EZCP

Lund University

Misc

Template

```cpp
//#pragma GCC optimize("Ofast")
#include <bits/stdc++.h>
#define all(x) begin(x),end(x)
using namespace std;
using ll = long long;
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
}
```

Compilation Script

```
g++ --std=c++20 -Wall -Wshadow -Wno-conversion -Wfatal-errors -fsanitize=address,undefined -ftrapv -g $1 -o
    ↪ ${1%.cpp}.bin
```

Binary Search

```cpp
while (lo < hi) {
    ll mid = lo + (hi - lo) / 2;
    if (f(mid)) // f should be false, then true
        hi = mid;
    else
        lo = mid + 1;
}
//lo is now the first index where f is true
```

Ternary Search

Find the smallest i in $[a, b]$ that maximizes $f(i)$, assuming that $f(a) < \cdots < f(i) \geq \cdots \geq f(b)$.
If there is a range of $f(i) \ldots f(j)$ that are equal, change according to the comments to get the last index instead of the first.

```cpp
template<class F>
ll ternarySearch(ll a, ll b, F f) {
    assert(a <= b);
    while (b - a >= 5) {
        ll mid = (a + b) / 2;
        if (f(mid) < f(mid+1)) // <= for last index
            a = mid;
        else
            b = mid+1;
    }
    //for (ll i = b; i > a; i--) to get last index
    for (ll i = a + 1; i <= b; i++)
        if (f(a) < f(i))
            a = i;
    return a;
}
```

```python
def ternarySearch(a, b, f):
    assert a <= b
    while b - a >= 5:
        mid = (a + b) // 2
        if f(mid) < f(mid+1): # <= for last index
            a = mid
        else:
            b = mid + 1
    #for i in range(b, a-1, -1): to get last index
    for i in range(a+1, b+1):
        if f(a) < f(i):
            a = i
    return a
```

Geometry

Geometry Template (Python)

```python
def vecsub(a, b):
    return (a[0] - b[0], a[1] - b[1])
def vecadd(a, b):
    return (a[0] + b[0], a[1] + b[1])
def dot(a, b):
    return a[0] * b[0] + a[1] * b[1]
def cross(a, b, o = (0, 0)):
    return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])
def len2(a):
    return a[0] ** 2 + a[1] ** 2
def dist2(a, b):
    return len2(vecsub(a, b))
def sign(x):
    return (x > 0) - (x < 0)
def zero(x):
    return abs(x) < 1E-9
```

Geometry Template (C++)

```cpp
template <typename T> struct point {
    T x, y;
    point() { x=y=0; }
    point(T xx, T yy) : x(xx), y(yy) { }
    template <typename U> point(point<U> o) : x(o.x), y(o.y) { }
    point operator+(point o) const { return { x+o.x, y+o.y }; }
    point operator-(point o) const { return { x-o.x, y-o.y }; }
    point operator*(T o) const { return { x*o, y*o }; }
    point operator/(T o) const { return { x/o, y/o }; }
    bool operator==(point o) const { return x==o.x && y==o.y; };
    bool operator<(point o) const { return tie(x, y) < tie(o.x, o.y); };
    T dot(point o) const { return x*o.x + y*o.y; }
    T cross(point b) const { return x*b.y - y*b.x; }
    T cross(point b, point o) const { return (*this-o).cross(b-o); }
    T len2() const { return x*x + y*y; }
};
using ipoint = point<ll>;
using dpoint = point<double>;
ll sign(auto x) { return (x>0) - (x<0); }
bool zero(double x) { return abs(x) < 1E-9; }
```

Check if point is on a line segment

```python
def onSegment(s, e, p):
    # return zero(distPS(s, e, p))   if floating-point is OK
    return cross(s, e, p) == 0 and dot(vecsub(s, p), vecsub(e, p)) <= 0
```

```cpp
bool onSegment(ipoint s, ipoint e, ipoint p) { return s.cross(e, p) == 0 && (s - p).dot(e - p) <= 0; }
```

Distance between point and line segment

Returns the distance from the point p to the line segment starting at s and ending at e.

```cpp
double distPS(ipoint s, ipoint e, ipoint p) {
    if (s == e)
        return sqrt((p - s).len2());
    auto se = e - s;
    auto sp = p - s;
    ll d = se.len2();
    ll t = min(d, max(0LL, (p - s).dot(e - s)));
    return sqrt((sp * d - se * t).len2()) / d;
}
```

```python
def distPS(s, e, p):
    if s == e:
        return sqrt(dist2(p, s))
    se, sp = vecsub(e, s), vecsub(p, s)
    d = len2(se)
    t = min(d, max(0, dot(vecsub(p,s), vecsub(e,s))))
    return sqrt(dist2(
        (sp[0]*d, sp[1]*d),
        (se[0]*t, se[1]*t)
    )) / d
```

Distance between point and line

Returns the signed distance from the point p to the line passing through the points a and b.

```cpp
double distPL(ipoint a, ipoint b, ipoint p) {
    return b.cross(p, a) / sqrt((a - b).len2());
}
```

```python
def distPL(a, b, p):
    return cross(b, p, a) / sqrt(dist2(a, b))
```

Intersection between two lines

If a unique intersection point of the lines going through `s1,e1` and `s2,e2` exists `(1,point)` is returned.
If no intersection point exists `(0,(0,0))` is returned and if infinitely many exist `(-1,(0,0))` is returned.

```cpp
pair<int, dpoint> intersectLL(ipoint s1, ipoint e1, ipoint s2, ipoint e2) {
    auto d = (e1 - s1).cross(e2 - s2);
    if (zero(d)) # parallel
        return { -int(zero(e1.cross(s2, s1))), {} };
    auto p = e1.cross(e2, s2);
    auto q = e2.cross(s1, s2);
    return { 1, dpoint(
        (s1.x * p + e1.x * q),
        (s1.y * p + e1.y * q)
    ) / d };
}
```

```python
def intersectLL(s1, e1, s2, e2):
    d = cross(vecsub(e1, s1), vecsub(e2, s2))
    if zero(d): # parallel
        return (-zero(cross(e1, s2, s1)), (0, 0))
    p, q = cross(e1, e2, s2), cross(e2, s1, s2)
    return (1,(
        (s1[0] * p + e1[0] * q) / d,
        (s1[1] * p + e1[1] * q) / d
    ))
```

Intersection between two line segments

If a unique intersection is found, returns a list with only this point. If the segments intersect in many points, returns a list of 2 elements containing the start and end of the common line segment. If no intersection, returns an empty list

```cpp
vector<dpoint> intersectSS(ipoint s1, ipoint e1, ipoint s2, ipoint e2) {
    auto oa = e2.cross(s1, s2);
    auto ob = e2.cross(e1, s2);
    auto oc = e1.cross(s2, s1);
    auto od = e1.cross(e2, s1);
    if (sign(oa)*sign(ob)<0 && sign(oc)*sign(od)<0) {
        return { dpoint(s1.x * ob - e1.x * oa,
                        s1.y * ob - e1.y * oa)
                 / double(ob - oa) };
    }
    set<ipoint> s;
    if (onSegment(s2, e2, s1)) s.insert(s1);
    if (onSegment(s2, e2, e1)) s.insert(e1);
    if (onSegment(s1, e1, s2)) s.insert(s2);
    if (onSegment(s1, e1, e2)) s.insert(e2);
    return {all(s)};
}
```

```python
def intersectSS(s1, e1, s2, e2):
    oa = cross(e2, s1, s2)
    ob = cross(e2, e1, s2)
    oc = cross(e1, s2, s1)
    od = cross(e1, e2, s1)
    if sign(oa)*sign(ob)<0 and sign(oc)*sign(od)<0:
        div = ob - oa
        return [(
            (s1[0] * ob - e1[0] * oa) / div,
            (s1[1] * ob - e1[1] * oa) / div
        )]
    s = set()
    if onSegment(s2, e2, s1): s.add(s1)
    if onSegment(s2, e2, e1): s.add(e1)
    if onSegment(s1, e1, s2): s.add(s2)
    if onSegment(s1, e1, e2): s.add(e2)
    return list(s)
```

Polygon area

Returns twice the signed area of a polygon. Clockwise enumeration gives negative area.

```cpp
ll polygonArea2(const vector<ipoint>& v) {
    ll a = 0;
    for (ll i = 0; i < (ll)v.size(); i++)
        a += v[i].cross(v[(i+1) % v.size()]);
    return a;
}
```

```python
def polygonArea2(v):
    return sum(map(lambda i: cross(v[i - 1], v[i]), range(len(v))))
```

Project point to line (or reflect)

Projects the point p onto the line passing through a and b. Set `refl=True` to get reflection of point p across the line instead.

```cpp
dpoint projPL(ipoint a, ipoint b, ipoint p, bool refl = false) {
    auto v = b - a;
    double s = (1 + refl) * b.cross(p, a) / v.len2();
    return { p.x + v.y * s, p.y - v.x * s };
}
```

```python
def projPL(a, b, p, refl = False):
    v = vecsub(b, a)
    s = (1 + refl) * cross(b, p, a) / len2(v)
    return (p[0] + v[1] * s, p[1] - v[0] * s)
```

Point inside polygon

Returns true if the point `pt` lies within the polygon `poly`. If strict is true, returns false for points on the boundary.

```python
def pointInPolygon(poly, pt, strict = True):
    c = False
    for i in range(len(poly)):
        q = poly[i - 1]
        if onSegment(q, poly[i], pt):
            return not strict
        c ^= ((pt[1] < q[1]) - (pt[1] < poly[i][1])) * cross(q, poly[i], pt) > 0
    return c
```

```cpp
bool pointInPolygon(const vector<ipoint>& poly, ipoint p, bool strict = true) {
    bool c = false;
    auto prev = poly.back();
    for (auto cur : poly) {
        if (onSegment(prev, cur, p)) return !strict;
        c ^= ((p.y < prev.y) - (p.y < cur.y)) * prev.cross(cur, p) > 0;
        prev = cur;
    }
    return c;
}
```

Circumcircle

Returns a circle from three points.

```cpp
pair<dpoint, double> circumcirle(array<dpoint,3> p) {
    auto b = p[2] - p[0], c = p[1] - p[0];
    double bc = b.cross(c);
    assert(!zero(bc)); // collinear
    auto x = b * c.len2() - c * b.len2();
    dpoint rc = dpoint(-x.y, x.x) / bc;
    return { p[0] + rc, sqrt(rc.len2()) };
}
```

```python
def circumcirle(p1, p2, p3):
    b, c = vecsub(p3, p1), vecsub(p2, p1)
    bc, c2, b2 = cross(b, c) * 2, len2(c), len2(b)
    assert not zero(bc) # collinear
    rc = ((c[1]*b2 - b[1]*c2)/bc,
          (b[0]*c2 - c[0]*b2)/bc)
    return (vecadd(p1,rc), sqrt(len2(rc)))
```

Intersection between two circles

Computes the pair of points at which two circles intersect. Returns `None` in case of no intersection.

```cpp
template <typename T> optional<array<dpoint, 2>>
intersectCC(point<T> c1, point<T> c2, T r1, T r2) {
    if (c1 == c2) {
        assert(r1 != r2);
        return { };
    }
    auto vec = c2 - c1;
    T d2 = vec.len2(), sm = r1 + r2, dif = r1 - r2;
    if (sm * sm < d2 || dif * dif > d2)
        return { };
    double p = double(d2 + r1*r1 - r2*r2) / (d2 * 2);
    dpoint mid(c1.x + vec.x * p, c1.y + vec.y * p);
    dpoint per = dpoint(-vec.y, vec.x) * sqrt(max(0.0, r1 * r1 - p * p * d2) / d2);
    return { { mid+per, mid-per } };
}
```

```python
def intersectCC(c1, c2, r1, r2):
    if c1 == c2:
        assert(r1 != r2)
        return None
    vec = vecsub(c2, c1)
    d2 = len2(vec)
    if (r1 + r2) ** 2 < d2 or (r1 - r2) ** 2 > d2:
        return None
    p = (d2 + r1 ** 2 - r2 ** 2) / (d2 * 2)
    h2 = r1 ** 2 - p * p * d2
    mid = (c1[0] + vec[0] * p, c1[1] + vec[1] * p)
    plen = sqrt(max(0, h2) / d2)
    per = (-vec[1] * plen, vec[0] * plen)
    return (vecadd(mid, per), vecsub(mid, per))
```

Intersection between circle and line

Computes the intersection between a circle and a line. Returns a list of either 0, 1, or 2 intersection points.

```cpp
vector<dpoint> intersectCL(dpoint c, double r, dpoint a, dpoint b) {
    dpoint ab = b - a;
    dpoint p = a + ab * (c-a).dot(ab) / ab.len2();
    double s = a.cross(b, c);
    double h2 = r*r - s*s / ab.len2();
    if (h2 < 0) return {};
    if (h2 == 0) return {p};
    dpoint h = ab * sqrt(h2 / ab.len2());
    return {p - h, p + h};
}
```

```python
def intersectCL(c, r, a, b):
    ab = vecsub(b, a)
    ps = dot(vecsub(c, a), ab) / len2(ab)
    p = (a[0] + ab[0] * ps, a[1] + ab[1] * ps)
    h2 = r ** 2 - cross(a, b, c) ** 2 / len2(ab)
    if h2 < 0: return []
    if h2 == 0: return [p]
    h2 = sqrt(h2 / ab.len2())
    h = (ab[0] * h2, ab[1] * h2);
    return [vecsub(p, h), vecadd(p, h)]
```

Circle tangents

Finds the external tangents of two circles, or internal if r2 is negated.
Can return 0, 1, or 2 tangents – 0 if one circle contains the other (or overlaps it, in the internal case, or if the circles are the same); 1 if the circles are tangent to each other (in which case .first = .second and the tangent line is perpendicular to the line between the centers). .first and .second give the tangency points at circle 1 and 2 respectively. To find the tangents of a circle with a point set r2 to 0.

```cpp
vector<pair<dpoint, dpoint>> circleTangents(dpoint c1, double r1, dpoint c2, double r2) {
    dpoint d = c2 - c1, dp = {-d.y, d.x};
    double dr = r1 - r2, d2 = d.len2();
    double h2 = d2 - dr * dr;
    if (d2 == 0 || h2 < 0)  return {};
    vector<pair<dpoint, dpoint>> out;
    for (double s : {-1, 1}) {
        dpoint v = (d * dr + dp * sqrt(h2) * s) / d2;
        out.push_back({c1 + v * r1, c2 + v * r2});
    }
    if (h2 == 0) out.pop_back();
    return out;
}
```

```python
def circleTangents(c1, r1, c2, r2):
    d = vecsub(c2, c1)
    dr, d2 = r1 - r2, len2(d)
    h2, s1, s2 = d2 - dr**2, r1 / d2, r2 / d2
    if d2 == 0 or h2 < 0: return []
    out = []
    for s in [-1, 1]:
        vx = d[0] * dr - d[1] * sqrt(h2) * s
        vy = d[1] * dr + d[0] * sqrt(h2) * s
        out.append((vecadd(c1, (vx*s1, vy*s1)),
                    vecadd(c2, (vx*s2, vy*s2))))
    if h2 == 0: out.pop()
    return out
```

Circle-polygon area

Returns the area of the intersection of a circle with a counter-clockwise polygon. Time complexity: $\mathcal{O}(n)$

```cpp
double circlePolyArea(dpoint c, double r, vector<dpoint> ps) {
    auto arg = [&](dpoint p, dpoint q) { return atan2(p.cross(q), p.dot(q)); };
    auto tri = [&](dpoint p, dpoint q) {
        double r2 = r * r / 2;
        dpoint d = q - p;
        double a = d.dot(p)/d.len2(), b = (p.len2()-r*r)/d.len2();
        double det = a * a - b;
        if (det <= 0) return arg(p, q) * r2;
        auto s = max(0., -a-sqrt(det)), t = min(1., -a+sqrt(det));
```

```
        if (t < 0 || 1 <= s) return arg(p, q) * r2;
        dpoint u = p + d * s, v = p + d * t;
        return arg(p,u) * r2 + u.cross(v)/2 + arg(v,q) * r2;
    };
    double sum = 0.0;
    for (size_t i = 0; i < ps.size(); i++)
        sum += tri(ps[i] - c, ps[(i + 1) % ps.size()] - c);
    return sum;
}
```

Convex hull

Returns a list of points on the convex hull in counter-clockwise order. Points on the edge of the hull between two other points are not considered part of the hull. Time complexity: $\mathcal{O}(n \log n)$

```
auto convexHull(vector<ipoint> pts) {
    if (pts.size() <= 1) return pts;
    sort(all(pts));
    decltype(pts) h;
    auto f = [&] (ll s) {
        for (auto p : pts) {
            while ((ll)h.size() >= s + 2 &&
    ↪ h.back().cross(p, h[h.size() - 2]) <= 0)
                h.pop_back();
            h.push_back(p);
        }
        h.pop_back();
    };
    f(0);
    reverse(all(pts));
    f(h.size());
    if (h.size() == 2 && h[0] == h[1]) h.pop_back();
    return h;
}
```

```
def convexHull(pts):
    if len(pts) <= 1:
        return pts
    pts.sort()
    t, s, h = 0, 0, [0] * (len(pts) + 1)
    for i in range(2):
        for p in pts:
            while t >= s + 2 and cross(h[t - 1], p,
    ↪ h[t - 2]) <= 0:
                t -= 1
            h[t], t = p, t + 1
        s = t = t - 1
        pts.reverse()
    return h[:t - (t == 2 and h[0] == h[1])]
```

Area of union of polygons

Calculates the area of the union of multiple polygons (not necessarily convex). The points within each polygon must be given in CCW order. Time complexity: $\mathcal{O}(n^2)$, where N is the total number of points.

```
double rat(dpoint a, dpoint b) { return sign(b.x) ? a.x/b.x : a.y/b.y; }
double polyUnion(vector<vector<dpoint>>& poly) {
    double ret = 0;
    for (size_t i = 0; i < poly.size(); i++)
        for (size_t v = 0; v < poly[i].size(); v++) {
            dpoint A = poly[i][v], B = poly[i][(v + 1) % poly[i].size()];
            vector<pair<double, int>> segs = {{0, 0}, {1, 0}};
            for (size_t j = 0; j < poly.size(); j++) if (i != j) {
                for (size_t u = 0; u < poly[j].size(); u++) {
                    dpoint C = poly[j][u], D = poly[j][(u + 1) % poly[j].size()];
                    int sc = sign(B.cross(C, A)), sd = sign(B.cross(D, A));
                    if (sc != sd) {
                        double sa = C.cross(D, A), sb = C.cross(D, B);
                        if (min(sc, sd) < 0)
                            segs.emplace_back(sa / (sa - sb), sign(sc - sd));
                    } else if (!sc && !sd && j<i && sign((B-A).dot(D-C))>0){
                        segs.emplace_back(rat(C - A, B - A), 1);
                        segs.emplace_back(rat(D - A, B - A), -1);
                    }
                }
            }
            sort(all(segs));
            for (auto& s : segs) s.first = min(max(s.first, 0.0), 1.0);
            double sum = 0;
            int cnt = segs[0].second;
            for (ll j = 1; j < (ll)segs.size(); j++) {
                if (!cnt) sum += segs[j].first - segs[j - 1].first;
                cnt += segs[j].second;
            }
            ret += A.cross(B) * sum;
        }
    return ret / 2;
}
```

Angle struct

Struct for representing angles using integer points and sorting them.

```
struct Angle {
    ll x, y, t;
    Angle(ll X, ll Y, ll T=0) : x(X), y(Y), t(T) {}
    Angle operator-(Angle b) const { return {x-b.x, y-b.y, t}; }
    ll half() const {
        assert(x || y);
        return y < 0 || (y == 0 && x < 0);
    }
    Angle t90() const { return {-y, x, t + (half() && x >= 0)}; }
    Angle t180() const { return {-x, -y, t + half()}; }
    Angle t360() const { return {x, y, t + 1}; }
};
bool operator<(Angle a, Angle b) {
    // add a.dist2() and b.dist2() to the tuples to also compare distances
    return make_tuple(a.t, a.half(), a.y * (ll)b.x) < make_tuple(b.t, b.half(), a.x * (ll)b.y);
}
// Given two points (a.x, a.y) and (b.x, b.y), calculates the smallest angle between them.
pair<Angle, Angle> segmentAngles(Angle a, Angle b) {
    if (b < a) swap(a, b);
    return (b < a.t180() ? make_pair(a, b) : make_pair(b, a.t360()));
}
Angle operator+(Angle a, Angle b) {
    Angle r(a.x + b.x, a.y + b.y, a.t);
    if (a.t180() < r) r.t--;
    return r.t180() < a ? r.t360() : r;
}
Angle angleDiff(Angle a, Angle b) {
    ll tu = b.t - a.t; a.t = b.t;
    return {a.x*b.x + a.y*b.y, a.x*b.y - a.y*b.x, tu - (b < a)};
}
```

Half-plane intersection

Computes the intersection of a set of half-planes, each on the form: $xP_x + yP_y \geq P_c$.

The resulting polygon extending to infinity is not handled. If that is possible add some extra half-planes far away and check for those in the result.

Each half-plane is given as a pair where the first value is (P_x, P_y) and the second value is P_c. In the returned vector, ret[i].first is the position of the i:th vertex and ret[i].second is the index in the input vector of the half-plane that created the edge between vertex i and (i+1)%ret.size().

There may be duplicate points in the output vector if there are multiple half-planes that intersect at the same point.

```
using lll = __int128_t;
// For double:
using P = dpoint;
double fraction(double e, double d) { return e / d; }
const double inf = INFINITY, neginf = -INFINITY;
// For fractions:
using P = point<fraction>;
const fraction inf = fraction(1, 0), neginf = fraction(-1, 0);

vector<pair<P, ll>> halfPlaneIntersection(const vector<pair<ipoint, ll>>& pts) {
    vector<ll> planesL, planesU;
    pair<double, ll> maxvx = {inf, -1}, minvx = {neginf, -1};
    for (ll i = 0; i < (ll)pts.size(); i++) {
        auto [n, c] = pts[i];
        if (n.y > 0) planesL.push_back(i);
        else if (n.y < 0) planesU.push_back(i);
        else {
            auto x = fraction(c, n.x);
            if (n.x < 0) maxvx = min(maxvx, make_pair(x, i));
            else if (n.x > 0) minvx = max(minvx, make_pair(x, i));
            else if (c > 0) return {};
        }
    }
    if (maxvx.first < minvx.first) return {};
    if (minvx.second != -1) planesU.emplace_back(minvx.second);
    if (maxvx.second != -1) planesL.emplace_back(maxvx.second);
    auto intersect = [&] (ll a, ll b) -> optional<P> {
```

```
        auto [an, ac] = pts[a]; auto [bn, bc] = pts[b];
        if (ll cr = bn.cross(an))
            return P(fraction((lll)bc*an.y - (lll)ac*bn.y, cr), fraction((lll)bc*an.x - (lll)ac*bn.x, -cr));
        return nullopt;
    };
    auto hull = [&] (vector<ll>& planes, bool rev) {
        sort(all(planes), [&] (ll a, ll b) {
            auto [ap, ac] = pts[a]; auto [bp, bc] = pts[b];
            return make_tuple(ap.x*bp.y, (lll)ac*(lll)bp.y) < make_tuple(bp.x * ap.y, (lll)bc*(lll)ap.y);
        });
        if (rev) reverse(all(planes));
        vector<pair<ll, P>> st;
        for (ll pi : planes) { start:
            if (st.empty())
                st.emplace_back(pi, P(neginf, 0));
            else if (auto i = intersect(pi, st.back().first))
                if (st.back().second.x <= i->x) { st.emplace_back(pi, *i); }
                else { st.pop_back(); goto start; }
        }
        return st;
    };
    auto stL = hull(planesL, true);
    auto stU = hull(planesU, false);
    assert(!stL.empty() && !stU.empty()); // otherwise infinite result
    optional<P> intersectL, intersectR;
    ll ril = stL.size() - 1, riu = stU.size() - 1, lil = 0, liu = 0;
    stL.emplace_back(-1, P(inf, 0));
    stU.emplace_back(-1, P(inf, 0));
    while (ril >= 0 && riu >= 0) {
        auto i = intersect(stL[ril].first, stU[riu].first);
        if (!i) break;
        if (stL[ril].second <= *i && stU[riu].second <= *i) {
            if (*i <= stL[ril + 1].second && *i <= stU[riu + 1].second)
                intersectR = i;
            break;
        } else
            (stL[ril].second.x < stU[riu].second.x ? riu : ril)--;
    }
    while (lil <= ril && liu <= riu) {
        auto i = intersect(stL[lil].first, stU[liu].first);
        if (!i) break;
        if (*i <= stL[lil + 1].second && *i <= stU[liu + 1].second) {
            intersectL = i;
            break;
        } else
            (stL[lil + 1].second.x < stU[liu + 1].second.x ? lil : liu)++;
    }
    if (!intersectR || !intersectL) return {};
    vector<pair<P, ll>> result;
    for (ll i = riu; i > liu; i--)
        result.emplace_back(stU[i].second, stU[i - 1].first);
    result.emplace_back(*intersectL, stL[lil].first);
    for (ll i = lil + 1; i <= ril; i++)
        result.emplace_back(stL[i].second, stL[i].first);
    result.emplace_back(*intersectR, stU[riu].first);
    return result;
}
```

Data Structures

Fenwick Tree

```cpp
struct FenwickTree {
    FenwickTree(ll n) : v(n + 1, 0) { }
    ll lsb(ll x) { return x & (-x); }
    ll prefixSum(ll n) { //sum of the first n items (nth not included)
        ll sum = 0;
        for (; n; n -= lsb(n))
            sum += v[n];
        return sum;
    }
    void adjust(ll i, ll delta) {
        for (i++; i < v.size(); i += lsb(i))
            v[i] += delta;
    }
    vector<ll> v;
};
```

Segment Tree

```cpp
struct SegTree {
    using T = ll;
    T f(T a, T b) { return a + b; }
    static constexpr T UNIT = 0;//neutral value for f

    vector<T> s; ll n;
    SegTree(ll len) : s(2 * len, UNIT), n(len) {}
    void set(ll pos, T val) {
        for (s[pos += n] = val; pos /= 2;)
            s[pos] = f(s[pos * 2], s[pos * 2 + 1]);
    }
    T query(ll lo, ll hi) { // hi not included
        T ra = UNIT, rb = UNIT;
        for (lo+=n, hi+=n; lo < hi; lo/=2, hi/=2) {
            if (lo % 2) ra = f(ra, s[lo++]);
            if (hi % 2) rb = f(s[--hi], rb);
        }
        return f(ra, rb);
    }
};
```

```python
class SegTree:
    def f(a, b):
        return a + b
    UNIT = 0 # neutral value for f

    def __init__(self, n):
        self.s = [self.UNIT] * (2 * n)
        self.n = n
    def set(self, pos, val):
        pos += self.n
        self.s[pos] = val
        while pos > 1:
            pos //= 2
            self.s[pos] = SegTree.f(self.s[pos * 2], self.s[pos * 2 + 1])
    def query(self, lo, hi): # hi not included
        ra, rb = self.UNIT, self.UNIT
        lo, hi = lo + self.n, hi + self.n
        while lo < hi:
            if lo % 2:
                ra = SegTree.f(ra, self.s[lo])
                lo += 1
            if hi % 2:
                hi -= 1
                rb = SegTree.f(self.s[hi], rb)
```

Sparse Table

```cpp
struct SparseTable {
    using T = ll;
    T f(T a, T b) { return min(a, b); }
    ll node(ll l, ll i) { return i + l * n; }
    ll n; vector<T> v;
    SparseTable(vector<T> values) : n(values.size()), v(move(values)) {
        ll d = log2(n);
        v.resize((d + 1) * n);
        for (ll L = 0, s = 1; L < d; L++, s *= 2) {
            for (ll i = 0; i < n; i++) {
                v[node(L + 1, i)] = f(v[node(L, i)], v[node(L, min(i + s, n - 1))]);
            }
        }
    }
    T query(ll lo, ll hi) { assert(hi > lo);
        ll l = (ll)log2(hi - lo);
        return f(v[node(l, lo)], v[node(l, hi - (1 << l))]);
    }
};
```

Line Container

Container where you can add lines of the form $kx + m$, and query maximum values at points x. All operations are $\mathcal{O}(\log(n))$. For doubles, use `inf = 1/.0` and `div(a,b) = a/b`

```
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<>> {
const ll inf = LLONG_MAX;
ll div(ll a, ll b) { // floored division
    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;
}
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;
}
};
```

Lazy Segment Tree

Segment tree with support for range updates. Use T = pair of value and index to get index from queries.
All ranges are `(lo, hi]` (hi is not included). `fQuery` defines the function to be used for queries (currently min) and `fUpdate` defines the function to be used for updates (currently addition).

```
struct LazyST {
    using T = ll;
    T f(T a, T b) { return min(a, b); }
    static const T QUERY_UNIT = LLONG_MAX; // neutral value for f

    struct Node {
        T val = QUERY_UNIT; // current value of this segment
        optional<T> p; // value being pushed down into this segment
    };
    int len; vector<Node> nodes;
    LazyST(int l) : len(pow(2, ceil(log2(l)))), nodes(len * 2) { }
    void update(int lo, int hi, T val) { u(lo, hi, val, 1, 0, len); }
    T query(int lo, int hi) { return q(lo, hi, 1, 0, len); }
private:
    #define LST_NEXT int l = n * 2; int r = l + 1; int mid = (nlo + nhi) / 2
    void push(int n, int nlo, int nhi) {
        if (!nodes[n].p) return;
        LST_NEXT;
        u(nlo, nhi, *nodes[n].p, l, nlo, mid);
        u(nlo, nhi, *nodes[n].p, r, mid, nhi);
        nodes[n].p = {};
    }
    void u(int qlo, int qhi, T val, int n, int nlo, int nhi) {
        if (nhi <= qlo || nlo >= qhi) return;
        if (nlo >= qlo && nhi <= qhi) {
            //for interval set:
            nodes[n].p = val;
            nodes[n].val = val; // val * (nhi - nlo) for sum queries
            //for interval add:
            nodes[n].p = nodes[n].p.value_or(0) + val;
            nodes[n].val += val; // val * (nhi - nlo) for sum queries
        } else {
            push(n, nlo, nhi); LST_NEXT;
            u(qlo, qhi, val, l, nlo, mid);
            u(qlo, qhi, val, r, mid, nhi);
            nodes[n].val = f(nodes[l].val, nodes[r].val);
        }
    }
    T q(int qlo, int qhi, int n, int nlo, int nhi) {
        if (nhi <= qlo || nlo >= qhi) return QUERY_UNIT;
        if (nlo >= qlo && nhi <= qhi) return nodes[n].val;
        push(n, nlo, nhi); LST_NEXT;
        return f(q(qlo, qhi, l, nlo, mid), q(qlo, qhi, r, mid, nhi));
    }
};
```

Heavy-Light Decomposition

Constructs a heavy-light decomposition of a tree and generates indices for nodes that are consecutive within each heavy path.

For node `i`, `hchild[i]` is the heavy child (or -1 for leaf nodes), `hpLeaf[i]` and `hpRoot[i]` are the leaf and root nodes of the heavy path passing through the node, and `arridx[i]` is the generated heavy path consecutive index (in the range $(0, n]$). Within one heavy path, the deepest node has the lowest `arridx[i]`;

`lca(a, b)` returns the lca of a and b. If `intv` is not null, `*intv` will receive up to $2\log_2(n)$ non-overlapping intervals such that there exists an interval `i` $\in$ `*intv` where `i.first <= arridx[x] < i.second` iff. the node `x` is on the path between a and b.
If `intvIncludeLCA` is false, the lca of a and b will not be included in these intervals.

```
struct HLD {
    vector<ll> depth, parent, arridx, hpLeaf, hpRoot, hchild;
    HLD(const vector<vector<ll>>& t) { // t is an adjacency list, or a child list for a tree rooted in node 0
        parent = hchild = vector<ll>(t.size(), -1);
        depth = arridx = hpLeaf = hpRoot = vector<ll>(t.size());
        vector<ll> sts(t.size(), 1), ci(t.size()), st{0}, trav{0};
        while (!st.empty()) {
            ll cur = st.back(), nx;
            if (ci[cur] == (ll)t[cur].size()) {
                st.pop_back();
                if (st.empty()) continue;
                sts[st.back()] += sts[cur];
                if (hchild[st.back()] == -1 || sts[cur] > sts[hchild[st.back()]])
                    hchild[st.back()] = cur;
            } else if ((nx = t[cur][ci[cur]++]) != parent[cur]) {
                depth[nx] = depth[cur] + 1;
                parent[nx] = cur;
                st.push_back(nx);
                trav.push_back(nx);
            }
        }
        iota(all(hpRoot), 0);
        iota(all(hpLeaf), 0);
        ll nai = 0;
        for (ll cur : trav) {
            if (hchild[cur] == -1) {
                arridx[cur] = nai;
                nai += depth[cur] - depth[hpRoot[cur]] + 1;
            }
            else
                hpRoot[hchild[cur]] = hpRoot[cur];
        }
        for (ll i = trav.size() - 1; i >= 0; i--) {
            if (hchild[trav[i]] == -1) continue;
            arridx[trav[i]] = arridx[hchild[trav[i]]] + 1;
            hpLeaf[trav[i]] = hpLeaf[hchild[trav[i]]];
        }
    }
    ll lca(ll a, ll b, vector<pair<ll, ll>>* intv = nullptr, bool intvIncludeLCA = true) {
        auto sdepth = [&] (ll i) { return i == -1 ? -1 : depth[i]; };
        auto addi = [&] (ll lo, ll hi) { if (intv && lo != hi) intv->emplace_back(lo, hi); };
        while (hpRoot[a] != hpRoot[b]) {
            ll nxa = parent[hpRoot[a]];
            ll nxb = parent[hpRoot[b]];
            if (sdepth(nxa) > sdepth(nxb)) {
                addi(arridx[a], arridx[hpRoot[a]] + 1);
                a = nxa;
            } else {
                addi(arridx[b], arridx[hpRoot[b]] + 1);
                b = nxb;
            }
        }
        if (depth[a] > depth[b])
            swap(a, b);
        addi(arridx[b], arridx[a] + intvIncludeLCA);
        return a;
    }
};
```

Treap

```cpp
struct Treap {
    Treap *l = 0, *r = 0;
    int val, y, c = 1;
    Treap(int v) : val(v), y(rand()) { }
};
int trCount(Treap* n) { // returns the number of nodes in treap n
    return n ? n->c : 0;
}
void trRecount(Treap* n) {
    n->c = trCount(n->l) + trCount(n->r) + 1;
}
Treap* trAt(Treap* n, int idx) { // returns the treap node at the specified index
    if (!n || idx == trCount(n->l)) return n;
    if (idx > trCount(n->l))
        return trAt(n->r, idx - trCount(n->l) - 1);
    return trAt(n->l, idx);
}
template<class F> void trForeach(Treap* n, F f) { // invokes f for every item in the treap n
    if (n) { trForeach(n->l, f); f(n->val); trForeach(n->r, f); }
}
pair<Treap*, Treap*> trSplit(Treap* n, int k) { // splits the treap n on index (or value) k
    if (!n) return {};
    if (trCount(n->l) >= k) { // use "if (n->val >= k) {" to split on value instead of index
        auto pa = trSplit(n->l, k);
        n->l = pa.second;
        trRecount(n);
        return {pa.first, n};
    } else {
        // use "auto pa = trSplit(n->r, k);" to split on value instead of index
        auto pa = trSplit(n->r, k - trCount(n->l) - 1);
        n->r = pa.first;
        trRecount(n);
        return {n, pa.second};
    }
}
Treap* trJoin(Treap* l, Treap* r) {
    if (!l) return r;
    if (!r) return l;
    if (l->y > r->y) {
        l->r = trJoin(l->r, r);
        trRecount(l);
        return l;
    } else {
        r->l = trJoin(l, r->l);
        trRecount(r);
        return r;
    }
}
// inserts the treap n into t at index pos (or value pos, depending on implementation of trSplit)
Treap* trInsert(Treap* t, Treap* n, int pos) {
    auto pa = trSplit(t, pos);
    return trJoin(trJoin(pa.first, n), pa.second);
}
```

Link Cut Tree

Represents a forest of unrooted trees. You can add and remove edges (as long as the result is still a forest), and check whether two nodes are in the same tree. All operations are amortized $\mathcal{O}(\log(n))$.

```cpp
struct Node { // Splay tree. Root's pp contains tree's parent.
    Node *p = 0, *pp = 0, *c[2];
    bool flip = 0;
    Node() { c[0] = c[1] = 0; fix(); }
    void fix() {
        if (c[0]) c[0]->p = this;
        if (c[1]) c[1]->p = this;
        // (+ update sum of subtree elements etc. if wanted)
    }
    void pushFlip() {
        if (!flip) return;
        flip = 0; swap(c[0], c[1]);
        if (c[0]) c[0]->flip ^= 1;
        if (c[1]) c[1]->flip ^= 1;
    }
    int up() { return p ? p->c[1] == this : -1; }
    void rot(int i, int b) {
        int h = i ^ b;
        Node *x = c[i], *y = b == 2 ? x : x->c[h], *z = b ? y : x;
        if ((y->p = p)) p->c[up()] = y;
        c[i] = z->c[i ^ 1];
        if (b < 2) {
            x->c[h] = y->c[h ^ 1];
            z->c[h ^ 1] = b ? x : this;
        }
        y->c[i ^ 1] = b ? this : x;
        fix(); x->fix(); y->fix();
        if (p) p->fix();
        swap(pp, y->pp);
    }
    void splay() { /// Splay this up to the root. Always finishes without flip set.
        for (pushFlip(); p; ) {
            if (p->p) p->p->pushFlip();
            p->pushFlip(); pushFlip();
            int c1 = up(), c2 = p->up();
            if (c2 == -1) p->rot(c1, 2);
            else p->p->rot(c2, c1 != c2);
        }
    }
    Node* first() { /// Return the min element of the subtree rooted at this, splayed to the top.
        pushFlip();
        return c[0] ? c[0]->first() : (splay(), this);
    }
};
struct LinkCut {
    vector<Node> node;
    LinkCut(int N) : node(N) {}
    void link(int u, int v) { // add an edge (u, v)
        assert(!connected(u, v));
        makeRoot(&node[u]);
        node[u].pp = &node[v];
    }
    void cut(int u, int v) { // remove an edge (u, v)
        Node *x = &node[u], *top = &node[v];
        makeRoot(top); x->splay();
        assert(top == (x->pp ?: x->c[0]));
        if (x->pp) x->pp = 0;
        else {
            x->c[0] = top->p = 0;
```

```
            x->fix();
        }
    }
    bool connected(int u, int v) { // are u, v in
    ↪ the same tree?
        Node* nu = access(&node[u])->first();
        return nu == access(&node[v])->first();
    }
    void makeRoot(Node* u) { /// Move u to root of
    ↪ represented tree.
        access(u);
        u->splay();
        if(u->c[0]) {
            u->c[0]->p = 0;
            u->c[0]->flip ^= 1;
            u->c[0]->pp = u;
            u->c[0] = 0;
            u->fix();
        }
    }
    Node* access(Node* u) { /// Move u to root aux
    ↪ tree. Return the root of the root aux tree.
        u->splay();
        while (Node* pp = u->pp) {
            pp->splay(); u->pp = 0;
            if (pp->c[1]) {
                pp->c[1]->p = 0; pp->c[1]->pp = pp; }
            pp->c[1] = u; pp->fix(); u = pp;
        }
        return u;
    }
};
```

Graph Algorithms

Maximum Flow (Dinic's Algorithm)

Constructor takes number of nodes, call `addEdge` to add edges and `calc` to find maximum flow. To obtain the actual flow, look at positive values of `Edge::cap` only.

Time complexity: $\mathcal{O}(VE\log U)$ where $U = \max|\text{cap}|$. $\mathcal{O}(\min(E^{1/2}, V^{2/3})E)$ if $U = 1$. $\mathcal{O}(\sqrt{V}E)$ for bipartite matching.

```
struct Dinic {
    struct Edge {
        int to, rev;
        ll c, oc;
        ll flow() { return max(oc - c, 0LL); }
    };
    vector<int> lvl, ptr, q;
    vector<vector<Edge>> adj;
    Dinic(int n) : lvl(n), ptr(n), q(n), adj(n) {}
    void addEdge(int a, int b, ll c, ll rcap = 0) {
        adj[a].push_back({b, (int)adj[b].size(), c, c});
        adj[b].push_back({a, (int)adj[a].size() - 1, rcap, rcap});
    }
    ll dfs(int v, int t, ll f) {
        if (v == t || !f) return f;
        for (int& i = ptr[v]; i < adj[v].size(); i++) {
            Edge& e = adj[v][i];
            if (lvl[e.to] == lvl[v] + 1)
                if (ll p = dfs(e.to, t, min(f, e.c))) {
                    e.c -= p, adj[e.to][e.rev].c += p;
                    return p;
                }
        }
        return 0;
    }
    ll calc(int s, int t) {
        ll flow = 0; q[0] = s;
        for (int L = 0; L < 31; L++) do { // 'int L=30' maybe faster for random data
            lvl = ptr = vector<int>(q.size());
            int qi = 0, qe = lvl[s] = 1;
            while (qi < qe && !lvl[t]) {
                int v = q[qi++];
                for (Edge e : adj[v])
                    if (!lvl[e.to] && e.c >> (30 - L))
                        q[qe++] = e.to, lvl[e.to] = lvl[v] + 1;
            }
            while (ll p = dfs(s, t, LLONG_MAX)) flow += p;
        } while (lvl[t]);
        return flow;
    }
    bool leftOfMinCut(int a) { return lvl[a] != 0; }
};
```

Bellman Ford

Calculates shortest paths from s in a graph that might have negative edge weights. Unreachable nodes get `dist = inf`; nodes reachable through negative-weight cycles get `dist = -inf`. Assumes $V^2 \max |w_i| < 2^{63}$. Time complexity: $\mathcal{O}(VE)$

```cpp
const ll inf = 1LL << 62;
struct Ed {
    int a, b, w;
    int s() { return a < b ? a : -a; }
};
struct Node { ll dist = inf; int prev = -1; };
void bellmanFord(vector<Node>& nodes, vector<Ed>& eds, int s) {
    nodes[s].dist = 0;
    sort(all(eds), [] (Ed a, Ed b) { return a.s() < b.s(); });
    int lim = nodes.size() / 2 + 2;
    for(int i = 0; i < lim; i++)
        for(auto& ed : eds) {
            Node cur = nodes[ed.a], &dest = nodes[ed.b];
            if (abs(cur.dist) == inf) continue;
            ll d = cur.dist + ed.w;
            if (d < dest.dist) {
                dest.prev = ed.a;
                dest.dist = (i < lim - 1 ? d : -inf);
            }
        }
    for(int i = 0; i < lim; i++)
        for(auto& e : eds)
            if (nodes[e.a].dist == -inf)
                nodes[e.b].dist = -inf;
}
```

Floyd Warshall

Calculates all-pairs shortest path in a directed graph. As output, `m[i][j]` is set to the shortest distance between i and j, `inf` if no path, or `-inf` if the path goes through a negative-weight cycle. Time complexity: $\mathcal{O}(N^3)$.

```python
from math import inf
def floydWarshall(m): # m[i][j] should be inf if i and j are not adjacent
    for i in range(len(m)):
        m[i][i] = min(m[i][i], 0)
    for k in range(len(m)):
        for i in range(len(m)):
            for j in range(len(m)):
                if m[i][k] != inf and m[k][j] != inf:
                    m[i][j] = min(m[i][j], max(m[i][k] + m[k][j], -inf))
    #only needed if weights can be negative:
    for k in range(len(m)):
        if m[k][k] < 0:
            for i in range(len(m)):
                for j in range(len(m)):
                    if m[i][k] != inf and m[k][j] != inf:
                        m[i][j] = -inf
```

```cpp
const ll inf = 1LL << 62;
void floydWarshall(vector<vector<ll>>& m) { // m[i][j] should be inf if i and j are not adjacent
    int n = m.size();
    for(int i = 0; i < n; i++)
        m[i][i] = min(m[i][i], 0LL);
    for(int k = 0; k < n; k++)
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
                if (m[i][k] != inf && m[k][j] != inf)
                    m[i][j] = min(m[i][j], max(m[i][k] + m[k][j], -inf));
    //only needed if weights can be negative:
    for(int k = 0; k < n; k++)
        if (m[k][k] < 0)
            for(int i = 0; i < n; i++)
                for(int j = 0; j < n; j++)
                    if (m[i][k] != inf && m[k][j] != inf)
                        m[i][j] = -inf;
}
```

2-SAT

Calculates a valid assignment to boolean variables in a 2-SAT problem. Negated variables are represented by bit-inversions (~x). Time complexity: $\mathcal{O}(N+E)$, where N is the number of boolean variables, and E is the number of clauses.

```
struct TwoSat {
int N;
vector<vector<int>> gr;
vector<int> values; // 0 = false, 1 = true
TwoSat(int n = 0) : N(n), gr(2 * n) {}
void either(int f, int j) {
    f = max(2 * f, -1-2*f);
    j = max(2 * j, -1-2*j);
    gr[f].push_back(j ^ 1);
    gr[j].push_back(f ^ 1);
}
void set_value(int x) { either(x, x); }
vector<int> val, comp, z; int time = 0;
int dfs(int i) {
    int low = val[i] = ++time, x;
    z.push_back(i);
    for(auto& e : gr[i])
        if (!comp[e])
            low = min(low, val[e] ?: dfs(e));
    if (low == val[i]) do {
        x = z.back(); z.pop_back();
        comp[x] = low;
        if (values[x>>1] == -1)
            values[x>>1] = x&1;
    } while (x != i);
    return val[i] = low;
}
bool solve() {
    values.assign(N, -1);
    val.assign(2 * N, 0); comp = val;
    for (int i = 0; i < 2 * N; ++i)
        if (!comp[i])
            dfs(i);
    for (int i = 0; i < N; ++i)
        if (comp[2 * i] == comp[2 * i + 1])
            return 0;
    return 1;
}
int add_var() { //optional
    gr.emplace_back();
    gr.emplace_back();
    return N++;
}
void at_most_one(const vector<int>& li) { //optional
    if (li.size() <= 1) return;
    int cur = ~li[0];
    for(size_t i = 2; i < li.size(); i++) {
        int next = add_var();
        either(cur, ~li[i]);
        either(cur, next);
        either(~li[i], next);
        cur = ~next;
    }
    either(cur, ~li[1]);
}
};
```

Usage:

```
TwoSat ts(number of boolean variables);
ts.either(0, ~3); // Var 0 is true or var 3 is false
ts.set_value(2); // Var 2 is true
ts.at_most_one({0,~1,2}); // <= 1 of vars 0, ~1 and 2 are true
ts.solve(); // Returns true iff it is solvable. ts.values holds the variables' values
```

Strongly Connected Components

Finds strongly connected components in a directed graph. Usage: `scc(graph, [&](vector<int>& v) { ... })` visits all components in reverse topological order. `comp[i]` holds the component index of a node (a component only has edges to components with lower index). `ncomps` will contain the number of components. Time complexity: $\mathcal{O}(E+V)$

```
vector<int> val, comp, z, cont; int Time, ncomps;
template<class G, class F> int dfs(int j, G& g, F& f) {
    int low = val[j] = ++Time, x; z.push_back(j);
    for(auto& e : g[j]) if (comp[e] < 0)
        low = min(low, val[e] ?: dfs(e,g,f));
    if (low == val[j]) {
        do {
            x = z.back(); z.pop_back();
            comp[x] = ncomps;
            cont.push_back(x);
        } while (x != j);
        f(cont); cont.clear();
        ncomps++;
    }
    return val[j] = low;
}
template<class G, class F> void scc(G& g, F f) {
    val.assign(g.size(), 0);
    comp.assign(g.size(), -1);
    Time = ncomps = 0;
    for(size_t i = 0; i < g.size(); i++)
        if (comp[i] < 0) dfs(i, g, f);
}
```

Biconnected Components

Finds all biconnected components in an undirected graph, and returns a list of edges in each. Time complexity: $\mathcal{O}(E+V)$. Note that a node can be in several components, and bridges are by default returned as a single-edge biconnected component.

```
struct BCC {
    const vector<vector<ll>>* adj;
    vector<ll> dfsNum;
    ll nnum = 0;
    vector<pair<ll, ll>> st;
    vector<vector<pair<ll, ll>>> bccs;
    ll dfs(ll cur, ll par) {
        ll top = dfsNum[cur] = ++nnum;
        for (ll nxt : (*adj)[cur]) {
```

```cpp
            if (nxt == par) continue;
            if (dfsNum[nxt]) {
                top = min(top, dfsNum[nxt]);
                if (dfsNum[nxt] < dfsNum[cur])
                    st.emplace_back(cur, nxt);
                continue;
            }
            ll si = st.size();
            ll up = dfs(nxt, cur);
            top = min(top, up);
            if (up == dfsNum[cur]) {
                bccs.emplace_back(st.begin() + si, st.end());
                bccs.back().emplace_back(cur, nxt);
                st.resize(si);
            } else if (up < dfsNum[cur]) {
                st.emplace_back(cur, nxt);
            } else { //the edge (cur,nxt) is a bridge
                bccs.push_back({make_pair(cur, nxt)}); //remove if bridges should not form BCCs
            }
        }
        return top;
    }
};
vector<vector<pair<ll, ll>>> findBCC(const vector<vector<ll>>& adj) {
    BCC bcc = { &adj, vector<ll>(adj.size()) };
    for (ll i = 0; i < (ll)adj.size(); i++)
        if (bcc.dfsNum[i] == 0)
            bcc.dfs(i, -1);
    return move(bcc.bccs);
}
```

Weighted Bipartite Matching

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. Requires $N \leq M$. Time complexity: $\mathcal{O}(N^2M)$

```cpp
pair<ll, vector<ll>> hungarian(const vector<vector<ll>> &a) {
    if (a.empty()) return {0, {}};
    ll n = a.size() + 1;
    ll m = a[0].size() + 1;
    vector<ll> u(n), v(m), p(m), ans(n - 1);
    for (ll i = 1; i < n; i++) {
        p[0] = i;
        ll j0 = 0;
        vector<ll> dist(m, LLONG_MAX), pre(m, -1);
        vector<bool> done(m + 1);
        do {
            done[j0] = true;
            ll i0 = p[j0], j1, delta = LLONG_MAX;
            for (ll 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 (ll 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) {
            ll j1 = pre[j0];
            p[j0] = p[j1], j0 = j1;
        }
    }
    for (ll j = 1; j < m; j++) if (p[j]) ans[p[j] - 1] = j - 1;
    return {-v[0], ans};
}
```

Minimum Cost Maximum Flow

Calculates min-cost max-flow. `cap[i][j]!=cap[j][i]` is allowed; double edges are not. To obtain the actual flow, look at positive values only. Time complexity: $\mathcal{O}(E^2)$. If costs can be negative, call `setpi` before `maxflow`. Negative cost cycles are not supported.

```
#include <bits/extc++.h>
const ll INF = LLONG_MAX / 4;
struct MCMF {
    int N;
    vector<vector<int>> ed, red;
    vector<vector<ll>> cap, flow, cost;
    vector<int> seen;
    vector<ll> dist, pi;
    vector<pair<int, int> > par;
    MCMF(int N) : N(N), ed(N), red(N), cap(N, vector<ll>(N)),
        flow(cap), cost(cap), seen(N), dist(N), pi(N), par(N) { }
    void addEdge(int from, int to, ll cap, ll cost) {
        this->cap[from][to] = cap;
        this->cost[from][to] = cost;
        ed[from].push_back(to);
        red[to].push_back(from);
    }
    void path(int s) {
        fill(all(seen), 0);
        fill(all(dist), INF);
        dist[s] = 0; ll di;
        __gnu_pbds::priority_queue<pair<ll, int>> q;
        vector<decltype(q)::point_iterator> its(N);
        q.push({0, s});
        auto relax = [&](int i, ll cap, ll cost, int dir) {
            ll val = di - pi[i] + cost;
            if (cap && val < dist[i]) {
                dist[i] = val;
                par[i] = {s, dir};
                if (its[i] == q.end())
                    its[i] = q.push({-dist[i], i});
                else
                    q.modify(its[i], {-dist[i], i});
            }
        };
        while (!q.empty()) {
            s = q.top().second; q.pop();
            seen[s] = 1;
            di = dist[s] + pi[s];
            for (auto& i : ed[s]) if (!seen[i])
                relax(i, cap[s][i] - flow[s][i], cost[s][i], 1);
            for (auto& i : red[s]) if (!seen[i])
                relax(i, flow[i][s], -cost[i][s], 0);
        }
        for(int i = 0; i < N; i++)
            pi[i] = min(pi[i] + dist[i], INF);
    }
    pair<ll, ll> maxflow(int s, int t) {
        ll totflow = 0, totcost = 0;
        while (path(s), seen[t]) {
            ll fl = INF;
            for (int p,r,x = t; tie(p,r) = par[x], x != s; x = p)
                fl = min(fl, r ? cap[p][x] - flow[p][x] : flow[x][p]);
            totflow += fl;
            for (int p,r,x = t; tie(p,r) = par[x], x != s; x = p) {
                if (r) flow[p][x] += fl;
                else flow[x][p] -= fl;
            }
        }
        for(int i = 0; i < N; i++)
            for(int j = 0; j < N; j++)
                totcost += cost[i][j] * flow[i][j];
        return { totflow, totcost };
    }
    void setpi(int s) { // optional, if some costs can be negative, call this before maxflow
        fill(all(pi), INF); pi[s] = 0;
        int it = N, ch = 1; ll v;
        while (ch-- && it--)
            for(int i = 0; i < N; i++) if (pi[i] != INF)
```

```
                for(auto& to : ed[i]) if (cap[i][to])
                    if ((v = pi[i] + cost[i][to]) < pi[to])
                        pi[to] = v, ch = 1;
        assert(it >= 0); // negative cost cycle
    }
};
```

Math

Fast Modudo Operations

```
using ull = unsigned long long;
ull modmul(ull a, ull b, ull M) {
    ll ret = a * b - M * ull(1.L / M * a * b);
    return ret + M * (ret < 0) - M * (ret >= (ll)M);
}
ull modpow(ull b, ull e, ull mod) {
    ull ans = 1;
    for (; e; b = modmul(b, b, mod), e /= 2)
        if (e & 1) ans = modmul(ans, b, mod);
    return ans;
}
```

Is Prime (Miller-Rabin)

Guaranteed to work for numbers up to $7 \cdot 10^{18}$. For larger numbers, use Python and extend A randomly.

```
#include "modmul.cpp"
bool isPrime(ull n) {
    if (n < 2 || n % 6 % 4 != 1)
        return (n | 1) == 3;
    ull s = __builtin_ctzll(n-1);
    for (ull a : {2, 325, 9375, 28178, 450775,
    ↪ 9780504, 1795265022}) {
        ull p = modpow(a % n, n >> s, n), i = s;
        while (p != 1 && p != n - 1 && a % n && i--)
            p = modmul(p, p, n);
        if (p != n-1 && i != s) return 0;
    }
    return 1;
}
```

```
def ctz(x): return (x & -x).bit_length() - 1
A = [2,325,9375,28178,450775,9780504,1795265022]
def isPrime(n):
    if n < 2 or n % 6 % 4 != 1:
        return (n | 1) == 3
    s = ctz(n-1)
    for a in A:
        p, i = pow(a % n, n >> s, n), s
        while p != 1 and p != n - 1 and a % n and i:
            p, i = p * p % n, i - 1
        if p != n-1 and i != s:
            return False
    return True
```

Prime Factorization (Pollard-rho)

Returns prime factors of a number, in arbitrary order.

```
#include "is_prime.cpp"
ull pollard(ull n) {
    auto f = [n](ull x) {return modmul(x, x, n)+1;};
    ull x = 0, y = 0, t = 30, prd = 2, i = 1;
    while (t++ % 40 || gcd(prd, n) == 1) {
        if (x == y) x = ++i, y = f(x);
        ull q = modmul(prd, max(x,y)-min(x,y), n);
        if (q) prd = q;
        x = f(x), y = f(f(y));
    }
    return gcd(prd, n);
}
vector<ull> factor(ull n) {
    if (n == 1) return {};
    if (isPrime(n)) return {n};
    ull x = pollard(n);
    auto l = factor(x), r = factor(n / x);
    l.insert(l.end(), all(r));
    return l;
}
```

```
from math import gcd
def pollard(n):
    f = lambda x: x * x % n + 1
    x, y, t, prd, i = 0, 0, 30, 2, 1
    while t % 40 or gcd(prd, n) == 1:
        if x == y:
            i += 1
            x, y = i, f(i)
        if q := prd * (max(x,y) - min(x,y)) % n:
            prd = q
        x, y = f(x), f(f(y))
        t += 1
    return gcd(prd, n)
def factor(n):
    if n == 1: return []
    if isPrime(n): return [n]
    x = pollard(n)
    return factor(x) + factor(n // x)
```

Extended Euclidean Algorithm

Finds the Greatest Common Divisor to the integers a and b. Also finds two integers x and y, such that $ax + by = \gcd(a, b)$. Returns a tuple of $(\gcd(a, b), x, y)$. If a and b are coprime, then x is the inverse of $a \pmod b$.

```cpp
ll extEuclid(ll a, ll b, ll& x, ll& y) {
    if (b) {
        ll d = extEuclid(b, a % b, y, x);
        return y -= a / b * x, d;
    }
    return x = 1, y = 0, a;
}
```

```python
def extEuclid(a, b):
    if b:
        d, x, y = extEuclid(b, a % b)
        return (d, y, x - a // b * y)
    return (a, 1, 0)
```

Chinese Remainder Theorem

Finds the smallest number x satisfying a system of congruences, each in the form $x \equiv r_i \pmod{m_i}$. All pairs of m_i must be coprime. `eq` is a list of tuples describing the equations, the i:th of which should be (r_i, m_i).

```cpp
//no overflow if the product of all eq.second < 2^62
ll crt(const vector<pair<ll, ll>>& eq) {
    ll p = 1, res = 0;
    for (auto e : eq) p *= e.second;
    for (auto e : eq) {
        ll pp = p / e.second, ppi, y;
        extEuclid(pp, e.second, ppi, y);
        res = (res + e.first * ppi * pp) % p;
    }
    return res;
}
```

```python
def crt(eq):
    p, res = 1, 0
    for rem, md in eq:
        p *= md
    for rem, md in eq:
        pp = p // md
        res = (res + rem*extEuclid(pp, md)[1]*pp) % p
    return res
```

Solve Linear System of Equations

Solves $Ax = b$. If there are multiple solutions, an arbitrary one is returned. Returns rank, or -1 if no solutions. Time complexity: $\mathcal{O}(n^2 m)$

```cpp
int solveLinear(vector<vector<double>> A,
    vector<double> b, vector<double>& x) {
    const double eps = 1e-12;
    int n = A.size(), m = x.size(), rank = 0, br, bc;
    if (n) assert((int)A[0].size() == m);
    vector<int> col(m); iota(all(col), 0);
    for(int i = 0; i < n; i++) {
        double v, bv = 0;
        for(int r = i; r < n; ++r)
            for(int c = i; c < m; c++)
                if ((v = fabs(A[r][c])) > bv)
                    br = r, bc = c, bv = v;
        if (bv <= eps) {
            for(int j = i; j < n; j++)
                if (fabs(b[j]) > eps) return -1;
            break;
        }
        swap(A[i], A[br]);
        swap(b[i], b[br]);
        swap(col[i], col[bc]);
        for(int j = 0; j < n; j++)
            swap(A[j][i], A[j][bc]);
        bv = 1 / A[i][i];
        // for all solutions do: for (ll j = 0; j <
    n; j++) { if (j != i) continue;
        for(int j = i + 1; j < n; j++) {
            double fac = A[j][i] * bv;
            b[j] -= fac * b[i];
            for(int k = i+1; k < (m); ++k)
                A[j][k] -= fac*A[i][k];
        }
        rank++;
    }
    // for all solutions do:
    // x.assign(m, undefined);
    // for(int i = 0; i < rank; i++) {
    //     for (int j = rank; j < m; j++)
    //         if (fabs(A[i][j]) > eps) goto fail;
    //     x[col[i]] = b[i] / A[i][i];
    //     fail:;
    // }
    x.assign(m, 0);
    for (int i = rank; i--;) {
        b[i] /= A[i][i];
        x[col[i]] = b[i];
        for (int j = 0; j < i; j++)
            b[j] -= A[j][i] * b[i];
    }
    return rank;
}
```

Matrix Multiplication

```python
def matmul(a, b):
    res = [[0] * len(a) for i in range(len(a))]
    for i in range(len(a)):
        for j in range(len(a)):
            for k in range(len(a)):
                res[i][j] += a[i][k] * b[k][j]
    return res
```

Spherical Distance

```
double sphericalDistance(double f1, double t1,
        double f2, double t2, double radius) {
    double dx = sin(t2)*cos(f2) - sin(t1)*cos(f1);
    double dy = sin(t2)*sin(f2) - sin(t1)*sin(f1);
    double dz = cos(t2) - cos(t1);
    double d = sqrt(dx*dx + dy*dy + dz*dz);
    return radius*2*asin(d/2);
}
```

Returns the shortest distance on the sphere with radius radius between the points with azimuthal angles (longitude) f1 (ϕ_1) and f2 (ϕ_2) from x axis and zenith angles (latitude) t1 (θ_1) and t2 (θ_2) from z axis (0 = north pole). All angles measured in radians. The algorithm starts by converting the spherical coordinates to cartesian coordinates so if that is what you have you can use only the two last rows. dx*radius is then the difference between the two points in the x direction and d*radius is the total distance between the points.

FFT

`fft(a)` computes $\hat{f}(k) = \sum_x a[x] \exp(2\pi i \cdot kx/N)$ for all k. Useful for convolution: `conv(a, b)=c`, where $c[x] = \sum a[i]b[x-i]$. Rounding is safe if $(\sum a_i^2 + \sum b_i^2)\log_2 N < 9 \cdot 10^{14}$ (in practice 10^{16}; higher for random inputs).
Time complexity: $\mathcal{O}(N \log N)$ with $N = |A| + |B|$ (about $1s$ for $N = 4 \cdot 10^6$)

```
typedef complex<double> C;
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 (int k = 2; k < n; k *= 2) {
        R.resize(n); rt.resize(n);
        auto x = polar(1.0L, M_PIl / k);
        for (int i = k; i < 2 * k; i++)
            rt[i] = R[i] = i & 1 ? R[i / 2] * x : R[i / 2];
    }
    vector<int> 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++) {
                auto x = (double*)&rt[j + k], y = (double*)&a[i + j + k];
                C z(x[0] * y[0] - x[1] * y[1], x[0] * y[1] + x[1] * y[0]);
                a[i + j + k] = a[i + j] - z;
                a[i + j] += z;
            }
}
vector<double> conv(const vector<double>& a, const vector<double>& b) {
    if (a.empty() || b.empty()) return { };
    vector<double> res(a.size() + a.size() - 1);
    int L = 32 - __builtin_clz(res.size()), n = 1 << L;
    vector<C> in(n), out(n);
    copy(all(a), begin(in));
    for (size_t i = 0; i < a.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 (size_t i = 0; i < res.size(); i++)
        res[i] = imag(out[i]) / (4 * n);
    return res;
}
```

ModFFT

fft(a) computes $\hat{f}(k) = \sum_x a[x]g^{xk}$ for all k, where $g = \text{root}^{(M-1)/N}$. N must be a power of 2.
For conv, M should be of the form $2^a b + 1$, and the convolution result should have size at most 2^a. Inputs must be in $[0, M)$.

```
constexpr ll M = 998244353, root = 62;
ll modpow(ll b, ll e) {
    ll ans = 1;
    for (; e; b = b * b % M, e /= 2)
        if (e & 1) ans = ans * b % M;
    return ans;
}
void fft(vector<ll> &a) {
    ll n = a.size(), L = 31 - __builtin_clz(n);
    static vector<ll> rt(2, 1);
    for (static ll k = 2, s = 2; k < n; k *= 2, s++) {
        rt.resize(n);
        ll z[] = {1, modpow(root, M >> s)};
        for (ll i = k; i < 2 * k; i++)
            rt[i] = rt[i / 2] * z[i & 1] % M;
    }
    vector<ll> rev(n);
    for (ll i = 0; i < n; i++)
        rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
    for (ll i = 0; i < n; i++)
        if (i < rev[i])
            swap(a[i], a[rev[i]]);
    for (ll k = 1; k < n; k *= 2)
        for (ll i = 0; i < n; i += 2 * k) for (ll j = 0; j < k; j++) {
            ll z = rt[j + k] * a[i + j + k] % M, &ai = a[i + j];
            a[i + j + k] = ai - z + (z > ai ? M : 0);
            ai += (ai + z >= M ? z - M : z);
        }
}
vector<ll> conv(const vector<ll> &a, const vector<ll> &b) {
    if (a.empty() || b.empty()) return {};
    ll s = (ll)(a.size() + b.size()) - 1;
    ll B = 32 - __builtin_clz(s);
    ll n = 1 << B;
    ll inv = modpow(n, M - 2);
    vector<ll> L(a), R(b), out(n);
    L.resize(n); R.resize(n);
    fft(L); fft(R);
    for (ll i = 0; i < n; i++)
        out[-i & (n - 1)] = (ll)L[i] * R[i] % M * inv % M;
    fft(out);
    return { out.begin(), out.begin() + s };
}
```

Fraction Binary Search

Given f and N, finds the smallest fraction $p/q \in [0, 1]$ such that $f(p/q)$ is true, and $p, q \leq N$.

```
struct Frac { ll p, q; };
template<class F> Frac fractionBinarySearch(F f, ll N) {
    bool dir = 1, A = 1, B = 1;
    Frac lo{0, 1}, hi{1, 1}; // Set hi to 1/0 to search (0, N]
    if (f(lo)) return lo;
    assert(f(hi));
    while (A || B) {
        ll adv = 0, step = 1; // move hi if dir, else lo
        for (ll si = 0; step; (step *= 2) >>= si) {
            adv += step;
            Frac mid{lo.p * adv + hi.p, lo.q * adv + hi.q};
            if (abs(mid.p) > N || mid.q > N || dir == !f(mid))
                adv -= step; si = 2;
        }
        hi.p += lo.p * adv;
        hi.q += lo.q * adv;
        dir = !dir;
        swap(lo, hi);
        A = B; B = !!adv;
    }
    return dir ? hi : lo;
}
```

Fraction

```
struct fraction {
    __int128_t e, d;
    fraction(__int128_t E, __int128_t D = 1) : e(E), d(D) {
        if (d < 0) { e = -e; d = -d; }
    }
    bool operator<(fraction o) const { return e * o.d < o.e * d; }
    bool operator==(fraction o) const { return e * o.d == o.e * d; }
    bool operator+(fraction o) const { return e * o.d < o.e * d; }
    fraction operator*(fraction o) const { return fraction(e * o.e, d * o.d); }
    fraction operator+(fraction o) const { return fraction(e * o.d + o.e * d, d * o.d); }
};
```

Polynomial Roots

Finds the real roots of a polynomial. Time complexity: $\mathcal{O}(n^2 \log(1/\epsilon))$. Usage (solves $x^2 - 3x + 2 = 0$): `poly_roots({{ 2, -3, 1 }},-1e9,1e9)`

```
struct Poly {
    vector<double> a;
    double operator()(double x) const {
        double val = 0;
        for(int i = a.size(); i--;)
            (val *= x) += a[i];
        return val;
    }
    void diff() {
        for (size_t i = 1; i < a.size(); i++)
            a[i - 1] = i * a[i];
        a.pop_back();
    }
};
vector<double> poly_roots(Poly p, double xmin, double xmax) {
    if (p.a.size() == 2) return { -p.a[0] / p.a[1] };
    vector<double> ret;
    Poly der = p;
    der.diff();
    auto dr = poly_roots(der, xmin, xmax);
    dr.push_back(xmin - 1);
    dr.push_back(xmax + 1);
    sort(all(dr));
    for (size_t i = 0; i < dr.size() - 1; i++) {
        double l = dr[i], h = dr[i + 1];
        bool sign = p(l) > 0;
        if (sign ^ (p(h) > 0)) {
            for (int it = 0; it < 60; it++) {
                double m = (l + h) / 2, f = p(m);
                if ((f <= 0) ^ sign) l = m;
                else h = m;
            }
            ret.push_back((l + h) / 2);
        }
    }
    return ret;
}
```

Matrix Inverse

Invert matrix A. Returns rank; result is stored in A unless singular (rank < n). Time complexity: $\mathcal{O}(n^3)$

```
ll matInv(vector<vector<double>>& A) {
    ll n = A.size();
    vector<ll> col(n);
    vector<vector<double>> tmp(n, vector<double>(n));
    for (ll i = 0; i < n; i++) {
        tmp[i][i] = 1;
        col[i] = i;
    }
    for (ll i = 0; i < n; i++) {
        ll r = i, c = i;
        for (ll j = i; j < n; j++)
            for (ll k = i; j < n; j++)
                if (fabs(A[j][k]) > fabs(A[r][c]))
                    r = j, c = k;
        if (fabs(A[r][c]) < 1e-12) return i;
        A[i].swap(A[r]); tmp[i].swap(tmp[r]);
        for (ll j = 0; j < n; j++) {
            swap(A[j][i], A[j][c]);
            swap(tmp[j][i], tmp[j][c]);
        }
        swap(col[i], col[c]);
        double v = A[i][i];
        for (ll j = i+1; j < n; j++) {
            double f = A[j][i] / v;
            A[j][i] = 0;
            for (ll k = i+1; k < n; k++)
                A[j][k] -= f*A[i][k];
            for (ll k = 0; k < n; k++)
                tmp[j][k] -= f*tmp[i][k];
        }
        for (ll j = i+1; j < n; j++) A[i][j] /= v;
        for (ll j = 0; j < n; j++) tmp[i][j] /= v;
        A[i][i] = 1;
    }
    for (ll i = n-1; i > 0; --i) {
        for (ll j = 0; j < i; j++) {
            double v = A[j][i];
            for (ll k = 0; k < n; k++)
                tmp[j][k] -= v*tmp[i][k];
        }
    }
    for (ll i = 0; i < n; i++)
        for (ll j = 0; j < n; j++)
            A[col[i]][col[j]] = tmp[i][j];
    return n;
}
```

Simplex

Solves a general linear maximization problem: maximize $c^T x$ subject to $Ax \le b, x \ge 0$.
Returns -inf if there is no solution, inf if there are arbitrarily good solutions, or the maximum value of $c^T x$ otherwise.
The input vector is set to an optimal x (or in the unbounded case, an arbitrary solution fulfilling the constraints).
Numerical stability is not guaranteed. For better performance, define variables such that $x = 0$ is viable.
vvd A = {{1,-1}, {-1,1}, {-1,-2}}; vd b = {1,1,-4}, c = {-1,-1}, x; T val = LPSolver(A, b, c).solve(x);
$O(NM * \#pivots)$, where a pivot may be e.g. an edge relaxation. $O(2^n)$ in the general case.

```
typedef double T; // long double, Rational, double + mod<P>...
typedef vector<T> vd;
typedef vector<vd> vvd;
#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define sz(x) (int)(x).size()

const T eps = 1e-8, inf = 1/.0;
#define MP make_pair
#define ltj(X) if(s == -1 || MP(X[j],N[j]) < MP(X[s],N[s])) s=j

struct LPSolver {
    int m, n;
    vector<ll> N, B;
    vvd D;

    LPSolver(const vvd& A, const vd& b, const vd& c) :
        m(sz(b)), n(sz(c)), N(n+1), B(m), D(m+2, vd(n+2)) {
            rep(i,0,m) rep(j,0,n) D[i][j] = A[i][j];
            rep(i,0,m) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i];}
            rep(j,0,n) { N[j] = j; D[m][j] = -c[j]; }
            N[n] = -1; D[m+1][n] = 1;
        }

    void pivot(int r, int s) {
        T *a = D[r].data(), inv = 1 / a[s];
        rep(i,0,m+2) if (i != r && abs(D[i][s]) > eps) {
            T *b = D[i].data(), inv2 = b[s] * inv;
            rep(j,0,n+2) b[j] -= a[j] * inv2;
            b[s] = a[s] * inv2;
        }
        rep(j,0,n+2) if (j != s) D[r][j] *= inv;
        rep(i,0,m+2) if (i != r) D[i][s] *= -inv;
        D[r][s] = inv;
        swap(B[r], N[s]);
    }

    bool simplex(int phase) {
        int x = m + phase - 1;
        for (;;) {
            int s = -1;
            rep(j,0,n+1) if (N[j] != -phase) ltj(D[x]);
            if (D[x][s] >= -eps) return true;
            int r = -1;
            rep(i,0,m) {
                if (D[i][s] <= eps) continue;
                if (r == -1 || MP(D[i][n+1] / D[i][s], B[i])
                             < MP(D[r][n+1] / D[r][s], B[r])) r = i;
            }
            if (r == -1) return false;
            pivot(r, s);
        }
    }

    T solve(vd &x) {
        int r = 0;
        rep(i,1,m) if (D[i][n+1] < D[r][n+1]) r = i;
        if (D[r][n+1] < -eps) {
            pivot(r, n);
            if (!simplex(2) || D[m+1][n+1] < -eps) return -inf;
            rep(i,0,m) if (B[i] == -1) {
                int s = 0;
                rep(j,1,n+1) ltj(D[i]);
```

```
                pivot(i, s);
            }
        }
        bool ok = simplex(1); x = vd(n);
        rep(i,0,m) if (B[i] < n) x[B[i]] = D[i][n+1];
        return ok ? D[m][n+1] : inf;
    }
};
```

Strings

Manacher

For each position in a string, computes p[0][i] = half length of longest even palindrome around pos i, p[1][i] = longest odd (half rounded down). Time: O(N)

```
#define rep(i, a, b) for(int i = a; i < (b); ++i)
array<vector<int>, 2> manacher(const string& s) {
    int n = s.size();
    array<vector<int>,2> p = {vector<int>(n+1), vector<int>(n)};
    rep(z,0,2) for (int i=0,l=0,r=0; i < n; i++) {
        int t = r-i+!z;
        if (i<r) p[z][i] = min(t, p[z][l+t]);
        int L = i-p[z][i], R = i+p[z][i]-!z;
        while (L>=1 && R+1<n && s[L-1] == s[R+1])
            p[z][i]++, L--, R++;
        if (R>r) l=L, r=R;
    }
    return p;
}
```

Polynomial Hash

```
using lll = __int128_t;
ll P = 1223372036854778 9LL;
ll B = 260;
struct PolyHash {
    vector<ll> hashes, ex;
    PolyHash(const string& s) : hashes(s.size() + 1), ex(s.size() + 1) {
        hashes[0] = 1; ex[0] = 1; ex[1] = B;
        for (size_t i = 0; i < s.size(); i++) {
            hashes[i + 1] = ((hashes[i] * B) % P + s[i] + 1) % P;
            ex[i + 1] = (ex[i] * B) % P;
        }
    }
    ll hash(ll lo, ll hi) {
        return ((lll)hashes[hi] - (lll)hashes[lo] * (lll)ex[hi - lo] % P + P) % P;
    }
};
```

Aho Corasick

Aho-Corasick automaton, used for multiple pattern matching.
Initialize with AhoCorasick ac(patterns); the automaton start node will be at index 0.
find(word) returns for each position the index of the longest word that ends there, or -1 if none.
findAll($-$, word) finds all words (up to $N\sqrt{N}$ many if no duplicate patterns)
that start at each position (shortest first).
Duplicate patterns are allowed; empty patterns are not.
To find the longest words that start at each position, reverse all input.
For large alphabets, split each symbol into chunks, with sentinel bits for symbol boundaries.
Time: construction takes $O(26N)$, where $N =$ sum of length of patterns.
find(x) is $O(N)$, where N = length of x. findAll is $O(NM)$.

```
#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define sz(x) (int)(x).size()
struct AhoCorasick {
    enum {alpha = 26, first = 'A'}; // change this!
    struct Node {
        // (nmatches is optional)
        int back, next[alpha], start = -1, end = -1, nmatches = 0;
        Node(int v) { memset(next, v, sizeof(next)); }
    };
    vector<Node> N;
    vector<int> backp;
    void insert(string& s, int j) {
        assert(!s.empty());
        int n = 0;
        for (char c : s) {
            int& m = N[n].next[c - first];
            if (m == -1) { n = m = sz(N); N.emplace_back(-1); }
            else n = m;
        }
```

```
        if (N[n].end == -1) N[n].start = j;
        backp.push_back(N[n].end);
        N[n].end = j;
        N[n].nmatches++;
    }
    AhoCorasick(vector<string>& pat) : N(1, -1) {
        rep(i,0,sz(pat)) insert(pat[i], i);
        N[0].back = sz(N);
        N.emplace_back(0);

        queue<int> q;
        for (q.push(0); !q.empty(); q.pop()) {
            int n = q.front(), prev = N[n].back;
            rep(i,0,alpha) {
                int &ed = N[n].next[i], y = N[prev].next[i];
                if (ed == -1) ed = y;
                else {
                    N[ed].back = y;
                    (N[ed].end == -1 ? N[ed].end : backp[N[ed].start])
                        = N[y].end;
                    N[ed].nmatches += N[y].nmatches;
                    q.push(ed);
                }
            }
        }
    }
    vector<int> find(string word) {
        int n = 0;
        vector<int> res; // ll count = 0;
        for (char c : word) {
            n = N[n].next[c - first];
            res.push_back(N[n].end);
            // count += N[n].nmatches;
        }
        return res;
    }
    vector<vector<int>> findAll(vector<string>& pat, string word) {
        vector<int> r = find(word);
        vector<vector<int>> res(sz(word));
        rep(i,0,sz(word)) {
            int ind = r[i];
            while (ind != -1) {
                res[i - sz(pat[ind]) + 1].push_back(ind);
                ind = backp[ind];
            }
        }
        return res;
    }
};
```

Suffix tree

Ukkonen's algorithm for online suffix tree construction.
Each node contains indices [l, r) into the string, and a list of child nodes.
Suffixes are given by traversals of this tree, joining [l, r) substrings.
The root is 0 (has l = -1, r = 0), non-existent children are -1.
To get a complete tree, append a dummy symbol – otherwise it may contain an incomplete path (still useful for substring matching, though).

```
#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define sz(x) (int)(x).size()
struct SuffixTree {
    enum { N = 200010, ALPHA = 26 }; // N ~ 2*maxlen+10
    int toi(char c) { return c - 'a'; }
    string a; // v = cur node, q = cur position
    int t[N][ALPHA],l[N],r[N],p[N],s[N],v=0,q=0,m=2;

    void ukkadd(int i, int c) { suff:
        if (r[v]<=q) {
            if (t[v][c]==-1) { t[v][c]=m;  l[m]=i;
                p[m++]=v; v=s[v]; q=r[v];  goto suff; }
            v=t[v][c]; q=l[v];
```

```
        }
        if (q==-1 || c==toi(a[q])) q++; else {
            l[m+1]=i;  p[m+1]=m;  l[m]=l[v];  r[m]=q;
            p[m]=p[v];  t[m][c]=m+1;  t[m][toi(a[q])]=v;
            l[v]=q;  p[v]=m;  t[p[m]][toi(a[l[m]])]=m;
            v=s[p[m]];  q=l[m];
            while (q<r[m]) { v=t[v][toi(a[q])];  q+=r[v]-l[v]; }
            if (q==r[m])  s[m]=v;  else s[m]=m+2;
            q=r[v]-(q-r[m]);  m+=2;  goto suff;
        }
    }

    SuffixTree(string a) : a(a) {
        fill(r,r+N,sz(a));
        memset(s, 0, sizeof s);
        memset(t, -1, sizeof t);
        fill(t[1],t[1]+ALPHA,0);
        s[0] = 1; l[0] = l[1] = -1; r[0] = r[1] = p[0] = p[1] = 0;
        rep(i,0,sz(a)) ukkadd(i, toi(a[i]));
    }

    // example: find longest common substring (uses ALPHA = 28)
    pair<int,int> best;
    int lcs(int node, int i1, int i2, int olen) {
        if (l[node] <= i1 && i1 < r[node]) return 1;
        if (l[node] <= i2 && i2 < r[node]) return 2;
        int mask = 0, len = node ? olen + (r[node] - l[node]) : 0;
        rep(c,0,ALPHA) if (t[node][c] != -1)
            mask |= lcs(t[node][c], i1, i2, len);
        if (mask == 3)
            best = max(best, {len, r[node] - len});
        return mask;
    }
    static pair<int,int> LCS(string s, string t) {
        SuffixTree st(s + (char)('z' + 1) + t + (char)('z' + 2));
        st.lcs(0, sz(s), sz(s) + 1 + sz(t), 0);
        return st.best;
    }
};
```

Estimates

$\sum_{d|n} d = O(n \log \log n)$.

The number of divisors of n is at most around 100 for $n < 5e4$, 500 for $n < 1e7$, 2000 for $n < 1e10$, 200 000 for $n < 1e19$.

Number of ways of writing n as a sum of positive integers, disregarding the order of the summands.

$$p(0) = 1, p(n) = \sum_{k \in \mathbb{Z} \setminus \{0\}} (-1)^{k+1} p(n - k(3k-1)/2)$$

$$p(n) \sim 0.145/n \cdot \exp(2.56\sqrt{n})$$

n	0	1	2	3	4	5	6	7	8	9	20	50	100
$p(n)$	1	1	2	3	5	7	11	15	22	30	627	∼2e5	∼2e8