    if (h < w) t = s, w = h;
                    idx[k] = idx[t];
                    idx[t++] = j;
                }
            }
        }
        for (int k = s; k < t; ++k) {
            j = idx[k];
            if (mr[j] < 0) goto aug;
        }
        int q = idx[s++], i = mr[q];
        for (int k = t; k < r; ++k) {
            j = idx[k];
            type h = residue(i, j) - residue(i, q) + w;
            if (h < d[j]) {
                d[j] = h;
                prev[j] = i;
                if(h == w) {
                    if(mr[j] < 0) goto aug;
                    idx[k] = idx[t];
                    idx[t++] = j;
                }
            }
        }
    }
}

```

```

    }
}
aug:
for (int k = 0; k < l; ++k)
    v[ idx[k] ] += d[ idx[k] ] - w;
int i;
do {
    mr[j] = i = prev[j];
    swap(j, ml[i]);
} while (i != f);
}
type opt = 0;
for (int i = 0; i < l; ++i)
    opt += c[i][ml[i]]; // (i, ml[i]) is a solution
return opt;
}
};

```

1.4 Graphs

1.4.1 2SAT

```

struct SATSolver {
    // Assumes that nodes are 0-indexed
    int n;
    int m;
    vector<bool> vis;
    vector<int> comp;
    vector<int> order;
    vector<int> component;
    vector<vector<int>> G;
    vector<vector<int>> Gt;

    SATSolver(int n, int m) : n(n), m(m) {

```

```

        // X_i = 2i
        // ~X_i = 2i + 1
        comp.resize(2 * n);
        vis.resize(2 * n, false);
        G.resize(2 * n, vector<int>());
        Gt.resize(2 * n, vector<int>());
    }

    void add_edge(int u, int v) {
        // u OR v
        G[u ^ 1].emplace_back(v);
        G[v ^ 1].emplace_back(u);
        Gt[v].emplace_back(u ^ 1);
        Gt[u].emplace_back(v ^ 1);
    }

    void DFS1(int u) {
        vis[u] = true;
        for(int v : G[u]) {
            if(vis[v]) continue;
            DFS1(v);
        }
        order.emplace_back(u);
    }

    void DFS2(int u) {
        vis[u] = true;
        for(int v : Gt[u]) {
            if(vis[v]) continue;
            DFS2(v);
        }
        component.emplace_back(u);
    }

    void get_scc() {

```

```

    for(int i = 0; i < 2 * n; i++) {
        if(vis[i]) continue;
        DFS1(i);
    }
    reverse(order.begin(), order.end());
    fill(vis.begin(), vis.end(), false);
    int component_id = 0;
    for(int u : order) {
        if(vis[u]) continue;
        component.clear();
        DFS2(u);
        for(int x : component) comp[x] =
            component_id;
        component_id += 1;
    }
}

vector<int> solve() {
    vector<int> res(n);
    get_scc();
    for(int i = 0; i < n; i++) {
        int val = 2 * i;
        if(comp[val] == comp[val ^ 1]) return
            vector<int>();
        if(comp[val] < comp[val ^ 1]) res[i] = 0;
        else res[i] = 1;
    }
    return res;
}
};

```

1.4.2 Articulation Points

```
int n;
```

```

int m;
int timer;
int low[N];
int tin[N];
bool vis[N];
bool cut_point[N];
vector<int> G[N];

void DFS(int u, int p = -1) {
    tin[u] = low[u] = timer++;
    vis[u] = true;
    int children = 0;
    for(int v : G[u]) {
        if(v == p) continue;
        if(vis[v]) {
            // Es una backedge, aporta a low[u]
            low[u] = min(low[u], tin[v]);
        }
        else {
            // Es una tree-edge, verificar si u es cut
            // point
            DFS(v, u);
            // Minimizamos el low del padre con el del
            // hijo
            low[u] = min(low[u], low[v]);
            // Ya tenemos procesado low[v]
            if(p != -1 and low[v] >= tin[u]) {
                // u es articulation si low[v] >=
                // tin[u] cuando u no es la raz
                cut_point[u] = true;
            }
            children++;
        }
    }
}

// Si es la raiz, es articulacion si tiene 2 o ms hijos

```



```

    if(p == -1 and children > 1) cut_point[u] = true;
}

```

1.4.3 Bellman Ford

```

int n;
int m;
int D[N]; // D[u] : Minima distancia de src a u usando <= k
          aristas en la k-sima iteracin
bool vis[N]; // vis[u] : El nodo se ha vuelto alcanzable por src
vector<tuple<int, int, int>> edges;

// retorna true si no hay ciclos negativos
bool bellman_ford(int src) {
    for(int i = 0; i < n; i++) {
        D[i] = -1;
        vis[i] = false;
    }
    D[src] = 0;
    vis[src] = true;
    for(int i = 1; i < n; i++) {
        for(auto e : edges) {
            int u, v, w;
            tie(u, v, w) = e;
            if(not vis[u]) continue;
            if(not vis[v] or D[v] > D[u] + w) {
                D[v] = D[u] + w;
                vis[v] = true;
            }
        }
    }
    for(auto e : edges) {
        int u, v, w;
        tie(u, v, w) = e;
    }
}

```

```

    if(not vis[u]) continue;
    if(not vis[v] or D[v] > D[u] + w) {
        return false; // Ciclo negativo alcanzable
                        por src
    }
}
return true;
}

```

1.4.4 Bridges

```

int n; // para guardar el numero de vertices
int m; // aristas
int k; // numero de componentes 2-conexas en el arbol final
int C[N]; // componente de cada vertice
int to[N], from[N]; // vertice de salida y llegada para cada
                    arista
int timer; // para el DFS
int low[N];
int tin[N];
bool vis[N];
bool bridge[N]; // es puente ?
vector<int> T[N]; // arbol final
vector<pair<int, int>> G[N]; // first = v, second = id de arista

void DFS(int u, int p = -1) {
    vis[u] = true;
    low[u] = tin[u] = timer++;
    for(auto edge : G[u]) {
        int v, e;
        tie(v, e) = edge;
        if(v == p) continue;
        if(vis[v]) {
            low[u] = min(low[u], tin[v]);
        }
    }
}

```

```

    }
    else {
        DFS(v, u);
        low[u] = min(low[u], low[v]);
        if(low[v] > tin[u]) {
            bridge[e] = true;
        }
    }
}

}

void build_tree() {
    DFS(1);
    for(int i = 1; i <= n; i++) vis[i] = false;
    k = 0;
    for(int i = 1; i <= n; i++) {
        if(vis[i]) continue;
        queue<int> Q;
        Q.emplace(i);
        vis[i] = true;
        while(!Q.empty()) {
            int u = Q.front(); Q.pop();
            C[u] = k;
            for(auto edge : G[u]) {
                int v, e;
                tie(v, e) = edge;
                if(bridge[e]) continue;
                if(vis[v]) continue;
                Q.emplace(v);
                vis[v] = true;
            }
        }
        k += 1;
    }
    for(int i = 1; i <= m; i++) {

```

```

        if(not bridge[i]) continue;
        int u = C[to[i]], v = C[from[i]];
        T[u].emplace_back(v);
        T[v].emplace_back(u);
    }
}

```

1.4.5 Dijkstra

```

int n;
int m;
int D[N];
vector<pair<int, int>> G[N];

// las distancias mnimas se guardan en D
// indexa en 0
void Dijkstra(int src) {
    for(int i = 0; i < n; i++) D[i] = -1;
    D[src] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>,
        greater<pair<int, int>>> Q; // Min PQ
    Q.emplace(0, src);
    while(!Q.empty()) {
        int dis, u;
        tie(dis, u) = Q.top(); Q.pop();
        if(dis != D[u]) continue; // Verificacion de que u
                                // no ha sido visitado todavia
        for(auto e : G[u]) {
            int v, w;
            tie(v, w) = e;
            if(D[v] == -1 or D[v] > D[u] + w) {
                D[v] = D[u] + w;
                Q.emplace(D[v], v);
            }
        }
    }
}

```

```

        }
    }
}

```

1.4.6 Floyd Warshall

```

const int N = 100 + 5;
const int inf = 2e9 + 10;

int n;
int m;
int d[N][N];

/*
 * inicializar el arreglo d con INF, a menos que i == j
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            d[i][j] = i == j ? 0 : inf;
        }
    }
 * para escoger siempre la menor
 * arista en caso de aristas mltiples
    for(int i = 0; i < m; i++) {
        int u, v, w;
        scanf("%d %d %d", &u, &v, &w);
        d[u][v] = min(d[u][v], w);
    }
 */

bool floyd_warshall(){
    for(int k = 0; k < n; k++) {
        for(int i = 0; i < n; i++) {

```

```

            for(int j = 0; j < n; j++) {
                if(d[i][k] == inf || d[k][j] ==
                    inf) continue;
                if(d[i][j] > d[i][k] + d[k][j])
                    d[i][j] = d[i][k] + d[k][j];
            }
        }
    }

    // Termina Floyd-Warshall
    // comprobacion de ciclos negativos
    for(int i = 0; i < n; i++) {
        if(d[i][i] < 0) {
            return false;
        }
    }
    return true;
}

```

1.4.7 Heavy Light Decomposition

```

/*
Para inicializar llamar build().
Agregar Segment Tree con un constructor vacio,
actualizaciones puntuales y declarar el valor neutro de forma
global.
Para consultas sobre aristas guardar el valor de cada arista
en su nodo hijo y cambiar pos[u] por pos[u]+1 en la linea 54.
*/

typedef int T; //tipo de dato del segtree
const int MX = 1e5+5;
vector<int> g[MX];

```

```

int par[MX], dep[MX], sz[MX];
int pos[MX], top[MX], value[MX];
vector<T> arr;
int idx;

int pre(int u, int p, int d) {
    par[u] = p; dep[u] = d;
    int aux = 1;
    for (auto &v : g[u]) {
        if (v != p) {
            aux += pre(v, u, d+1);
            if (sz[v] >= sz[g[u][0]]) swap(v, g[u][0]);
        }
    }
    return sz[u] = aux;
}

void hld(int u, int p, int t) {
    arr[idx] = value[u]; //vector para inicializar el segtree
    pos[u] = idx++;
    top[u] = t < 0 ? t = u : t;
    for (auto &v : g[u]) {
        if (v != p) {
            hld(v, u, t);
            t = -1;
        }
    }
}

segtree sgt;

void build(int n, int root) {
    idx = 0;
    arr.resize(n);
    pre(root, root, 0);
}

```

```

    hld(root, root, -1);
    sgt = segtree(arr);
}

T query(int u, int v) {
    T ans = neutro;
    while (top[u] != top[v]) {
        if (dep[top[u]] > dep[top[v]]) swap(u, v);
        ans = min(ans, sgt.query(pos[top[v]], pos[v]));
        v = par[top[v]];
    }
    if (dep[u] > dep[v]) swap(u, v);
    ans = min(ans, sgt.query(pos[u], pos[v]));
    return ans;
}

void upd(int u, T val) {
    sgt.upd(pos[u], val);
}

```

1.4.8 LCA

```

/*
Dados los nodos u y v de un arbol determina cual es el ancestro
comun mas bajo entre u y v.
*Tambien puede determinar la arista de peso maximo/minimo entre
los nodos u y v (Para esto quitar los "//")
Se debe ejecutar la funcion dfs() primero, el padre de la raiz
es s mismo, w es el valor a almacenar del padre.
*/
const int N = 4e5+2, inf = 1e9, LOG2 = 20;
int dep[N]; // Profundidad de cada nodo
int par[LOG2][N]; // Sparse table para guardar los padres
//int rmq[LOG2][N]; // Sparse table para guardar pesos

```

```

struct edge { int v, w; };
vector<edge> g[N];

void dfs(int u, int p, int d, int w){
    dep[u] = d;
    par[0][u] = p;
    // rmq[0][u] = w;
    for(int j = 1; j < LOG2; j++){
        par[j][u] = par[j-1][par[j-1][u]];
        // rmq[j][u] = max(rmq[j-1][u], rmq[j-1][par[j-1][u]]);
    }
    for(auto &ed: g[u]){
        int v = ed.v;
        int val = ed.w;
        if(v == p)continue;
        dfs(v, u, d+1, val);
    }
}

int lca(int u, int v){
    // int ans = -1;
    if(dep[v] < dep[u])swap(u, v);
    int d = dep[v]-dep[u];
    for(int j = LOG2-1; j >= 0; j--){
        if(d >> j & 1){
            // ans = max(ans, rmq[j][v]);
            v = par[j][v];
        }
    }
    // if(u == v)return ans;
    if(u == v)return u;
    for(int j = LOG2-1; j >= 0; j--){
        if(par[j][u] != par[j][v]){
            // ans = max({ans, rmq[j][u], rmq[j][v]});

```

```

        u = par[j][u];
        v = par[j][v];
    }
}
// return max({ans, rmq[1][u], rmq[0][v]}); // si la info es
// de los nodos
// return max({ans, rmq[0][u], rmq[0][v]}); // si la info es
// de las aristas
return par[0][u];
}

```

1.4.9 Prim

// Prim clsico, retorna el MST de un grafo

```

int n;
int m;
int q[N];
bool vis[N];
int wedge[N];
vector<pair<int, int>> G[N];

int Prim(int src) {
    memset(wedge, -1, sizeof wedge);
    wedge[src] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>,
        greater<pair<int, int>>> Q;
    Q.emplace(0, src);
    while(!Q.empty()) {
        int we, u;
        tie(we, u) = Q.top(); Q.pop();
        vis[u] = true;
        for(auto e : G[u]) {
            int v, w;

```

```

        tie(v, w) = e;
        if(wedge[v] == -1 or wedge[v] > w) {
            wedge[v] = w;
            Q.emplace(wedge[v], v);
        }
    }
    for(int i = 1; i <= n; i++) {
        if(not vis[i]) return -1;
    }
    return accumulate(wedge + 1, wedge + n + 1, 0);
}

/*
 *
 * Caso especial para un grafo completo
 * no es posible construir el grafo (memoria)
 * pero es posible un algoritmo  $O(n^2)$ 
 */
int n;
int x[N];
int y[N];
bool vis[N];
int edge[N]; // Minima arista que cruza desde S hasta mi nodo

int dis(int i, int j) {
    return abs(x[i] - x[j]) + abs(y[i] - y[j]);
}

int Prim(int src) {
    vis[src] = true;
    for(int i = 0; i < n; i++) edge[i] = dis(src, i);

```

```

    for(int i = 1; i < n; i++) {
        // El nodo al que llega la arista ligera es al
        // argmin(edge[i]) pero con vis[i] = false
        int v = -1;
        for(int j = 0; j < n; j++) {
            if(vis[j]) continue;
            if(v == -1 or edge[v] > edge[j]) v = j;
        }
        vis[v] = true;
        for(int j = 0; j < n; j++) {
            if(vis[j]) continue;
            edge[j] = min(edge[j], dis(v, j));
        }
    }
    return accumulate(edge, edge + n, 0);
}

```

1.4.10 SCC

```

int n;
int m;
bool in[N];
bool vis[N];
int comp[N];
vector<int> order;
vector<int> component;
vector<int> G[2][N]; // en G[1] es el grafo transpuesto

void DFS(int id, int u) {
    vis[u] = id ^ 1;
    for(int v : G[id][u]) {
        if(vis[v] == (id ^ 1)) continue;
        DFS(id, v);
    }
}

```

```

        if(id == 0) order.emplace_back(u);
        else component.emplace_back(u);
    }

    int solve() {
        order.clear();
        component.clear();
        for(int i = 0; i < n; i++) {
            if(vis[i]) continue;
            DFS(0, i);
        }
        reverse(order.begin(), order.end());
        vector<vector<int>> res;
        for(int u : order) {
            if(not vis[u]) continue;
            component.clear();
            DFS(1, u);
            for(int x : component) comp[x] = res.size();
            res.emplace_back(component);
        }
        for(int i = 0; i < n; i++) {
            for(int v : G[0][i]) {
                if(comp[i] == comp[v]) continue;
                in[comp[v]] = true;
            }
        }
        int cnt = 0;
        for(int i = 0; i < res.size(); i++) cnt += !in[i];
        return cnt;
    }
}

```

1.4.11 Topological Sort

```
vector<int> toposort(int n, int m, vector<vector<int>> &G) {
```

```

    vector<int> in_degree(n, 0);
    for(int i = 0; i < n; i++) {
        for(int v : G[i]) in_degree[v] += 1;
    }
    queue<int> Q;
    vector<int> res;
    for(int i = 0; i < n; i++) {
        if(in_degree[i] == 0) {
            Q.emplace(i);
        }
    }
    while(!Q.empty()) {
        int u = Q.front(); Q.pop();
        res.emplace_back(u);
        for(int v : G[u]) {
            in_degree[v] -= 1;
            if(in_degree[v] == 0) {
                Q.emplace(v);
            }
        }
    }
    return res.size() < n ? vector<int>() : res;
}

```

1.5 Number theory

1.5.1 Chinese Remainder Theorem

```

/*
Encuentra un x tal que para cada i : x es congruente con A_i mod
M_i
Devuelve {x, lcm}, donde x es la solucion con modulo lcm (lcm =
LCM(M_0, M_1, ...)). Dado un k : x + k*lcm es solucion
tambien.

```

Si la solución no existe o la entrada no es válida devuelve {-1, -1}

Agregar Extended Euclides.

```
*/

pair<int, int> crt(vector<int> A, vector<int> M) {
    int n = A.size(), ans = A[0], lcm = M[0];
    for (int i = 1; i < n; i++) {
        int d = euclid(lcm, M[i]);
        if ((A[i] - ans) % d) return {-1, -1};
        int mod = lcm / d * M[i];
        ans = (ans + x * (A[i] - ans) / d % (M[i] / d) * lcm) %
            mod;
        if (ans < 0) ans += mod;
        lcm = mod;
    }
    return {ans, lcm};
}
```

1.5.2 Extended Euclidean

// Recursivo

```
int gcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int d = gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}
```

// Iterativo

```
int gcd(int a, int b, int& x, int& y) {
    x = 1, y = 0;
    int x1 = 0, y1 = 1, a1 = a, b1 = b;
    while (b1) {
        int q = a1 / b1;
        tie(x, x1) = make_tuple(x1, x - q * x1);
        tie(y, y1) = make_tuple(y1, y - q * y1);
        tie(a1, b1) = make_tuple(b1, a1 - q * b1);
    }
    return a1;
}
```

// UFPS

// El algoritmo de Euclides extendido retorna el gcd(a, b) y calcula los coeficientes enteros X y Y que satisfacen la ecuación: $aX + bY = \text{gcd}(a, b)$.

```
int x, y;
/// O(log(max(a, b)))
int euclid(int a, int b) {
    if (b == 0) { x = 1; y = 0; return a; }
    int d = euclid(b, a % b);
    int aux = x;
    x = y;
    y = aux - a / b * y;
    return d;
}
```

1.5.3 Lineal Sieve

```
const int N = 100000000 + 5;
vector<int> primes;
```



```

bitset<N> composite;

void lineal(int n){
    for(int i = 2; i <= n; i++) {
        if(not composite[i]) primes.emplace_back(i);
        for(int p : primes) {
            if(i * p > n) break;
            composite[i * p] = true;
            if(i % p == 0) break;
        }
    }
}

```

1.5.4 Miller Rabin

// El algoritmo de Miller-Rabin determina si un numero es primo o no. Agregar Modular Exponentiation (para m ll) y Modular Multiplication.

```

/// O(log^3(n))
bool test(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;
    for (int i = 0; i < s-1; i++) {
        x = mulmod(x, x, n);
        if (x == 1) return false;
        if (x+1 == n) return true;
    }
    return false;
}

```

```

bool is_prime(ll n) {
    if (n == 1) return false;
    int ar[] = {2,3,5,7,11,13,17,19,23};
    for (auto &p : ar) if (!test(n, p)) return false;
    return true;
}

```

1.5.5 Mobius

```

/*
La funcion mu de Mobius devuelve 0 si n es divisible por algun
cuadrado (x^2).
Si n es libre de cuadrados entonces devuelve 1 o -1 si n tiene
un numero par o impar de factores primos distintos.
* Calcular Mobius para todos los numeros menores o iguales a MX
con Sieve of Eratosthenes.
*/
const int MX = 1e6;
short mu[MX+1] = {0, 1};
/// O(MX log(log(MX)))
void mobius() {
    for (int i = 1; i <= MX; i++) {
        if (!mu[i]) continue;
        for (int j = i*2; j <= MX; j += i) {
            mu[j] -= mu[i];
        }
    }
}

```

1.5.6 Modular Multiplication

```

/* Calcula (a*b) % m sin overflow cuando m es ll. */

```

```

/// 0(1)
ll mulmod(ll a, ll b, ll m) {
    ll r = a*b-(ll)((long double)a*b/m+.5)*m;
    return r < 0 ? r+m : r;
}

```

1.5.7 Natural Sieve

```

const int N = 100000000 + 5;
bitset<N> composite;

void natural(int n){
    // (WARNING) Todos los pares son primos
    for(int i = 3; i * i <= n; i += 2) {
        if(not composite[i]) {
            for(int j = i * i; j <= n; j += i)
                composite[j] = true;
        }
    }
}

```

1.5.8 Pollard Rho

```

/*
La funcion Rho de Pollard calcula un divisor no trivial de n.
Agrega Modular Multiplication.
*/
ll gcd(ll a, ll b) { return a ? gcd(b%a, a) : b; }

ll rho(ll n) {
    if (!(n&1)) 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;
        d = gcd(abs(x-y), n);
    }
    return d == n ? rho(n) : d;
}

* Version optimizada

ll add(ll a, ll b, ll m) { return (a += b) < m ? a : a-m; }

ll rho(ll n) {
    static ll s[MX];
    while (1) {
        ll x = rand()%n, y = x, c = rand()%n;
        ll *px = s, *py = s, v = 0, p = 1;
        while (1) {
            *py++ = y = add(mulmod(y, y, n), c, n);
            *py++ = y = add(mulmod(y, y, n), c, n);
            if ((x = *px++) == y) break;
            ll t = p;
            p = mulmod(p, abs(y-x), n);
            if (!p) return gcd(t, n);
            if (++v == 26) {
                if ((p = gcd(p, n)) > 1 && p < n) return p;
                v = 0;
            }
        }
        if (v && (p = gcd(p, n)) > 1 && p < n) return p;
    }
}

```

1.6 Strings

1.6.1 Aho Corasick

```

/*
    El trie (o prefix tree) guarda un diccionario de strings
    como un arbol enraizado.
    Aho corasick permite encontrar las ocurrencias de todos los
    strings del trie en un string s.
*/
const int alpha = 26; //cantidad de letras del lenguaje
const char L = 'a'; //primera letra del lenguaje

struct node {
    int next[alpha], end;
    //int link, exit, cnt; //para aho corasick
    int& operator[](int i) { return next[i]; }
};

vector<node> trie = {node()};

void add_str(string &s, int id = 1) {
    int u = 0;
    for (auto ch : s) {
        int c = ch-L;
        if (!trie[u][c]) {
            trie[u][c] = trie.size();
            trie.push_back(node());
        }
        u = trie[u][c];
    }
    trie[u].end = id; //con id > 0
    //trie[u].cnt++; //para aho corasick
}

```

```

// aho corasick
void build_ac() {
    queue<int> q; q.push(0);
    while (q.size()) {
        int u = q.front(); q.pop();
        for (int c = 0; c < alpha; ++c) {
            int v = trie[u][c];
            if (!v) trie[u][c] = trie[trie[u].link][c];
            else q.push(v);
            if (!u || !v) continue;
            trie[v].link = trie[trie[u].link][c];
            trie[v].exit = trie[trie[v].link].end ?
                trie[v].link : trie[trie[v].link].exit;
            trie[v].cnt += trie[trie[v].link].cnt;
        }
    }
}

vector<int> cnt; //cantidad de ocurrencias en s para cada patron

void run_ac(string &s) {
    int u = 0, sz = s.size();
    for (int i = 0; i < sz; ++i) {
        int c = s[i]-L;
        while (u && !trie[u][c]) u = trie[u].link;
        u = trie[u][c];
        int x = u;
        while (x) {
            int id = trie[x].end;
            if (id) cnt[id-1]++;
            x = trie[x].exit;
        }
    }
}

```

1.6.2 Hashing

```


*/
Convierte el string en un polinomio, en O(n), tal que podemos
comparar substrings como valores numericos en O(1).
Primero llamar calc_xpow() (una unica vez) con el largo maximo
de los strings dados.
*/
inline int add(int a, int b, const int &mod) { return a+b >= mod
    ? a+b-mod : a+b; }
inline int sbt(int a, int b, const int &mod) { return a-b < 0 ?
    a-b+mod : a-b; }
inline int mul(int a, int b, const int &mod) { return 1ll*a*b %
    mod; }

const int X[] = {257, 359};
const int MOD[] = {(int)1e9+7, (int)1e9+9};
vector<int> xpow[2];

struct hashing {
    vector<int> h[2];

    hashing(string &s) {
        int n = s.size();
        for (int j = 0; j < 2; ++j) {
            h[j].resize(n+1);
            for (int i = 1; i <= n; ++i) {
                h[j][i] = add(mul(h[j][i-1], X[j], MOD[j]),
                    s[i-1], MOD[j]);
            }
        }
    }
    //Hash del substring en el rango [i, j)
    ll value(int l, int r) {


```

```


        int a = sbt(h[0][r], mul(h[0][l], xpow[0][r-l], MOD[0]),
            MOD[0]);
        int b = sbt(h[1][r], mul(h[1][l], xpow[1][r-l], MOD[1]),
            MOD[1]);
        return (1ll(a)<<32) + b;
    }
};

void calc_xpow(int mxlen) {
    for (int j = 0; j < 2; ++j) {
        xpow[j].resize(mxlen+1, 1);
        for (int i = 1; i <= mxlen; ++i) {
            xpow[j][i] = mul(xpow[j][i-1], X[j], MOD[j]);
        }
    }
}


```

1.6.3 Manacher

```


// O(n) => solo string tamanio impar
vector<int> manacher_odd(string s) {
    int n = s.size();
    s = "$" + s + "^";
    vector<int> p(n + 2);
    int l = 1, r = 1;
    for(int i = 1; i <= n; i++) {
        p[i] = max(0, min(r - i, p[l + (r - i)]));
        while(s[i - p[i]] == s[i + p[i]]) {
            p[i]++;
        }
        if(i + p[i] > r) {
            l = i - p[i], r = i + p[i];
        }
    }
}


```

```

    return vector<int>(begin(p) + 1, end(p) - 1);
}

// Para cualquier tamaño par o impar
vector<int> manacher(string s) {
    string t;
    for(auto c: s) {
        t += string("#") + c;
    }
    auto res = manacher_odd(t + "#");
    return vector<int>(begin(res) + 1, end(res) - 1);
}

```

1.6.4 Prefix Function

```

vector<int> prefix_function(string s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}

```

1.6.5 Suffix Array

```

const int MAXL = 300;

struct suffixArray {
    string s;
    int n, MX;
    vector<int> ra, tra, sa, tsa, lcp;

    suffixArray(string &s) {
        s = _s + "$";
        n = s.size();
        MX = max(MAXL, n)+2;
        ra = tra = sa = tsa = lcp = vector<int>(n);
        build();
    }

    void radix_sort(int k) {
        vector<int> cnt(MX, 0);
        for(int i = 0; i < n; i++)
            cnt[(i+k < n) ? ra[i+k]+1 : 1]++;
        for(int i = 1; i < MX; i++)
            cnt[i] += cnt[i-1];
        for(int i = 0; i < n; i++)
            tsa[cnt[(sa[i]+k < n) ? ra[sa[i]+k] : 0]++] = sa[i];
        sa = tsa;
    }

    void build() {
        for (int i = 0; i < n; i++)
            ra[i] = s[i], sa[i] = i;
        for (int k = 1, r; k < n; k <= 1) {
            radix_sort(k);
            radix_sort(0);
            tra[sa[0]] = r = 0;
            for (int i = 1; i < n; i++) {

```

```

        if (ra[sa[i]] != ra[sa[i-1]] || ra[sa[i]+k] !=
            ra[sa[i-1]+k]) ++r;
        tra[sa[i]] = r;
    }
    ra = tra;
    if (ra[sa[n-1]] == n-1) break;
}

int& operator[] (int i) { return sa[i]; }

void build_lcp() {
    lcp[0] = 0;
    for (int i = 0, k = 0; i < n; i++) {
        if (!ra[i]) continue;
        while (s[i+k] == s[sa[ra[i]-1]+k]) k++;
        lcp[ra[i]] = k;
        if (k) k--;
    }
}

//Longest Common Substring: construir el suffixArray s = s1
+ "#" + s2 + "$" y m = s2.size()
pair<int, int> lcs() {
    int mx = -1, ind = -1;
    for (int i = 1; i < n; i++) {
        if (((sa[i] < n-m-1) != (sa[i-1] < n-m-1)) && mx <
            lcp[i]) {
            mx = lcp[i]; ind = i;
        }
    }
    return {mx, ind};
}
};

```

1.6.6 Suffix Automaton

```

struct suffixAutomaton {
    struct node {
        int len, link; bool end;
        map<char, int> next;
        int cnt; ll in, out;
    };

    vector<node> sa;
    int last; ll substrs = 0;

    suffixAutomaton() {}
    suffixAutomaton(string &s) {
        sa.reserve(s.size()*2);
        last = add_node();
        sa[0].link = -1;
        sa[0].in = 1;
        for (char &c : s) add_char(c);
        for (int p = last; p; p = sa[p].link) sa[p].end = 1;
    }

    int add_node() { sa.pb({}); return sa.size()-1; }

    void add_char(char c) {
        int u = add_node(), p = last;
        sa[u].len = sa[last].len + 1;
        while (p != -1 && !sa[p].next.count(c)) {
            sa[p].next[c] = u;
            sa[u].in += sa[p].in;
            substrs += sa[p].in;
            p = sa[p].link;
        }
        if (p != -1) {
            int q = sa[p].next[c];

```

```

    if (sa[p].len + 1 != sa[q].len) {
        int clone = add_node();
        sa[clone] = sa[q];
        sa[clone].len = sa[p].len + 1;
        sa[clone].in = 0;
        sa[q].link = sa[u].link = clone;
        while (p != -1 && sa[p].next[c] == q) {
            sa[p].next[c] = clone;
            sa[q].in -= sa[p].in;
            sa[clone].in += sa[p].in;
            p = sa[p].link;
        }
        } else sa[u].link = q;
    }
    last = u;
}

void run(string &s) {
    int u = 0;
    for (int i = 0; i < s.size(); ++i) {
        while (u && !sa[u].next.count(s[i])) u = sa[u].link;
        if (sa[u].next.count(s[i])) u = sa[u].next[s[i]];
    }
}

int match_str(string &s) {
    int u = 0, n = s.size();
    for (int i = 0; i < n; ++i) {
        if (!sa[u].next.count(s[i])) return 0;
        u = sa[u].next[s[i]];
    }
    return count_occ(u);
}

int count_occ(int u) {

```

```

    if (sa[u].cnt != 0) return sa[u].cnt;
    sa[u].cnt = sa[u].end;
    for (auto &v : sa[u].next)
        sa[u].cnt += count_occ(v.S);
    return sa[u].cnt;
}

ll count_paths(int u) {
    if (sa[u].out != 0) return sa[u].out;
    for (auto &v : sa[u].next)
        sa[u].out += count_paths(v.S) + 1;
    return sa[u].out;
}

node& operator[](int i) { return sa[i]; }
};

```

1.6.7 Trie

```

const int N = 1e6 + 100;
int trie[N][26]; // N = suma de longitudes
bool stop[N];
int ct = 0;

void insert(string word) {
    int node = 0;
    for(int i = 0; i < (int)word.size(); i++) {
        if(!trie[node][word[i] - 'a']) trie[node][word[i] - 'a'] = ++ct;
        node = trie[node][word[i] - 'a'];
    }
    stop[node] = true;
}

```

1.6.8 Z Function

```
vector<int> z_function(string s) {  
    int n = s.size();  
    vector<int> z(n);  
    int l = 0, r = 0;  
    for(int i = 1; i < n; i++) {  
        if(i < r) {  
            z[i] = min(r - i, z[i - l]);  
        }  
        while(i + z[i] < n && s[z[i]] == s[i + z[i]]) {
```

```
            z[i]++;  
        }  
        if(i + z[i] > r) {  
            l = i;  
            r = i + z[i];  
        }  
    }  
    return z;  
}
```
