Competições de Algoritmos e Programação UFRJ

Universidade Federal do Rio de Janeiro

Meu nome é shadow. Sou aquele que deixou todo o seu passado para trás. Não faço esse contest por ninguém, não estou preso a nada

Enzo Vieira, Caio David, Luís Rafael

Setembro, 2025

Sumário

1 Geometria

1.1 Minkowski Sum

```
// Computa A+B = {a+b : a \in A, b \in B}, em que
// A e B sao poligonos convexos
// A+B eh um poligono convexo com no max |A|+|B| pontos
//
// O(|A|+|B|)
// Do cadeno do Brunas Maletas UFMG

vector<pt> minkowski(vector<pt> p, vector<pt> q) {
    auto fix = [](vector<pt>& P) {
        rotate(P.begin(), min_element(P.begin(), P.end()), P.end());
        P.push_back(P[0]), P.push_back(P[1]);
    };
    fix(p), fix(q);
    vector<pt> ret;
    int i = 0, j = 0;
    while (i < p.size()-2 or j < q.size()-2) {
        ret.push_back(p[i] + q[j]);
        auto c = ((p[i+1] - p[i]) ^ (q[j+1] - q[j]));
        if (c >= 0) i = min<int>(i+1, p.size()-2);
        if (c <= 0) j = min<int>(j+1, q.size()-2);
    }
    return ret;
}

ld dist_convex(vector<pt> p, vector<pt> q) {
    for (pt& i : p) i = i * -1;
    auto s = minkowski(p, q);
    if (inpol(s, pt(0, 0))) return 0;
    ld ans = DINF;
    for (int i = 0; i < s.size(); i++) ans = min(ans,
            disttoseg(pt(0, 0), line(s[(i+1)%s.size()], s[i])));
    return ans;
}
```

1.2 Primitiva Double

```
typedef double ld;
const ld DINF = 1e18;
const ld pi = acos(-1.0);
const ld eps = 1e-9;

#define sq(x) ((x)*(x))

bool eq(ld a, ld b) {
    return abs(a - b) <= eps;
}

struct pt { // ponto
    ld x, y;
    pt(ld x_ = 0, ld y_ = 0) : x(x_), y(y_) {}
    bool operator < (const pt p) const {
        if (!eq(x, p.x)) return x < p.x;
        if (!eq(y, p.y)) return y < p.y;
        return 0;
    }
```

```cpp
    bool operator == (const pt p) const {
        return eq(x, p.x) and eq(y, p.y);
    }
    pt operator + (const pt p) const { return pt(x+p.x, y+p.y); }
    pt operator - (const pt p) const { return pt(x-p.x, y-p.y); }
    pt operator * (const ld c) const { return pt(x*c  , y*c  ); }
    pt operator / (const ld c) const { return pt(x/c  , y/c  ); }
    ld operator * (const pt p) const { return x*p.x + y*p.y; }
    ld operator ^ (const pt p) const { return x*p.y - y*p.x; }
    friend istream& operator >> (istream& in, pt& p) {
        return in >> p.x >> p.y;
    }
};

struct line { // reta
    pt p, q;
    line() {}
    line(pt p_, pt q_) : p(p_), q(q_) {}
    friend istream& operator >> (istream& in, line& r) {
        return in >> r.p >> r.q;
    }
};

// PONTO & VETOR

ld dist(pt p, pt q) { // distancia
    return hypot(p.y - q.y, p.x - q.x);
}

ld dist2(pt p, pt q) { // quadrado da distancia
    return sq(p.x - q.x) + sq(p.y - q.y);
}

ld norm(pt v) { // norma do vetor
    return dist(pt(0, 0), v);
}

ld angle(pt v) { // angulo do vetor com o eixo x
    ld ang = atan2(v.y, v.x);
    if (ang < 0) ang += 2*pi;
    return ang;
}

ld sarea(pt p, pt q, pt r) { // area com sinal
    return ((q-p)^(r-q))/2;
}

bool col(pt p, pt q, pt r) { // se p, q e r sao colin.
    return eq(sarea(p, q, r), 0);
}

bool ccw(pt p, pt q, pt r) { // se p, q, r sao ccw
    return sarea(p, q, r) > eps;
}

pt rotate(pt p, ld th) { // rotaciona o ponto th radianos
    return pt(p.x * cos(th) - p.y * sin(th),
            p.x * sin(th) + p.y * cos(th));
}

pt rotate90(pt p) { // rotaciona 90 graus
    return pt(-p.y, p.x);
}

// RETA

bool isvert(line r) { // se r eh vertical
    return eq(r.p.x, r.q.x);
}

bool isinseg(pt p, line r) { // se p pertence ao seg de r
    pt a = r.p - p, b = r.q - p;
    return eq((a ^ b), 0) and (a * b) < eps;
}

ld get_t(pt v, line r) { // retorna t tal que t*v pertence a reta r
    return (r.p^r.q) / ((r.p-r.q)^v);
}

pt proj(pt p, line r) { // projecao do ponto p na reta r
    if (r.p == r.q) return r.p;
    r.q = r.q - r.p; p = p - r.p;
    pt proj = r.q * ((p*r.q) / (r.q*r.q));
    return proj + r.p;
}

pt inter(line r, line s) { // r inter s
    if (eq((r.p - r.q) ^ (s.p - s.q), 0)) return pt(DINF, DINF);
    r.q = r.q - r.p, s.p = s.p - r.p, s.q = s.q - r.p;
    return r.q * get_t(r.q, s) + r.p;
}

bool interseg(line r, line s) { // se o seg de r intersecta o seg de s
    if (isinseg(r.p, s) or isinseg(r.q, s)
```

```
        or isinseg(s.p, r) or isinseg(s.q, r)) return 1;

    return ccw(r.p, r.q, s.p) != ccw(r.p, r.q, s.q) and
            ccw(s.p, s.q, r.p) != ccw(s.p, s.q, r.q);
}

ld disttoline(pt p, line r) { // distancia do ponto a reta
    return 2 * abs(sarea(p, r.p, r.q)) / dist(r.p, r.q);
}

ld disttoseg(pt p, line r) { // distancia do ponto ao seg
    if ((r.q - r.p)*(p - r.p) < 0) return dist(r.p, p);
    if ((r.p - r.q)*(p - r.q) < 0) return dist(r.q, p);
    return disttoline(p, r);
}

ld distseg(line a, line b) { // distancia entre seg
    if (interseg(a, b)) return 0;

    ld ret = DINF;
    ret = min(ret, disttoseg(a.p, b));
    ret = min(ret, disttoseg(a.q, b));
    ret = min(ret, disttoseg(b.p, a));
    ret = min(ret, disttoseg(b.q, a));

    return ret;
}

// POLIGONO

// corta poligono com a reta r deixando os pontos p tal que
// ccw(r.p, r.q, p)
vector<pt> cut_polygon(vector<pt> v, line r) { // O(n)
    vector<pt> ret;
    for (int j = 0; j < v.size(); j++) {
        if (ccw(r.p, r.q, v[j])) ret.push_back(v[j]);
        if (v.size() == 1) continue;
        line s(v[j], v[(j+1)%v.size()]);
        pt p = inter(r, s);
        if (isinseg(p, s)) ret.push_back(p);
    }
    ret.erase(unique(ret.begin(), ret.end()), ret.end());
    if (ret.size() > 1 and ret.back() == ret[0]) ret.pop_back();
    return ret;
}

// distancia entre os retangulos a e b (lados paralelos aos eixos)
// assume que ta representado (inferior esquerdo, superior direito)
ld dist_rect(pair<pt, pt> a, pair<pt, pt> b) {
    ld hor = 0, vert = 0;
    if (a.second.x < b.first.x) hor = b.first.x - a.second.x;
    else if (b.second.x < a.first.x) hor = a.first.x - b.second.x;
    if (a.second.y < b.first.y) vert = b.first.y - a.second.y;
    else if (b.second.y < a.first.y) vert = a.first.y - b.second.y;
    return dist(pt(0, 0), pt(hor, vert));
}

ld polarea(vector<pt> v) { // area do poligono
    ld ret = 0;
    for (int i = 0; i < v.size(); i++)
        ret += sarea(pt(0, 0), v[i], v[(i + 1) % v.size()]);
    return abs(ret);
}

// se o ponto ta dentro do poligono: retorna 0 se ta fora,
// 1 se ta no interior e 2 se ta na borda
int inpol(vector<pt>& v, pt p) { // O(n)
    int qt = 0;
    for (int i = 0; i < v.size(); i++) {
        if (p == v[i]) return 2;
        int j = (i+1)%v.size();
        if (eq(p.y, v[i].y) and eq(p.y, v[j].y)) {
            if ((v[i]-p)*(v[j]-p) < eps) return 2;
            continue;
        }
        bool baixo = v[i].y+eps < p.y;
        if (baixo == (v[j].y+eps < p.y)) continue;
        auto t = (p-v[i])^(v[j]-v[i]);
        if (eq(t, 0)) return 2;
        if (baixo == (t > eps)) qt += baixo ? 1 : -1;
    }
    return qt != 0;
}

bool interpol(vector<pt> v1, vector<pt> v2) { // se dois poligonos se intersectam -
↪  O(n*m)
    int n = v1.size(), m = v2.size();
    for (int i = 0; i < n; i++) if (inpol(v2, v1[i])) return 1;
    for (int i = 0; i < n; i++) if (inpol(v1, v2[i])) return 1;
    for (int i = 0; i < n; i++) for (int j = 0; j < m; j++)
        if (interseg(line(v1[i], v1[(i+1)%n]), line(v2[j], v2[(j+1)%m]))) return 1;
    return 0;
}
```

```cpp
ld distpol(vector<pt> v1, vector<pt> v2) { // distancia entre poligonos
    if (interpol(v1, v2)) return 0;

    ld ret = DINF;

    for (int i = 0; i < v1.size(); i++) for (int j = 0; j < v2.size(); j++)
        ret = min(ret, distseg(line(v1[i], v1[(i + 1) % v1.size()]),
                    line(v2[j], v2[(j + 1) % v2.size()])));
    return ret;
}

vector<pt> convex_hull(vector<pt> v) { // convex hull - O(n log(n))
    sort(v.begin(), v.end());
    v.erase(unique(v.begin(), v.end()), v.end());
    if (v.size() <= 1) return v;
    vector<pt> l, u;
    for (int i = 0; i < v.size(); i++) {
        while (l.size() > 1 and !ccw(l.end()[-2], l.end()[-1], v[i]))
            l.pop_back();
        l.push_back(v[i]);
    }
    for (int i = v.size() - 1; i >= 0; i--) {
        while (u.size() > 1 and !ccw(u.end()[-2], u.end()[-1], v[i]))
            u.pop_back();
        u.push_back(v[i]);
    }
    l.pop_back(); u.pop_back();
    for (pt i : u) l.push_back(i);
    return l;
}

struct convex_pol {
    vector<pt> pol;

        // nao pode ter ponto colinear no convex hull
    convex_pol() {}
    convex_pol(vector<pt> v) : pol(convex_hull(v)) {}

        // se o ponto ta dentro do hull - O(log(n))
    bool is_inside(pt p) {
        if (pol.size() == 0) return false;
        if (pol.size() == 1) return p == pol[0];
        int l = 1, r = pol.size();
        while (l < r) {
            int m = (l+r)/2;
            if (ccw(p, pol[0], pol[m])) l = m+1;
            else r = m;
        }
        if (l == 1) return isinseg(p, line(pol[0], pol[1]));
        if (l == pol.size()) return false;
        return !ccw(p, pol[l], pol[l-1]);
    }
        // ponto extremo em relacao a cmp(p, q) = p mais extremo q
        // (copiado de https://github.com/gustavoM32/caderno-zika)
    int extreme(const function<bool(pt, pt)>& cmp) {
        int n = pol.size();
        auto extr = [&](int i, bool& cur_dir) {
            cur_dir = cmp(pol[(i+1)%n], pol[i]);
            return !cur_dir and !cmp(pol[(i+n-1)%n], pol[i]);
        };
        bool last_dir, cur_dir;
        if (extr(0, last_dir)) return 0;
        int l = 0, r = n;
        while (l+1 < r) {
            int m = (l+r)/2;
            if (extr(m, cur_dir)) return m;
            bool rel_dir = cmp(pol[m], pol[l]);
            if ((!last_dir and cur_dir) or
                    (last_dir == cur_dir and rel_dir == cur_dir)) {
                l = m;
                last_dir = cur_dir;
            } else r = m;
        }
        return l;
    }
    int max_dot(pt v) {
        return extreme([&](pt p, pt q) { return p*v > q*v; });
    }
    pair<int, int> tangents(pt p) {
        auto L = [&](pt q, pt r) { return ccw(p, r, q); };
        auto R = [&](pt q, pt r) { return ccw(p, q, r); };
        return {extreme(L), extreme(R)};
    }
};

// CIRCUNFERENCIA

pt getcenter(pt a, pt b, pt c) { // centro da circunf dado 3 pontos
    b = (a + b) / 2;
    c = (a + c) / 2;
    return inter(line(b, b + rotate90(a - b)),
            line(c, c + rotate90(a - c)));
}
```

```
vector<pt> circ_line_inter(pt a, pt b, pt c, ld r) { // intersecao da circunf (c, r)
↪  e reta ab
    vector<pt> ret;
    b = b-a, a = a-c;
    ld A = b*b;
    ld B = a*b;
    ld C = a*a - r*r;
    ld D = B*B - A*C;
    if (D < -eps) return ret;
    ret.push_back(c+a+b*(-B+sqrt(D+eps))/A);
    if (D > eps) ret.push_back(c+a+b*(-B-sqrt(D))/A);
    return ret;
}

vector<pt> circ_inter(pt a, pt b, ld r, ld R) { // intersecao da circunf (a, r) e
↪  (b, R)
    vector<pt> ret;
    ld d = dist(a, b);
    if (d > r+R or d+min(r, R) < max(r, R)) return ret;
    ld x = (d*d-R*R+r*r)/(2*d);
    ld y = sqrt(r*r-x*x);
    pt v = (b-a)/d;
    ret.push_back(a+v*x + rotate90(v)*y);
    if (y > 0) ret.push_back(a+v*x - rotate90(v)*y);
    return ret;
}

bool operator <(const line& a, const line& b) { // comparador pra reta
        // assume que as retas tem p < q
    pt v1 = a.q - a.p, v2 = b.q - b.p;
    if (!eq(angle(v1), angle(v2))) return angle(v1) < angle(v2);
    return ccw(a.p, a.q, b.p); // mesmo angulo
}
bool operator ==(const line& a, const line& b) {
    return !(a < b) and !(b < a);
}

// comparador pro set pra fazer sweep line com segmentos
struct cmp_sweepline {
    bool operator () (const line& a, const line& b) const {
            // assume que os segmentos tem p < q
        if (a.p == b.p) return ccw(a.p, a.q, b.q);
        if (!eq(a.p.x, a.q.x) and (eq(b.p.x, b.q.x) or a.p.x+eps < b.p.x))
            return ccw(a.p, a.q, b.p);
        return ccw(a.p, b.q, b.p);
    }
};

// comparador pro set pra fazer sweep angle com segmentos
pt dir;
struct cmp_sweepangle {
    bool operator () (const line& a, const line& b) const {
        return get_t(dir, a) + eps < get_t(dir, b);
    }
};
```

1.3 Primitiva Inteiro

```
#define sq(x) ((x)*(ll)(x))

struct pt { // ponto
    int x, y;
    pt(int x_ = 0, int y_ = 0) : x(x_), y(y_) {}
    bool operator < (const pt p) const {
        if (x != p.x) return x < p.x;
        return y < p.y;
    }
    bool operator == (const pt p) const {
        return x == p.x and y == p.y;
    }
    pt operator + (const pt p) const { return pt(x+p.x, y+p.y); }
    pt operator - (const pt p) const { return pt(x-p.x, y-p.y); }
    pt operator * (const int c) const { return pt(x*c, y*c); }
    ll operator * (const pt p) const { return x*(ll)p.x + y*(ll)p.y; }
    ll operator ^ (const pt p) const { return x*(ll)p.y - y*(ll)p.x; }
    friend istream& operator >> (istream& in, pt& p) {
        return in >> p.x >> p.y;
    }
};

struct line { // reta
    pt p, q;
    line() {}
    line(pt p_, pt q_) : p(p_), q(q_) {}
    friend istream& operator >> (istream& in, line& r) {
        return in >> r.p >> r.q;
    }
};

// PONTO & VETOR
```

```
ll dist2(pt p, pt q) { // quadrado da distancia
    return sq(p.x - q.x) + sq(p.y - q.y);
}

ll sarea2(pt p, pt q, pt r) { // 2 * area com sinal
    return (q-p)^(r-q);
}

bool col(pt p, pt q, pt r) { // se p, q e r sao colin.
    return sarea2(p, q, r) == 0;
}

bool ccw(pt p, pt q, pt r) { // se p, q, r sao ccw
    return sarea2(p, q, r) > 0;
}

int quad(pt p) { // quadrante de um ponto
    return (p.x<0)^3*(p.y<0);
}

bool compare_angle(pt p, pt q) { // retorna se ang(p) < ang(q)
    if (quad(p) != quad(q)) return quad(p) < quad(q);
    return ccw(q, pt(0, 0), p);
}

pt rotate90(pt p) { // rotaciona 90 graus
    return pt(-p.y, p.x);
}

// RETA

bool isinseg(pt p, line r) { // se p pertence ao seg de r
    pt a = r.p - p, b = r.q - p;
    return (a ^ b) == 0 and (a * b) <= 0;
}

bool interseg(line r, line s) { // se o seg de r intersecta o seg de s
    if (isinseg(r.p, s) or isinseg(r.q, s)
        or isinseg(s.p, r) or isinseg(s.q, r)) return 1;

    return ccw(r.p, r.q, s.p) != ccw(r.p, r.q, s.q) and
            ccw(s.p, s.q, r.p) != ccw(s.p, s.q, r.q);
}

int segpoints(line r) { // numero de pontos inteiros no segmento
    return 1 + __gcd(abs(r.p.x - r.q.x), abs(r.p.y - r.q.y));
}

double get_t(pt v, line r) { // retorna t tal que t*v pertence a reta r
    return (r.p^r.q) / (double) ((r.p-r.q)^v);
}

// POLIGONO

// quadrado da distancia entre os retangulos a e b (lados paralelos aos eixos)
// assume que ta representado (inferior esquerdo, superior direito)
ll dist2_rect(pair<pt, pt> a, pair<pt, pt> b) {
    int hor = 0, vert = 0;
    if (a.second.x < b.first.x) hor = b.first.x - a.second.x;
    else if (b.second.x < a.first.x) hor = a.first.x - b.second.x;
    if (a.second.y < b.first.y) vert = b.first.y - a.second.y;
    else if (b.second.y < a.first.y) vert = a.first.y - b.second.y;
    return sq(hor) + sq(vert);
}

ll polarea2(vector<pt> v) { // 2 * area do poligono
    ll ret = 0;
    for (int i = 0; i < v.size(); i++)
        ret += sarea2(pt(0, 0), v[i], v[(i + 1) % v.size()]);
    return abs(ret);
}

// se o ponto ta dentro do poligono: retorna 0 se ta fora,
// 1 se ta no interior e 2 se ta na borda
int inpol(vector<pt>& v, pt p) { // O(n)
    int qt = 0;
    for (int i = 0; i < v.size(); i++) {
        if (p == v[i]) return 2;
        int j = (i+1)%v.size();
        if (p.y == v[i].y and p.y == v[j].y) {
            if ((v[i]-p)*(v[j]-p) <= 0) return 2;
            continue;
        }
        bool baixo = v[i].y < p.y;
        if (baixo == (v[j].y < p.y)) continue;
        auto t = (p-v[i])^(v[j]-v[i]);
        if (!t) return 2;
        if (baixo == (t > 0)) qt += baixo ? 1 : -1;
    }
    return qt != 0;
}

vector<pt> convex_hull(vector<pt> v) { // convex hull - O(n log(n))
```

```cpp
    sort(v.begin(), v.end());
    v.erase(unique(v.begin(), v.end()), v.end());
    if (v.size() <= 1) return v;
    vector<pt> l, u;
    for (int i = 0; i < v.size(); i++) {
        while (l.size() > 1 and !ccw(l.end()[-2], l.end()[-1], v[i]))
            l.pop_back();
        l.push_back(v[i]);
    }
    for (int i = v.size() - 1; i >= 0; i--) {
        while (u.size() > 1 and !ccw(u.end()[-2], u.end()[-1], v[i]))
            u.pop_back();
        u.push_back(v[i]);
    }
    l.pop_back(); u.pop_back();
    for (pt i : u) l.push_back(i);
    return l;
}

ll interior_points(vector<pt> v) { // pontos inteiros dentro de um poligono simples
    ll b = 0;
    for (int i = 0; i < v.size(); i++)
        b += segpoints(line(v[i], v[(i+1)%v.size()])) - 1;
    return (polarea2(v) - b) / 2 + 1;
}

struct convex_pol {
    vector<pt> pol;

        // nao pode ter ponto colinear no convex hull
    convex_pol() {}
    convex_pol(vector<pt> v) : pol(convex_hull(v)) {}

        // se o ponto ta dentro do hull - O(log(n))
    bool is_inside(pt p) {
        if (pol.size() == 0) return false;
        if (pol.size() == 1) return p == pol[0];
        int l = 1, r = pol.size();
        while (l < r) {
            int m = (l+r)/2;
            if (ccw(p, pol[0], pol[m])) l = m+1;
            else r = m;
        }
        if (l == 1) return isinseg(p, line(pol[0], pol[1]));
        if (l == pol.size()) return false;
        return !ccw(p, pol[l], pol[l-1]);
    }
        // ponto extremo em relacao a cmp(p, q) = p mais extremo q
        // (copiado de https://github.com/gustavoM32/caderno-zika)
    int extreme(const function<bool(pt, pt)>& cmp) {
        int n = pol.size();
        auto extr = [&](int i, bool& cur_dir) {
            cur_dir = cmp(pol[(i+1)%n], pol[i]);
            return !cur_dir and !cmp(pol[(i+n-1)%n], pol[i]);
        };
        bool last_dir, cur_dir;
        if (extr(0, last_dir)) return 0;
        int l = 0, r = n;
        while (l+1 < r) {
            int m = (l+r)/2;
            if (extr(m, cur_dir)) return m;
            bool rel_dir = cmp(pol[m], pol[l]);
            if ((!last_dir and cur_dir) or
                    (last_dir == cur_dir and rel_dir == cur_dir)) {
                l = m;
                last_dir = cur_dir;
            } else r = m;
        }
        return l;
    }
    int max_dot(pt v) {
        return extreme([&](pt p, pt q) { return p*v > q*v; });
    }
    pair<int, int> tangents(pt p) {
        auto L = [&](pt q, pt r) { return ccw(p, r, q); };
        auto R = [&](pt q, pt r) { return ccw(p, q, r); };
        return {extreme(L), extreme(R)};
    }
};

bool operator <(const line& a, const line& b) { // comparador pra reta
        // assume que as retas tem p < q
    pt v1 = a.q - a.p, v2 = b.q - b.p;
    bool b1 = compare_angle(v1, v2), b2 = compare_angle(v2, v1);
    if (b1 or b2) return b1;
    return ccw(a.p, a.q, b.p); // mesmo angulo
}
bool operator ==(const line& a, const line& b) {
    return !(a < b) and !(b < a);
}

// comparador pro set pra fazer sweep line com segmentos
struct cmp_sweepline {
    bool operator () (const line& a, const line& b) const {
```

```
            // assume que os segmentos tem p < q
        if (a.p == b.p) return ccw(a.p, a.q, b.q);
        if (a.p.x != a.q.x and (b.p.x == b.q.x or a.p.x < b.p.x))
            return ccw(a.p, a.q, b.p);
        return ccw(a.p, b.q, b.p);
    }
};

// comparador pro set pra fazer sweep angle com segmentos
pt dir;
struct cmp_sweepangle {
    bool operator () (const line& a, const line& b) const {
        return get_t(dir, a) < get_t(dir, b);
    }
};
```

1.4 Simple Polygon

```
// Verifica se um poligono com n pontos eh simples
//
// O(n log n)
// Direto do Caderno do Brullas Mano

bool operator < (const line& a, const line& b) { // comparador pro sweepline
    if (a.p == b.p) return ccw(a.p, a.q, b.q);
    if (!eq(a.p.x, a.q.x) and (eq(b.p.x, b.q.x) or a.p.x+eps < b.p.x))
        return ccw(a.p, a.q, b.p);
    return ccw(a.p, b.q, b.p);
}

bool simple(vector<pt> v) {
    auto intersects = [&](pair<line, int> a, pair<line, int> b) {
        if ((a.second+1)%v.size() == b.second or
            (b.second+1)%v.size() == a.second) return false;
        return interseg(a.first, b.first);
    };
    vector<line> seg;
    vector<pair<pt, pair<int, int>>> w;
    for (int i = 0; i < v.size(); i++) {
        pt at = v[i], nxt = v[(i+1)%v.size()];
        if (nxt < at) swap(at, nxt);
        seg.push_back(line(at, nxt));
        w.push_back({at, {0, i}});
        w.push_back({nxt, {1, i}});
            // casos degenerados estranhos
        if (isinseg(v[(i+2)%v.size()], line(at, nxt))) return 0;
        if (isinseg(v[(i+v.size()-1)%v.size()], line(at, nxt))) return 0;
    }
    sort(w.begin(), w.end());
    set<pair<line, int>> se;
    for (auto i : w) {
        line at = seg[i.second.second];
        if (i.second.first == 0) {
            auto nxt = se.lower_bound({at, i.second.second});
            if (nxt != se.end() and intersects(*nxt, {at, i.second.second})) return
↪   0;
            if (nxt != se.begin() and intersects(*(--nxt), {at, i.second.second}))
↪   return 0;
            se.insert({at, i.second.second});
        } else {
            auto nxt = se.upper_bound({at, i.second.second}), cur = nxt, prev =
↪   --cur;
            if (nxt != se.end() and prev != se.begin()
                and intersects(*nxt, *(--prev))) return 0;
            se.erase(cur);
        }
    }
    return 1;
}
```

2 Matematica

2.1 Combinatorics

```
const int maxn = 1e6;
vector<ll> fact(maxn+1), ifact(maxn+1);

ll fastexp(ll b, ll e){
    ll res = 1;
    while(e){
        if(e&1) res = (res * b)%mod;
        b = (b * b)%mod;
        e/=2;
    }
    return res;
}
```

```cpp
ll inv(ll x){
    return fastexp(x, mod-2);
}

ll choose(ll a, ll b){
    if(a < b) return 0;
    return fact[a] * ifact[b] %mod * ifact[a-b] %mod;
}

void build(){

    fact[0] = 1;
    for(int i = 1; i <= maxn; i++) fact[i] = (fact[i-1] * i)%mod;
    ifact[maxn] = inv(fact[maxn]);
    for(int i = maxn-1; i >= 0; i--) ifact[i] = (ifact[i+1] * (i+1))%mod;

}
```

2.2 Crivo + Fatoração

```cpp
struct Sieve{
    int maxn;
    vector <int> is_prime, min_div;
    Sieve(int n){
        this->maxn = n;
        is_prime.assign(n+1, 1);
        min_div.resize(n+1);

        for(int i = 0; i <= n; i++)
            min_div[i] = i;

        is_prime[0] = is_prime[1] = 0;
        for (int i = 2; i <= n; i++) {
            if (is_prime[i] && (long long)i * i <= n) {
                for (int j = i * i; j <= n; j += i){
                    if(is_prime[j]) min_div[j] = i;
                    is_prime[j] = false;
                }
            }
        }
    }

    vector<pair<int,int>> factorize(int n){
        assert(n <= maxn);
        vector <pair<int,int>> fact;
        while(n > 1){
            if(fact.empty() || fact.back().first != min_div[n]){
                fact.push_back({min_div[n], 1});
            }else{
                fact.back().second += 1;
            }
            n /= min_div[n];
        }
        return fact;
    }
};
```

2.3 Euclides Estendido

```cpp
//Retorna o GCD de a e b, e os coeficientes x e y
//tais que ax + by = gcd(a, b).
//Complexidade: O(log(min(a, b)))

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

2.4 FFT + NTT

```cpp
struct FFT{
    typedef complex<double> C;
    typedef vector<double> vd;
    typedef vector<long long int> vl;
    typedef vector<int> vi;
```

```
/*
 * Author: Ludo Pulles, chilli, Simon Lindholm
 * Date: 2019-01-09
 * License: CC0
 * Source: http://neerc.ifmo.ru/trains/toulouse/2017/fft2.pdf (do read, it's
↪ excellent)
   Accuracy bound from http://www.daemonology.net/papers/fft.pdf
 * Description: fft(a) computes $\hat f(k) = \sum_x a[x] \exp(2\pi i \cdot k
↪ x / N)$ for all $k$. N must be a power of 2.
   Useful for convolution:
   \texttt{conv(a, b) = c}, where $c[x] = \sum a[i]b[x-i]$.
   For convolution of complex numbers or more than two vectors: FFT, multiply
   pointwise, divide by n, reverse(start+1, end), FFT back.
   Rounding is safe if $(\sum a_i^2 + \sum b_i^2)\log_2{N} < 9\cdot10^{14}$
   (in practice $10^{16}$; higher for random inputs).
   Otherwise, use NTT/FFTMod.
 * Time: O(N \log N) with $N = |A|+|B|$ ($\tilde 1s$ for $N=2^{22}$)
 * Status: somewhat tested
 * Details: An in-depth examination of precision for both FFT and FFTMod can
↪ be found
 * here
↪ (https://github.com/simonlindholm/fft-precision/blob/master/fft-precision.md)
*/
void fft(vector<C>& a) {
    int n = a.size(), L = 31 - __builtin_clz(n);
    static vector<complex<long double>> R(2, 1);
    static vector<C> rt(2, 1);  // (^ 10% faster if double)
    for (static int k = 2; k < n; k *= 2) {
        R.resize(n); rt.resize(n);
        auto x = polar(1.0L, acos(-1.0L) / k);
        for(int i=k; i<2*k; i++) rt[i] = R[i] = i&1 ? R[i/2] * x : R[i/2];
    }
    vi rev(n);
    for(int i = 0; i < n; i++) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
    for(int i = 0; i < n; i++)  if (i < rev[i]) swap(a[i], a[rev[i]]);
    for (int k = 1; k < n; k *= 2)
        for (int i = 0; i < n; i += 2 * k) for(int j = 0; j < k; j++) {
                // C z = rt[j+k] * a[i+j+k]; // (25% faster if hand-rolled)  ///
↪ include-line
            auto x = (double *)&rt[j+k], y = (double *)&a[i+j+k];        ///
↪ exclude-line
            C z(x[0]*y[0] - x[1]*y[1], x[0]*y[1] + x[1]*y[0]);           ///
↪ exclude-line
            a[i + j + k] = a[i + j] - z;
            a[i + j] += z;
        }
}

vd conv(const vd& a, const vd& b) {
    if (a.empty() || b.empty()) return {};
    vd res(a.size() + b.size() - 1);
    int L = 32 - __builtin_clz(res.size()), n = 1 << L;
    vector<C> in(n), out(n);
    copy(a.begin(), a.end(), begin(in));
    for(int i = 0; i < b.size(); i++) in[i].imag(b[i]);
    fft(in);
    for (C& x : in) x *= x;
    for(int i = 0; i < n; i++) out[i] = in[-i & (n - 1)] - conj(in[i]);
    fft(out);
    for(int i = 0; i < res.size(); i++) res[i] = imag(out[i]) / (4 * n);
    return res;
}

vl conv(const vl& a, const vl& b) {
    if (a.empty() || b.empty()) return {};
    vd res(a.size() + b.size() - 1);
    int L = 32 - __builtin_clz(res.size()), n = 1 << L;
    vector<C> in(n), out(n);
    copy(a.begin(), a.end(), begin(in));
    for(int i = 0; i < b.size(); i++) in[i].imag(b[i]);
    fft(in);
    for (C& x : in) x *= x;
    for(int i = 0; i < n; i++) out[i] = in[-i & (n - 1)] - conj(in[i]);
    fft(out);
    for(int i = 0; i < res.size(); i++) res[i] = imag(out[i]) / (4 * n);
    vl r(a.size() + b.size() - 1);
    for(int i = 0; i < res.size(); i++) r[i] = (ll)(res[i]+.5);
    return r;
}

    /*
    * Author: chilli
    * Date: 2019-04-25
    * License: CC0
    * Source: http://neerc.ifmo.ru/trains/toulouse/2017/fft2.pdf
    * Description: Higher precision FFT, can be used for convolutions modulo
↪ arbitrary integers
    * as long as $N\log_2N\cdot \text{mod} < 8.6 \cdot 10^{14}$ (in practice
↪ $10^{16}$ or higher).
    * Inputs must be in $[0, \text{mod})$.
    * Time: O(N \log N), where $N = |A|+|B|$ (twice as slow as NTT or FFT)
    * Status: stress-tested
    * Details: An in-depth examination of precision for both FFT and FFTMod can
↪ be found
```

```
        * here
↪   (https://github.com/simonlindholm/fft-precision/blob/master/fft-precision.md)
    */
        // multiplica dois polinomios modulo algum inteiro
    template<int M> vl convMod(const vl &a, const vl &b) {
        if (a.empty() || b.empty()) return {};
        vl res(a.size() + b.size() - 1);
        int B=32-__builtin_clz(res.size()), n=1<<B, cut=int(sqrt(M));
        vector<C> L(n), R(n), outs(n), outl(n);
        for(int i = 0; i < a.size(); i++) L[i] = C((int)a[i] / cut, (int)a[i] % cut);
        for(int i = 0; i < b.size(); i++) R[i] = C((int)b[i] / cut, (int)b[i] % cut);
        fft(L), fft(R);
        for(int i = 0; i < n; i++) {
            int j = -i & (n - 1);
            outl[j] = (L[i] + conj(L[j])) * R[i] / (2.0 * n);
            outs[j] = (L[i] - conj(L[j])) * R[i] / (2.0 * n) / 1i;
        }
        fft(outl), fft(outs);
        for(int i = 0; i < res.size(); i++) {
            ll av = ll(real(outl[i])+.5), cv = ll(imag(outs[i])+.5);
            ll bv = ll(imag(outl[i])+.5) + ll(real(outs[i])+.5);
            res[i] = ((av % M * cut + bv) % M * cut + cv) % M;
        }
        return res;
    }


};
```

2.5 Gauss

```
//Complexidade: O(n^3), onde n é o número de variáveis

template<typename T>
pair<int, vector<T>> gauss(vector<vector<T>> a, vector<T> b) {
    const double eps = 1e-6;
    int n = a.size(), m = a[0].size();
    for (int i = 0; i < n; i++) a[i].push_back(b[i]);

    vector<int> where(m, -1);
    for (int col = 0, row = 0; col < m and row < n; col++) {
        int sel = row;
        for (int i=row; i<n; ++i)
            if (abs(a[i][col]) > abs(a[sel][col])) sel = i;
        if (abs(a[sel][col]) < eps) continue;
        for (int i = col; i <= m; i++)
            swap(a[sel][i], a[row][i]);
        where[col] = row;

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

    vector<T> ans(m, 0);
    for (int i = 0; i < m; i++) if (where[i] != -1)
        ans[i] = a[where[i]][m] / a[where[i]][i];
    for (int i = 0; i < n; i++) {
        T sum = 0;
        for (int j = 0; j < m; j++)
            sum += ans[j] * a[i][j];
        if (abs(sum - a[i][m]) > eps)
            return pair(0, vector<T>());
    }

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

2.6 Interpolação

```
//
// Interpolação is a numerical method to
// know the result of a function of degree n
// just by knowing  n+1 point from it
//
//
// Proof of Uniques: say we have another polynome
// of degree <=k M(x). So in M(x) - L(x) = 0 in k+1
// points, but the only function that has K+1 roots
// with degree <=k is f(x) = 0, so
// M(x) - L(X) = 0 -> M(x) = L(x)

struct Interpolation
```

```
{

        //naive implementation O(n^2)
    void interpolate(vector<pair<ll,ll>> &P, int x){

        ll ans = 0;
        for(int i = 0; i < P.size(); i++){
            ll li = 1;
            for(int j = 0; j < P.size(); j++){
                if(i == j) continue;
                li *= (x - P[j].first);
                li /= (P[i].first - P[j].first);
            }
            li *= P[i].second;
            ans += li;
        }
        return ans;

    }

};
```

2.7 Matrix

```
//Utilizado principalmente em recorrencias lineares
//
//https://www.codemarathon.com.br/conteudos/matematica/recorrencia-linear
//


const int D = 2;
const int MOD = 1000000007;
struct Matriz{
  int mat[D][D];
  int* operator[](int i){
    return mat[i];
  }
  Matriz operator*(Matriz oth){
    Matriz res;
    for(int i=0; i<D; i++){
      for(int j=0; j<D; j++){
        res[i][j] = 0;
        for(int k=0; k<D; k++)
          res[i][j] = (res[i][j]+(mat[i][k]*1LL*oth[k][j])%MOD)%MOD;
      }
    }
    return res;
  }
  Matriz exp(long long e){
    Matriz res;
    for(int i=0; i<D; i++)
      for(int j=0; j<D; j++)
        res[i][j] = (i==j);
    Matriz base = *this;
    while(e > 0){
      if(e & 1LL)
        res = res * base;
      base = base*base;
      e = e>>1;
    }
    return res;
  }
};
```

2.8 NTT

```
{
    typedef vector<long long int> vl;
    typedef vector<int> vi;

        /*
    * Author: chilli
    * Date: 2019-04-16
    * License: CC0
    * Source: based on KACTL's FFT
    * Description: ntt(a) computes $\hat f(k) = \sum_x a[x] g^{xk}$ for all $k$,
 ↪  where $g=\text{root}^{(mod-1)/N}$.
    * N must be a power of 2.
    * Useful for convolution modulo specific nice primes of the form $2^a b+1$,
    * where the convolution result has size at most $2^a$. For arbitrary modulo, see
 ↪  FFTMod.
    \texttt{conv(a, b) = c}, where $c[x] = \sum a[i]b[x-i]$.
    For manual convolution: NTT the inputs, multiply
    pointwise, divide by n, reverse(start+1, end), NTT back.
    * Inputs must be in [0, mod).
    * Time: O(N \log N)
    * Status: stress-tested
    */
```

```
    const ll mod = (119 << 23) + 1, root = 62; // = 998244353
        // For p < 2^30 there is also e.g. 5 << 25, 7 << 26, 479 << 21
        // and 483 << 21 (same root). The last two are > 10^9.
    void ntt(vl &a) {
        int n = a.size(), L = 31 - __builtin_clz(n);
        static vl rt(2, 1);
        for (static int k = 2, s = 2; k < n; k *= 2, s++) {
            rt.resize(n);
            ll z[] = {1, modpow(root, mod >> s)};
            for(int i = k; i < 2*k; i++) rt[i] = rt[i / 2] * z[i & 1] % mod;
        }
        vi rev(n);
        for(int i = 0; i < n; i++) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
        for(int i = 0; i < n; i++) if (i < rev[i]) swap(a[i], a[rev[i]]);
        for (int k = 1; k < n; k *= 2)
            for (int i = 0; i < n; i += 2 * k) for(int j = 0; j < k; j++) {
                ll z = rt[j + k] * a[i + j + k] % mod, &ai = a[i + j];
                a[i + j + k] = ai - z + (z > ai ? mod : 0);
                ai += (ai + z >= mod ? z - mod : z);
            }
    }
    vl conv_ntt(const vl &a, const vl &b) {
        if (a.empty() || b.empty()) return {};
        int s = a.size() + b.size() - 1, B = 32 - __builtin_clz(s),
            n = 1 << B;
        int inv = modpow(n, mod - 2);
        vl L(a), R(b), out(n);
        L.resize(n), R.resize(n);
        ntt(L), ntt(R);
        for(int i = 0; i < n; i++)
            out[-i & (n - 1)] = (ll)L[i] * R[i] % mod * inv % mod;
        ntt(out);
        return {out.begin(), out.begin() + s};
    }
    ll modpow(ll b, ll e) {
        ll ans = 1;
        for (; e; b = b * b % mod, e /= 2)
            if (e & 1) ans = ans * b % mod;
        return ans;
    }
};
```

2.9 Pollard Ho

```
//Complexidade: O(n^(1/4)) em média, O(n^(1/2)) no pior caso

ll mul(ll a, ll b, ll m) {
    ll ret = a*b - ll((long double)1/m*a*b+0.5)*m;
    return ret < 0 ? ret+m : ret;
}

ll pow(ll x, ll y, ll m) {
    if (!y) return 1;
    ll ans = pow(mul(x, x, m), y/2, m);
    return y%2 ? mul(x, ans, m) : ans;
}

bool prime(ll n) {
    if (n < 2) return 0;
    if (n <= 3) return 1;
    if (n % 2 == 0) return 0;

    ll r = __builtin_ctzll(n - 1), d = n >> r;
    for (int a : {2, 325, 9375, 28178, 450775, 9780504, 1795265022}) {
        ll x = pow(a, d, n);
        if (x == 1 or x == n - 1 or a % n == 0) continue;

        for (int j = 0; j < r - 1; j++) {
            x = mul(x, x, n);
            if (x == n - 1) break;
        }
        if (x != n - 1) return 0;
    }
    return 1;
}

ll rho(ll n) {
    if (n == 1 or prime(n)) return n;
    auto f = [n](ll x) {return mul(x, x, n) + 1;};

    ll x = 0, y = 0, t = 30, prd = 2, x0 = 1, q;
    while (t % 40 != 0 or gcd(prd, n) == 1) {
        if (x==y) x = ++x0, y = f(x);
        q = mul(prd, abs(x-y), n);
        if (q != 0) prd = q;
        x = f(x), y = f(f(y)), t++;
    }
    return gcd(prd, n);
```

```
}

vector<ll> fact(ll n) {
    if (n == 1) return {};
    if (prime(n)) return {n};
    ll d = rho(n);
    vector<ll> l = fact(d), r = fact(n / d);
    l.insert(l.end(), r.begin(), r.end());
    return l;
}
```

3 Grafos

3.1 2 Sat

```
//(a ou b) e (c ou d) e (~a ou c) ...
//Complexidade: O(n + m), onde n é o número de variáveis e m é o número de
↪  implicações
//
// (+a  ou -b) -> add_edge(1, a, 0, b)
//
//Status: tested - https://cses.fi/problemset/result/8784228/

struct SAT2{

    int n, cont;
    vector<char> resp;
    vector<int> marc, ord, comp;
    vector<vector<int>> grafo, rgrafo,scc;

    SAT2(int n) :  n(n), marc(2*n+2), grafo(2*n+2), rgrafo(2*n+2), comp(2*n+2),
↪  resp(2*n + 2){}

    void add_edge(int sx, int x, int sy, int y){ // '+' = 1, '-' = 0

        grafo[y+n*(!sy)].push_back(x+n*sx); //~y -> x
        grafo[x+n*(!sx)].push_back(y+n*sy); //~x -> y

        rgrafo[x+n*sx].push_back(y+n*(!sy));
        rgrafo[y+n*sy].push_back(x+n*(!sx));

    }

    void dfs1(int v){
        marc[v] = 1;
        for(auto viz : grafo[v]){
            if(!marc[viz]) dfs1(viz);
        }
        ord.push_back(v);
    }

    void dfs2(int v, int c){
        comp[v] = c;
        for(auto viz : rgrafo[v]){
            if(!comp[viz]) dfs2(viz, c);
        }
    }

    void build(){

        cont = 0;

        for(int i = 1; i <=2*n ;i++){
            if(!marc[i]) dfs1(i);
        }

        reverse(ord.begin(), ord.end());

        for(int v : ord){
            if(!comp[v]){
                dfs2(v, ++cont);
            }
        }

        bool can = true;
        for(int i= 1; i <=n; i++){
            if(comp[i] == comp[i+n]) can = false;
            resp[i]=comp[i]<comp[i+n]?'+':'-';
                //positiva é comp[i+n], está escolhendo a variavel que não
                //tem um caminho de implicação que resulta em impossivel
        }

        if(can){
            for(int i = 1; i <=n; i++){
                cout << resp[i] << " ";
            }
        }else{
            cout << "IMPOSSIBLE" << endl;
        }
    }
```

```
};
```

3.2 Bellman-Ford

```
//Podemos encontrar ciclos negativos guardando os pais de cada vértice.

struct Edge{
    int v, u, cost;
    Edge(int v, int u, int cost): v(v), u(u), cost(cost) {}
};

struct Ford
{
    const ll INFL = 1e18;
    int n, m;
    vector<Edge> edges;
    vector<ll> dist;

    Ford(int n, int m) : n(n), m(m), dist(n+1, INFL) {}

    void add_edge(int v, int u, int cost){
        edges.emplace_back(v,u,cost);
    }

    ll bellman(int s, int t){
        dist[s] = 0;

            //Encontrar distancias
        for(int k=1; k < n; k++){
            for(Edge e : edges){
                int a = e.v, b = e.u, c = e.cost;
                if(dist[a] != MINFL && dist[b] > dist[a] + c){
                    dist[b] = dist[a] + c;
                }
            }
        }

            //Se conseguirmos melhorar após n-1, significa que existe ciclo negativo
        for(Edge e : edges){
            int a = e.v, b =e.u, c=e.cost;
            if(dist[a] != MINFL && dist[b] > dist[a]+c){
                return -1;
            }
        }

        return dist[t];

    }

};
```

3.3 BFS

```
queue<int> q;
vector<bool> used(n);

q.push(s);
used[s] = true;
while (!q.empty()) {
    int v = q.front();
    q.pop();
    for (int u : adj[v]) {
        if (!used[u]) {
            used[u] = true;
            q.push(u);
        }
    }
}
```

3.4 Bridge Tree

```
//Complexidade: O(V + E), onde V é o número de vértices e E é o número de arestas.
//A árvore de pontes é um grafo que representa as componentes conexas de um grafo
↪  original,
//onde cada aresta é formada por uma ponte do grafo original.

struct BridgeTree{

    int n;
    int count = 0;
    vector<int> marc, tin, low, is_bridge;
    vector<vector<pair<int,int>>> grafo;
    vector<vector<int>> BT;
    vector<pair<int,int>> edge;
```

```
vector<int> BTcomponent;

BridgeTree(int n) : n(n), grafo(n+1), marc(n+1), tin(n+1), low(n+1),
↪ BTcomponent(n+1){}

void add_edge(int a, int b){
    grafo[a].push_back({b, edge.size()});
    grafo[b].push_back({a, edge.size()});
    edge.push_back({a,b});
    is_bridge.push_back(0);
}

void dfs(int x, int p){
    marc[x] = 1;
    tin[x] = low[x] = ++count;
    int children = 0;
    for(auto [viz, e] : grafo[x]){
        if(viz == p) continue;
        if(marc[viz]){
            low[x] = min(low[x], tin[viz]);
        }else{
            dfs(viz,x);
            low[x] = min(low[x], low[viz]);
            if(low[viz] > tin[x]){
                is_bridge[e] = 1;
            }
            children++;
        }
    }
}

void find_bridges(){
    for(ll i=1; i<=n; i++){
        if(!marc[i]) dfs(i,0);
    }
}

void BTdfs(int v, int comp){
    BTcomponent[v] = comp;
    for(auto [viz, e] : grafo[v]){
        if(BTcomponent[viz] || is_bridge[e]) continue;
        BTdfs(viz, comp);
    }
}

void BrigeTree(){
        int comp = 0;
        for(int i = 1; i <= n; i++){
            if(!BTcomponent[i]) BTdfs(i, ++comp);
        }

        BT.resize(comp+1);

        for(int i = 1; i <= n; i++){
            for(auto [j,e] : grafo[i]){
                if(is_bridge[e]){
                    BT[BTcomponent[i]].push_back(BTcomponent[j]);
                    BT[BTcomponent[j]].push_back(BTcomponent[i]);
                }
            }
        }
    }
};
```

3.5 Dinic

```
// Grafo com capacidades 1: O(min(M*sqrt(M), M*N^(2/3)))
// Todo vértice tem grau de entrada ou saída 1 e a maior capacidade é 1: O(sqrt(N)*M)
template<typename T>
struct Dinic{
    struct Edge {int v, u; T cap, flow;};
    int m=0;
    vector<Edge> edges;
    vector<vector<int> > vec;
    vector<int> lv, pos;
    queue<int> fila;

    Dinic() {}

    Dinic(int n) : vec(n), lv(n), pos(n) {}

    void add_edge(int v, int u, T cap) {
        edges.push_back({v, u, cap, 0});
        edges.push_back({u, v, 0, 0});
        vec[v].push_back(m);
        vec[u].push_back(m+1);
        m+=2;
    }

    int bfs(int t){
```

```
    while(!fila.empty()){
        int v=fila.front();
        fila.pop();
        for(int i:vec[v]){
            if(edges[i].cap-edges[i].flow<1) continue;
            if(lv[edges[i].u]!=-1) continue;

            lv[edges[i].u]=lv[v]+1;
            fila.push(edges[i].u);
        }
    }
    return lv[t]!=-1;
}

T dfs(int v, int t, T menor) {
    if(!menor) return 0;
    if(v==t) return menor;

    for(int& j=pos[v]; j<(int)vec[v].size(); j++){
        int i=vec[v][j];
        int u=edges[i].u;

        if(lv[v]+1!=lv[u] || edges[i].cap-edges[i].flow<1) continue;

        T agr=dfs(u, t, min(menor, edges[i].cap-edges[i].flow));
        if(!agr) continue;

        edges[i].flow+=agr;
        edges[i^1].flow-=agr;

        return agr;
    }
    return 0;
}

T max_flow(int s, int t){
    T flow=0;
    while(1){
        fill(lv.begin(), lv.end(), -1);

        lv[s]=0;
        fila.push(s);

        if(!bfs(t)) break;

        fill(pos.begin(), pos.end(), 0);
            while(T atual=dfs(s, t, INF)) flow+=atual; //remember to change INF
        }
        return flow;
    }
    auto recap(){
        vector<pair<int, int> > resp;
        for(int i=0; i<(int)edges.size(); i+=2){
            if(lv[edges[i].v]>=0 && lv[edges[i].u]==-1) resp.push_back({edges[i].v,
↪   edges[i].u});
        }
        return resp;
    }
};
```

3.6 Dykstra

```
//Algoritmo de Caminho mínimo para grafos compesos não negativos. Um para todos
//Complexidade: O(n log n) onde n é o número de vértices do grafo.

template<typename T> struct Dykstra
{
    ll INF = 1e18;

    int n;
    vector<ll> dist;
    vector<vector<pair<int,int>>> g;

    Dykstra(int n) : n(n), dist(n+1,INF), g(n+1) {}

    void addEdge(ll v, ll u, ll p){
        g[v].push_back({u,p});
        g[u].push_back({v,p});
    }

    void run(ll v){

            //preparing structures
        priority_queue<pair<ll,ll>, vector<pair<ll,ll>>,
        greater<pair<ll,ll>>> fila;

            //setting up
        fila.push({0,v});

        while (!fila.empty())
```

```
        {
            ll vert = fila.top().second;
            ll price = fila.top().first;
            fila.pop();

            if(dist[vert] != INF) continue;

            dist[vert] = price;

            for(auto viz : g[vert]){
                ll nxt = viz.first;
                ll cost = viz.second;
                fila.push({price + cost, nxt});
            }
        }

    }

};
```

3.7 Euler Path

```
 * Author: Simon Lindholm
 * Date: 2019-12-31
 * License: CC0
 * Source: folklore
 * Description: Eulerian undirected/directed path/cycle algorithm.
 * Input should be a vector of (dest, global edge index), where
 * for undirected graphs, forward/backward edges have the same index.
 * Returns a list of nodes in the Eulerian path/cycle with src at both start and
↪  end, or
 * empty list if no cycle/path exists.
 * To get edge indices back, add .second to s and ret.
 * Time: O(V + E)
 * Status: stress-tested
 *
 * Condições para a existencia de um caminho/cicutio euleriano:
 *
 *          |        Direcionado            |         Não Direcionado
 * ---------+-------------------------------+---------------------------
 *          | "existem 0 ou 1 vértices      |
 * Caminho  |  com difrença 1 entre grau    | "existem 0 ou 2 vértices de grau impar"
 *          |  de entrada e saida"          |
 * ---------+------------------------------+-----------------------------------
 *          | "Grau de entrada e saida      |
 * Circuito |  de todos os vértices         | "não existe vértice de grau impar"
 *          |  são iguais"                  |
 *

 */
vector<int> eulerWalk(vector<vector<pii>>& gr, int nedges, int src=0) {
    int n = gr.size();
    vector<int> D(n), its(n), eu(nedges), ret, s = {src};
    D[src]++; // to allow Euler paths, not just cycles
    while (!s.empty()) { ///start-hash
        int x = s.back(), y, e, &it = its[x], end = int(gr[x].size());
        if (it == end){ ret.push_back(x); s.pop_back(); continue; }
        tie(y, e) = gr[x][it++];
        if (!eu[e])
            D[x]--, D[y]++, eu[e] = 1, s.push_back(y);
    } ///end-hash
    for(auto &x : D) if (x < 0 || int(ret.size()) != nedges+1) return {};
    return {ret.rbegin(), ret.rend()};
}
```

3.8 Floyd Warshall

```
//Algoritmo todos para todos de distancia mínima
//Se houver ciclos negativos, para algum vertice a  -> dist[a][a] < 0
//Complexidade: O(n^3)

template<typename T> struct FloydWarshall
{

    const int MAXN = 500;
    cont ll INF = 1e18;
    vector dist(maxn, vector<ll>(maxn, INF));

    void floydWarshall( ){

        for(int i = 0; i < MAXN; i++)dist[i][i] = 0;

        for(int k = 1; k < MAXN; k++)
            for(int i = 1; i < MAXN; i++)
                for(int j = 1; j < MAXN; j++){
                    if(dist[i][k] < INF && dist[k][j] < INF)
                        dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
```

```
            }
        }
};
```

3.9 Hopcroft Karp

```
 * Author: Chen Xing
 * Date: 2009-10-13
 * License: CC0
 * Source: N/A
 * Description: Fast bipartite matching algorithm. Graph $g$ should be a list
 * of neighbors of the left partition, and $btoa$ should be a vector full of
 * -1's of the same size as the right partition. Returns the size of
 * the matching. $btoa[i]$ will be the match for vertex $i$ on the right side,
 * or $-1$ if it's not matched.
 * Usage: vector<int> btoa(m, -1); hopcroftKarp(g, btoa);
 * Status: Tested on oldkattis.adkbipmatch and SPOJ:MATCHING
 * Time: O(\sqrt{V}E)
 */
struct Hop{

    using vi = vector<int>;

    int n, m;
    vector<vi> g;
    vi btoa;

    Hop(int n, int m) : n(n), m(m), g(n+1), btoa(m+1, -1) {}

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

    bool dfs(int a, int L, vi &A, vi &B) { ///start-hash
        if (A[a] != L) return 0;
        A[a] = -1;
        for(auto &b : g[a]) if (B[b] == L + 1) {
            B[b] = 0;
            if (btoa[b] == -1 || dfs(btoa[b], L+1, A, B))
                return btoa[b] = a, 1;
        }
        return 0;
    } ///end-hash

    int solve() { ///start-hash
        int res = 0;
        vector<int> A(g.size()), B(int(btoa.size())), cur, next;
        for (;;) {
            fill(A.begin(), A.end(), 0), fill(B.begin(), B.end(), 0);
            cur.clear();
            for(auto &a : btoa) if (a != -1) A[a] = -1;
            for (int a = 0; a < g.size(); ++a) if (A[a] == 0) cur.push_back(a);
            for (int lay = 1;; ++lay) {
                bool islast = 0; next.clear();
                for(auto &a : cur) for(auto &b : g[a]) {
                    if (btoa[b] == -1) B[b] = lay, islast = 1;
                    else if (btoa[b] != a && !B[b])
                        B[b] = lay, next.push_back(btoa[b]);
                }
                if (islast) break;
                if (next.empty()) return res;
                for(auto &a : next) A[a] = lay;
                cur.swap(next);
            }
            for(int a = 0; a < int(g.size()); ++a)
                res += dfs(a, 0, A, B);
        }
    } ///end-hash
};
```

3.10 Hungarian Matching

```
 * Source: https://github.com/bqi343/USACO/blob/master/Implementations/content/graph
↪  s%20(12)/Matching/Hungarian.h
 * Description: Given a weighted bipartite graph, matches every node on
 * the left with a node on the right such that no
 * nodes are in two matchings and the sum of the edge weights is minimal. Takes
 * cost[N][M], where cost[i][j] = cost for L[i] to be matched with R[j] and
 * returns (min cost, match), where L[i] is matched with
 * R[match[i]]. Negate costs for max cost.
 * Time: O(N^2M)
 * Status: Tested on kattis:cordonbleu, stress-tested
*/
// o valor na posição i do vector retornado indica a coluna do elemento da linha i
↪  que foi escolhido
```

```
template<class cost_t> pair<cost_t, vector<int>> hungarian(const
↪ vector<vector<cost_t>> &a){
    int n = a.size() + 1, m = a[0].size() + 1;

    vector<int> p(m), ans(n - 1);
    vector<cost_t> u(n), v(m);
    for(int i = 1; i < n; ++i) {
        p[0] = i; int j0 = 0;
        vector<cost_t> dist(m, 1e9);
        vector<int> pre(m, -1);
        vector<bool> done(m + 1);
        do {
            done[j0] = true;
            int i0 = p[j0], j1;
            cost_t delta = 1e9;
            for(int j = 1; j < m; ++j) if (!done[j]) {
                auto cur = a[i0-1][j-1] - u[i0] - v[j];
                if (cur < dist[j]) dist[j] = cur, pre[j] = j0;
                if (dist[j] < delta) delta = dist[j], j1 = j;
            }
            for(int j = 0; j < m; ++j)
                if (done[j]) u[p[j]] += delta, v[j] -= delta;
                else dist[j] -= delta;
            j0 = j1;
        } while (p[j0]);
        while (j0) {
            int j1 = pre[j0]; p[j0] = p[j1], j0 = j1;
        }
    }
    for(int j = 1; j < m; ++j) if (p[j]) ans[p[j]-1] = j-1;
    return {-v[0], ans};
}
```

3.11 Kosaraju

```
//Retorna também em scc as componentes em ordem topologica
//Complexidade: O(n+m)

struct Kosa{

    int n, cont;
    vector<int> marc, ord, comp;
    vector<vector<int>> grafo, rgrafo,scc;

    Kosa(int n) : n(n), marc(n+1), grafo(n+1), rgrafo(n+1), comp(n+1), scc(n+1) {}

    void add_edge(int a, int b){
        grafo[a].push_back(b);
        rgrafo[b].push_back(a);
    }

    void dfs1(int v){
        marc[v] = 1;
        for(auto viz : grafo[v]){
            if(!marc[viz]) dfs1(viz);
        }
        ord.push_back(v);
    }

    void dfs2(int v, int c){
        comp[v] = c;
        for(auto viz : rgrafo[v]){
            if(!comp[viz]) dfs2(viz, c);
        }
    }

    void build(){

        cont = 0;

        for(int i = 1; i <=n ;i++){
            if(!marc[i]) dfs1(i);
        }

        reverse(ord.begin(), ord.end());

        for(int v : ord){
            if(!comp[v]){
                dfs2(v, ++cont);
            }
        }

        for(int i = 1; i <=n; i++){
            for(int j : grafo[i]){
                if(comp[i] == comp[j]) continue;
                scc[comp[i]].push_back(comp[j]);
            }
        }

    }
```

```
};
```

3.12 Kuhn

```
struct bm_t
{
    int N, M, T;
    vector<vector<int>> grafo;
    vector<int> match, seen;
    bm_t(int a, int b) : N(a), M(a+b), T(0), grafo(M),match(M, -1), seen(M, -1) {}

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

    bool dfs(int cur){
        if(seen[cur] == T) return false;
        seen[cur] = T;
        for(int nxt : grafo[cur]) if(match[nxt] == -1){
            match[nxt] = cur;
            match[cur] = nxt;
            return true;
        }
        for(int nxt : grafo[cur]) if(dfs(match[nxt])){
            match[nxt] = cur;
            match[cur] = nxt;
            return true;
        }
        return false;
    }

    int solve(){
        int res = 0;
        for(int cur = 1; cur;){
            cur = 0; ++T;
            for(int i = 0; i < N; ++i) if(match[i] == -1)
                cur += dfs(i);
            res += cur;
        }
        return res;
    }
};
```

3.13 Min-cist max-flow

```
//Time: O (F (V + E)logV ), being F the amount of flow.

template<class flow_t, class cost_t> struct min_cost {
    static constexpr flow_t FLOW_EPS = flow_t(1e-10);
    static constexpr flow_t FLOW_INF = numeric_limits<flow_t>::
        max();
    static constexpr cost_t COST_EPS = cost_t(1e-10);
    static constexpr cost_t COST_INF = numeric_limits<cost_t>::
        max();
    int n, m{}; vector<int> ptr, nxt, zu;
    vector<flow_t> capa; vector<cost_t> cost;

    min_cost(int N) : n(N),ptr(n,-1),dist(n),vis(n),pari(n) {}

    void add_edge(int u, int v, flow_t w, cost_t c) {
        nxt.push_back(ptr[u]); zu.push_back(v); capa.push_back(w);
        cost.push_back(c); ptr[u] = m++;
        nxt.push_back(ptr[v]); zu.push_back(u); capa.push_back(0);
        cost.push_back(-c); ptr[v] = m++;
    }

    vector<cost_t> pot, dist; vector<bool> vis; vector<int> pari;
    vector<flow_t> flows; vector<cost_t> slopes;
        // You can pass t = =1 to find a shortest
    void shortest(int s, int t) {//path to each vertex . // hash=1
        using E = pair<cost_t, int>;
        priority_queue<E, vector<E>, greater<E>> que;
        for(int u = 0; u < n; ++u){dist[u]=COST_INF; vis[u]=false;}
        for (que.emplace(dist[s] = 0, s); !que.empty(); ) {
            const cost_t c = que.top().first;
            const int u = que.top().second; que.pop();
            if (vis[u]) continue;
            vis[u] = true; if (u == t) return;
            for (int i = ptr[u]; ~i; i = nxt[i]) if (capa[i] > FLOW_EPS) {
                const int v = zu[i];
                const cost_t cc = c + cost[i] + pot[u] - pot[v];
                if(dist[v] > cc){que.emplace(dist[v]=cc,v);pari[v]=i;}
            }
        }
    }
```

```
        // hash=1 = 89f16a
    auto run(int s, int t, flow_t limFlow = FLOW_INF) { // hash=2
        pot.assign(n, 0); flows = {0}; slopes.clear();
        while (true) {
            bool upd = false;
            for (int i = 0; i < m; ++i) if (capa[i] > FLOW_EPS) {
                const int u = zu[i ^ 1], v = zu[i];
                const cost_t cc = pot[u] + cost[i];
                if(pot[v] > cc + COST_EPS) { pot[v] = cc; upd = true; }
            } if (!upd) break;
        }
        flow_t flow = 0; cost_t tot_cost = 0;
        while (flow < limFlow) {
            shortest(s, t); flow_t f = limFlow - flow;
            if (!vis[t]) break;
            for(int u = 0; u < n; ++u)pot[u] += min(dist[u],dist[t]);
            for (int v = t; v != s; ) { const int i = pari[v];
                if (f > capa[i]) { f = capa[i]; } v = zu[i^1];
                }
            for (int v = t; v != s; ) { const int i = pari[v];
                capa[i] -= f; capa[i^1] += f; v = zu[i^1];
                }
            flow += f; tot_cost += f * (pot[t] - pot[s]);
            flows.push_back(flow); slopes.push_back(pot[t] - pot[s]);
        } return make_pair(flow, tot_cost);
    } // hash=2 = 285527
};
```

3.14 Pontes e Articulação

```
//Complexidade: O(V + E), onde V é o número de vértices e E é o número de arestas.
struct ArticPont{

    int n;
    int count = 0;
    vector<int> marc, tin, low, artic;
    vector<vector<int>> grafo;
    vector<pair<int,int>> bridges;

    ArticPont(int n) : n(n), grafo(n+1), marc(n+1), tin(n+1), low(n+1), artic(n+1) {}

    void add_edge(int a, int b){
        grafo[a].push_back(b);
        grafo[b].push_back(a);
    }

    void dfs(ll x, ll p){
        marc[x] = 1;
        tin[x] = low[x] = ++count;
        ll children = 0;
        for(ll viz : grafo[x]){
            if(viz == p) continue;
            if(marc[viz]){
                low[x] = min(low[x], tin[viz]);
            }else{
                dfs(viz,x);
                low[x] = min(low[x], low[viz]);
                if(low[viz] > tin[x]){
                    bridges.push_back({min(viz,x), max(viz, x)});
                }
                if(low[viz] >= tin[x] && p) artic[x] = 1;
                children++;
            }
        }
        if(!p && children>1) artic[x] = 1;
    }

    void find_brig_and_artc(){
        for(ll i=1; i<=n; i++){
            if(!marc[i]) dfs(i,0);
        }
    }

};
```

3.15 Topo Sort

```
//It returns a vector with the vertices in topological order.
//Complexity: O(n + m), where n is the number of vertices and m is the number of
↪ edges.
struct TopoSort
{
    int n;
    vector<int> grau;
    vector<vector<int>> grafo;

    TopoSort(int n): n(n), grau(n+1), grafo(n+1){}
```

```
    void add_edge(int a, int b){
        grau[b]++;
        grafo[a].push_back(b);
    }

    vector<int> top_sort(){
        vector<int> resp;
        queue<int> fila;
        for(int i=1; i<=n;i++){
            if(!grau[i])fila.push(i);
        }
        while (!fila.empty())
        {
            int u = fila.front();
            resp.push_back(u);
            fila.pop();
            for(int viz : grafo[u]){
                grau[viz]--;
                if(!grau[viz])fila.push(viz);
            }
        }
        if(resp.size() < n){
            return {};
        }else{
            return resp;
        }
    }
};
```

4 DP

4.1 Digit DP

```
ll solve(string &s, int i, int tight, int last, int started){
    if(i==(int)s.size()) return 1;

    if(!tight && dp[i][last][started]!=-1) return dp[i][last][started];

    int lim=(tight?s[i]-'0':9);

    ll resp=0;
    for(int j=0; j<=lim; j++){
        if(started && j==last) continue;
        resp+=solve(s, i+1, tight&(j==lim), j, (started|j)>0);
    }

    if(!tight) return dp[i][last][started]=resp;
    return resp;
}

ll func(ll a, ll b){
    string agr1=to_string(a-1);
    memset(dp, -1, sizeof(dp));
    ll ans1 = solve(agr1, 0, 1, 10, 0);

    string agr2=to_string(b);
    memset(dp, -1, sizeof(dp));
    ll ans2 = solve(agr2, 0, 1, 10, 0);

    return ans2-ans1;
}
```

4.2 Submask DP

```
/*
Iterate for all strict subsets of mask
Complexity: O(3^n)
*/

for (int mask = 0; mask < (1 << n); mask++) {
    for (int submask = mask; submask != 0; submask = (submask - 1) & mask) {
        int subset = mask ^ submask;
            // do whatever you need to do here
    }
}
```

4.3 Subset Sum - Sqrt(n)

```
//Subset sum - Implementation O(n) memory and O(S * sqrt(N)) runtime
//Uses sliding window technique to optimize the subset sum problem.

vector<pair<int,int>> sack; // {item, frequency}
```

```
vector<int> dp(S+1, 0);


for(int i = 0; i < sack.size(); i++){
    vector<int> ndp(n+1);
    auto [item, freq] = sack[i];
    for(int j = 0; j < item; j++){
        int numTrues = 0;
        for(int k = j; k <= n; k += item){
            ndp[k] = dp[k];
            if(numTrues > 0) ndp[k] = true;
            if(k - freq*item >= 0) numTrues -= dp[k - freq*item];
            numTrues += dp[k];
        }
    }
    swap(ndp, dp);
}
```

5 Arvore

5.1 Centroid Decomposition

```
struct Centroid{
    int n;
    vector<int> used, pai, sub;
    vector<vector<int >> vec;

    Centroid(int n) : n(n), used(n+1), pai(n+1), sub(n+1), vec(n+1) {}

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

    int dfs_sz(int x, int p=0){
        sub[x]=1;
        for(int i:vec[x]){
            if(i==p || used[i]) continue;
            sub[x]+=dfs_sz(i, x);
        }
        return sub[x];
    }

    int find_c(int x, int total, int p=0){
        for(int i:vec[x]){
            if(i==p || used[i]) continue;
            if(2*sub[i]>total) return find_c(i, total, x);
        }
        return x;
    }

    void build(int x=1, int p=0){
        int c=find_c(x, dfs_sz(x));

            //do something

        used[c]=1;
        pai[c]=p;
        for(int i:vec[c]){
            if(!used[i]) build(i, c);
        }
    }
};
```

5.2 Lowest Common Ancestor

```
struct LCA{

    int n;
    const int sz = 32;
    vector<int> marc, height;
    vector<vector<int>> g, bl;

        //Trocar se a raiz nao for 1
    LCA(int n) : n(n), g(n+1), bl(sz, vector<int> (n+1, 1)), marc(n+1), height(n+1){}

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

        //Trocar se a raiz nao for 1
    void build(int x = 1){
        marc[x] = 1;
        for(int i = 1; i < sz; i++){
            bl[i][x] = bl[i-1][bl[i-1][x]];
        }
```

```cpp
        for(auto viz : g[x]){
            if(marc[viz]) continue;
            bl[0][viz] = x;
            height[viz] = height[x]+1;
            build(viz);
        }
    }

    int find_lca(int a, int b){
        if(height[a] < height[b]) swap(a,b);

        int dif = height[a] - height[b];
        for(int i = 0; i < sz; i++){
            if((1<<i) & dif){
                a = bl[i][a];
            }
        }

        assert(height[a] == height[b]);
        if(a == b) return a;

        for(int i = sz-1; i >=0; i--){
            if(bl[i][a] == bl[i][b]) continue;
            a = bl[i][a];
            b = bl[i][b];
        }

        assert(a != b);
        assert(bl[0][a] == bl[0][b]);
        return bl[0][a];
    }

    int dist(int a, int b){
        int l = find_lca(a,b);
        return height[a] + height[b] - 2*height[l];
    }

};
```

6 Strings

6.1 Hashing

```cpp
//Cria o hashing de uma string
//ha[0] = 0
//ha[1] = s[0]
//ha[2] = p*s[0] + s[1]
//ha[3] = p^2*s[0] + p*s[1] + s[2]


template<int MOD> struct Hashing{
    ll base, n;
    vector<ll> pow, ha;

        /*
    for random base:
    mt19937 rng((uint32_t)chrono::steady_clock::now().time_since_epoch().count());
    const ll B = uniform_int_distribution<ll>(0, M - 1)(rng);
    */

    Hashing(string & s, int a) : n(s.size()), base(a) ,pow(n+1), ha(n+1){

        pow[0] = 1;
        for(int i = 0; i < n; i++){
            ha[i+1] = (ha[i] * base + s[i])%MOD;
            pow[i+1] = (pow[i] * base)%MOD;
        }
    }

        //Retorna o Hashing da substring [a, b), indexado em 0
    int getRange(int a, int b){
        assert(a <= b);
        ll hash =  (ha[b] - (ha[a] * pow[b-a])%MOD)%MOD;
        return hash < 0 ? hash + MOD : hash;
    }

};
```

6.2 KMP

```cpp
vector<int> find_pi(string s){

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

vector<int> kmp(string t, string p){

    vector<int> pi= find_pi(p + '$'), match;
    for(int i = 0, j = 0; i < t.size(); i++){
        while(j > 0 && t[i] != p[j]) j = pi[j-1];
        if(t[i] == p[j]) j++;
        if(j == p.size()) match.push_back(i-j+1);
    }
    return match;
};

struct autKMP {
    vector<vector<int>> nxt;

    autKMP(string& s) : nxt(26, vector<int>(s.size()+1)) {
        vector<int> p = pi(s);
        nxt[s[0]-'a'][0] = 1;
        for (char c = 0; c < 26; c++)
            for (int i = 1; i <= s.size(); i++)
                nxt[c][i] = c == s[i]-'a' ? i+1 : nxt[c][p[i-1]];
    }
};
```

6.3 Manacher

```cpp
//Complexidade: O(n), onde n é o tamanho da string

vector<int> manacher_odd(string s) {
    int n = s.size();
    s = "$" + s + "^";
    vector<int> p(n + 2);
    int l = 0, r = 1;
    for(int i = 1; i <= n; i++) {
        p[i] = 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);
}

pair<vector<int>, vector<int>> manacher(string s) {
    string t;
    for(auto c: s) {
        t += string("#") + c;
    }
    vector<int> res = manacher_odd(t + "#");
    vector<int> dodd(s.size()), deven(s.size());
    for(int i = 0; i < s.size(); i++){
        dodd[i] = res[2*i + 1]/2;
        deven[i] = (res[2*i]-1)/2;
    }

    return {dodd, deven};
}
```

6.4 Trie

```cpp
struct Vertex {
    int next[K];
    ll output = 0;

    Vertex() {
        fill(begin(next), end(next), -1);
    }
};

struct Trie{

    int n;
```

```
    const int K = 26;
    vector<Vertex> t;

    Trie() : t(1){}

    void add_string(string s){
        int p = 0;
        for(int i = 0; i < s.size(); i++){
            if(t[p].next[s[i] - 'a'] == -1){
                t[p].next[s[i] - 'a'] = t.size();
                t.push_back(Vertex());
            }
            p = t[p].next[s[i] - 'a'];
        }
        t[p].output++;
    }
};
```

6.5 Z

```
//e é igual ao prefixo da string original.
//Complexidade: O(n), onde n é o tamanho da string

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

7 DataStructures

7.1 BIT

```
//dada uma função f associativa em um sobre um
//conjunto com elemento neutro e inversos
//Querry - O(log(n))++suporta apenas querry de update singular
//Update - O(log(n))

struct FenwickTree {
    vector<int> bit;
    int n;

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

    FenwickTree(vector<int> const &a) : FenwickTree(a.size()){
        for (int i = 0; i < n; i++) {
            bit[i] += a[i];
            int r = i | (i + 1);
            if (r < n) bit[r] += bit[i];
        }
    }


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

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

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

7.2 BIT - Range Update

```
vector<int> bit1, bit2;
void init(int n){
    bit1.assign(n+1, 0);
    bit2.assign(n+1, 0);
}

int rsq(vector<int> &bit, int i){
    int ans = 0;
    for(; i; i-=i&-i)
        ans += bit[i];
    return ans;
}

void update(vector<int> &bit, int i, int v){
    for(; i < bit.size(); i+=i&-i)
        bit[i] += v;
}

void update(int i, int j, int v){
    update(bit1, i, v);
    update(bit1, j+1, -v);
    update(bit2, i, v*(i-1));
    update(bit2, j+1, -v*j);
}

int rsq(int i){
    return rsq(bit1, i)*i - rsq(bit2, i);
}

int rsq(int i, int j){
    return rsq(j) - rsq(i-1);
}
```

7.3 BIT 2D

```
#define pii pair<ll,ll>
#define upper(v, x) (upper_bound(begin(v), end(v), x) - begin(v))

struct BIT2D{
    vector<ll> ord;
    vector<vector<ll>> bit,coord;
    BIT2D(vector<pii> pts){
        sort(begin(pts),end(pts));

        for(auto [x,y] : pts)
            if(ord.empty() || x != ord.back())
                ord.push_back(x);

        bit.resize(ord.size() + 1);
        coord.resize(ord.size() + 1);

        sort(begin(pts),end(pts), [&](pii &a , pii& b){
            return a.second < b.second;
        });

        for(auto [x,y] : pts)
            for(int i = upper(ord,x); i < bit.size(); i += i & -i)
                if(coord[i].empty() || coord[i].back() != y)
                    coord[i].push_back(y);

        for(int i = 0; i < bit.size(); i++) bit[i].assign(coord[i].size() + 1,0);
    }

    void update(ll X, ll Y, ll v){
        for(int i = upper(ord, X); i < bit.size(); i += i & -i)
            for(int j = upper(coord[i], Y); j < bit[i].size(); j += j & -j)
                bit[i][j] += v;
    }

    ll query(ll X, ll Y){
        ll sum = 0;
        for(int i = upper(ord,X); i > 0; i -= i & -i)
            for(int j = upper(coord[i], Y); j > 0; j -= j & -j)
                sum += bit[i][j];
        return sum;
    }

    ll queryArea(ll xi , ll yi, ll xf, ll yf){
        return query(xf,yf) - query(xf, yi-1) - query(xi-1, yf) + query(xi-1, yi-1);
    }
};
```

7.4 Line Container

```
 * Author: Simon Lindholm
 * Date: 2017-04-20
 * License: CC0
 * Source: own work
 * Description: Container where you can add lines of the form kx+m, and query
↪  maximum values at points x.
 *  Useful for dynamic programming (``convex hull trick'').
 * Time: O(\log N)
 * Status: stress-tested
 */

struct Line {
    mutable ll k, m, p;
    bool operator<(const Line& o) const { return k < o.k; }
    bool operator<(ll x) const { return p < x; }
};

struct LineContainer : multiset<Line, less<>> {

    static const ll inf = LLONG_MAX; //for doubles 1/.0

    ll div(ll a, ll b) { //for doubles return a/b
        return a / b - ((a ^ b) < 0 && a % b); }

    bool isect(iterator x, iterator y) {
        if (y == end()) { x->p = inf; return false; }
        if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
        else x->p = div(y->m - x->m, x->k - y->k);
        return x->p >= y->p;
    }

        //para achar o mínimo, é preciso fazer insert({-k, -m, 0}), além disso
↪   multiplicar por -1 o resultado da query
    void add(ll k, ll m) {
        auto z = insert({k, m, 0}), y = z++, x = y;
        while (isect(y, z)) z = erase(z);
        if (x != begin() && isect(--x, y)) isect(x, y = erase(y));
        while ((y = x) != begin() && (--x)->p >= y->p)
            isect(x, erase(y));
    }

    ll query(ll x) {
        assert(!empty());
        auto l = *lower_bound(x);
        return l.k * x + l.m;
    }
};
```

7.5 Merge Sort Tree

```
//Segtree node for Merge-Sort
struct Node{
    vector<int> vec;
    Node operator+(Node other) const{
        vector<int> novo(vec.size() + other.vec.size());
        merge(vec.begin(), vec.end(), other.vec.begin(), other.vec.end(),
↪   novo.begin());
        return {novo};
    }
    Node operator=(int x){
    return {this->vec = {x}};
    }
};
```

7.6 Mo

```
const int blockSize = 500;

struct Query
{
    int l, r, idx;

    bool operator<(Query other) const{
        return make_pair(l/blockSize, r) < make_pair(other.l/blockSize, other.r);
    };
};

struct Mo{
        //TODO: declare the data structures
    Mo(){

    }

    void add(int idx){
            //TODO: add an element to the data structure
```

```
    }

    void remove(int idx){
            //TODO: remove an element from the data structure
    }

    int get_answer(){
            //TODO: get answer from the data structure
    }

    vector<int> solve(vector<Query> queries) {
        vector <int> answers(queries.size());
        sort(queries.begin(), queries.end());

            //TODO: initialize data structue

        int cur_l = 0;
        int cur_r = -1;
        for (Query q : queries) {
            while (cur_l > q.l) {
                cur_l--;
                add(cur_l);
            }
            while (cur_r < q.r) {
                cur_r++;
                add(cur_r);
            }
            while (cur_l < q.l) {
                remove(cur_l);
                cur_l++;
            }
            while (cur_r > q.r) {
                remove(cur_r);
                cur_r--;
            }
            answers[q.idx] = get_answer();
        }
        return answers;
    }
};
```

7.7 Prefix Sum 2D

```
struct pref2D{
    int n, m;
    vector <vector<int>> mat, pref;

    pref2D(int n, int m, vector<vector<int>> tmp){
        this-> n = n; this->m = m;
        mat = tmp;

        pref.resize(n+1);
        for(auto& v : pref) v.resize(m+1, 0);

        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= m; j++){
                pref[i][j] = pref[i-1][j] + pref[i][j-1] - pref[i-1][j-1] +
↪   mat[i-1][j-1];
            }
        }
    }

    int query(int rowl, int rowr, int coll, int colr){
            //rowl++, rowr++, coll++, colr++;
        if(rowl > rowr) swap(rowl, rowr);
        if(coll > colr) swap(colr, coll);
        return pref[rowr][colr] - pref[rowl-1][colr] - pref[rowr][coll-1] +
↪   pref[rowl-1][coll-1];
    }
};
```

7.8 SegTree

```
struct SegTree{
  int n;
  struct Node{
    int val;
    Node operator+(Node other) const{
      return {this->val + other.val};
    }
    Node operator=(int x){
      return {this->val = x};
    }
  };
```

```
Node neutral = {0};
vector <Node> t;

SegTree(vector <int> a){
  n = a.size();
  t.resize(4*n);
  build(a, 1, 0, n-1);
}

void build(vector <int>& a, int v, int tl, int tr) {
  if (tl == tr) {
    t[v] = a[tl];
  } else {
    int tm = (tl + tr) / 2;
    build(a, v*2, tl, tm);
    build(a, v*2+1, tm+1, tr);
    t[v] = t[v*2] + t[v*2+1];
  }
}

Node query(int l, int r){
  return query(1, 0, n-1, l, r);
}

Node query(int v, int tl, int tr, int l, int r){
  if (l > r)
    return neutral;
  if (l == tl && r == tr) {
    return t[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);
}

void update(int pos, int val){
  update(1, 0, n-1, pos, val);
}

void update(int v, int tl, int tr, int pos, int new_val){
  if (tl == tr) {
    t[v] = new_val;
  } else {
    int tm = (tl + tr) / 2;
    if (pos <= tm)
      update(v*2, tl, tm, pos, new_val);
    else
      update(v*2+1, tm+1, tr, pos, new_val);
    t[v] = t[v*2] + t[v*2+1];
  }
}
};
```

7.9 SegTree c/ Lazy

```
struct SegTree{
  int n;
  struct Node{
    int val;
    Node operator+(Node other) const{
      return {this->val + other.val};
    }
    Node operator=(int x){
      return {this->val = x};
    }
  };
  Node neutral = {0};
  vector <Node> t;
  vector<int> lazy;

  SegTree(vector <int> a){
    n = a.size();
    t.resize(4*n);
    lazy.resize(4*n);
    build(a, 1, 0, n-1);
  }

  void build(vector <int>& a, int v, int tl, int tr) {
    if (tl == tr) {
      t[v] = a[tl];
    } else {
      int tm = (tl + tr) / 2;
      build(a, v*2, tl, tm);
      build(a, v*2+1, tm+1, tr);
      t[v] = t[v*2] + t[v*2+1];
    }
  }

  void unlazy(int v, int tl, int tr){
    if(lazy[v] == 0) return;
```

```
        //Update current range
    t[v].val += (tr-tl+1) * lazy[v];

        //Pass lazy to child if any
    if(tl != tr){
        lazy[2*v] += lazy[v];
        lazy[2*v+1] += lazy[v];
    }

        //Reset lazy
    lazy[v] = 0;
  }

  Node query(int l, int r){
    return query(1, 0, n-1, l, r);
  }

  Node query(int v, int tl, int tr, int l, int r){
    unlazy(v, tl, tr);
    if (l > r)
      return neutral;
    if (l == tl && r == tr) {
      return t[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);
  }

  void RangeUpdate(int l, int r, int new_val){
    RangeUpdate(1,0,n-1,l,r,new_val);
  }

  void RangeUpdate(int v, int tl, int tr, int l, int r, int new_val){
    unlazy(v, tl, tr);
    if (l > r)
      return;
    if (l == tl && r == tr) {
        lazy[v] += new_val; //Change here
        unlazy(v, tl, tr);
        return;
    }
    int tm = (tl + tr) / 2;
    RangeUpdate(v*2, tl, tm, l, min(r, tm), new_val);
    RangeUpdate(v*2+1, tm+1, tr, max(l, tm+1), r, new_val);
    t[v] = t[2*v] + t[2*v+1];
  }

  void PointUpdate(int pos, int val){
    PointUpdate(1, 0, n-1, pos, val);
  }

  void PointUpdate(int v, int tl, int tr, int pos, int new_val){
    unlazy(v, tl, tr);
    if (tl == tr) {
      t[v] = new_val;
    } else {
      int tm = (tl + tr) / 2;
      if (pos <= tm)
        PointUpdate(v*2, tl, tm, pos, new_val);
      else
        PointUpdate(v*2+1, tm+1, tr, pos, new_val);
      t[v] = t[v*2] + t[v*2+1];
    }
  }
};
```

7.10 SegTree Sparse

```
struct Node {
    int left, right;
    int sum = 0;
    Node *left_child = nullptr, *right_child = nullptr;

    Node(int lb, int rb) {
        left = lb;
        right = rb;
    }

    void extend() {
        if (!left_child && left + 1 < right) {
            int t = (left + right) / 2;
            left_child = new Node(left, t);
            right_child = new Node(t, right);
        }
    }

    void add(int k, int x) {
        extend();
        sum += x;
        if (left_child) {
```

```
            if (k < left_child->right)
                left_child->add(k, x);
            else
                right_child->add(k, x);
        }
    }

    int get_sum(int lq, int rq) {
        if (lq <= left && right <= rq)
            return sum;
        if (max(left, lq) >= min(right, rq))
            return 0;
        extend();
        return left_child->get_sum(lq, rq) + right_child->get_sum(lq, rq);
    }
};
```

7.11 Sparse Table

```
struct SparseTable{
    int K = 25, n;
    vector <vector<int>> st; //st[i][j] = min on range [j, j + 2^i-1]
    vector <int> lg2; //lg2[i] = floor(log2(i))

    SparseTable(vector <int> arr){
        n = arr.size();
        st.resize(K+1);
        for(auto& v : st) v.resize(n);

        st[0] = arr;
        for(int i = 1; i <= K; i++){
            for(int j = 0; j + (1 << i) - 1 < n; j++){
                st[i][j] = min(st[i-1][j], st[i-1][j + (1 << (i - 1))]);
            }
        }

        lg2.resize(n+1);
        lg2[1] = 0;
        for(int i = 2; i <= n; i++){
            lg2[i] = lg2[i/2] + 1;
        }
    }

    int query(int l, int r){
        int i = lg2[r-l+1];
        return min(st[i][l], st[i][r-(1<<i)+1]);
    }

    int querylog(int l , int r){

        int ans = st[0][l];
        int dif = r-l+1;

        for(int i = 0; i < K; i++){
            if((1<<i) & dif){
                ans = min(ans,st[i][l]);
                l = l + (1<<i);
            }
        }

        return ans;
    }
};
```

7.12 Union Find

```
//Complexidade: O($\alpha$(n)), onde $\alpha$ é a função de Ackermann inversa
struct DSU
{
    int n;
    vector<int> pai, rank;

    DSU(int n) : n(n), pai(n+1), rank(n+1,1){
        for(int i = 1; i <=n; i++){
            pai[i] = i;
        }
    }

    int find(int a){
        if(pai[a] == a) return a;
        return pai[a] = find(pai[a]);
    }

    void uu(int a, int b){
        a = find(a), b = find(b);
        if(a == b) return;
        if(rank[a] > rank[b]) swap(a,b);
        rank[b] += rank[a];
```

```
        pai[a] = b;
    }

};
```

8 Extra

8.1 XorBasis.h

```
//Xor Basis

struct Basis{
  vector <int> basis;
  Basis(){

  }
  Basis(int x){
    add(x);
  }
  Basis operator+(Basis other) const{
    Basis res;
    for(int x : basis){
      res.add(x);
    }
    for(int x : other.basis){
      res.add(x);
    }
    return res;
  }
  void add(int x){
    for(auto& i : basis){
      x = min(x, x^i);
    }
    if(x){
      basis.push_back(x);
    }
  }
};
```

8.2 TernarySearch.h

```
//Ternary Search

double ternary(double l, double r){
    // < for maximum and > for minimum value
    int cont = 300;
    while (cont --)
```

```
    {
        double m1 = l + (r-l)/3;
        double m2 = r - (r-l)/3;
        double f1 = f(m1);
        double f2 = f(m2);
        if(f1>f2){
            l = m1;
        }else{
            r = m2;
        }
    }
    return l;
}

/**
 * Author: Simon Lindholm
 * Date: 2015-05-12
 * License: CC0
 * Source: own work
 * Description:
 * Find the smallest i in $[a,b]$ that maximizes $f(i)$, assuming that $f(a) < \dots < f(i) \ge \dots \ge f(b)$.
 * To reverse which of the sides allows non-strict inequalities, change the < marked with (A) to <=, and reverse
 * the loop at (B).
 * To minimize $f$, change it to >, also at (B).
 * If you are dealing with real numbers, you'll need to pick $m_1 = (2a + b)/3.0$ and $m_2 = (a + 2b)/3.0$.
 * Consider setting a constant number of iterations for the search, usually $[200,300]$ iterations are sufficient
 * for problems with error limit as $10^{-6}$.
 * Status: tested
 * Usage: int ind = ternSearch(0,n-1,[\&](int i){return a[i];});
 * Time: O(\log(b-a))
 */

int ternSearch(int a, int b) {
    assert(a <= b);
    while (b - a >= 5) {
        int mid = (a + b) / 2;
        if (f(mid) < f(mid+1)) a = mid; // (A)
        else b = mid+1;
    }
    for(int i = a+1; i <= b; ++i)
        if (f(a) < f(i)) a = i; // (B)
    return a;
}
```

8.3 Brute.h

```
//Brute
set -e
g++ code.cpp -o code
g++ brute.cpp -o brute
g++ gen.cpp -o gen
for((i = 1; ; ++i)) do
    echo "Test: " $i
    ./gen $i > input_file
    cat input_file
    ./code < input_file > myAnswer
    ./brute < input_file > correctAnswer
    diff -Z myAnswer correctAnswer > /dev/null || break
    cat input_file
    cat myAnswer
    cat correctAnswer
    echo "Passed test" $i
done
echo "WA:"
cat input_file
echo "My:"
cat myAnswer
echo "correct:"
cat correctAnswer
```