Team notebook

March 16, 2023

Contents

1 dp

1.1 divide-and-conquer

```
#include "../common.h"

const ll INF = 1e18;

// for every index i assign an optimal index j, such that cost(i, j) is
// minimal for every i. the property that if i2 >= i1 then j2 >= j1 is
// exploited (monotonic condition).
//
// calculate optimal index for all indices in range [l, r) knowing that
// the optimal index for every index in this range is within [optl, optr).
//
// time: O(N log N)
void calc(vector<int> &opt, int l, int r, int optl, int optr) {
    if (l == r) return;
    int i = (l + r) / 2;
    ll optc = INF;
    int optj;
    repx(j, optl, optr) {
        ll c = i + j; // cost(i, j)
        if (c < optc) optc = c, optj = j;
    }
    opt[i] = optj;
    calc(opt, l, i, optl, optj + 1);
    calc(opt, i + 1, r, optj, optr);
}
```

2 geo2d

2.1 circle

```
#include "line.cpp"
#include "point.cpp"

struct C {
    P o;
    T r;

    C(P o, T r) : o(o), r(r) {}
    C() : C(P(), T()) {}

    // intersects the circle with a line, assuming they intersect
    // the intersections are sorted with respect to the direction of the
    // line
    pair<P, P> line_inter(L l) const {
        P c = l.closest_to(o);
        T c2 = (c - o).magsq();
        P e = sqrt(max(r * r - c2, T())) * l.d.unit();
        return {c - e, c + e};
    }

    // checks whether the given line collides with the circle
    // negative: 2 intersections
    // zero: 1 intersection
    // positive: 0 intersections
    // UNTESTED but very simple
    T line_collide(L l) const {
        T c2 = (l.closest_to(o) - o).magsq();
        return c2 - r * r;
    }

    // calculates the two intersections between two circles
    // the circles must intersect in one or two points!
    // REALLY UNTESTED
    pair<P, P> inter(C h) const {
        P d = h.o - o;
        T c = (r * r - h.r * h.r) / d.magsq();
        return h.line_inter({(1 + c) / 2 * d, d.rot()});
    }

    // check if the given circles intersect
```

```
bool collide(C h) const {
    return (h.o - o).magsq() <= (h.r + r) * (h.r + r);
}

// get one of the two tangents that cross through the point
// the point must not be inside the circle
// a = -1: cw (relative to the circle) tangent
// a = 1: ccw (relative to the circle) tangent
P point_tangent(P p, T a) const {
    T c = r * r / p.magsq();
    return o + c * (p - o) - a * sqrt(c * (1 - c)) * (p - o).rot();
}

// get one of the 4 tangents between the two circles
// a = 1: exterior tangents
// a = -1: interior tangents (requires no area overlap)
// b = 1: ccw tangent
// b = -1: cw tangent
// the line origin is on this circumference, and the direction
// is a unit vector towards the other circle
L tangent(C c, T a, T b) const {
    T dr = a * r - c.r;
    P d = c.o - o;
    P n = (d * dr + b * d.rot() * sqrt(d.magsq() - dr * dr)).unit();
    return {o + n * r, -b * n.rot()};
}

// find the circumcircle of the given **non-degenerate** triangle
static C thru_points(P a, P b, P c) {
    L l((a + b) / 2, (b - a).rot());
    P p = l.intersection(L((a + c) / 2, (c - a).rot()));
    return {p, (p - a).mag()};
}

// find the two circles that go through the given point, are tangent
// to the given line and have radius ‘r‘
// the point-line distance must be at most ‘r‘!
// the circles are sorted in the direction of the line
static pair<C, C> thru_point_line_r(P a, L t, T r) {
    P d = t.d.rot().unit();
    if (d * (a - t.o) < 0) d = -d;
    auto p = C(a, r).line_inter({t.o + d * r, t.d});
    return {{p.first, r}, {p.second, r}};
}

    // find the two circles that go through the given points and have
    // radius ‘r‘
    // the circles are sorted by angle with respect to the first point
    // the points must be at most at distance ‘r‘!
    static pair<C, C> thru_points_r(P a, P b, T r) {
        auto p = C(a, r).line_inter({(a + b) / 2, (b - a).rot()});
        return {{p.first, r}, {p.second, r}};
    }
};
```

2.2 convex-hull

```
#include "point.cpp"

// get the convex hull with the least amount of vertices for the given
    set of points.
// probably misbehaves if points are not all distinct!
vector<P> convex_hull(vector<P> &ps) {
    int N = ps.size(), n = 0, k = 0;
    if (N <= 2) return ps;
    rep(i, N) if (make_pair(ps[i].y, ps[i].x) < make_pair(ps[k].y,
        ps[k].x)) k = i;
    swap(ps[k], ps[0]);
    sort(++ps.begin(), ps.end(), [&](P l, P r) {
        T x = (r - l) / (ps[0] - l), d = (r - l) * (ps[0] - l);
        return x > 0 || x == 0 && d < 0;
    });

    vector<P> H;
    for (P p : ps) {
        while (n >= 2 && (H[n - 1] - p) / (H[n - 2] - p) >= 0)
            H.pop_back(), n--;
        H.push_back(p), n++;
    }
    return H;
}
```

2.3 delaunay

```
#include "point.cpp"
```

```
const T INF = 1e18;

typedef ll lll; // if all coordinates are < 2e4
// typedef __int128_t lll; // if on a 64-bit platform

struct Q {
    Q *rot, *o;
    P p = {INF, INF};
    bool mark;
    P &F() { return r()->p; }
    Q *&r() { return rot->rot; }
    Q *prev() { return rot->o->rot; }
    Q *next() { return r()->prev(); }
};

T cross(P a, P b, P c) {
    return (b - a) / (c - a);
}

bool circ(P p, P a, P b, P c) { // is p in the circumcircle?
    lll p2 = p.magsq(), A = a.magsq() - p2,
        B = b.magsq() - p2, C = c.magsq() - p2;
    return cross(p, a, b) * C + cross(p, b, c) * A + cross(p, c, a) * B >
        0;
}

Q *makeEdge(Q *&H, P orig, P dest) {
    Q *r = H ? H : new Q{new Q{new Q{new Q{0}}}};
    H = r->o;
    r->r()->r() = r;
    repx(i, 0, 4) r = r->rot, r->p = {INF, INF}, r->o = i & 1 ? r :
        r->r();
    r->p = orig;
    r->F() = dest;
    return r;
}

void splice(Q *a, Q *b) {
    swap(a->o->rot->o, b->o->rot->o);
    swap(a->o, b->o);
}

Q *connect(Q *&H, Q *a, Q *b) {
    Q *q = makeEdge(H, a->F(), b->p);
    splice(q, a->next());
    splice(q->r(), b);
    return q;
}

pair<Q *, Q *> rec(Q *&H, const vector<P> &s) {
    if (s.size() <= 3) {
        Q *a = makeEdge(H, s[0], s[1]), *b = makeEdge(H, s[1], s.back());
        if (s.size() == 2) return {a, a->r()};
        splice(a->r(), b);
        auto side = cross(s[0], s[1], s[2]);
        Q *c = side ? connect(H, b, a) : 0;
        return {side < 0 ? c->r() : a, side < 0 ? c : b->r()};
    }

#define J(e) e->F(), e->p
#define valid(e) (cross(e->F(), J(base)) > 0)
    Q *A, *B, *ra, *rb;
    int half = s.size() / 2;
    tie(ra, A) = rec(H, {s.begin(), s.end() - half});
    tie(B, rb) = rec(H, {s.begin() + s.size() - half, s.end()});
    while ((cross(B->p, J(A)) < 0 && (A = A->next())) ||
           (cross(A->p, J(B)) > 0 && (B = B->r()->o)))
        ;
    Q *base = connect(H, B->r(), A);
    if (A->p == ra->p) ra = base->r();
    if (B->p == rb->p) rb = base;

#define DEL(e, init, dir)                         \
    Q *e = init->dir;                             \
    if (valid(e))                                 \
        while (circ(e->dir->F(), J(base), e->F())) { \
            Q *t = e->dir;                        \
            splice(e, e->prev());                 \
            splice(e->r(), e->r()->prev());       \
            e->o = H;                             \
            H = e;                                \
            e = t;                                \
        }
    for (;;) {
        DEL(LC, base->r(), o);
        DEL(RC, base, prev());
        if (!valid(LC) && !valid(RC)) break;
        if (!valid(LC) || (valid(RC) && circ(J(RC), J(LC))))
            base = connect(H, RC, base->r());
        else
```

```
        base = connect(H, base->r(), LC->r());
    }
    return {ra, rb};
#undef J
#undef valid
#undef DEL
}

// there must be no duplicate points
// returns no triangles in the case of all collinear points
// produces counter-clockwise triangles ordered in triples
// maximizes the minimum angle across all triangulations
// the euclidean mst is a subset of these edges
// O(N log N)
// UNTESTED
vector<P> triangulate(vector<P> pts) {
    sort(pts.begin(), pts.end(), [](P a, P b) {
        return make_pair(a.x, a.y) < make_pair(b.x, b.y);
    });
    assert(unique(pts.begin(), pts.end()) == pts.end());
    if (pts.size() < 2) return {};
    Q *H = 0;
    Q *e = rec(H, pts).first;
    vector<Q *> q = {e};
    int qi = 0;
    while (cross(e->o->F(), e->F(), e->p) < 0) e = e->o;
#define ADD                         \
    {                               \
        Q *c = e;                   \
        do {                        \
            c->mark = 1;            \
            pts.push_back(c->p);    \
            q.push_back(c->r());    \
            c = c->next();          \
        } while (c != e);           \
    }
    ADD;
    pts.clear();
    while (qi < (int)q.size())
        if (!(e = q[qi++])->mark) ADD;
    return pts;
#undef ADD
}
```

2.4 halfplane-intersect

```
#include "line.cpp"
#include "point.cpp"

const T INF = 1e9;

// obtain the convex polygon that results from intersecting the given list
// of halfplanes, represented as lines that allow their left side
// assumes the halfplane intersection is bounded
vector<P> halfplane_intersect(vector<L> &H) {
    L bb(P(-INF, -INF), P(INF, 0));
    rep(k, 4) H.push_back(bb), bb.o = bb.o.rot(), bb.d = bb.d.rot();

    sort(H.begin(), H.end(), [](L a, L b) { return a.d.angcmp(b.d) < 0;
     });
    deque<L> q;
    int n = 0;
    rep(i, H.size()) {
        while (n >= 2 && H[i].side(q[n - 1].intersection(q[n - 2])) > 0)
            q.pop_back(), n--;
        while (n >= 2 && H[i].side(q[0].intersection(q[1])) > 0)
            q.pop_front(), n--;
        if (n > 0 && H[i].parallel(q[n - 1])) {
            if (H[i].d * q[n - 1].d < 0) return {};
            if (H[i].side(q[n - 1].o) > 0) q.pop_back(), n--;
            else continue;
        }
        q.push_back(H[i]), n++;
    }

    while (n >= 3 && q[0].side(q[n - 1].intersection(q[n - 2])) > 0)
        q.pop_back(), n--;
    while (n >= 3 && q[n - 1].side(q[0].intersection(q[1])) > 0)
        q.pop_front(), n--;
    if (n < 3) return {};

    vector<P> ps(n);
    rep(i, n) ps[i] = q[i].intersection(q[(i + 1) % n]);
    return ps;
}
```

2.5 line

```
#include "point.cpp"

// a segment or an infinite line
// does not handle point segments correctly!
struct L {
    P o, d;

    L() : o(), d() {}
    L(P o, P d) : o(o), d(d) {}

    // UNTESTED
    L(P ab, T c) : d(ab.rot()), o(ab * -c / ab.magsq()) {}
    pair<P, T> line_eq() { return {-d.rot(), d.rot() * o}; }

    // returns a number indicating which side of the line the point is in
    // negative: left
    // positive: right
    T side(P r) const { return (r - o) / d; }

    // returns the intersection coefficient
    // in the range [0, d / r.d]
    // if d / r.d is zero, the lines are parallel
    T inter(L r) const { return (r.o - o) / r.d; }

    // get the single intersection point
    // lines must not be parallel
    P intersection(L r) const { return o + d * inter(r) / (d / r.d); }

    // check if lines are parallel
    bool parallel(L r) const { return abs(d / r.d) <= EPS; }

    // check if segments intersect
    bool seg_collide(L r) const {
        T z = d / r.d;
        if (abs(z) <= EPS) {
            if (abs(side(r.o)) > EPS) return false;
            T s = (r.o - o) * d, e = s + r.d * d;
            if (s > e) swap(s, e);
            return s <= d * d + EPS && e >= -EPS;
        }
        T s = inter(r), t = -r.inter(*this);
        if (z < 0) s = -s, t = -t, z = -z;
        return s >= -EPS && s <= z + EPS && t >= -EPS && t <= z + EPS;
    }

    // full segment intersection
    // produces a point segment if the intersection is a point
    // however it **does not** handle point segments as input!
    bool seg_inter(L r, L *out) const {
        T z = d / r.d;
        if (abs(z) <= EPS) {
            if (abs(side(r.o)) > EPS) return false;
            if (r.d * d < 0) r = {r.o + r.d, -r.d};
            P s = o * d < r.o * d ? r.o : o;
            P e = (o + d) * d < (r.o + r.d) * d ? o + d : r.o + r.d;
            if (s * d > e * d) return false;
            return *out = L(s, e - s), true;
        }
        T s = inter(r), t = -r.inter(*this);
        if (z < 0) s = -s, t = -t, z = -z;
        if (s >= -EPS && s <= z + EPS && t >= -EPS && t <= z + EPS)
            return *out = L(o + d * s / z, P()), true;
        return false;
    }

    // check if the given point is on the segment
    bool point_on_seg(P r) const {
        if (abs(side(r)) > EPS) return false;
        if ((r - o) * d < -EPS) return false;
        if ((r - o - d) * d > EPS) return false;
        return true;
    }

    // get the point in this line that is closest to a given point
    P closest_to(P r) const { return r + (o - r) * d.rot() * d.rot() /
        d.magsq(); }
};
```

2.6 point

```
#include "../common.h"

typedef ll T;
const T EPS = 0;

struct P {
```

```cpp
    T x, y;

    P(T x, T y) : x(x), y(y) {}
    P() : P(0, 0) {}

    friend ostream &operator<<(ostream &s, const P &r) {
        return s << r.x << " " << r.y;
    }
    friend istream &operator>>(istream &s, P &r) { return s >> r.x >>
        r.y; }

    P operator+(P r) const { return {x + r.x, y + r.y}; }
    P operator-(P r) const { return {x - r.x, y - r.y}; }
    P operator*(T r) const { return {x * r, y * r}; }
    P operator/(T r) const { return {x / r, y / r}; }
    P operator-() const { return {-x, -y}; }
    friend P operator*(T l, P r) { return {l * r.x, l * r.y}; }

    P rot() const { return {-y, x}; }
    T operator*(P r) const { return x * r.x + y * r.y; }
    T operator/(P r) const { return rot() * r; }

    T magsq() const { return x * x + y * y; }
    T mag() const { return sqrt(magsq()); }
    P unit() const { return *this / mag(); }

    bool half() const { return abs(y) <= EPS && x < -EPS || y < -EPS; }
    T angcmp(P r) const {
        int h = (int)half() - r.half();
        return h ? h : r / *this;
    }

    bool operator==(P r) const { return abs(x - r.x) <= EPS && abs(y -
        r.y) <= EPS; }

    double angle() const { return atan2(y, x); }
    static P from_angle(double a) { return {cos(a), sin(a)}; }
};
```

2.7 polygon

```cpp
#include "point.cpp"
```

```cpp
// get the area of a simple polygon in ccw order
// returns negative area for cw polygons
T area(const vector<P> &ps) {
    int N = ps.size();
    T a = 0;
    rep(i, N) a += (ps[i] - ps[0]) / (ps[(i + 1) % N] - ps[i]);
    return a / 2;
}

// checks whether a point is inside a simple polygon
// returns -1 if inside, 0 if on border, 1 if outside
// O(N)
// UNTESTED
int in_poly(const vector<P> &ps, P p) {
    int N = ps.size(), w = 0;
    rep(i, N) {
        P s = ps[i] - p, e = ps[(i + 1) % N] - p;
        if (s == P()) return 0;
        if (s.y == 0 && e.y == 0) {
            if (min(s.x, e.x) <= 0 && 0 <= max(s.x, e.x)) return 0;
        } else {
            bool b = s.y < 0;
            if (b != (e.y < 0)) {
                T z = s / e;
                if (z == 0) return 0;
                if (b == (z > 0)) w += b ? 1 : -1;
            }
        }
    }
    return w ? -1 : 1;
}

// check if a point is in a convex polygon
struct InConvex {
    vector<P> ps;
    T ll, lh, rl, rh;
    int N, m;

    // preprocess polygon
    // O(N)
    InConvex(const vector<P> &p) : ps(p), N(ps.size()), m(0) {
        assert(N >= 2);
        rep(i, N) if (ps[i].x < ps[m].x) m = i;
        rotate(ps.begin(), ps.begin() + m, ps.end());
        rep(i, N) if (ps[i].x > ps[m].x) m = i;
```

```
        ll = lh = ps[0].y, rl = rh = ps[m].y;
        for (P p : ps) {
            if (p.x == ps[0].x) ll = min(ll, p.y), lh = max(lh, p.y);
            if (p.x == ps[m].x) rl = min(rl, p.y), rh = max(rh, p.y);
        }
    }
    InConvex() {}

    // check if point belongs in polygon
    // returns -1 if inside, 0 if on border, 1 if outside
    // O(log N)
    int in_poly(P p) {
        if (p.x < ps[0].x || p.x > ps[m].x) return 1;
        if (p.x == ps[0].x) return p.y < ll || p.y > lh;
        if (p.x == ps[m].x) return p.y < rl || p.y > rh;
        int r = upper_bound(ps.begin(), ps.begin() + m, p, [](P a, P b) {
            return a.x < b.x; }) - ps.begin();
        T z = (ps[r - 1] - ps[r]) / (p - ps[r]);
        if (z >= 0) return !!z;
        r = upper_bound(ps.begin() + m, ps.end(), p, [](P a, P b) { return
            a.x > b.x; }) - ps.begin();
        z = (ps[r - 1] - ps[r % N]) / (p - ps[r % N]);
        if (z >= 0) return !!z;
        return -1;
    }
};
```

2.8 sweep

```
#include "point.cpp"

// iterate over all pairs of points
// `op` is called with all ordered pairs of different indices `(i, j)`
// additionally, the `ps` vector is kept sorted by signed distance
// to the line formed by `i` and `j`
// for example, if the vector from `i` to `j` is pointing right,
// the `ps` vector is sorted from smallest `y` to largest `y`
// note that, because the `ps` vector is sorted by signed distance,
// `j` is always equal to `i + 1`
// this means that the amount of points to the left of the line is always
    `N - i`
template <class OP>
void all_pair_points(vector<P> &ps, OP op) {
    int N = ps.size();
    sort(ps.begin(), ps.end(), [](P a, P b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    vector<pair<int, int>> ss;
    rep(i, N) rep(j, N) if (i != j) ss.push_back({i, j});
    stable_sort(ss.begin(), ss.end(), [&](auto a, auto b) {
        return (ps[a.second] - ps[a.first]).angle_lt(ps[b.second] -
            ps[b.first]);
    });
    vector<int> p(N);
    rep(i, N) p[i] = i;
    for (auto [i, j] : ss) {
        op(p[i], p[j]);
        swap(ps[p[i]], ps[p[j]]);
        swap(p[i], p[j]);
    }
}
```

2.9 theorems

```
// Pick's theorem
//
//    For a simple polygon with integer vertices, the following
    relationship holds:
//
//    A = I + B / 2 - 1
//
//    A: Area of the polygon
//    I: Integer points strictly inside the polygon
//    B: Integer points on the boundary of the polygon
```

3 graph

3.1 bellman-ford

```
#include "../common.h"

const ll INF = 1e18;
```

```
struct Edge {
    int u, v;
    ll w;
};

// find distance from source node to all nodes.
// supports negative edge weights.
// returns true if a negative cycle is detected.
//
// time: O(V E)
bool bellman_ford(int N, int s, vector<Edge> &E, vector<ll> &D,
     vector<int> &P) {
    P.assign(N, -1), D.assign(N, INF), D[s] = 0;
    rep(i, N - 1) {
        bool f = true;
        rep(ei, E.size()) {
            auto &e = E[ei];
            ll n = D[e.u] + e.w;
            if (D[e.u] < INF && n < D[e.v]) D[e.v] = n, P[e.v] = ei, f =
                false;
        }
        if (f) return false;
    }
    return true;
}
```

3.2 dijkstra

```
#include "../common.h"

const ll INF = 1e18;

// calculate shortest distances from a source node to every other node in
// O(E log V). requires an array of size N to store results.
void dijkstra(const vector<vector<pair<ll, int>>> &G, vector<ll> &D, int
     src) {
    priority_queue<pair<ll, int>> q;
    D.assign(G.size(), INF);
    D[src] = 0, q.push({0, src});
    while (!q.empty()) {
        auto [d, u] = q.top();
        d = -d, q.pop();
        if (d > D[u]) continue;
        for (auto [w, v] : G[u])
            if (d + w < D[v]) {
                D[v] = d + w;
                q.push({-D[v], v});
            }
    }
}
```

3.3 dinic

```
#include "../common.h"

const ll INF = 1e18;

struct Edge {
    int u, v;
    ll c, f = 0;
};

// maximum flow algorithm.
//
// time: O(E V^2)
//       O(E V^(2/3)) / O(E sqrt(E)) unit capacities
//       O(E sqrt(V))                unit networks (hopcroft-karp)
//
// unit network: c in {0, 1} and forall v, len(incoming(v)) <= 1 or
     len(outgoing(v)) <= 1
//
// min-cut: find all nodes reachable from the source in the residual graph
struct Dinic {
    int N, s, t;
    vector<vector<int>> G;
    vector<Edge> E;
    vector<int> lvl, ptr;

    Dinic() {}
    Dinic(int N, int s, int t) : N(N), s(s), t(t), G(N) {}

    void add_edge(int u, int v, ll c) {
        G[u].push_back(E.size());
        E.push_back({u, v, c});
        G[v].push_back(E.size());
        E.push_back({v, u, 0});
```

```
    }

    ll push(int u, ll p) {
        if (u == t || p <= 0) return p;
        while (ptr[u] < G[u].size()) {
            int ei = G[u][ptr[u]++];
            Edge &e = E[ei];
            if (lvl[e.v] != lvl[u] + 1) continue;
            ll a = push(e.v, min(e.c - e.f, p));
            if (a <= 0) continue;
            e.f += a, E[ei ^ 1].f -= a;
            return a;
        }
        return 0;
    }

    ll maxflow() {
        ll f = 0;
        while (true) {
            // bfs to build levels
            lvl.assign(N, -1);
            queue<int> q;
            lvl[s] = 0, q.push(s);
            while (!q.empty()) {
                int u = q.front();
                q.pop();
                for (int ei : G[u]) {
                    Edge &e = E[ei];
                    if (e.c - e.f <= 0 || lvl[e.v] != -1) continue;
                    lvl[e.v] = lvl[u] + 1, q.push(e.v);
                }
            }
            if (lvl[t] == -1) break;

            // dfs to find blocking flow
            ptr.assign(N, 0);
            while (ll ff = push(s, INF)) f += ff;
        }
        return f;
    }
};
```

3.4 floyd-warshall

```
#include "../common.h"

const ll INF = 1e18;

// calculate distances between every pair of nodes in O(V^3) time and
    O(V^2)
// memory.
// requires an NxN array to store results.
// works with negative edges, but not negative cycles.
void floyd(const vector<vector<pair<int, ll>>> &G, vector<vector<ll>>
    &dists) {
    int N = G.size();
    rep(i, N) rep(j, N) dists[i][j] = i == j ? 0 : INF;
    rep(i, N) for (auto edge : G[i]) dists[i][edge.first] = edge.second;
    rep(k, N) rep(i, N) rep(j, N) {
        dists[i][j] = min(dists[i][j], dists[i][k] + dists[k][j]);
    }
}
```

3.5 heavy-light

```
#include "../common.h"

struct Hld {
    vector<int> P, H, D, pos, top;

    Hld() {}
    void init(vector<vector<int>> &G) {
        int N = G.size();
        P.resize(N), H.resize(N), D.resize(N), pos.resize(N),
            top.resize(N);
        D[0] = -1, dfs(G, 0);
        int t = 0;
        rep(i, N) if (H[P[i]] != i) {
            int j = i;
            while (j != -1) {
                top[j] = i, pos[j] = t++;
                j = H[j];
            }
        }
    }
```

```cpp
int dfs(vector<vector<int>> &G, int i) {
    int w = 1, mw = 0;
    D[i] = D[P[i]] + 1, H[i] = -1;
    for (int c : G[i]) {
        if (c == P[i]) continue;
        P[c] = i;
        int sw = dfs(G, c);
        if (sw > mw) H[i] = c, mw = sw;
        w += sw;
    }
    return w;
}

template <class OP>
void path(int u, int v, OP op) {
    while (top[u] != top[v]) {
        if (D[top[u]] > D[top[v]]) swap(u, v);
        op(pos[top[v]], pos[v] + 1);
        v = P[top[v]];
    }
    if (D[u] > D[v]) swap(u, v);
    op(pos[u], pos[v] + 1); // value on vertex
    // op(pos[u]+1, pos[v] + 1); // value on path
}

// segment tree
template <class T, class S>
void update(S &seg, int i, T val) {
    seg.update(pos[i], val);
}

// segment tree lazy
template <class T, class S>
void update(S &seg, int u, int v, T val) {
    path(u, v, [&](int l, int r) { seg.update(l, r, val); });
}

template <class T, class S>
T query(S &seg, int u, int v) {
    T ans = 0;                                         // neutral
        element
    path(u, v, [&](int l, int r) { ans += seg.query(l, r); }); //
        query op
    return ans;
}
```

```cpp
};
```

3.6 hungarian

```cpp
#include "../common.h"

const ll INF = 1e18;

// find a maximum gain perfect matching in the given bipartite complete
    graph.
// input: gain matrix (G_{xy} = benefit of joining vertex x in set X with
    vertex
// y in set Y).
// output: maximum gain matching in members ‘xy[x]‘ and ‘yx[y]‘.
// runtime: O(N^3)
struct Hungarian {
    int N, qi, root;
    vector<vector<ll>> gain;
    vector<int> xy, yx, p, q, slackx;
    vector<ll> lx, ly, slack;
    vector<bool> S, T;

    void add(int x, int px) {
        S[x] = true, p[x] = px;
        rep(y, N) if (lx[x] + ly[y] - gain[x][y] < slack[y]) {
            slack[y] = lx[x] + ly[y] - gain[x][y], slackx[y] = x;
        }
    }

    void augment(int x, int y) {
        while (x != -2) {
            yx[y] = x;
            swap(xy[x], y);
            x = p[x];
        }
    }

    void improve() {
        S.assign(N, false), T.assign(N, false), p.assign(N, -1);
        qi = 0, q.clear();
        rep(x, N) if (xy[x] == -1) {
            q.push_back(root = x), p[x] = -2, S[x] = true;
            break;
```

```
        }
        rep(y, N) slack[y] = lx[root] + ly[y] - gain[root][y], slackx[y] =
            root;

        while (true) {
            while (qi < q.size()) {
                int x = q[qi++];
                rep(y, N) if (lx[x] + ly[y] == gain[x][y] && !T[y]) {
                    if (yx[y] == -1) return augment(x, y);
                    T[y] = true, q.push_back(yx[y]), add(yx[y], x);
                }
            }

            ll d = INF;
            rep(y, N) if (!T[y]) d = min(d, slack[y]);
            rep(x, N) if (S[x]) lx[x] -= d;
            rep(y, N) if (T[y]) ly[y] += d;
            rep(y, N) if (!T[y]) slack[y] -= d;

            rep(y, N) if (!T[y] && slack[y] == 0) {
                if (yx[y] == -1) return augment(slackx[y], y);
                T[y] = true;
                if (!S[yx[y]]) q.push_back(yx[y]), add(yx[y], slackx[y]);
            }
        }
    }

    Hungarian(vector<vector<ll>> g)
        : N(g.size()),
          gain(g),
          xy(N, -1),
          yx(N, -1),
          lx(N, -INF),
          ly(N),
          slack(N),
          slackx(N) {
        rep(x, N) rep(y, N) lx[x] = max(lx[x], ly[y]);
        rep(i, N) improve();
    }
};
```

3.7 kuhn

```
#include "../common.h"

// get a maximum cardinality matching in a bipartite graph.
// input: adjacency lists.
// output: matching (in `mt` member).
// runtime: O(V E)
struct Kuhn {
    int N, size;
    vector<vector<int>> G;
    vector<bool> seen;
    vector<int> mt;

    bool visit(int i) {
        if (seen[i]) return false;
        seen[i] = true;
        for (int to : G[i])
            if (mt[to] == -1 || visit(mt[to])) {
                mt[to] = i;
                return true;
            }
        return false;
    }

    Kuhn(vector<vector<int>> adj) : G(adj), N(G.size()), mt(N, -1) {
        rep(i, N) {
            seen.assign(N, false);
            size += visit(i);
        }
    }
};
```

3.8 lca

```
#include "../common.h"

// calculates the lowest common ancestor for any two nodes in O(log N)
    time,
// with O(N log N) preprocessing
struct Lca {
    int L;
    vector<vector<int>> up;
    vector<pair<int, int>> time;
```

```
    Lca() {}
    void init(const vector<vector<int>> &G) {
        int N = G.size();
        L = N <= 1 ? 0 : 32 - __builtin_clz(N - 1);
        up.resize(L + 1);
        rep(l, L + 1) up[l].resize(N);
        time.resize(N);
        int t = 0;
        visit(G, 0, 0, t);
        rep(l, L) rep(i, N) up[l + 1][i] = up[l][up[l][i]];
    }

    void visit(const vector<vector<int>> &G, int i, int p, int &t) {
        up[0][i] = p;
        time[i].first = t++;
        for (int edge : G[i]) {
            if (edge == p) continue;
            visit(G, edge, i, t);
        }
        time[i].second = t++;
    }

    bool is_anc(int up, int dn) {
        return time[up].first <= time[dn].first &&
               time[dn].second <= time[up].second;
    }

    int get(int i, int j) {
        if (is_anc(i, j)) return i;
        if (is_anc(j, i)) return j;
        int l = L;
        while (l >= 0) {
            if (is_anc(up[l][i], j))
                l--;
            else
                i = up[l][i];
        }
        return up[0][i];
    }
};
```

3.9 maxflow-mincost

```
// untested

#include "../common.h"

const ll INF = 1e18;

struct Edge {
    int u, v;
    ll c, w, f = 0;
};

// find the minimum-cost flow among all maximum-flow flows.
//
// time: O(F V E)          F is the maximum flow
//       O(V E + F E log V) if bellman-ford is replaced by johnson
struct Flow {
    int N, s, t;
    vector<vector<int>> G;
    vector<Edge> E;
    vector<ll> d;
    vector<int> p;

    Flow() {}
    Flow(int N, int s, int t) : N(N), s(s), t(t), G(N) {}

    void add_edge(int u, int v, ll c, ll w) {
        G[u].push_back(E.size());
        E.push_back({u, v, c, w});
        G[v].push_back(E.size());
        E.push_back({v, u, 0, -w});
    }

    void calcdists() {
        // replace bellman-ford with johnson for better time
        d.assign(N, INF);
        p.assign(N, -1);
        d[s] = 0;
        rep(i, N - 1) rep(ei, E.size()) {
            Edge &e = E[ei];
            ll n = d[e.u] + e.w;
            if (d[e.u] < INF && e.c - e.f > 0 && n < d[e.v]) d[e.v] = n,
                p[e.v] = ei;
        }
    }
```

```
    ll maxflow() {
        ll ff = 0;
        while (true) {
            calcdists();
            if (p[t] == -1) break;

            ll f = INF;
            int cur = t;
            while (p[cur] != -1) {
                Edge &e = E[p[cur]];
                f = min(f, e.c - e.f);
                cur = e.u;
            }

            int cur = t;
            while (p[cur] != -1) {
                E[p[cur]].f += f;
                E[p[cur] ^ 1].f -= f;
            }

            ff += f;
        }
        return ff;
    }
};
```

3.10 push-relabel

```
#include "../common.h"

const ll INF = 1e18;

// maximum flow algorithm.
// to run, use `maxflow()`.
//
// time: O(V^2 sqrt(E)) <= O(V^3)
// memory: O(V^2)
struct PushRelabel {
    vector<vector<ll>> cap, flow;
    vector<ll> excess;
    vector<int> height;

    PushRelabel() {}

    void resize(int N) { cap.assign(N, vector<ll>(N)); }

    // push as much excess flow as possible from u to v.
    void push(int u, int v) {
        ll f = min(excess[u], cap[u][v] - flow[u][v]);
        flow[u][v] += f;
        flow[v][u] -= f;
        excess[v] += f;
        excess[u] -= f;
    }

    // relabel the height of a vertex so that excess flow may be pushed.
    void relabel(int u) {
        int d = INT32_MAX;
        rep(v, cap.size()) if (cap[u][v] - flow[u][v] > 0) d =
            min(d, height[v]);
        if (d < INF) height[u] = d + 1;
    }

    // get the maximum flow on the network specified by `cap` with source
        `s`
    // and sink `t`.
    // node-to-node flows are output to the `flow` member.
    ll maxflow(int s, int t) {
        int N = cap.size(), M;
        flow.assign(N, vector<ll>(N));
        height.assign(N, 0), height[s] = N;
        excess.assign(N, 0), excess[s] = INF;
        rep(i, N) if (i != s) push(s, i);

        vector<int> q;
        while (true) {
            // find the highest vertices with excess
            q.clear(), M = 0;
            rep(i, N) {
                if (excess[i] <= 0 || i == s || i == t) continue;
                if (height[i] > M) q.clear(), M = height[i];
                if (height[i] >= M) q.push_back(i);
            }
            if (q.empty()) break;
            // process vertices
            for (int u : q) {
                bool relab = true;
                rep(v, N) {
                    if (excess[u] <= 0) break;
```

```
                if (cap[u][v] - flow[u][v] > 0 && height[u] > height[v])
                    push(u, v), relab = false;
            }
            if (relab) {
                relabel(u);
                break;
            }
        }
    }

    ll f = 0;
    rep(i, N) f += flow[i][t];
    return f;
  }
};
```

3.11 strongly-connected-components

```
#include "../common.h"

// compute strongly connected components.
// time: O(V + E), memory: O(V)
//
// after building:
// comp = map from vertex to component (components are toposorted, root
   first, leaf last)
// N = number of components
// G = condensation graph (component DAG)
//
// byproducts:
// vgi = transposed graph
// order = reverse topological sort (leaf first, root last)
//
// others:
// vn = number of vertices
// vg = original vertex graph
struct Scc {
    int vn, N;
    vector<int> order, comp;
    vector<vector<int>> vg, vgi, G;

    void toposort(int u) {
        if (comp[u]) return;
        comp[u] = -1;
        for (int v : vg[u]) toposort(v);
        order.push_back(u);
    }

    bool carve(int u) {
        if (comp[u] != -1) return false;
        comp[u] = N;
        for (int v : vgi[u]) {
            carve(v);
            if (comp[v] != N) G[comp[v]].push_back(N);
        }
        return true;
    }

    Scc() {}
    Scc(vector<vector<int>> &g) : vn(g.size()), vg(g), comp(vn), vgi(vn),
      G(vn), N(0) {
        rep(u, vn) toposort(u);
        rep(u, vn) for (int v : vg[u]) vgi[v].push_back(u);
        invrep(i, vn) N += carve(order[i]);
    }
};
```

3.12 two-sat

```
#include "../common.h"
#include "strongly_connected_components.cpp"

// calculate the solvability of a system of logical equations, where
   every equation is of the form `a or b`.
// `neg`: get negation of `u`
// `then`: `u` implies `v`
// `any`: `u` or `v`
// `set`: `u` is true
//
// after `solve` (O(V+E)) returns true, `sol` contains one possible
   solution.
// determining all solutions is O(V*E) hard (requires computing
   reachability in a DAG).
struct TwoSat {
    int N;
    vector<vector<int>> G;
```

```
    Scc scc;
    vector<bool> sol;

    TwoSat(int n) : N(n), G(2 * n), sol(n) {}
    TwoSat() {}

    int neg(int u) { return (u + N) % (2 * N); }
    void then(int u, int v) { G[u].push_back(v),
        G[neg(v)].push_back(neg(u)); }
    void any(int u, int v) { then(neg(u), v); }
    void set(int u) { G[neg(u)].push_back(u); }

    bool solve() {
        scc = Scc(G);
        rep(u, N) if (scc.comp[u] == scc.comp[neg(u)]) return false;
        rep(u, N) sol[u] = (scc.comp[u] > scc.comp[neg(u)]);
        return true;
    }
};
```

4 implementation

4.1 dsu

```
#include "../common.h"

struct Dsu {
    vector<int> p, r;

    // initialize the disjoint-set-union to all unitary sets
    void reset(int N) {
        p.resize(N), r.assign(N, 0);
        rep(i, N) p[i] = i;
    }

    // find the leader node corresponding to node `i`
    int find(int i) {
        if (p[i] != i) p[i] = find(p[i]);
        return p[i];
    }

    // perform union on the two sets that `i` and `j` belong to
    void unite(int i, int j) {
        i = find(i), j = find(j);
        if (i == j) return;
        if (r[i] > r[j]) swap(i, j);
        if (r[i] == r[j]) r[j] += 1;
        p[i] = j;
    }
};
```

4.2 fenwick-tree

```
#include "../common.h"

template <class T>
struct Ft {
    vector<T> t;

    T neutral() { return 0; }

    Ft() {}
    Ft(int N) : t(N + 1, neutral()) {}

    T query(int r) {
        r = min(r, N);
        T x = 0; // neutral
        for (; r > 0; r -= r & -r)
            x = x + t[r];
        return x;
    }
    T query(int l, int r) { return query(r) - query(l); }

    void update(int i, T x) {
        for (i++;)
    }
};
```

4.3 mo

```
#include "../common.h"

struct Query {
```

```
    int l, r, idx;
};

// answer segment queries using only ‘add(i)‘, ‘remove(i)‘ and ‘get()‘
// functions.
//
// complexity: O((N + Q) * sqrt(N) * F)
//   N = length of the full segment
//   Q = amount of queries
//   F = complexity of the ‘add‘, ‘remove‘ functions
template <class A, class R, class G, class T>
void mo(vector<Query> &queries, vector<T> &ans, A add, R remove, G get) {
    int Q = queries.size(), B = (int)sqrt(Q);
    sort(queries.begin(), queries.end(), [&](Query &a, Query &b) {
        return make_pair(a.l / B, a.r) < make_pair(b.l / B, b.r);
    });
    ans.resize(Q);

    int l = 0, r = 0;
    for (auto &q : queries) {
        while (r < q.r) add(r), r++;
        while (l > q.l) l--, add(l);
        while (r > q.r) r--, remove(r);
        while (l < q.l) remove(l), l++;
        ans[q.idx] = get();
    }
}
```

4.4 persistent-segment-tree-lazy

```
#include "../common.h"

template <class T>
struct Node {
    T x, lz;
    int l = -1, r = -1;
};

template <class T>
struct Pstl {
    int N;
    vector<Node<T>> a;
    vector<int> head;

    T qneut() { return 0; }
    T merge(T l, T r) { return l + r; }
    T uneut() { return 0; }
    T accum(T u, T x) { return u + x; }
    T apply(T x, T lz, int l, int r) { return x + (r - l) * lz; }

    int build(int vl, int vr) {
        if (vr - vl == 1) a.push_back({qneut(), uneut()}); // node
            construction
        else {
            int vm = (vl + vr) / 2, l = build(vl, vm), r = build(vm, vr);
            a.push_back({merge(a[l].x, a[r].x), uneut(), l, r}); // query
                merge
        }
        return a.size() - 1;
    }

    T query(int l, int r, int v, int vl, int vr, T acc) {
        if (l >= vr || r <= vl) return qneut();              // query
            neutral
        if (l <= vl && r >= vr) return apply(a[v].x, acc, vl, vr); //
            update op
        acc = accum(acc, a[v].lz);                            // update
            merge
        int vm = (vl + vr) / 2;
        return merge(query(l, r, a[v].l, vl, vm, acc), query(l, r, a[v].r,
            vm, vr, acc)); // query merge
    }

    int update(int l, int r, T x, int v, int vl, int vr) {
        if (l >= vr || r <= vl || r <= l) return v;
        a.push_back(a[v]);
        v = a.size() - 1;
        if (l <= vl && r >= vr) {
            a[v].x = apply(a[v].x, x, vl, vr); // update op
            a[v].lz = accum(a[v].lz, x);    // update merge
        } else {
            int vm = (vl + vr) / 2;
            a[v].l = update(l, r, x, a[v].l, vl, vm);
            a[v].r = update(l, r, x, a[v].r, vm, vr);
            a[v].x = merge(a[a[v].l].x, a[a[v].r].x); // query merge
        }
        return v;
    }
```

```
    Pstl() {}
    Pstl(int N) : N(N) { head.push_back(build(0, N)); }

    T query(int t, int l, int r) {
        return query(l, r, head[t], 0, N, uneut()); // update neutral
    }
    int update(int t, int l, int r, T x) {
        return head.push_back(update(l, r, x, head[t], 0, N)), head.size()
            - 1;
    }
};
```

4.5 persistent-segment-tree

```
#include "../common.h"

// usage:
// Pst<Node<ll>> pst;
// pst = {N};
// int newtime = pst.update(time, index, value);
// Node<ll> result = pst.query(newtime, left, right);

template <class T>
struct Node {
    T x;
    int l = -1, r = -1;

    Node() : x(0) {}
    Node(T x) : x(x) {}
    Node(Node a, Node b, int l = -1, int r = -1) : x(a.x + b.x), l(l),
        r(r) {}
};

template <class U>
struct Pst {
    int N;
    vector<U> a;
    vector<int> head;

    int build(int vl, int vr) {
        if (vr - vl == 1) a.push_back(U()); // node construction
        else {
            int vm = (vl + vr) / 2, l = build(vl, vm), r = build(vm, vr);
            a.push_back(U(a[l], a[r], l, r)); // query merge
        }
        return a.size() - 1;
    }

    U query(int l, int r, int v, int vl, int vr) {
        if (l >= vr || r <= vl) return U(); // query neutral
        if (l <= vl && r >= vr) return a[v];
        int vm = (vl + vr) / 2;
        return U(query(l, r, a[v].l, vl, vm), query(l, r, a[v].r, vm,
            vr)); // query merge
    }

    int update(int i, U x, int v, int vl, int vr) {
        a.push_back(a[v]);
        v = a.size() - 1;
        if (vr - vl == 1) a[v] = x; // update op
        else {
            int vm = (vl + vr) / 2;
            if (i < vm) a[v].l = update(i, x, a[v].l, vl, vm);
            else a[v].r = update(i, x, a[v].r, vm, vr);
            a[v] = U(a[a[v].l], a[a[v].r], a[v].l, a[v].r); // query merge
        }
        return v;
    }

    Pst() {}
    Pst(int N) : N(N) { head.push_back(build(0, N)); }

    U query(int t, int l, int r) {
        return query(l, r, head[t], 0, N);
    }
    int update(int t, int i, U x) {
        return head.push_back(update(i, x, head[t], 0, N)), head.size() -
            1;
    }
};
```

4.6 search

```
#include "common.h"
```

```
// search x in a[i]
//
// first a[i] > x:  upper_bound(a, x)
// first a[i] >= x: lower_bound(a, x)
//  last a[i] < x: --lower_bound(a, x)
//  last a[i] <= x: --upper_bound(a, x)

// note: searching for the largest [l, r] such that f(l) > a & f(r) < b
    where
// [a, b] is a range in f() space may result in negative [l, r] ranges.

// searches for a value in an [l, r] range (both inclusive).
//
// the `isleft(m)` function evaluates whether `m` is strictly to the left
    of the
// target value.
int binsearch_left(int l, int r, bool isleft(int)) {
    while (l != r) {
        int m = (l + r) / 2;
        if (isleft(m)) {
            l = m + 1;
        } else {
            r = m;
        }
    }
    return l;
}

// searches for a value in an [l, r] range (both inclusive).
//
// the `isright(m)` function evaluates whether `m` is strictly to the
    right of
// the target value.
//
// note the `+1` when computing `m`, which avoids infinite loops.
// the only difference with `binsearch_left` is how the evaluation
    function is
// specified. both are functionally identical.
int binsearch_right(int l, int r, bool isright(int)) {
    while (l != r) {
        int m = (l + r + 1) / 2;
        if (isright(m)) {
            r = m - 1;
        } else {
            l = m;
        }
    }
    return l;
}

// continuous ternary (golden section) search.
//
// searches for a minimum value of the given unimodal function (monotonic
// positive derivative).
template <typename T, typename U>
pair<T, U> ctersearch(int iter, T l, T r, U f(T)) {
    const T INVG = 0.61803398874989484820;

    T m = l + (r - l) * INVG;
    U lv = f(l), rv = f(r), mv = f(m);
    rep(i, iter) {
        T x = l + (m - l) * INVG;
        U xv = f(x);
        if (xv > mv) l = r, lv = rv, r = x, rv = xv;
        else r = m, rv = mv, m = x, mv = xv;
    }
    return {m, mv};
}
```

4.7 segment-tree-lazy

```
#include "../common.h"

// 0-based, inclusive-exclusive
// usage:
// Stl3<ll> a;
// a = {N};
template <class T>
struct Stl3 {
    // immediate, lazy
    vector<pair<T, T>> a;

    T qneutral() { return 0; }
    T merge(T l, T r) { return l + r; }
    T uneutral() { return 0; }
    void update(pair<T, T> &u, T val, int l, int r) { u.first += val * (r
        - l), u.second += val; }
```

```
    St13() {}
    St13(int N) : a(4 * N, {qneutral(), uneutral()}) {} // node neutral

    void push(int v, int vl, int vm, int vr) {
        update(a[2 * v], a[v].second, vl, vm); // node update
        update(a[2 * v + 1], a[v].second, vm, vr); // node update
        a[v].second = uneutral();                 // update neutral
    }

    // query for range [l, r)
    T query(int l, int r, int v = 1, int vl = 0, int vr = -1) {
        if (vr == -1) vr = a.size() / 4;
        if (l <= vl && r >= vr) return a[v].first; // query op
        if (l >= vr || r <= vl) return qneutral(); // query neutral
        int vm = (vl + vr) / 2;
        push(v, vl, vm, vr);
        return merge(query(l, r, 2 * v, vl, vm), query(l, r, 2 * v + 1,
            vm, vr)); // item merge
    }

    // update range [l, r) using val
    void update(int l, int r, T val, int v = 1, int vl = 0, int vr = -1) {
        if (vr == -1) vr = a.size() / 4;
        if (l >= vr || r <= vl || r <= l) return;
        if (l <= vl && r >= vr) update(a[v], val, vl, vr); // node update
        else {
            int vm = (vl + vr) / 2;
            push(v, vl, vm, vr);
            update(l, r, val, 2 * v, vl, vm);
            update(l, r, val, 2 * v + 1, vm, vr);
            a[v].first = merge(a[2 * v].first, a[2 * v + 1].first); //
                node merge
        }
    }
};

struct Node {
    ll x, lazy;

    Node() : x(neutral()), lazy(0) {} // query neutral, update neutral
    Node(ll x_) : Node() { x = x_; }
    Node(Node &l, Node &r) : Node() { refresh(l, r); } // node merge
        construction
    void refresh(Node &l, Node &r) { x = merge(l.x, r.x); } // node merge
    void update(ll val, int l, int r) { x += val * (r - l), lazy += val;
        } // update-query, update accumulate
    ll take() {
        ll z = 0; // update neutral
        swap(lazy, z);
        return z;
    }

    ll query() { return x; }
    static ll neutral() { return 0; }         // query neutral
    static ll merge(ll l, ll r) { return l + r; } // query merge
};

template <class T, class Node>
struct Stl {
    vector<Node> node;

    void reset(int N) { node.assign(4 * N, {}); } // node neutral

    void build(const vector<T> &a, int v = 1, int vl = 0, int vr = -1) {
        node.resize(4 * a.size()), vr = vr == -1 ? node.size() / 4 : vr;
        if (vr - vl == 1) {
            node[v] = {a[vl]}; // node construction
            return;
        }
        int vm = (vl + vr) / 2;
        build(a, 2 * v, vl, vm);
        build(a, 2 * v + 1, vm, vr);
        node[v] = {node[2 * v], node[2 * v + 1]}; // node merge
            construction
    }

    void push(int v, int vl, int vm, int vr) {
        T lazy = node[v].take();          // update neutral
        node[2 * v].update(lazy, vl, vm); // node update
        node[2 * v + 1].update(lazy, vm, vr); // node update
    }

    // query for range [l, r)
    T query(int l, int r, int v = 1, int vl = 0, int vr = -1) {
        if (vr == -1) vr = node.size() / 4;
        if (l <= vl && r >= vr) return node[v].query(); // query op
        if (l >= vr || r <= vl) return Node::neutral(); // query neutral
        int vm = (vl + vr) / 2;
        push(v, vl, vm, vr);
```

```
        return Node::merge(query(l, r, 2 * v, vl, vm), query(l, r, 2 * v +
            1, vm, vr)); // item merge
    }

    // update range [l, r) using val
    void update(int l, int r, T val, int v = 1, int vl = 0, int vr = -1) {
        if (vr == -1) vr = node.size() / 4;
        if (l >= vr || r <= vl || r <= l) return;
        if (l <= vl && r >= vr) node[v].update(val, vl, vr); // node update
        else {
            int vm = (vl + vr) / 2;
            push(v, vl, vm, vr);
            update(l, r, val, 2 * v, vl, vm);
            update(l, r, val, 2 * v + 1, vm, vr);
            node[v].refresh(node[2 * v], node[2 * v + 1]); // node merge
        }
    }
};
```

4.8 segment-tree

```
#include "../common.h"

// usage:
// St<Node<ll>> st;
// st = {N};
// st.update(index, new_value);
// Node<ll> result = st.query(left, right);

template <class T>
struct Node {
    T x;
    Node() : x(0) {}
    Node(T x) : x(x) {}
    Node(Node a, Node b) : x(a.x + b.x) {}
};

template <class U>
struct St {
    vector<U> a;

    St3() {}
    St3(int N) : a(4 * N, U()) {} // node neutral

    // query for range [l, r)
    U query(int l, int r, int v = 1, int vl = 0, int vr = -1) {
        if (vr == -1) vr = a.size() / 4;
        if (l <= vl && r >= vr) return a[v]; // item construction
        int vm = (vl + vr) / 2;
        if (l >= vr || r <= vl) return U();                           //
            item neutral
        return U(query(l, r, 2 * v, vl, vm), query(l, r, 2 * v + 1, vm,
            vr)); // item merge
    }

    // set element i to val
    void update(int i, U val, int v = 1, int vl = 0, int vr = -1) {
        if (vr == -1) vr = a.size() / 4;
        if (vr - vl == 1) a[v] = val; // item update
        else {
            int vm = (vl + vr) / 2;
            if (i < vm) update(i, val, 2 * v, vl, vm);
            else update(i, val, 2 * v + 1, vm, vr);
            a[v] = U(a[2 * v], a[2 * v + 1]); // node merge
        }
    }
};
```

4.9 sparse-table

```
#include "../common.h"

// handle immutable range maximum queries (or any idempotent query) in
    O(1)
template <class T>
struct Sparse {
    vector<vector<T>> st;

    T op(T a, T b) { return max(a, b); }

    Sparse() {}

    void reset(int N) { st = {vector<T>(N)}; }
    void set(int i, T val) { st[0][i] = val; }

    // O(N log N) time
```

```
    // O(N log N) memory
    void init() {
        int N = st[0].size();
        int npot = N <= 1 ? 1 : 32 - __builtin_clz(N);
        st.resize(npot);
        repx(i, 1, npot) rep(j, N + 1 - (1 << i)) st[i].push_back(
            op(st[i - 1][j], st[i - 1][j + (1 << (i - 1))])); // query op
    }

    // query maximum in the range [l, r) in O(1) time
    // range must be nonempty!
    T query(int l, int r) {
        int i = 31 - __builtin_clz(r - l);
        return op(st[i][l], st[i][r - (1 << i)]); // query op
    }
};
```

4.10 unordered-map

```
#include "../common.h"

// hackproof rng
static mt19937
    rng(chrono::steady_clock::now().time_since_epoch().count());

// deterministic rng
uint64_t splitmix64(uint64_t *x) {
    uint64_t z = (*x += 0x9e3779b97f4a7c15);
    z = (z ^ (z >> 30)) * 0xbf58476d1ce4e5b9;
    z = (z ^ (z >> 27)) * 0x94d049bb133111eb;
    return z ^ (z >> 31);
}

// hackproof unordered map hash
struct Hash {
    size_t operator()(const ll &x) const {
        static const uint64_t RAND =
            chrono::steady_clock::now().time_since_epoch().count();
        uint64_t z = x + RAND + 0x9e3779b97f4a7c15;
        z = (z ^ (z >> 30)) * 0xbf58476d1ce4e5b9;
        z = (z ^ (z >> 27)) * 0x94d049bb133111eb;
        return z ^ (z >> 31);
    }
};

// hackproof unordered_map
template <class T, class U>
using umap = unordered_map<T, U, Hash>;

// hackproof unordered_set
template <class T>
using uset = unordered_set<T, Hash>;

// an unordered map with small integer keys that avoids hashing, but
    allows O(N)
// iteration and clearing, with N being the amount of items (not the
    maximum
// key).
template <class T>
struct Map {
    int N;
    vector<bool> used;
    vector<int> keys;
    vector<T> vals;

    Map() : N(0) {}
    // O(C)
    void recap(int C) {
        C += 1, used.resize(C), keys.resize(C), vals.resize(C);
    }

    // O(1)
    T &operator[](int k) {
        if (!used[k]) used[k] = true, keys[N++] = k, vals[k] = T();
        return vals[k];
    }

    // O(N)
    void clear() {
        while (N) used[keys[--N]] = false;
    }

    // O(N)
    template <class OP>
    void iterate(OP op) {
        rep(i, N) op(keys[i], vals[keys[i]]);
    }
};
```

5 imprimible

6 math

6.1 arithmetic

```
#include "../common.h"

// floor(log2(n)) without precision loss
inline int floor_log2(int n) { return n <= 1 ? 0 : 31 - __builtin_clz(n);
    }
// ceil(log2(n)) without precision loss
inline int ceil_log2(int n) { return n <= 1 ? 0 : 32 - __builtin_clz(n -
    1); }

inline ll floordiv(ll a, ll b) {
   ll d = a / b;
   return d * b == a ? d : d - ((a < 0) ^ (b < 0));
}

inline ll ceildiv(ll a, ll b) {
   ll d = a / b;
   return d * b == a ? d : d - ((a < 0) ^ (b < 0)) + 1;
}

// a^e through binary exponentiation.
ll binexp(ll a, ll e) {
   ll res = 1; // neutral element
   while (e) {
      if (e & 1) res = res * a; // multiplication
      a = a * a;             // multiplication
      e >>= 1;
   }
   return res;
}

#ifndef NOMAIN_ARITH

int main() {
   // Floor division
   assert(floordiv(3, 5) == 0);
   assert(floordiv(7, 5) == 1);
   assert(floordiv(-2, 5) == -1);
   assert(floordiv(-7, 3) == -3);
   assert(floordiv(-6, 3) == -2);
   assert(floordiv(-0, 7) == 0);
   assert(floordiv(2, -5) == -1);
   assert(floordiv(7, -3) == -3);
   assert(floordiv(6, -3) == -2);
   assert(floordiv(0, -7) == 0);

   // Ceil division
   assert(ceildiv(3, 5) == 1);
   assert(ceildiv(7, 5) == 2);
   assert(ceildiv(-2, 5) == 0);
   assert(ceildiv(-7, 3) == -2);
   assert(ceildiv(-6, 3) == -2);
   assert(ceildiv(-0, 7) == 0);
   assert(ceildiv(2, -5) == 0);
   assert(ceildiv(7, -3) == -2);
   assert(ceildiv(6, -3) == -2);
   assert(ceildiv(0, -7) == 0);

   // Count divisors
   assert(count_divisors(1) == 1);
   assert(count_divisors(2) == 2);
   assert(count_divisors(3) == 2);
   assert(count_divisors(4) == 3);
   assert(count_divisors(5) == 2);
   assert(count_divisors(6) == 4);
   assert(count_divisors(7) == 2);
   assert(count_divisors(16) == 5);
   assert(count_divisors(42) == 8);
   assert(count_divisors(101) == 2);
}

#endif
```

6.2 bigint

```
#include "../common.h"

using u32 = uint32_t;
using u64 = uint64_t;

// signed bigint
struct bigint {
```

```
vector<u32> digits;
u32 neg;

bigint() : neg(0) {}
bigint(ll x) : digits{lo(x), hi(x)}, neg(x < 0 ? ~0 : 0) {
    this->trim(); }
bigint(vector<u32> d) : digits(d), neg(0) {}

static u32 lo(u64 dw) { return (u32)dw; }
static u32 hi(u64 dw) { return (u32)(dw >> 32); }

// remove leading zeros from representation
void trim() {
   while (digits.size() && digits.back() == neg) digits.pop_back();
}

void add(const bigint &rhs, u32 c = 0) {
   int ls = digits.size();
   int rs = rhs.digits.size();
   rep(i, max(ls, rs)) {
      if (i >= ls) digits.push_back(neg);
      u64 r = (u64)digits[i] + (i < rs ? rhs.digits[i] : rhs.neg) +
          c;
      digits[i] = lo(r), c = hi(r);
   }
   u64 ec = (u64)c + neg + rhs.neg;
   neg = ((hi(ec) ^ neg ^ rhs.neg) & 1 ? ~0 : 0);
   if (lo(ec) != neg) digits.push_back(lo(ec));
}
bigint &operator+=(const bigint &rhs) {
   this->add(rhs);
   return *this;
}
bigint &operator+=(u32 rhs) {
   this->add({}, rhs);
   return *this;
}

void negate() {
   rep(i, digits.size()) digits[i] = ~digits[i];
   neg = ~neg;
   this->add({}, 1);
}

bigint negated() const {
   bigint out = *this;
   out.negate();
   return out;
}

bigint &operator-=(const bigint &rhs) {
   this->negate();
   *this += rhs;
   this->negate();
   return *this;
}

bigint &operator*=(bigint &rhs) {
   static bigint lhs;
   swap(*this, lhs), digits.clear(), neg = 0;
   u32 r = rhs.neg, s = 0;
   if (lhs.neg) s ^= lhs.neg, lhs.negate();
   if (rhs.neg) s ^= rhs.neg, rhs.negate();
   rep(j, rhs.digits.size()) {
      u64 c = 0;
      int ls = digits.size();
      int rs = lhs.digits.size();
      repx(i, j, max(ls, rs + j)) {
         if (i >= ls) digits.push_back(0);
         u64 r =
            (u64)digits[i] +
            (u64)(i - j < rs ? lhs.digits[i - j] : 0) *
               rhs.digits[j] +
            c;
         digits[i] = lo(r), c = hi(r);
      }
      if (c != 0) digits.push_back(c);
   }
   if (r) rhs.negate();
   if (s) negate();
   return *this;
}

bigint &operator/=(bigint &rhs) {
   divmod(rhs);
   return *this;
}
bigint &operator%=(bigint &rhs) {
   *this = divmod(rhs);
   return *this;
```

```
}

int divmod_trunc(int rhs) {
    u32 s = (rhs < 0 ? ~0 : 0) ^ this->neg, q = abs(rhs);
    u64 r = 0;
    if (this->neg) this->negate();
    invrep(i, digits.size()) {
        r = (r << 32) | digits[i];
        digits[i] = r / q, r %= q;
    }
    if (s) {
        this->negate();
        return -(int)r;
    }
    return (int)r;
}

// compares `this` with `rhs`
// `this < rhs`: -1
// `this == rhs`: 0
// `this > rhs`: 1
int cmp(const bigint &rhs) const {
    if (neg && !rhs.neg) return -1;
    if (!neg && rhs.neg) return 1;
    int ls = digits.size(), rs = rhs.digits.size();
    invrep(i, max(ls, rs)) {
        u32 l = i < ls ? digits[i] : neg;
        u32 r = i < rs ? rhs.digits[i] : rhs.neg;
        if (l < r) return -1;
        if (l > r) return 1;
    }
    return 0;
}

bool operator==(const bigint &rhs) const { return cmp(rhs) == 0; }
bool operator!=(const bigint &rhs) const { return cmp(rhs) != 0; }
bool operator<(const bigint &rhs) const { return cmp(rhs) == -1; }
bool operator>=(const bigint &rhs) const { return cmp(rhs) != -1; }
bool operator>(const bigint &rhs) const { return cmp(rhs) == 1; }
bool operator<=(const bigint &rhs) const { return cmp(rhs) != 1; }

friend ostream &operator<<(ostream &s, const bigint &self) {
    if (self == bigint()) return s << "0";
    bigint x = self;
    if (x.neg) {
        x.negate();
        s << "-";
    }
    vector<int> digs;
    while (x != bigint()) digs.push_back(x.divmod_trunc(10));
    invrep(i, digs.size()) s << digs[i];
    return s;
}

// truncating division and modulo
bigint divmod(bigint &rhs) {
    assert(rhs != bigint());
    u32 sr = rhs.neg, s = neg ^ rhs.neg;
    if (neg) negate();
    if (sr) rhs.negate();
    bigint l = 0, r = *this, x;
    r += 1u;
    while (l != r) {
        bigint m = l;
        m += r;
        rep(i, m.digits.size()) m.digits[i] =
            (m.digits[i] >> 1) |
            (i + 1 < m.digits.size() ? m.digits[i + 1] << 31 : 0);
        x = m, x *= rhs;
        if (x <= *this) {
            l = (m += 1);
        } else {
            r = m;
        }
    }
    l -= 1, swap(l, *this);
    r = *this, r *= rhs, l -= r;
    trim(), l.trim();
    if (sr) rhs.negate();
    if (s) negate(), l.negate();
    return l;
}
};
```

6.3 crt

```
#define NOMAIN_BIGINT
```

```
#include "mod.cpp"

pair<ll, ll> solve_crt(const vector<pair<ll, ll>> &eqs) {
    ll a0 = eqs[0].first, p0 = eqs[0].second;
    repx(i, 1, eqs.size()) {
        ll a1 = eqs[i].first, p1 = eqs[i].second;
        ll k1, k0;
        ll d = ext_gcd(p1, p0, k1, k0);
        a0 -= a1;
        if (a0 % d != 0) return {-1, -1};
        p0 = p0 / d * p1;
        a0 = a0 / d * k1 % p0 * p1 % p0 + a1;
        a0 = (a0 % p0 + p0) % p0;
    }
    return {a0, p0};
}
```

6.4 discrete-log

```
#include "../common.h"
#include "../implementation/unordered_map.cpp"
#include "mod.cpp"

// discrete logarithm log_a(b).
// solve b ^ x = a (mod M) for the smallest x.
// returns -1 if no solution is found.
//
// time: O(sqrt(M))
ll dlog(ll a, ll b, ll M) {
    ll k = 1, s = 0;
    while (true) {
        ll g = __gcd(b, M);
        if (g <= 1) break;
        if (a == k) return s;
        if (a % g != 0) return -1;
        a /= g, M /= g, s += 1, k = b / g * k % M;
    }
    ll N = sqrt(M) + 1;

    umap<ll, ll> r;
    rep(q, N + 1) {
        r[a] = q;
        a = a * b % M;
    }

    ll bN = binexp(b, N, M), bNp = k;
    repx(p, 1, N + 1) {
        bNp = bNp * bN % M;
        if (r.count(bNp)) return N * p - r[bNp] + s;
    }
    return -1;
}
```

6.5 gauss

```
#include "../common.h"

const double EPS = 1e-9;

// solve a system of equations.
// complexity: O(min(N, M) * N * M)
//
// `a` is a list of rows
// the last value in each row is the result of the equation
// return values:
//  0 -> no solutions
//  1 -> unique solution, stored in `ans`
// -1 -> infinitely many solutions, one of which is stored in `ans`
int gauss(vector<vector<double>> a, vector<double> &ans) {
    int N = a.size(), M = a[0].size() - 1;

    vector<int> where(M, -1);
    for (int col = 0, row = 0; col < M && row < N; col++, row++) {
        int sel = row;
        repx(i, row, N) if (abs(a[i][col]) > abs(a[sel][col])) sel = i;
        if (abs(a[sel][col]) < EPS) continue;
        repx(i, col, M + 1) swap(a[sel][i], a[row][i]);
        where[col] = row;

        rep(i, N) if (i != row) {
            double c = a[i][col] / a[row][col];
            repx(j, col, M + 1) a[i][j] -= a[row][j] * c;
        }
    }

    ans.assign(M, 0);
```

```
    rep(i, M) if (where[i] != -1) ans[i] = a[where[i]][M] /
        a[where[i]][i];
    rep(i, N) {
        double sum = 0;
        rep(j, M) sum += ans[j] * a[i][j];
        if (abs(sum - a[i][M]) > EPS) return 0;
    }

    rep(i, M) if (where[i] == -1) return -1;
    return 1;
}
```

6.6 matrix

```
#include "../common.h"

using T = ll;

struct Mat {
    int N, M;
    vector<vector<T>> v;

    Mat(int n, int m) : N(n), M(m), v(N, vector<T>(M)) {}
    Mat(int n) : Mat(n, n) { rep(i, N) v[i][i] = 1; }

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

    Mat operator*(Mat &r) {
        assert(M == r.N);
        int n = N, m = r.M, p = M;
        Mat a(n, m);
        rep(i, n) rep(j, m) {
            a[i][j] = T();                              // neutral
            rep(k, p) a[i][k] = a[i][j] + v[i][k] * r[k][j]; // mul, add
        }
        return a;
    }

    Mat binexp(ll e) {
        assert(N == M);
        Mat a = *this, res(N); // neutral
        while (e) {
            if (e & 1) res = res * a; // mul
            a = a * a;              // mul
            e >>= 1;
        }
        return res;
    }

    friend ostream &operator<<(ostream &s, Mat &a) {
        rep(i, a.N) {
            rep(j, a.M) s << a[i][j] << " ";
            s << endl;
        }
        return s;
    }
};
```

6.7 mod

```
#include "../common.h"

ll binexp(ll a, ll e, ll M) {
    assert(e >= 0);
    ll res = 1 % M;
    while (e) {
        if (e & 1) res = res * a % M;
        a = a * a % M;
        e >>= 1;
    }
    return res;
}

ll multinv(ll a, ll M) { return binexp(a, M - 2, M); }

// calculate gcd(a, b).
// also, calculate x and y such that:
// a * x + b * y == gcd(a, b)
//
// time: O(log min(a, b))
// (ignoring complexity of arithmetic)
ll ext_gcd(ll a, ll b, ll &x, ll &y) {
    if (b == 0) {
        x = 1, y = 0;
        return a;
    }
```

```
    ll d = ext_gcd(b, a % b, y, x);
    y -= a / b * x;
    return d;
}

// compute inverse with any M.
// a and M must be coprime for inverse to exist!
ll multinv_euc(ll a, ll M) {
    ll x, y;
    ext_gcd(a, M, x, y);
    return x;
}

// multiply two big numbers (~10^18) under a large modulo, without
     resorting to
// bigints.
ll bigmul(ll x, ll y, ll M) {
    ll z = 0;
    while (y) {
        if (y & 1) z = (z + x) % M;
        x = (x << 1) % M, y >>= 1;
    }
    return z;
}

struct Mod {
    int a;
    static const int M = 1e9 + 7;

    Mod(ll aa) : a((aa % M + M) % M) {}

    Mod operator+(Mod rhs) const { return (a + rhs.a) % M; }
    Mod operator-(Mod rhs) const { return (a - rhs.a + M) % M; }
    Mod operator-() const { return Mod(0) - *this; }
    Mod operator*(Mod rhs) const { return (ll)a * rhs.a % M; }

    Mod operator+=(Mod rhs) { return *this = *this + rhs; }
    Mod operator-=(Mod rhs) { return *this = *this - rhs; }
    Mod operator*=(Mod rhs) { return *this = *this * rhs; }

    Mod bigmul(ll big) const { return ::bigmul(a, big, M); }

    Mod binexp(ll e) const { return ::binexp(a, e, M); }
    // Mod multinv() const { return ::multinv(a, M); } // prime M
    Mod multinv() const { return ::multinv_euc(a, M); } // possibly
        composite M
};

// dynamic modulus
struct DMod {
    int a, M;

    DMod(ll aa, ll m) : M(m), a((aa % m + m) % m) {}

    DMod operator+(DMod rhs) const { return {(a + rhs.a) % M, M}; }
    DMod operator-(DMod rhs) const { return {(a - rhs.a + M) % M, M}; }
    DMod operator-() const { return DMod(0, M) - *this; }
    DMod operator*(DMod rhs) const { return {(ll)a * rhs.a % M, M}; }

    DMod operator+=(DMod rhs) { return *this = *this + rhs; }
    DMod operator-=(DMod rhs) { return *this = *this - rhs; }
    DMod operator*=(DMod rhs) { return *this = *this * rhs; }

    DMod bigmul(ll big) const { return {::bigmul(a, big, M), M}; }

    DMod binexp(ll e) const { return {::binexp(a, e, M), M}; }
    DMod multinv() const { return {::multinv(a, M), M}; } // prime M
    // DMod multinv() const { return {::multinv_euc(a, M), M}; } //
        possibly composite M
};
```

6.8 poly

```
#include "../common.h"

#define NOMAIN_MOD
#include "mod.cpp"

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

// compute the DFT of a power-of-two-length sequence.
// if ‘inv‘ is true, computes the inverse DFT.
//
// the DFT of a polynomial A(x) = A0 + A1*x + A2*x^2 + ... + An*x^n is
     the array
```

```
// of the polynomial A evaluated in all nths roots of unity: [A(w0),
    A(w1),
// A(w2), ..., A(wn-1)], where w0 = 1 and w1 is the nth principal root of
    unity.
void fft(vector<cd> &a, bool inv) {
    int N = a.size(), k = 0;
    assert(N == 1 << __builtin_ctz(N));

    rep(i, N) {
        int b = N >> 1;
        while (k & b) k ^= b, b >>= 1;
        k ^= b;
        if (i < k) swap(a[i], a[k]);
    }

    for (int l = 2; l <= N; l <<= 1) {
        double ang = 2 * PI / l * (inv ? -1 : 1);
        cd wl(cos(ang), sin(ang));
        for (int i = 0; i < N; i += l) {
            cd w(1);
            repx(j, 0, l / 2) {
                cd u = a[i + j], v = a[i + j + l / 2] * w;
                a[i + j] = u + v;
                a[i + j + l / 2] = u - v;
                w *= wl;
            }
        }
    }

    if (inv)
        for (cd &x : a) x /= N;
}

const ll MOD = 7340033, ROOT = 5, ROOTPOW = 1 << 20;

void find_root_of_unity(ll M) {
    ll c = M - 1, k = 0;
    while (c % 2 == 0) c /= 2, k += 1;

    // find proper divisors of M - 1
    vector<int> divs;
    repx(d, 1, c) {
        if (d * d > c) break;
        if (c % d == 0) rep(i, k + 1) divs.push_back(d << i);
    }
    rep(i, k) divs.push_back(c << i);

    // find any primitive root of M
    ll G = -1;
    repx(g, 2, M) {
        bool ok = true;
        for (int d : divs) ok &= (binexp(g, d, M) != 1);
        if (ok) {
            G = g;
            break;
        }
    }
    assert(G != -1);

    ll w = binexp(G, c, M);
    cerr << M << " = c * 2^k + 1" << endl;
    cerr << " c = " << c << endl;
    cerr << " k = " << k << endl;
    cerr << "w^(2^k) == 1" << endl;
    cerr << " w = " << w << endl;
}

// compute the DFT of a power-of-two-length sequence, modulo a special
    prime
// number with principal root.
//
// the modulus _must_ be a prime number with an Nth root of unity, where
    N is a
// power of two. the FFT can only be performed on arrays of size <= N.
void ntt(vector<ll> &a, bool inv) {
    int N = a.size(), k = 0;
    assert(N == 1 << __builtin_ctz(N) && N <= ROOTPOW);
    rep(i, N) a[i] = (a[i] % MOD + MOD) % MOD;

    repx(i, 1, N) {
        int b = N >> 1;
        while (k & b) k ^= b, b >>= 1;
        k ^= b;
        if (i < k) swap(a[i], a[k]);
    }

    for (int l = 2; l <= N; l <<= 1) {
        ll wl = inv ? multinv(ROOT, MOD) : ROOT;
        for (ll i = ROOTPOW; i > l; i >>= 1) wl = wl * wl % MOD;
        for (int i = 0; i < N; i += l) {
```

```
            ll w = 1;
            repx(j, 0, l / 2) {
                ll u = a[i + j], v = a[i + j + l / 2] * w % MOD;
                a[i + j] = (u + v) % MOD;
                a[i + j + l / 2] = (u - v + MOD) % MOD;
                w = w * wl % MOD;
            }
        }
    }

    ll ninv = multinv(N, MOD);
    if (inv)
        for (ll &x : a) x = x * ninv % MOD;
}

void convolve(vector<ll> &a, vector<ll> b, int n) {
    n = 1 << (32 - __builtin_clz(2 * n - 1));
    a.resize(n), b.resize(n);
    ntt(a, false), ntt(b, false);
    rep(i, n) a[i] *= b[i];
    ntt(a, true), ntt(b, true);
}

using T = ll;
T pmul(T a, T b) { return a * b % MOD; }
T padd(T a, T b) { return (a + b) % MOD; }
T psub(T a, T b) { return (a - b + MOD) % MOD; }
T pinv(T a) { return multinv(a, MOD); }

struct Poly {
    vector<T> a;

    Poly() {}
    Poly(T c) : a(c) { trim(); }
    Poly(vector<T> c) : a(c) { trim(); }

    void trim() {
        while (!a.empty() && a.back() == 0) a.pop_back();
    }
    int deg() const { return a.empty() ? -1000000 : a.size() - 1; }
    Poly sub(int l, int r) const {
        r = min(r, (int)a.size()), l = min(l, r);
        return vector<T>(a.begin() + l, a.begin() + r);
    }
    Poly trunc(int n) const { return sub(0, n); }
    Poly shl(int n) const {
        Poly out = *this;
        out.a.insert(out.a.begin(), n, 0);
        return out;
    }
    Poly rev(int n, bool r = false) const {
        Poly out(*this);
        if (r) out.a.resize(max(n, (int)a.size()));
        reverse(out.a.begin(), out.a.end());
        return out.trunc(n);
    }

    Poly &operator+=(const Poly &rhs) {
        auto &b = rhs.a;
        a.resize(max(a.size(), b.size()));
        rep(i, b.size()) a[i] = padd(a[i], b[i]); // add
        trim();
        return *this;
    }
    Poly &operator-=(const Poly &rhs) {
        auto &b = rhs.a;
        a.resize(max(a.size(), b.size()));
        rep(i, b.size()) a[i] = psub(a[i], b[i]); // sub
        trim();
        return *this;
    }
    Poly &operator*=(const Poly &rhs) {
        int n = deg() + rhs.deg() + 1;
        if (n <= 0) return *this = Poly();
        n = 1 << (n <= 1 ? 0 : 32 - __builtin_clz(n - 1));
        vector<T> b = rhs.a;
        a.resize(n), b.resize(n);
        ntt(a, false), ntt(b, false);         // fft
        rep(i, a.size()) a[i] = pmul(a[i], b[i]); // mul
        ntt(a, true), trim();                 // invfft
        return *this;
    }
    Poly inv(int n) const {
        assert(deg() >= 0);
        Poly ans = pinv(a[0]); // inverse
        int b = 1;
        while (b < n) {
            Poly C = (ans * trunc(2 * b)).sub(b, 2 * b);
            ans -= (ans * C).trunc(b).shl(b);
            b *= 2;
```

```
    }
    return ans.trunc(n);
}

Poly operator+(const Poly &rhs) const { return Poly(*this) += rhs; }
Poly operator-(const Poly &rhs) const { return Poly(*this) -= rhs; }
Poly operator*(const Poly &rhs) const { return Poly(*this) *= rhs; }

pair<Poly, Poly> divmod(const Poly &b) const {
    if (deg() < b.deg()) return {Poly(), *this};
    int d = deg() - b.deg() + 1;
    Poly D = (rev(d) * b.rev(d).inv(d)).trunc(d).rev(d, true);
    return {D, *this - D * b};
}
Poly operator/(const Poly &b) const { return divmod(b).first; }
Poly operator%(const Poly &b) const { return divmod(b).second; }
Poly &operator/=(const Poly &b) { return *this = divmod(b).first; }
Poly &operator%=(const Poly &b) { return *this = divmod(b).second; }

T eval(T x) {
    T y = 0;
    invrep(i, a.size()) y = padd(pmul(y, x), a[i]); // add, mul
    return y;
}
Poly &build(vector<Poly> &tree, vector<T> &x, int v, int l, int r) {
    if (l == r) return tree[v] = vector<T>{-x[l], 1};
    int m = (l + r) / 2;
    return tree[v] = build(tree, x, 2 * v, l, m) *
                     build(tree, x, 2 * v + 1, m + 1, r);
}
void subeval(vector<Poly> &tree, vector<T> &x, vector<T> &y, int v,
     int l,
             int r) {
    if (l == r) {
        y[l] = eval(x[l]);
        return;
    }
    int m = (l + r) / 2;
    (*this % tree[2 * v]).subeval(tree, x, y, 2 * v, l, m);
    (*this % tree[2 * v + 1]).subeval(tree, x, y, 2 * v + 1, m + 1, r);
}
// evaluate m points in O(k (log k)^2) with k = max(n, m).
vector<T> multieval(vector<T> &x) {
    int N = x.size();
    if (deg() < 0) return vector<T>(N, 0);
        vector<Poly> tree(4 * N);
        build(tree, x, 1, 0, N - 1);
        vector<T> y(N);
        subeval(tree, x, y, 1, 0, N - 1);
        return y;
    }

    friend ostream &operator<<(ostream &s, const Poly &p) {
        s << "(";
        bool first = true;
        rep(i, p.a.size()) {
            if (p.a[i] == 0) continue;
            if (!first) s << " + ";
            s << p.a[i];
            if (i > 0) s << " x";
            if (i > 1) s << "^" << i;
            first = false;
        }
        s << ")";
        return s;
    }
};

#ifndef NOMAIN_POLY

int main() {
    Poly p1({1, 4});
    Poly p2({-3, 2});
    Poly p3({12, 12, 12, 1});
    Poly p4({128, 40, 29, 2, 0});

    cout << p1 << " * " << p2 << " = " << p1 * p2 << endl;

    vector<ll> xs = {-4, -3, -2, -1, 0, 1, 2, 3, 4};
    for (ll &x : xs) x = (x % MOD + MOD) % MOD;
    vector<ll> ys = p2.multieval(xs);
    cout << "P(x) = " << p2 << endl;
    cout << "x -> P(x):" << endl;
    rep(i, xs.size()) { cout << " " << xs[i] << " -> " << ys[i] << endl; }

    cerr << endl;
    find_root_of_unity(7340033);

    cerr << endl;
    find_root_of_unity(998244353);
```

```
}

#endif
```

6.9 primes

```
#include "../common.h"

// counts the divisors of a positive integer in O(sqrt(n))
ll count_divisors(ll x) {
    ll divs = 1, i = 2;
    for (ll divs = 1, i = 2; x > 1; i++) {
        if (i * i > x) {
            divs *= 2;
            break;
        }
        for (ll d = divs; x % i == 0; x /= i) divs += d;
    }
    return divs;
}

// gets the prime factorization of a number in O(sqrt(n))
vector<pair<ll, int>> factorize(ll x) {
    vector<pair<ll, int>> f;
    for (ll k = 2; x > 1; k++) {
        if (k * k > x) {
            f.push_back({x, 1});
            break;
        }
        int n = 0;
        while (x % k == 0) x /= k, n++;
        if (n > 0) f.push_back({k, n});
    }
    return f;
}

// iterate over all divisors of a number.
//
// divisor count upper bound: n^(1.07 / ln ln n)
template <class OP>
void divisors(ll x, OP op) {
    auto facts = factorize(x);
    vector<int> f(facts.size());
    while (true) {
        ll y = 1;
        rep(i, f.size()) rep(j, f[i]) y *= facts[i].first;
        op(y);

        int i;
        for (i = 0; i < f.size(); i++) {
            f[i] += 1;
            if (f[i] <= facts[i].second) break;
        }
        if (i == f.size()) break;
    }
}

// computes euler totative function phi(x), counting the amount of
    integers in
// [1, x] that are coprime with x.
//
// time: O(sqrt(x))
ll phi(ll x) {
    ll phi = 1, k = 2;
    for (; x > 1; k++) {
        if (k * k > x) {
            phi *= x - 1;
            break;
        }
        ll k1 = 1, k0 = 0;
        while (x % k == 0) x /= k, k0 = k1, k1 *= k;
        phi *= k1 - k0;
    }
    return phi;
}

// computes primality up to N.
// considers 0 and 1 prime.
// O(N log N)
void sieve(int N, vector<bool> &prime) {
    prime.assign(N + 1, true);
    repx(n, 2, N + 1) if (prime[n]) for (int k = 2 * n; k <= N; k += n)
        prime[k] = false;
}
```

6.10 segment

```
#include "../common.h"

// in-place segment intersection.
void intersect(pair<int, int>& a, pair<int, int> b) {
    a = {max(a.first, b.first), min(a.second, b.second)};
}

// in-place segment "union".
// finds the shortest segment that contains both `a` and `b`.
//
// for [a, b) segments: change > to >= and <= to <
void unite(pair<int, int>& a, pair<int, int> b) {
    if (a.first > a.second)
        a = b;
    else if (b.first <= b.second)
        a = {min(a.first, b.first), max(a.second, b.second)};
}

// segment containment.
//
// [a, b] in [c, d]
// subset or equal: a >= c && b <= d || a > b
// proper subset: a >  c && b < d || a > b && c <= d
//
// [a, b) in [c, d)
// subset or equal: a >= c && b <= d || a >= b
// proper subset: a >  c && b < d || a >= b && c < d
bool is_subset(pair<int, int> sub, pair<int, int> sup) {
    return sub.first >= sup.first && sub.second <= sup.second ||
           sub.second < sub.first;
}
bool is_subset_proper(pair<int, int> sub, pair<int, int> sup) {
    return sub.first > sup.first && sub.second < sup.second ||
           sub.second < sub.first && sup.first <= sup.second;
}
```

6.11 theorems

```
// Burnside lemma
//
//    For a set X, with members x in X, and a group G, with operations g
    in G, where g(x): X -> X.
//    F_g is the set of x which are fixed points of g (ie. { x in X /
    g(x) = x }).
//    The number of orbits (connected components in the graph formed by
    assigning each x a node and
//    a directed edge between x and g(x) for every g) is called M.
//    M = the average of the fixed points of all g = (|F_g1| + |F_g2| +
    ... + |F_gn|) / |G|
//
//    If x are images and g are simmetries, then M corresponds to the
    amount of objects, |G|
//    corresponds to the amount of simmetries, and F_g corresponds to
    the amount of simmetrical
//    images under the simmetry g.
//
// Rational root theorem
//
//    All rational roots of the polynomials with integer coefficients:
//
//    a0 * x^0 + a1 * x^1 + a2 * x^2 + ... + an * x^n = 0
//
//    If these roots are represented as p / q, with p and q coprime,
//    - p is an integer factor of a0
//    - q is an integer factor of an
//
//    Note that if a0 = 0, then x = 0 is a root, the polynomial can be
    divided by x and the theorem
//    applies once again.
//
// Legendre's formula
//
//    Considering a prime p, the largest power p^k that divides n! is
    given by:
//
//    k = floor(n/p) + floor(n/p^2) + floor(n/p^3) + ...
//
//    Which can be computed in O(log n / log p) time
```

7 r

```
#!/bin/bash

OK=0
```

```
if [ "$1" -nt tmp ] || [ "$2" = "f" ]
then
    echo "compiling..." >&2
    g++ -g -std=c++17 -o tmp "$1"
    OK=$?
fi

ulimit -s 1000000
(exit $OK) && echo "running..." >&2 && ./tmp
```

8 strings

8.1 hash

```
#include "../common.h"

// compute substring hashes in O(1).
// hashes are compatible between different strings.
struct Hash {
    ll HMOD;
    int N;
    vector<int> h;
    vector<int> p;

    Hash() {}
    // O(N)
    Hash(const string& s, ll HMOD_ = 1000003931)
        : N(s.size() + 1), HMOD(HMOD_), p(N), h(N) {
        static const ll P =
            chrono::steady_clock::now().time_since_epoch().count() % (1 <<
                29);
        p[0] = 1;
        rep(i, N - 1) p[i + 1] = p[i] * P % HMOD;
        rep(i, N - 1) h[i + 1] = (h[i] + (ll)s[i] * p[i]) % HMOD;
    }

    // O(1)
    pair<ll, int> get(int i, int j) { return {(h[j] - h[i] + HMOD) %
        HMOD, i}; }

    bool cmp(pair<ll, int> x0, pair<ll, int> x1) {
        int d = x0.second - x1.second;
        ll& lo = d < 0 ? x0.first : x1.first;
        lo = lo * p[abs(d)] % HMOD;
        return x0.first == x1.first;
    }
};

// compute hashes in multiple prime modulos simultaneously, to reduce the
    chance
// of collisions.
struct HashM {
    int N;
    vector<Hash> sub;

    HashM() {}
    // O(K N)
    HashM(const string& s, const vector<ll>& mods) : N(mods.size()),
        sub(N) {
        rep(i, N) sub[i] = Hash(s, mods[i]);
    }

    // O(K)
    vector<pair<ll, int>> get(int i, int j) {
        vector<pair<ll, int>> hs(N);
        rep(k, N) hs[k] = sub[k].get(i, j);
        return hs;
    }

    bool cmp(const vector<pair<ll, int>>& x0, const vector<pair<ll,
        int>>& x1) {
        rep(i, N) if (!sub[i].cmp(x0[i], x1[i])) return false;
        return true;
    }

    bool cmp(int i0, int j0, int i1, int j1) {
        rep(i, N) if (!sub[i].cmp(sub[i].get(i0, j0),
                                  sub[i].get(i1, j1))) return false;
        return true;
    }
};

#ifndef NOMAIN_HASH

int main() {
```

```
    const vector<ll> HMOD = {1000001237, 1000003931};
    //  0123456789012345678901 2
    string s = "abracadabra abracadabra";
    HashM h(s, HMOD);
    rep(i0, s.size() + 1) repx(j0, i0, s.size() + 1) rep(i1, s.size() + 1)
        repx(j1, i1, s.size() + 1) {
        bool eq = h.cmp(h.get(i0, j0), h.get(i1, j1));
        bool eq2 = s.substr(i0, j0 - i0) == s.substr(i1, j1 - i1);
        if (eq != eq2) {
            cout << " hash says strings \"" << s.substr(i0, j0 - i0) <<
                "\" "
                << (eq ? "==" : "!=") << " \"" << s.substr(i1, j1 - i1)
                << "\" but in reality they are " << (eq2 ? "==" : "!=")
                << endl;
        }
    }
}

#endif
```

8.2 hash2d

```
#include "../common.h"

using Hash = pair<ll, int>;

struct Block {
    int x0, y0, x1, y1;
};

struct Hash2d {
    ll HMOD;
    int W, H;
    vector<int> h;
    vector<int> p;

    Hash2d() {}
    Hash2d(const string& s, int W_, int H_, ll HMOD_ = 1000003931)
        : W(W_ + 1), H(H_ + 1), HMOD(HMOD_) {
        static const ll P =
            chrono::steady_clock::now().time_since_epoch().count() % (1 <<
                29);
        p.resize(W * H);
        p[0] = 1;
        rep(i, W * H - 1) p[i + 1] = p[i] * P % HMOD;
        h.assign(W * H, 0);
        repx(y, 1, H) repx(x, 1, W) {
            ll c = (ll)s[(y - 1) * (W - 1) + x - 1] * p[y * W + x] % HMOD;
            h[y * W + x] = (HMOD + h[y * W + x - 1] + h[(y - 1) * W + x] -
                            h[(y - 1) * W + x - 1] + c) %
                           HMOD;
        }
    }

    bool isout(Block s) {
        return s.x0 < 0 || s.x0 >= W || s.x1 < 0 || s.x1 >= W || s.y0 < 0
            ||
               s.y0 >= H || s.y1 < 0 || s.y1 >= H;
    }

    Hash get(Block s) {
        return {(2 * HMOD + h[s.y1 * W + s.x1] - h[s.y1 * W + s.x0] -
                 h[s.y0 * W + s.x1] + h[s.y0 * W + s.x0]) %
                    HMOD,
                s.y0 * W + s.x0};
    }

    bool cmp(Hash x0, Hash x1) {
        int d = x0.second - x1.second;
        ll& lo = d < 0 ? x0.first : x1.first;
        lo = lo * p[abs(d)] % HMOD;
        return x0.first == x1.first;
    }
};

struct Hash2dM {
    int N;
    vector<Hash2d> sub;

    Hash2dM() {}
    Hash2dM(const string& s, int W, int H, const vector<ll>& mods)
        : N(mods.size()), sub(N) {
        rep(i, N) sub[i] = Hash2d(s, W, H, mods[i]);
    }

    bool isout(Block s) { return sub[0].isout(s); }
```

```
    vector<Hash> get(Block s) {
        vector<Hash> hs(N);
        rep(i, N) hs[i] = sub[i].get(s);
        return hs;
    }

    bool cmp(const vector<Hash>& x0, const vector<Hash>& x1) {
        rep(i, N) if (!sub[i].cmp(x0[i], x1[i])) return false;
        return true;
    }

    bool cmp(Block s0, Block s1) {
        rep(i, N) if (!sub[i].cmp(sub[i].get(s0), sub[i].get(s1))) return
            false;
        return true;
    }
};

#ifndef NOMAIN_HASH2D

const vector<ll> HMOD = {1000002649, 1000000933, 1000003787, 1000002173};

int main() {}

#endif
```

8.3 kmp

```
#include "common.h"

// compute the prefix function for string `s`:
// for every character substring [0 : i] (both inclusive), compute the
    longest proper prefix that
// is also a suffix.
// O(N)
//
// computing `prefunc` on a string of the type `wwww#ttttttt` will give
    for
// every `t` the amount of characters from `w` that match (ie. search for
    the
// string `w` inside the string `t`).
void prefunc(const string &s, vector<int> &p)
{
    int N = s.size(), j;
    p.resize(N), p[0] = 0;
    repx(i, 1, N)
    {
        for (j = p[i - 1]; j > 0 && s[j] != s[i];)
            j = p[j - 1];
        p[i] = j + (s[j] == s[i]);
    }
}
```

8.4 palin

```
#include "../common.h"

// find maximal palindromes (and therefore all palindromes) in O(n).
// returns a vector of positions, with one position for every character
    and in
// between characters.
//
// a   b   c   c   c
// 0 1 2 3 4 5 6 7 8
// 1 0 1 0 1 2 3 2 1
void manacher(const string& s, vector<int>& p) {
    int N = s.size(), P = 2 * N - 1;
    p.assign(P, 0);
    int l = 0, r = -1;
    rep(i, P) {
        int d = (r >= i ? min(p[l + r - i], r - i + 2) : i % 2);
        while (i - d >= 0 && i + d < P && s[(i - d) / 2] == s[(i + d) / 2])
            d += 2;
        p[i] = d;
        if (i + d - 2 > r) l = i - d + 2, r = i + d - 2;
    }
    rep(i, P) p[i] -= 1;
}

#ifndef NOMAIN_PALIN

void test(const string& s) {
    vector<int> p;
    manacher(s, p);
```

```
    cout << "palindromes of string \"" << s << "\":" << endl;
    rep(i, p.size()) {
        for (int k = i % 2; k < p[i]; k += 2) {
            cout << " \"" << s.substr((i - k) / 2, k + 1) << "\"" << endl;
        }
    }
}

int main() {
    test("hello");
    test("abracadabra");
    test("abcba");
    test("abba");
    test("cada");
}

#endif
```

8.5 sufarr

```
#include "../common.h"

// build the suffix array
// suffixes are sorted, with each suffix represented by its starting
    position
vector<int> suffixarray(const string& s) {
    int N = s.size() + 1; // optional: include terminating NUL
    vector<int> p(N), p2(N), c(N), c2(N), cnt(256);
    rep(i, N) cnt[s[i]] += 1;
    repx(b, 1, 256) cnt[b] += cnt[b - 1];
    rep(i, N) p[--cnt[s[i]]] = i;
    repx(i, 1, N) c[p[i]] = c[p[i - 1]] + (s[p[i]] != s[p[i - 1]]);
    for (int k = 1; k < N; k <<= 1) {
        int C = c[p[N - 1]] + 1;
        cnt.assign(C + 1, 0);
        for (int& pi : p) pi = (pi - k + N) % N;
        for (int cl : c) cnt[cl + 1] += 1;
        rep(i, C) cnt[i + 1] += cnt[i];
        rep(i, N) p2[cnt[c[p[i]]]++] = p[i];
        c2[p2[0]] = 0;
        repx(i, 1, N) c2[p2[i]] =
            c2[p2[i - 1]] + (c[p2[i]] != c[p2[i - 1]] ||
                             c[(p2[i] + k) % N] != c[(p2[i - 1] + k) % N]);
        swap(c, c2), swap(p, p2);
    }
    p.erase(p.begin()); // optional: erase terminating NUL
    return p;
}

// build the lcp
// `lcp[i]` represents the length of the longest common prefix between
    suffix i
// and suffix i+1 in the suffix array `p`. the last element of `lcp` is
    zero by
// convention
vector<int> makelcp(const string& s, const vector<int>& p) {
    int N = p.size(), k = 0;
    vector<int> r(N), lcp(N);
    rep(i, N) r[p[i]] = i;
    rep(i, N) {
        if (r[i] + 1 >= N) {
            k = 0;
            continue;
        }
        int j = p[r[i] + 1];
        while (i + k < N && j + k < N && s[i + k] == s[j + k]) k += 1;
        lcp[r[i]] = k;
        if (k) k -= 1;
    }
    return lcp;
}

#ifndef NOMAIN_SUFARR

void test(const string& s) {
    cout << "suffix array for string \"" << s << "\" (length " << s.size()
        << "):" << endl;
    vector<int> sa = suffixarray(s);
    vector<int> lcp = makelcp(s, sa);
    rep(i, sa.size()) {
        int j = sa[i];
        if (i > 0) cout << "  " << lcp[i - 1] << endl;
        cout << " \"" << s.substr(j) << "\"" << endl;
    }
}

int main() {
```

```
    test("hello");
    test("abracadabra");
}

#endif
```

9 template

```
#pragma GCC optimize("Ofast")
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;

#define repx(i, a, b) for (int i = a; i < b; i++)
#define rep(i, n) repx(i, 0, n)
#define invrepx(i, a, b) for (int i = b - 1; i >= a; i--)
#define invrep(i, n) invrepx(i, 0, n)

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    int TC;
    cin >> TC;
    while (TC--)
    {
    }
}
```