

Team notebook

November 6, 2025

Contents

1 DP	1
1.1 DivideAndConquerDP	1
1.2 LineContainer	1
1.3 lis	1
2 Data	1
2.1 Fenwick2D	1
2.2 ImplicitTreap	1
2.3 MeldableHeap	2
2.4 OrderStatisticTree	2
2.5 PalindromeTree	2
2.6 RMQ	3
2.7 RMQ	3
2.8 SegmentTree	3
2.9 WalkOnBIT	3
3 Geometry	4
3.1 AngleBisector	4
3.2 Centroid	4
3.3 Circle	4
3.4 ClosestPair	5
3.5 ConvexPolygon	6
3.6 ExtremeVertex	6
3.7 GeometricMedian	6
3.8 GeometryTemplate	6
3.9 HalfPlane	7
3.10 IsPoint	7
3.11 Line	7
3.12 LineLineIntersection	8
3.13 MaximumCircleCover	8
3.14 MaximumInscribedCircle	8
3.15 MinimumEnclosingCircle	8
3.16 MinimumEnclosingRectangle	8
3.17 MinkowskiSum	9
3.18 MonotoneChain	9
3.19 Point2D	9
3.20 PointInsideHull	9
3.21 PointPolygonTangents	10
3.22 PolarSort	10
3.23 PolygonCircleIntersection	10

3.24 PolygonCut	10
3.25 PolygonDiameter	10
3.26 PolygonDistances	10
3.27 PolygonLineIntersection	11
3.28 PolygonUnion	11
3.29 PolygonWidth	11
3.30 Ray	12
3.31 Segment	12
3.32 SmallestEnclosingCircle	12
3.33 TriangleCircleIntersection	12
3.34 Utilities	13
4 Graph	13
4.1 2SAT	13
4.2 Dinic	14
4.3 EulerPath	14
4.4 EulerPathDirected	14
4.5 GeneralMatching	15
4.6 GlobalMinCut	15
4.7 HopcroftKarp	16
4.8 JointsnBridges	16
4.9 MCMF slow	16
4.10 MCMF	17
4.11 SCC	18
4.12 TopoSort	18
4.13 spfa	18
5 Math	19
5.1 Euclid	19
5.2 FFT	19
5.3 Factorization	19
5.4 FastSubsetTransform	20
5.5 Interpolate	20
5.6 Lucas	20
5.7 Matrix	20
5.8 MillerRabin	20
5.9 Mobius	21
5.10 ModInverse	21
5.11 ModMull	21
5.12 ModularArithmetic	21
5.13 NTT	21
5.14 Notes	22
6 Miscellaneous	24
6.1 FastInput	24
6.2 template	24
7 String	24
7.1 AhoCorasick	24
7.2 KMP	25
7.3 Manacher	25
7.4 StringHashing	25
7.5 SuffixArray	25
7.6 Z	25

1 DP

1.1 DivideAndConquerDP

```
const ll INF = 1e18;

void calc(int i, int l, int r, int optL, int optR) {
    if (l > r) return;
    int mid = (l + r) / 2;
    f[i][mid] = INF; // change to -INF to find max
    int opt = -1;
    for (int k = optL; k <= min(mid, optR); ++k) {
        ll c = f[i - 1][k] + cost(k + 1, mid);
        if (c < f[i][mid]) {
            f[i][mid] = c;
            opt = k;
        }
    }
    calc(i, l, mid - 1, optL, opt);
    calc(i, mid + 1, r, opt, optR);
}
```

1.2 LineContainer

```
struct Line {
    mutable ll k, m, p;
    bool operator<(const Line& o) const { return k < o.k; } // change '<' to '>' to get min
    bool operator<(ll x) const { return p < x; }
};

struct LineContainer : multiset<Line, less<>> {
    // (for doubles, use inf = 1/.0, div(a,b) = a/b)
    static 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()) return x->p = inf, 0;
        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;
    }
};
```

1.3 lis

```
template<typename T>
int lis(const vector<T>& a) {
    vector<T> u;
    for (const T& x : a) {
        auto it = lower_bound(u.begin(), u.end(), x);
        if (it == u.end()) {
            u.push_back(x);
        } else {
            *it = x;
        }
    }
    return (int) u.size();
}
```

2 Data

2.1 Fenwick2D

```
struct Fenwick2D {
#define gb(x) (x) & -(x)
    vector<vector<int>> nodes;
    vector<vector<int>> bit;
    int sx;
    void init(int _sx) {
        sx = _sx;
        nodes.resize(sx + 1);
        bit.resize(sx + 1);
    }
    void init_nodes() {
        for (int i = 1; i <= sx; ++i) {
            sort(all(nodes[i]));
            nodes[i].resize(unique(all(nodes[i])) -
                nodes[i].begin());
            bit[i].resize(sz(nodes[i]) + 1);
        }
    }
    void fake_update(int x, int y) {
        for (; x <= sx; x += gb(x)) nodes[x].push_back(y);
    }
    void fake_get(int x, int y) {
        for (; x > 0; x -= gb(x)) nodes[x].push_back(y);
    }
    void update(int x, int yy, int val) {
        for (; x <= sx; x += gb(x))
            for (int y = lower_bound(all(nodes[x]), yy) -
                nodes[x].begin() + 1;
                 y <= sz(nodes[x]); y += gb(y))
                bit[x][y] = max(bit[x][y], val);
    }
    int get(int x, int yy) {
        int res = 0;
        for (; x > 0; x -= gb(x))
```

```
        for (int y = upper_bound(all(nodes[x]), yy) -
            nodes[x].begin(); y > 0;
             y -= gb(y))
            res = max(res, bit[x][y]);
    }
};
```

2.2 ImplicitTreap

```
// Implicit Treap
// Tested: https://oj.vnoi.info/problem/sqrt\_b
struct Treap {
    ll val;
    int prior, size;
    ll sum;
    Treap *left, *right;
    Treap(ll val)
        : val(val), prior(rng()), size(1), sum(val),
          left(NULL), right(NULL){};
    int size(Treap *t) { return t == NULL ? 0 : t->size; }
    void down(Treap *t) {
        // do lazy propagation here
    }
    void refine(Treap *t) {
        if (t == NULL) return;
        t->size = 1;
        t->sum = t->val;
        if (t->left != NULL) {
            t->size += t->left->size;
            t->sum += t->left->sum;
        }
        if (t->right != NULL) {
            t->size += t->right->size;
            t->sum += t->right->sum;
        }
    }
    void split(Treap *t, Treap *&left, Treap *&right, int val) {
        if (t == NULL) return void(left = right = NULL);
        down(t);
        if (size(t->left) < val) {
            split(t->right, t->right, right, val - size(t->left) - 1);
            left = t;
        } else {
            split(t->left, left, t->left, val);
            right = t;
        }
        refine(t);
    }
    void merge(Treap *&t, Treap *left, Treap *right) {
        if (left == NULL) {
            t = right;
            return;
        }
        if (right == NULL) {
```

```

    t = left;
    return;
}

down(left);
down(right);
if (left->prior < right->prior) {
    merge(left->right, left->right, right);
    t = left;
} else {
    merge(right->left, left, right->left);
    t = right;
}
refine(t);
}

array<Treap *, 2> split(Treap *root, int val) {
    array<Treap *, 2> t;
    split(root, t[0], t[1], val);
    return t;
}
array<Treap *, 3> split(Treap *root, int l, int r) {
    array<Treap *, 3> t;
    Treap *tmp;

    split(root, t[0], t[1], l - 1);
    tmp = t[1];
    split(tmp, t[1], t[2], r - l + 1);

    return t;
}
Treap *root;

```

2.3 MeldableHeap

```

mt19937 gen(0x94949);
template<typename T>
struct Node {
    Node *l, *r;
    T v;
    Node(T x): l(0), r(0), v(x){}
};
template<typename T>
Node<T>* Meld(Node<T>* A, Node<T>* B) {
    if (!A) return B; if (!B) return A;
    if (B->v < A->v) swap(A, B);
    if (gen() & 1) A->l = Meld(A->l, B);
    else A->r = Meld(A->r, B);
    return A;
}
template<typename T>
struct Heap {
    Node<T> *r; int s;
    Heap(): r(0), s(0){}
    void push(T x) {
        r = Meld(new Node<T>(x), r);
        ++s;
    }
    int size(){ return s; }
    bool empty(){ return s == 0; }
}

```

```

    T top(){ return r->v; }
    void pop() {
        Node<T>* p = r;
        r = Meld(r->l, r->r);
        delete p;
        --s;
    }
    void Meld(Heap x) {
        s += x->s;
        r = Meld(r, x->r);
    }
}

```

2.4 OrderStatisticTree

```

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <class T>
using Tree =
    tree<T, null_type, less<T>, rb_tree_tag,
          tree_order_statistics_node_update>;
void example() {
    Tree<int> t, t2;
    t.insert(8);
    auto it = t.insert(10).first;
    assert(it == t.lower_bound(9));
    assert(t.order_of_key(10) == 1);
    assert(t.order_of_key(11) == 2);
    assert(*t.find_by_order(0) == 8);
    t.join(t2); // assuming T < T2 or T > T2, merge t2 into t
}

```

2.5 PalindromeTree

```

template <int MAXC = 26>
struct PalindromicTree {
    PalindromicTree(const string& str) : _sz(str.size() +
      5), next(_sz, vector<int>(MAXC, 0)), link(_sz, 0),
      qlink(_sz, 0), cnt(_sz, 0), right_id(_sz, 0),
      len(_sz, 0), s(_sz, 0) {
        init();
        for (int i = 0; i < (int)str.size(); ++i) {
            add(str[i], i);
        }
        count();
    }
    int _sz;
    // returns vector of (left, right, frequency)
    vector<tuple<int, int, int>> get_palindromes() {
        vector<tuple<int, int, int>> res;
        dfs(0, res);
        dfs(1, res);
        return res;
    }
}

```

```

void dfs(int u, vector<tuple<int, int, int>>& res) {
    if (u > 1) { // u = 0 and u = 1 are two empty nodes
        res.emplace_back(right_id[u] - len[u] + 1,
                         right_id[u], cnt[u]);
    }
    for (int i = 0; i < MAXC; ++i) {
        if (next[u][i]) dfs(next[u][i], res);
    }
}

int last, n, p;
vector<vector<int>> next, dlink;
vector<int> link, qlink, cnt, right_id, len, s;

int newnode(int l, int right) {
    len[p] = l;
    right_id[p] = right;
    return p++;
}
void init() {
    p = 0;
    newnode(0, -1), newnode(-1, -1);
    n = last = 0;
    s[n] = -1, link[0] = 1;
}
int getlink(int x) {
    while (s[n - len[x] - 1] != s[n]) {
        if (s[n - len[link[x]] - 1] == s[n]) {
            x = link[x];
        } else
            x = qlink[x];
    }
    return x;
}
void add(char c, int right) {
    c -= 'a';
    s[++n] = c;
    int cur = getlink(last);
    if (!next[cur][(int)c]) {
        int now = newnode(len[cur] + 2, right);
        link[now] = next[getlink(link[cur])][(int)c];
        next[cur][(int)c] = now;
        if (s[n - len[link[now]]] == s[n -
          len[link[link[now]]]]) {
            qlink[now] = qlink[link[now]];
        } else {
            qlink[now] = link[link[now]];
        }
    }
    last = next[cur][(int)c];
    cnt[last]++;
}
void count() {
    for (int i = p - 1; i >= 0; i--) {
        cnt[link[i]] += cnt[i];
    }
}

```

2.6 RMQ

```
// RMQ O(1): 1-indexed
// remember to change the constants, types
using ll = long long;
#define For(i, j, k) for ( int i = (j) ; i <= (k) ; i++ )
#define Fol(i, j, k) for ( int i = (j) ; i >= (k) ; i-- )
namespace RMQ
{
    using T = int; constexpr int N = 2e6 + 6; // change
    this
    inline bool cmp(T x, T y) { return x < y; } // change to '>'
    to query max
    inline T calc(T x, T y) { return cmp(x, y) ? x : y;
    }
    T val[N], pre[N], st[__lg(( N >> 5 ) + 9) + 1][(( N
    >> 5 ) + 9]; unsigned f[N];
    __attribute__((target("bmi")))
    lb(unsigned x) { return __builtin_ctz(x); }
    __attribute__((target("lzcnt")))
    hb(unsigned x) { return __builtin_clz(x) ^ 31;
    }
    inline void build(int n, T *a)
    {
        int m = (n - 1) >> 5, o = hb(m + 1),
            stk[33]; copy(a + 1, a + n + 1, val);
        For(i, 0, n - 1) pre[i] = i & 31 ?
            calc(pre[i - 1], val[i]) : val[i];
        For(i, 0, m) st[0][i] = pre[min(n - 1, i <<
            5 | 31)];
        For(i, 1, o) For(j, 0, m + 1 - (1 << i))
            st[i][j] = calc(st[i - 1][j], st[i -
            1][j + (1 << (i - 1))]);
        For(i, 0, n - 1)
            if (i & 31)
            {
                f[i] = f[i - 1];
                while (o &&
                    !cmp(val[stk[o]],
                        val[i])) f[i] &= ~(1u
                        << (stk[o - 1] & 31));
                f[i] |= 1u << ((stk[o + 1] =
                    i) & 31);
            }
            else f[i] = 1u << ((stk[o + 1] = i
                ) & 31);
    }
    inline T qry(int l, int r)
    {
        if ((--l >> 5) == (--r >> 5)) return
            val[l + hb(f[r] >> (1 & 31))];
        T z = calc(pre[r], val[l + hb(f[l + 31]) >>
            (1 & 31)]);
        if ((l = (l >> 5) + 1) == (r >>= 5))
            return z;
        int t = hb(r - 1); return calc(z,
            calc(st[t][1], st[t][r - (1 << t)]));
    }
}
// build: RMQ::build(n, a), a is an array (not a vector!)
// query: RMQ::qry(l, r)
```

2.7 RMQ

```
ll a[N], st[LG + 1][N];

void pre() {
    for (int i = 1; i <= n; ++i) st[0][i] = a[i];
    for (int j = 1; j <= LG; ++j)
        for (int i = 1; i + (1 << j) - 1 <= n; ++i)
            st[j][i] = __gcd(st[j - 1][i], st[j - 1][i + (1 << (j
                - 1))]);
}

ll query(int l, int r) {
    int k = __lg(r - l + 1);
    return __gcd(st[k][l], st[k][r - (1 << k) + 1]);
}
```

2.8 SegmentTree

```
struct Tree {
    typedef int T;
    static constexpr T unit = INT_MIN;
    T f(T a, T b) { return max(a, b); } // (any associative
    fn)
    vector<T> s;
    int n;
    Tree(int n = 0, T def = unit) : s(2 * n, def), n(n) {}
    void update(int pos, T val) {
        for (s[pos += n] = val; pos /= 2; s[pos] = f(s[pos *
            2], s[pos * 2 + 1]));
    }
    T query(int b, int e) { // query [b , e)
        T ra = unit, rb = unit;
        for (b += n, e += n; b < e; b /= 2, e /= 2) {
            if (b % 2) ra = f(ra, s[b++]);
            if (e % 2) rb = f(s[--e], rb);
        }
        return f(ra, rb);
    }
};
```

2.9 WalkOnBIT

```
template<class T>
class FenwickTree {
public:
    int n;
    vector<T> bit;
    int LOG;
    FenwickTree(int _n) {
        init(_n);
    }
    void init(int _n) {
        n = _n;
        bit = vector<T>(n, 0);
    }
```

```
LOG = __lg(n);
}
T sum(int idx) {
    assert(0 <= idx && idx < n);
    T ret = 0;
    for (; idx >= 0; idx &= idx + 1, --idx)
        ret += bit[idx];
    return ret;
}
T sum(int l, int r) {
    return sum(r) - sum(l - 1);
}
void add(int idx, int delta) {
    for (; idx < n; idx |= idx + 1)
        bit[idx] += delta;
}
int bit_search(T v) { // lower_bound in the prefix
    sums array of A
    T sum = 0;
    int pos = -1;
    for(int i = LOG; i >= 0; i--){
        if (pos + (1 << i) < n && sum +
            bit[pos + (1 << i)] < v) {
            sum += bit[pos + (1 << i)];
            pos += (1 << i);
        }
    }
    return pos + 1; // +1 because 'pos' will
    have position of largest value less
    than 'v',
};
```

3 Geometry

3.1 AngleBisector

```
// bisector vector of <abc
PT angle_bisector(PT &a, PT &b, PT &c){
    PT p = a - b, q = c - b;
    return p + q * sqrt(dot(p, p) / dot(q, q));
}
```

3.2 Centroid

```
// centroid of a (possibly non-convex) polygon,
// assuming that the coordinates are listed in a clockwise
// or
// counterclockwise fashion. Note that the centroid is
// often known as
// the "center of gravity" or "center of mass".
PT centroid(vector<PT> &p) {
    int n = p.size(); PT c(0, 0);
    double sum = 0;
```

```

for (int i = 0; i < n; i++) sum += cross(p[i], p[(i + 1) % n]);
double scale = 3.0 * sum;
for (int i = 0; i < n; i++) {
    int j = (i + 1) % n;
    c = c + (p[i] + p[j]) * cross(p[i], p[j]);
}
return c / scale;
}

```

3.3 Circle

```

struct circle {
    PT p;
    double r;
    circle() {}
    circle(PT _p, double _r) : p(_p), r(_r){};
    // center (x, y) and radius r
    circle(double x, double y, double _r) : p(PT(x, y)),
        r(_r){};
    // circumcircle of a triangle
    // the three points must be unique
    circle(PT a, PT b, PT c) {
        b = (a + b) * 0.5;
        c = (a + c) * 0.5;
        line_line_intersection(b, b + rotatecw90(a - b), c, c +
            rotatecw90(a - c), p);
        r = dist(a, p);
    }
    // inscribed circle of a triangle
    circle(PT a, PT b, PT c, bool t) {
        line u, v;
        double m = atan2(b.y - a.y, b.x - a.x), n = atan2(c.y -
            a.y, c.x - a.x);
        u.a = a;
        u.b = u.a + (PT(cos((n + m) / 2.0), sin((n + m) /
            2.0)));
        v.a = b;
        m = atan2(a.y - b.y, a.x - b.x), n = atan2(c.y - b.y,
            c.x - b.x);
        v.b = v.a + (PT(cos((n + m) / 2.0), sin((n + m) /
            2.0)));
        line_line_intersection(u.a, u.b, v.a, v.b, p);
        r = dist_from_point_to_seg(a, b, p);
    }
    bool operator==(circle v) { return p == v.p && sign(r - v.r) == 0; }
    double area() { return PI * r * r; }
    double circumference() { return 2.0 * PI * r; }
};

// 0 if outside, 1 if on circumference, 2 if inside circle
int circle_point_relation(PT p, double r, PT b) {
    double d = dist(p, b);
    if (sign(d - r) < 0) return 2;
    if (sign(d - r) == 0) return 1;
    return 0;
}

// 0 if outside, 1 if on circumference, 2 if inside circle
int circle_line_relation(PT p, double r, PT a, PT b) {

```

```

    double d = dist_from_point_to_line(a, b, p);
    if (sign(d - r) < 0) return 2;
    if (sign(d - r) == 0) return 1;
    return 0;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> circle_line_intersection(PT c, double r, PT a,
    PT b) {
    vector<PT> ret;
    b = b - a;
    a = a - c;
    double A = dot(b, b), B = dot(a, b);
    double C = dot(a, a) - r * r, D = B * B - A * C;
    if (D < -eps) return ret;
    ret.push_back(c + a + b * (-B + sqrt(D + eps)) / A);
    if (D > eps) ret.push_back(c + a + b * (-B - sqrt(D)) /
        A);
    return ret;
}

// 5 - outside and do not intersect
// 4 - intersect outside in one point
// 3 - intersect in 2 points
// 2 - intersect inside in one point
// 1 - inside and do not intersect
int circle_circle_relation(PT a, double r, PT b, double R)
{
    double d = dist(a, b);
    if (sign(d - r - R) > 0) return 5;
    if (sign(d - r - R) == 0) return 4;
    double l = fabs(r - R);
    if (sign(d - r - R) < 0 && sign(d - l) > 0) return 3;
    if (sign(d - l) == 0) return 2;
    if (sign(d - l) < 0) return 1;
    assert(0);
    return -1;
}

vector<PT> circle_circle_intersection(PT a, double r, PT
    b, double R) {
    if (a == b && sign(r - R) == 0) return {PT(1e18, 1e18)};
    vector<PT> ret;
    double d = sqrt(dist2(a, b));
    if (d > r + R || d + min(r, R) < max(r, R)) return ret;
    double x = (d * d - R * R + r * r) / (2 * d);
    double y = sqrt(r * r - x * x);
    PT v = (b - a) / d;
    ret.push_back(a + v * x + rotateccw90(v) * y);
    if (y > 0) ret.push_back(a + v * x - rotateccw90(v) * y);
    return ret;
}

// returns two circle c1, c2 through points a, b and of
// radius r
// 0 if there is no such circle, 1 if one circle, 2 if two
// circle
int get_circle(PT a, PT b, double r, circle &c1, circle
    &c2) {
    vector<PT> v = circle_circle_intersection(a, r, b, r);
    int t = v.size();
    if (!t) return 0;
    c1.p = v[0], c1.r = r;
    if (t == 2) c2.p = v[1], c2.r = r;
    return t;
}

```

```

}

// returns two circle c1, c2 which is tangent to line u,
// goes through
// point q and has radius r1; 0 for no circle, 1 if c1 =
// c2 , 2 if c1 != c2
int get_circle(line u, PT q, double r1, circle &c1, circle
    &c2) {
    double d = dist_from_point_to_line(u.a, u.b, q);
    if (sign(d - r1 * 2.0) > 0) return 0;
    if (sign(d) == 0) {
        cout << u.v.x << ' ' << u.v.y << '\n';
        c1.p = q + rotateccw90(u.v).truncate(r1);
        c2.p = q + rotatecw90(u.v).truncate(r1);
        c1.r = c2.r = r1;
        return 2;
    }
    line u1 = line(u.a + rotateccw90(u.v).truncate(r1), u.b +
        + rotateccw90(u.v).truncate(r1));
    line u2 = line(u.a + rotatecw90(u.v).truncate(r1), u.b +
        + rotatecw90(u.v).truncate(r1));
    circle cc = circle(q, r1);
    PT p1, p2;
    vector<PT> v;
    v = circle_line_intersection(q, r1, u1.a, u1.b);
    if (v.size()) v = circle_line_intersection(q, r1, u2.a,
        u2.b);
    v.push_back(v[0]);
    p1 = v[0], p2 = v[1];
    c1 = circle(p1, r1);
    if (p1 == p2) {
        c2 = c1;
        return 1;
    }
    c2 = circle(p2, r1);
    return 2;
}

// returns area of intersection between two circles
double circle_circle_area(PT a, double r1, PT b, double
    r2) {
    double d = (a - b).norm();
    if (r1 + r2 < d + eps) return 0;
    if (r1 + d < r2 + eps) return PI * r1 * r1;
    if (r2 + d < r1 + eps) return PI * r2 * r2;
    double theta_1 = acos((r1 * r1 + d * d - r2 * r2) / (2 *
        r1 * d));
    theta_2 = acos((r2 * r2 + d * d - r1 * r1) / (2 *
        r2 * d));
    return r1 * r1 * (theta_1 - sin(2 * theta_1) / 2.) + r2 *
        r2 * (theta_2 - sin(2 * theta_2) / 2.);
}

// tangent lines from point q to the circle
int tangent_lines_from_point(PT p, double r, PT q, line
    &u, line &v) {
    int x = sign(dist2(p, q) - r * r);
    if (x < 0) return 0; // point in circle
    if (x == 0) { // point on circle
        u = line(q, q + rotateccw90(q - p));
        v = u;
        return 1;
    }
    double d = dist(p, q);
    double l = r * r / d;

```

```

double h = sqrt(r * r - 1 * 1);
u = line(q, p + ((q - p).truncate(1) + (rotateccw90(q -
    p).truncate(h)))); 
v = line(q, p + ((q - p).truncate(1) + (rotatecw90(q -
    p).truncate(h)))); 
return 2;
}
// returns outer tangents line of two circles
// if inner == 1 it returns inner tangent lines
int tangents_lines_from_circle(PT c1, double r1, PT c2,
    double r2, bool inner, line &u, line &v) {
if (inner) r2 = -r2;
PT d = c2 - c1;
double dr = r1 - r2, d2 = d.norm2(), h2 = d2 - dr * dr;
if (d2 == 0 || h2 < 0) {
    assert(h2 != 0);
    return 0;
}
vector<pair<PT, PT>> out;
for (int tmp : {-1, 1}) {
    PT v = (d * dr + rotateccw90(d) * sqrt(h2) * tmp) / d2;
    out.push_back({c1 + v * r1, c2 + v * r2});
}
u = line(out[0].first, out[0].second);
if (out.size() == 2) v = line(out[1].first,
    out[1].second);
return 1 + (h2 > 0);
}
// O(n^2 log n)
struct CircleUnion {
    int n;
    double x[2020], y[2020], r[2020];
    int covered[2020];
    vector<pair<double, double>> seg, cover;
    double arc, pol;
    inline int sign(double x) { return x < -eps ? -1 : x >
        eps; }
    inline int sign(double x, double y) { return sign(x -
        y); }
    inline double SQ(const double x) { return x * x; }
    inline double dist(double x1, double y1, double x2,
        double y2) {
        return sqrt(SQ(x1 - x2) + SQ(y1 - y2));
    }
    inline double angle(double A, double B, double C) {
        double val = (SQ(A) + SQ(B) - SQ(C)) / (2 * A * B);
        if (val < -1) val = -1;
        if (val > +1) val = +1;
        return acos(val);
    }
    CircleUnion() {
        n = 0;
        seg.clear(), cover.clear();
        arc = pol = 0;
    }
    void init() {
        n = 0;
        seg.clear(), cover.clear();
        arc = pol = 0;
    }
    void add(double xx, double yy, double rr) {
        x[n] = xx, y[n] = yy, r[n] = rr, covered[n] = 0, n++;
    }
};

```

```

}
void getarea(int i, double lef, double rig) {
    arc += 0.5 * r[i] * r[i] * (rig - lef - sin(rig - lef));
    double x1 = x[i] + r[i] * cos(lef), y1 = y[i] + r[i] *
        sin(lef);
    double x2 = x[i] + r[i] * cos(rig), y2 = y[i] + r[i] *
        sin(rig);
    pol += x1 * y2 - x2 * y1;
}
double solve() {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (!sign(x[i] - x[j]) && !sign(y[i] - y[j]) &&
                !sign(r[i] - r[j])) {
                r[i] = 0.0;
                break;
            }
        }
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (i != j && sign(r[j] - r[i]) >= 0 &&
                    sign(dist(x[i], y[i], x[j], y[j]) - (r[j] -
                        r[i])) <= 0) {
                    covered[i] = 1;
                    break;
                }
            }
            for (int i = 0; i < n; i++) {
                if (sign(r[i]) && !covered[i]) {
                    seg.clear();
                    for (int j = 0; j < n; j++) {
                        if (i != j) {
                            double d = dist(x[i], y[i], x[j], y[j]);
                            if (sign(d - (r[j] + r[i])) >= 0 || sign(d -
                                abs(r[j] - r[i])) <= 0) {
                                continue;
                            }
                            double alpha = atan2(y[j] - y[i], x[j] - x[i]);
                            double beta = angle(r[i], d, r[j]);
                            pair<double, double> tmp(alpha - beta, alpha +
                                beta);
                            if (sign(tmp.first) <= 0 && sign(tmp.second) <=
                                0) {
                                seg.push_back(pair<double, double>(2 * PI +
                                    tmp.first, 2 * PI + tmp.second));
                            } else if (sign(tmp.first) < 0) {
                                seg.push_back(pair<double, double>(2 * PI +
                                    tmp.first, 2 * PI));
                                seg.push_back(pair<double, double>(0,
                                    tmp.second));
                            } else {
                                seg.push_back(tmp);
                            }
                        }
                    }
                    sort(seg.begin(), seg.end());
                    double rig = 0;
                    for (vector<pair<double, double>>::iterator iter =
                        seg.begin(); iter != seg.end(); iter++) {
                        if (sign(rig - iter->first) >= 0) {

```

```

                            rig = max(rig, iter->second);
                        } else {
                            getarea(i, rig, iter->first);
                            rig = iter->second;
                        }
                    }
                    if (!sign(rig)) {
                        arc += r[i] * r[i] * PI;
                    } else {
                        getarea(i, rig, 2 * PI);
                    }
                }
            }
            return pol / 2.0 + arc;
        }
    } CU;

```

3.4 ClosestPair

```

typedef Point<ll> P;
pair<P, P> closest(vector<P> v) {
    assert(sz(v) > 1);
    set<P> S;
    sort(all(v), [](P a, P b) { return a.y < b.y; });
    pair<ll, pair<P, P>> ret{LLONG_MAX, {P(), P()}};
    int j = 0;
    for (P p : v) {
        P d{1 + (11)sqrt(ret.first), 0};
        while (v[j].y <= p.y - d.x) S.erase(v[j++]);
        auto lo = S.lower_bound(p - d), hi = S.upper_bound(p +
            d);
        for (; lo != hi; ++lo) ret = min(ret, {(*lo -
            p).dist2(), {*lo, p}});
        S.insert(p);
    }
    return ret.second;
}

```

3.5 ConvexPolygon

```

// checks if convex or not
bool is_convex(vector<PT>& p) {
    bool s[3];
    s[0] = s[1] = s[2] = 0;
    int n = p.size();
    for (int i = 0; i < n; i++) {
        int j = (i + 1) % n;
        int k = (j + 1) % n;
        s[sign(cross(p[j] - p[i], p[k] - p[i])) + 1] = 1;
        if (s[0] && s[2])
            return 0;
    }
    return 1;
}
// -1 if strictly inside, 0 if on the polygon, 1 if
// strictly outside

```

```
// it must be strictly convex, otherwise make it strictly
// convex first
int is_point_in_convex(vector<PT>& p, const PT& x) { // O(log n)
    int n = p.size();
    assert(n >= 3);
    int a = orientation(p[0], p[1], x), b =
        orientation(p[0], p[n - 1], x);
    if (a < 0 || b > 0)
        return 1;
    int l = 1, r = n - 1;
    while (l + 1 < r) {
        int mid = l + r >> 1;
        if (orientation(p[0], p[mid], x) >= 0) l = mid;
        else r = mid;
    }
    int k = orientation(p[l], p[r], x);
    if (k <= 0)
        return -k;
    if (l == 1 && a == 0)
        return 0;
    if (r == n - 1 && b == 0)
        return 0;
    return -1;
}
```

3.6 ExtremeVertex

```
// id of the vertex having maximum dot product with z
// polygon must need to be convex
// top - upper right vertex
// for minimum dot prout negate z and return -dot(z,
// p[id])
int extreme_vertex(vector<PT> &p, const PT &z, const int
    top) { // O(log n)
    int n = p.size();
    if (n == 1) return 0;
    double ans = dot(p[0], z); int id = 0;
    if (dot(p[top], z) > ans) ans = dot(p[top], z), id =
        top;
    int l = 1, r = top - 1;
    while (l < r) {
        int mid = l + r >> 1;
        if (dot(p[mid + 1], z) >= dot(p[mid], z)) l = mid +
            1;
        else r = mid;
    }
    if (dot(p[l], z) > ans) ans = dot(p[l], z), id = l;
    l = top + 1, r = n - 1;
    while (l < r) {
        int mid = l + r >> 1;
        if (dot(p[(mid + 1) % n], z) >= dot(p[mid], z)) l =
            mid + 1;
        else r = mid;
    }
    l %= n;
    if (dot(p[l], z) > ans) ans = dot(p[l], z), id = l;
    return id;
}
```

3.7 GeometricMedian

```
// it returns a point such that the sum of distances
// from that point to all points in p is minimum
// O(n log^2 MX)
PT geometric_median(vector<PT> p) {
    auto tot_dist = [&](PT z) {
        double res = 0;
        for (int i = 0; i < p.size(); i++) res +=
            dist(p[i], z);
        return res;
    };
    auto findY = [&](double x) {
        double yl = -1e5, yr = 1e5;
        for (int i = 0; i < 60; i++) {
            double ym1 = yl + (yr - yl) / 3;
            double ym2 = yr - (yr - yl) / 3;
            double d1 = tot_dist(PT(x, ym1));
            double d2 = tot_dist(PT(x, ym2));
            if (d1 < d2) yr = ym2;
            else yl = ym1;
        }
        return pair<double, double> (yl, tot_dist(PT(x,
            yl)));
    };
    double xl = -1e5, xr = 1e5;
    for (int i = 0; i < 60; i++) {
        double xm1 = xl + (xr - xl) / 3;
        double xm2 = xr - (xr - xl) / 3;
        double yl, d1, y2, d2;
        auto z = findY(xm1); yl = z.first; d1 = z.second;
        z = findY(xm2); y2 = z.first; d2 = z.second;
        if (d1 < d2) xr = xm2;
        else xl = xm1;
    }
    return {xl, findY(xl).first};
}
```

3.8 GeometryTemplate

```
const long double PI = acos(-1);
struct Vector {
    using type = long long;
    type x, y;
    Vector operator-(const Vector &other) const {
        return {x - other.x, y - other.y};
    }
    type operator*(const Vector &other) const {
        return x * other.y - other.x * y;
    }
    type operator%(const Vector &other) const {
        return x * other.x + y * other.y;
    }
    bool operator==(const Vector &other) const {
        return x == other.x and y == other.y;
    }
    bool operator!=(const Vector &other) const { return
        !(*this == other); }
```

```
friend type cross(const Vector &A, const Vector &B,
    const Vector &C) {
    return (B - A) * (C - A);
}
friend type dist(Vector A) { return A.x * A.x + A.y *
    A.y; }
friend type dot(const Vector &A, const Vector &B, const
    Vector &C) {
    Vector u = (B - A), v = (C - A);
    return u % v;
}
friend istream &operator>>(istream &is, Vector &V) {
    is >> V.x >> V.y;
    return is;
}
friend ostream &operator<<(ostream &os, Vector &V) {
    os << V.x << ' ' << V.y;
    return os;
}
friend double angle(const Vector &A, const Vector &B,
    const Vector &C) {
    double x = dot(B, A, C) / sqrt(dist(A - B) * dist(C -
        B));
    return acos(min(1.0, max(-1.0, x))) * 180.0 / PI;
}
using Point = Vector;
const Point origin = {0, 0};

long double area(Point A, Point B, Point C) {
    long double res =
        cross(origin, A, B) + cross(origin, B, C) +
        cross(origin, C, A);
    return abs(res) / 2.0;
}
```

3.9 HalfPlane

```
// contains all points p such that: cross(b - a, p - a) >=
    0
struct HP {
    PT a, b;
    HP() {}
    HP(PT a, PT b) : a(a), b(b) {}
    HP(const HP& rhs) : a(rhs.a), b(rhs.b) {}
    int operator < (const HP& rhs) const {
        PT p = b - a;
        PT q = rhs.b - rhs.a;
        int fp = (p.y < 0 || (p.y == 0 && p.x < 0));
        int fq = (q.y < 0 || (q.y == 0 && q.x < 0));
        if (fp != fq) return fp == 0;
        if (cross(p, q)) return cross(p, q) > 0;
        return cross(p, rhs.b - a) < 0;
    }
    PT line_line_intersection(PT a, PT b, PT c, PT d) {
        b = b - a; d = c - d; c = c - a;
        return a + b * cross(c, d) / cross(b, d);
    }
    PT intersection(const HP &v) {
```

```

    return line_line_intersection(a, b, v.a, v.b);
}
}

int check(HP a, HP b, HP c) {
    return cross(a.b - a.a, b.intersection(c) - a.a) >
        -eps; // -eps to include polygons of zero area
        (straight lines, points)
}

// consider half-plane of counter-clockwise side of each
// line
// if lines are not bounded add infinity rectangle
// returns a convex polygon, a point can occur multiple
// times though
// complexity: O(n log(n))
vector<PT> half_plane_intersection(vector<HP> h) {
    sort(h.begin(), h.end());
    vector<HP> tmp;
    for (int i = 0; i < h.size(); i++) {
        if (!i || cross(h[i].b - h[i].a, h[i - 1].b - h[i -
            1].a)) {
            tmp.push_back(h[i]);
        }
    }
    h = tmp;
    vector<HP> q(h.size() + 10);
    int qh = 0, qe = 0;
    for (int i = 0; i < h.size(); i++) {
        while (qe - qh > 1 && !check(h[i], q[qe - 2], q[qe -
            - 1])) qe--;
        while (qe - qh > 1 && !check(h[i], q[qh], q[qh +
            1])) qh++;
        q[qe++] = h[i];
    }
    while (qe - qh > 2 && !check(q[qh], q[qe - 2], q[qe -
        1])) qe--;
    while (qe - qh > 2 && !check(q[qe - 1], q[qh], q[qh +
        1])) qh++;
    vector<HP> res;
    for (int i = qh; i < qe; i++) res.push_back(q[i]);
    vector<PT> hull;
    if (res.size() > 2) {
        for (int i = 0; i < res.size(); i++) {
            hull.push_back(res[i].intersection(res[(i + 1) %
                ((int)res.size())]));
        }
    }
    return hull;
}

```

3.10 IsPoint

```

// -1 if strictly inside, 0 if on the polygon, 1 if
// strictly outside
int is_point_in_triangle(PT a, PT b, PT c, PT p) {
    if (sign(cross(b - a, c - a)) < 0) swap(b, c);
    int c1 = sign(cross(b - a, p - a));
    int c2 = sign(cross(c - b, p - b));
    int c3 = sign(cross(a - c, p - c));
    if (c1 < 0 || c2 < 0 || c3 < 0) return 1;

```

```

    if (c1 + c2 + c3 != 3) return 0;
    return -1;
}

bool is_point_on_polygon(vector<PT> &p, const PT& z) {
    int n = p.size();
    for (int i = 0; i < n; i++) {
        if (is_point_on_seg(p[i], p[(i + 1) % n], z))
            return 1;
    }
    return 0;
}

// returns 1e9 if the point is on the polygon
int winding_number(vector<PT> &p, const PT& z) { // O(n)
    if (is_point_on_polygon(p, z)) return 1e9;
    int n = p.size(), ans = 0;
    for (int i = 0; i < n; ++i) {
        int j = (i + 1) % n;
        bool below = p[i].y < z.y;
        if (below != (p[j].y < z.y)) {
            auto orient = orientation(z, p[j], p[i]);
            if (orient == 0) return 0;
            if (below == (orient > 0)) ans += below ? 1 :
                -1;
        }
    }
    return ans;
}

// -1 if strictly inside, 0 if on the polygon, 1 if
// strictly outside
int is_point_in_polygon(vector<PT> &p, const PT& z) { // O(n)
    int k = winding_number(p, z);
    return k == 1e9 ? 0 : k == 0 ? 1 : -1;
}

```

3.11 Line

```

struct line {
    PT a, b; // goes through points a and b
    PT v; double c; //line form: direction vec [cross] (x,
                    y) = c
    line() {}
    //direction vector v and offset c
    line(PT v, double c) : v(v), c(c) {
        auto p = get_points();
        a = p.first; b = p.second;
    }
    // equation ax + by + c = 0
    line(double _a, double _b, double _c) : v({_b, -_a}),
        c(_c) {
        auto p = get_points();
        a = p.first; b = p.second;
    }
    // goes through points p and q
    line(PT p, PT q) : v(q - p), c(cross(v, p)), a(p), b(q)
                    {}
}

```

```

pair<PT, PT> get_points() { //extract any two
    points from this line
    PT p, q; double a = -v.y, b = v.x; // ax + by = c
    if (sign(a) == 0) {
        p = PT(0, c / b);
        q = PT(1, c / b);
    }
    else if (sign(b) == 0) {
        p = PT(c / a, 0);
        q = PT(c / a, 1);
    }
    else {
        p = PT(0, c / b);
        q = PT(1, (c - a) / b);
    }
    return {p, q};
}
//ax + by + c = 0
array<double, 3> get_abc() {
    double a = -v.y, b = v.x;
    return {a, b, c};
}
// 1 if on the left, -1 if on the right, 0 if on the
// line
int side(PT p) { return sign(cross(v, p) - c); }
// line that is perpendicular to this and goes through
// point p
line perpendicular_through(PT p) { return {p, p +
    perp(v)}; }
// translate the line by vector t i.e. shifting it by
// vector t
line translate(PT t) { return {v, c + cross(v, t)}; }
// compare two points by their orthogonal projection on
// this line
// a projection point comes before another if it comes
// first according to vector v
bool cmp_by_projection(PT p, PT q) { return dot(v, p) <
    dot(v, q); }
line shift_left(double d) {
    PT z = v.perp().truncate(d);
    return line(a + z, b + z);
}

```

3.12 LineLineIntersection

```

// intersection point between ab and cd assuming unique
// intersection exists
bool line_line_intersection(PT a, PT b, PT c, PT d, PT
    &ans) {
    double a1 = a.y - b.y, b1 = b.x - a.x, c1 = cross(a, b);
    double a2 = c.y - d.y, b2 = d.x - c.x, c2 = cross(c, d);
    double det = a1 * b2 - a2 * b1;
    if (det == 0) return 0;
    ans = PT((b1 * c2 - b2 * c1) / det, (c1 * a2 - a1 * c2) /
        det);
    return 1;
}

```

3.13 MaximumCircleCover

```
// find a circle of radius r that contains as many points
// as possible
// O(n^2 log n);
double maximum_circle_cover(vector<PT> p, double r, circle
    &c) {
    int n = p.size();
    int ans = 0;
    int id = 0; double th = 0;
    for (int i = 0; i < n; ++i) {
        // maximum circle cover when the circle goes
        // through this point
        vector<pair<double, int>> events = {{-PI, +1}, {PI,
            -1}};
        for (int j = 0; j < n; ++j) {
            if (j == i) continue;
            double d = dist(p[i], p[j]);
            if (d > r * 2) continue;
            double dir = (p[j] - p[i]).arg();
            double ang = acos(d / 2 / r);
            double st = dir - ang, ed = dir + ang;
            if (st > PI) st -= PI * 2;
            if (st <= -PI) st += PI * 2;
            if (ed > PI) ed -= PI * 2;
            if (ed <= -PI) ed += PI * 2;
            events.push_back({st - eps, +1}); // take care
                // of precisions!
            events.push_back({ed, -1});
            if (st > ed) {
                events.push_back({-PI, +1});
                events.push_back({+PI, -1});
            }
        }
        sort(events.begin(), events.end());
        int cnt = 0;
        for (auto &&e: events) {
            cnt += e.second;
            if (cnt > ans) {
                ans = cnt;
                id = i; th = e.first;
            }
        }
    }
    PT w = PT(p[id].x + r * cos(th), p[id].y + r * sin(th));
    c = circle(w, r); //best_circle
    return ans;
}
```

3.14 MaximumInscribedCircle

```
// radius of the maximum inscribed circle in a convex
// polygon
double maximum_inscribed_circle(vector<PT> p) {
    int n = p.size();
    if (n <= 2) return 0;
    double l = 0, r = 20000;
    while (r - l > eps) {
```

```
    double mid = (l + r) * 0.5;
    vector<HP> h;
    const int L = 1e9;
    h.push_back(HP(PT(-L, -L), PT(L, -L)));
    h.push_back(HP(PT(L, -L), PT(L, L)));
    h.push_back(HP(PT(L, L), PT(-L, L)));
    h.push_back(HP(PT(-L, L), PT(-L, -L)));
    for (int i = 0; i < n; i++) {
        PT z = (p[(i + 1) % n] - p[i]).perp();
        z = z.truncate(mid);
        PT y = p[i] + z, q = p[(i + 1) % n] + z;
        h.push_back(HP(p[i] + z, p[(i + 1) % n] + z));
    }
    vector<PT> nw = half_plane_intersection(h);
    if (!nw.empty()) l = mid;
    else r = mid;
}
return l;
```

3.15 MinimumEnclosingCircle

```
// given n points, find the minimum enclosing circle of
// the points
// call convex_hull() before this for faster solution
// expected O(n)
circle minimum_enclosing_circle(vector<PT> &p) {
    random_shuffle(p.begin(), p.end());
    int n = p.size();
    circle c(p[0], 0);
    for (int i = 1; i < n; i++) {
        if (sign(dist(c.p, p[i]) - c.r) > 0) {
            c = circle(p[i], 0);
            for (int j = 0; j < i; j++) {
                if (sign(dist(c.p, p[j]) - c.r) > 0) {
                    c = circle((p[i] + p[j]) / 2, dist(p[i],
                        p[j]) / 2);
                    for (int k = 0; k < j; k++) {
                        if (sign(dist(c.p, p[k]) - c.r) > 0) {
                            c = circle(p[i], p[j], p[k]);
                        }
                    }
                }
            }
        }
    }
    return c;
}
```

3.16 MinimumEnclosingRectangle

```
// minimum perimeter
double minimum_enclosing_rectangle(vector<PT> &p) {
    int n = p.size();
    if (n <= 2) return perimeter(p);
    int mndot = 0; double tmp = dot(p[1] - p[0], p[0]);
```

```
for (int i = 1; i < n; i++) {
    if (dot(p[i] - p[0], p[i]) <= tmp) {
        tmp = dot(p[i] - p[0], p[i]);
        mndot = i;
    }
}
double ans = inf;
int i = 0, j = 1, mxdot = 1;
while (i < n) {
    PT cur = p[(i + 1) % n] - p[i];
    while (cross(cur, p[(j + 1) % n] - p[j]) >= 0) j =
        (j + 1) % n;
    while (dot(p[(mxdot + 1) % n], cur) >=
        dot(p[mndot], cur)) mxdot = (mxdot + 1) % n;
    while (dot(p[(mndot + 1) % n], cur) <=
        dot(p[mndot], cur)) mndot = (mndot + 1) % n;
    ans = min(ans, 2.0 * ((dot(p[mxdot], cur) /
        cur.norm() - dot(p[mndot], cur) / cur.norm()) +
        dist_from_point_to_line(p[i], p[(i + 1) %
        n], p[j])));
    i++;
}
return ans;
}
```

3.17 MinkowskiSum

```
// a and b are strictly convex polygons of DISTINCT points
// returns a convex hull of their minkowski sum with
// distinct points
vector<PT> minkowski_sum(vector<PT> &a, vector<PT> &b) {
    int n = (int)a.size(), m = (int)b.size();
    int i = 0, j = 0; //assuming a[i] and b[j] both are
        //left, bottom)-most points
    vector<PT> c;
    c.push_back(a[i] + b[j]);
    while (1) {
        PT p1 = a[i] + b[(j + 1) % m];
        PT p2 = a[(i + 1) % n] + b[j];
        int t = orientation(c.back(), p1, p2);
        if (t >= 0) j = (j + 1) % m;
        if (t <= 0) i = (i + 1) % n, p1 = p2;
        if (t == 0) p1 = a[i] + b[j];
        if (p1 == c[0]) break;
        c.push_back(p1);
    }
    return c;
}
```

3.18 MonotoneChain

```
vector<Point> convex_hull(vector<Point> p, int n){
    sort(p.begin(), p.end(), [](const Point &A, const Point
        &B){
```

$$\text{return } A.x \neq B.x ? A.x < B.x : A.y < B.y;$$

$$\});$$

```

Point st = p[0], en = p[n - 1];
vector<Point> up = {p[0]};
vector<Point> down = {p[0]};
for(int i = 1; i < n; ++i){
    if(i == n - 1 or cross(st, p[i], en) < 0){
        while((int)up.size() >= 2 and
              cross(up[up.size() - 2], up.back(), p[i]) >= 0)
            up.pop_back();
        up.push_back(p[i]);
    }
    if(i == n - 1 or cross(st, p[i], en) > 0){
        while((int)down.size() >= 2 and
              cross(down[down.size() - 2], down.back(), p[i]) <= 0)
            down.pop_back();
        down.push_back(p[i]);
    }
}
p.clear();
for(int i = 0; i < (int)up.size(); ++i)
    p.push_back(up[i]);
for(int i = down.size() - 2; i >= 1; --i)
    p.push_back(down[i]);
// return hull in clockwise order
return p;
}

```

3.19 Point2D

```

const double inf = 1e100;
const double eps = 1e-9;
const double PI = acos((double)-1.0);
int sign(double x) { return (x > eps) - (x < -eps); }
struct PT {
    double x, y;
    PT() : x(0), y(0) {}
    PT(double x, double y) : x(x), y(y) {}
    PT(const PT &p) : x(p.x), y(p.y) {}
    PT operator + (const PT &a) const { return PT(x + a.x,
          y + a.y); }
    PT operator - (const PT &a) const { return PT(x - a.x,
          y - a.y); }
    PT operator * (const double a) const { return PT(x * a,
          y * a); }
    friend PT operator * (const double &a, const PT &b) {
        return PT(a * b.x, a * b.y); }
    PT operator / (const double a) const { return PT(x / a,
          y / a); }
    bool operator == (PT a) const { return sign(a.x - x) ==
        0 && sign(a.y - y) == 0; }
    bool operator != (PT a) const { return !(*this == a); }
    bool operator < (PT a) const { return sign(a.x - x) ==
        0 ? y < a.y : x < a.x; }
    bool operator > (PT a) const { return sign(a.x - x) ==
        0 ? y > a.y : x > a.x; }
    double norm() { return sqrt(x * x + y * y); }
    double norm2() { return x * x + y * y; }
    PT perp() { return PT(-y, x); }
}

```

```

double arg() { return atan2(y, x); }
PT truncate(double r) { // returns a vector with norm r
    and having same direction
    double k = norm();
    if (!sign(k)) return *this;
    r /= k;
    return PT(x * r, y * r);
}
inline double dot(PT a, PT b) { return a.x * b.x + a.y *
    b.y; }
inline double dist2(PT a, PT b) { return dot(a - b, a -
    b); }
inline double dist(PT a, PT b) { return sqrt(dot(a - b, a -
    b)); }
inline double cross(PT a, PT b) { return a.x * b.y - a.y *
    b.x; }
inline double cross2(PT a, PT b, PT c) { return cross(b -
    a, c - a); }
inline int orientation(PT a, PT b, PT c) { return
    sign(cross(b - a, c - a)); }
PT perp(PT a) { return PT(-a.y, a.x); }
PT rotateccw90(PT a) { return PT(-a.y, a.x); }
PT rotatecw90(PT a) { return PT(a.y, -a.x); }
PT rotateccw(PT a, double t) { return PT(a.x * cos(t) -
    a.y * sin(t), a.x * sin(t) + a.y * cos(t)); }
PT rotatecw(PT a, double t) { return PT(a.x * cos(t) + a.y *
    sin(t), -a.x * sin(t) + a.y * cos(t)); }
double SQ(double x) { return x * x; }
double rad_to_deg(double r) { return (r * 180.0 / PI); }
double deg_to_rad(double d) { return (d * PI / 180.0); }
double get_angle(PT a, PT b) {
    double costheta = dot(a, b) / a.norm() / b.norm();
    return acos(max((double)-1.0, min((double)1.0,
        costheta)));
}
bool is_point_in_angle(PT b, PT a, PT c, PT p) { // does
    point p lie in angle <bac
    assert(orientation(a, b, c) != 0);
    if (orientation(a, c, b) < 0) swap(b, c);
    return orientation(a, c, p) >= 0 && orientation(a, b,
        p) <= 0;
}

```

3.20 PointInsideHull

```

bool on_segment(const Point &A, const Point &B, const
    Point &C) { return cross(A, B, C) == 0 and dot(C, A,
        B) <= 0; }

bool check(vector<Point> &hull, Point &a) {
    int n = sz(hull);
    if (n == 1) return hull[0] == a;
    if (n == 2) return on_segment(hull[0], hull[1], a);
    if (cross(hull[0], hull[1], a) > 0) return 0;
    if (cross(hull[n - 1], hull[0], a) >= 0) return
        on_segment(hull[n - 1], hull[0], a);
    int l = 2, r = n - 1, ans = -1;
    while (l <= r) {

```

```

        int mid = (l + r) / 2;
        if (cross(hull[0], hull[mid], a) >= 0) {
            ans = mid;
            r = mid - 1;
        } else
            l = mid + 1;
    }
    debug(hull[0], hull[ans - 1], hull[ans], a, ans);
    return cross(hull[ans - 1], hull[ans], a) < 0 or
        on_segment(hull[ans - 1], hull[ans], a);
}

```

3.21 PointPolygonTangents

```

pair<PT, PT> convex_line_intersection(vector<PT> &p, PT a,
    PT b) {
    return {{0, 0}, {0, 0}};
}

pair<PT, int> point_poly_tangent(vector<PT> &p, PT Q, int
    dir, int l, int r) {
    while (r - l > 1) {
        int mid = (l + r) >> 1;
        bool pvs = orientation(Q, p[mid], p[mid - 1]) !=
            -dir;
        bool nxt = orientation(Q, p[mid], p[mid + 1]) !=
            -dir;
        if (!pvs && nxt) return {p[mid], mid};
        if (!(pvs || nxt)) {
            auto p1 = point_poly_tangent(p, Q, dir, mid +
                1, r);
            auto p2 = point_poly_tangent(p, Q, dir, l, mid -
                1);
            return orientation(Q, p1.first, p2.first) ==
                dir ? p1 : p2;
        }
        if (!pvs) {
            if (orientation(Q, p[mid], p[1]) == dir) r =
                mid - 1;
            else if (orientation(Q, p[1], p[r]) == dir) r =
                mid - 1;
            else l = mid + 1;
        }
        if (!nxt) {
            if (orientation(Q, p[mid], p[1]) == dir) l =
                mid + 1;
            else if (orientation(Q, p[1], p[r]) == dir) r =
                mid - 1;
            else l = mid + 1;
        }
    }
    pair<PT, int> ret = {p[l], l};
    for (int i = l + 1; i <= r; i++) ret = orientation(Q,
        ret.first, p[i]) != dir ? make_pair(p[i], i) :
        ret;
    return ret;
}
// (cw, ccw) tangents from a point that is outside this
convex polygon

```

```
// returns indexes of the points
pair<int, int> tangents_from_point_to_polygon(vector<PT>
    &p, PT Q) {
    int cw = point_poly_tangent(p, Q, 1, 0, (int)p.size() - 1).second;
    int ccw = point_poly_tangent(p, Q, -1, 0, (int)p.size() - 1).second;
    return make_pair(cw, ccw);
}
```

3.22 PolarSort

```
bool half(PT p) {
    return p.y > 0.0 || (p.y == 0.0 && p.x < 0.0);
}
void polar_sort(vector<PT> &v) { // sort points in
    counter-clockwise
    sort(v.begin(), v.end(), [](PT a, PT b) {
        return make_tuple(half(a), 0.0, a.norm2()) <
            make_tuple(half(b), cross(a, b), b.norm2());
    });
}
void polar_sort(vector<PT> &v, PT o) { // sort points in
    counter-clockwise with respect to point o
    sort(v.begin(), v.end(), [&](PT a, PT b) {
        return make_tuple(half(a - o), 0.0, (a - o).norm2()) < make_tuple(half(b - o), cross(a - o, b - o), (b - o).norm2());
    });
}
```

3.23 PolygonCircleIntersection

```
// intersection between a simple polygon and a circle
double polygon_circle_intersection(vector<PT> &v, PT p,
    double r) {
    int n = v.size();
    double ans = 0.00;
    PT org = {0, 0};
    for(int i = 0; i < n; i++) {
        int x = orientation(p, v[i], v[(i + 1) % n]);
        if(x == 0) continue;
        double area = triangle_circle_intersection(org, r,
            v[i] - p, v[(i + 1) % n] - p);
        if (x < 0) ans -= area;
        else ans += area;
    }
    return abs(ans);
}
```

3.24 PolygonCut

```
// returns a vector with the vertices of a polygon with
// everything
// to the left of the line going from a to b cut away.
vector<PT> cut(vector<PT> &p, PT a, PT b) {
    vector<PT> ans;
    int n = (int)p.size();
    for (int i = 0; i < n; i++) {
        double c1 = cross(b - a, p[i] - a);
        double c2 = cross(b - a, p[(i + 1) % n] - a);
        if (sign(c1) >= 0) ans.push_back(p[i]);
        if (sign(c1 * c2) < 0) {
            if (!is_parallel(p[i], p[(i + 1) % n], a, b)) {
                PT tmp; line_line_intersection(p[i], p[(i + 1) % n], a, b, tmp);
                ans.push_back(tmp);
            }
        }
    }
    return ans;
}
```

3.25 PolygonDiameter

```
// Maximum distance of 2 points
double diameter(vector<PT> &p) {
    int n = (int)p.size();
    if (n == 1) return 0;
    if (n == 2) return dist(p[0], p[1]);
    double ans = 0;
    int i = 0, j = 1;
    while (i < n) {
        while (cross(p[(i + 1) % n] - p[i], p[(j + 1) % n]
            - p[j]) >= 0) {
            ans = max(ans, dist2(p[i], p[j]));
            j = (j + 1) % n;
        }
        ans = max(ans, dist2(p[i], p[j]));
        i++;
    }
    return sqrt(ans);
}
```

3.26 PolygonDistances

```
// minimum distance from a point to a convex polygon
// it assumes point lie strictly outside the polygon
double dist_from_point_to_polygon(vector<PT> &p, PT z) {
    double ans = inf;
    int n = p.size();
    if (n <= 3) {
        for(int i = 0; i < n; i++) ans = min(ans,
            dist_from_point_to_seg(p[i], p[(i + 1) % n],
            z));
        return ans;
    }
    auto [r, l] = tangents_from_point_to_polygon(p, z);
```

```
    if(l > r) r += n;
    while (l < r) {
        int mid = (l + r) >> 1;
        double left = dist2(p[mid % n], z), right =
            dist2(p[(mid + 1) % n], z);
        ans = min({ans, left, right});
        if(left < right) r = mid;
        else l = mid + 1;
    }
    ans = sqrt(ans);
    ans = min(ans, dist_from_point_to_seg(p[l % n], p[(l + 1) % n], z));
    ans = min(ans, dist_from_point_to_seg(p[l % n], p[(l - 1 + n) % n], z));
    return ans;
}

// minimum distance from convex polygon p to line ab
// returns 0 is it intersects with the polygon
// top - upper right vertex
double dist_from_polygon_to_line(vector<PT> &p, PT a, PT b, int top) { // O(log n)
    PT orth = (b - a).perp();
    if (orientation(a, b, p[0]) > 0) orth = (a - b).perp();
    int id = extreme_vertex(p, orth, top);
    if (dot(p[id] - a, orth) > 0) return 0.0; //if orth and
        a are in the same half of the line, then poly and
        line intersects
    return dist_from_point_to_line(a, b, p[id]); //does not
        intersect
}
// minimum distance from a convex polygon to another
// convex polygon
// the polygon does not overlap or touch
// tested in https://toph.co/p/the-wall
double dist_from_polygon_to_polygon(vector<PT> &p1,
    vector<PT> &p2) { // O(n log n)
    double ans = inf;
    for (int i = 0; i < p1.size(); i++) {
        ans = min(ans, dist_from_point_to_polygon(p1,
            p1[i]));
    }
    for (int i = 0; i < p2.size(); i++) {
        ans = min(ans, dist_from_point_to_polygon(p1,
            p2[i]));
    }
    return ans;
}
// maximum distance from a convex polygon to another
// convex polygon
double maximum_dist_from_polygon_to_polygon(vector<PT> &u,
    vector<PT> &v){ // O(n)
    int n = (int)u.size(), m = (int)v.size();
    double ans = 0;
    if (n < 3 || m < 3) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) ans = max(ans,
                dist2(u[i], v[j]));
        }
        return sqrt(ans);
    }
    if (u[0].x > v[0].x) swap(n, m), swap(u, v);
    int i = 0, j = 0, step = n + m + 10;
```

```

while (j + 1 < m && v[j].x < v[j + 1].x) j++;
while (step--) {
    if (cross(u[(i + 1)%n] - u[i], v[(j + 1)%m] - v[j]) 
        >= 0) j = (j + 1) % m;
    else i = (i + 1) % n;
    ans = max(ans, dist2(u[i], v[j]));
}
return sqrt(ans);
}

```

3.27 PolygonLineIntersection

```

// not necessarily convex, boundary is included in the
// intersection
// returns total intersected length
double polygon_line_intersection(vector<PT> p, PT a, PT b)
{
    int n = p.size();
    p.push_back(p[0]);
    line l = line(a, b);
    double ans = 0.0;
    vector< pair<double, int> > vec;
    for (int i = 0; i < n; i++) {
        int s1 = sign(cross(b - a, p[i] - a));
        int s2 = sign(cross(b - a, p[i+1] - a));
        if (s1 == s2) continue;
        line t = line(p[i], p[i + 1]);
        PT inter = (t.v * l.c - l.v * t.c) / cross(l.v,
            t.v);
        double tmp = dot(inter, l.v);
        int f;
        if (s1 > s2) f = s1 && s2 ? 2 : 1;
        else f = s1 && s2 ? -2 : -1;
        vec.push_back(make_pair(tmp, f));
    }
    sort(vec.begin(), vec.end());
    for (int i = 0, j = 0; i + 1 < (int)vec.size(); i++){
        j += vec[i].second;
        if (j) ans += vec[i + 1].first - vec[i].first;
    }
    ans = ans / sqrt(dot(l.v, l.v));
    p.pop_back();
    return ans;
}

```

3.28 PolygonUnion

```

// calculates the area of the union of n polygons (not
// necessarily convex).
// the points within each polygon must be given in CCW
// order.
// complexity: O(N^2), where N is the total number of
// points
double rat(PT a, PT b, PT p) {
    return !sign(a.x - b.x) ? (p.y - a.y) / (b.y - a.y)
        : (p.x - a.x) / (b.x - a.x);
}

```

```

};

double polygon_union(vector<vector<PT>> &p) {
    int n = p.size();
    double ans=0;
    for(int i = 0; i < n; ++i) {
        for (int v = 0; v < (int)p[i].size(); ++v) {
            PT a = p[i][v], b = p[i][(v + 1) % p[i].size()];
            vector<pair<double, int>> segs;
            segs.emplace_back(0, 0), segs.emplace_back(1,
                0);
            for(int j = 0; j < n; ++j) {
                if(i != j) {
                    for(size_t u = 0; u < p[j].size(); ++u) {
                        PT c = p[j][u], d = p[j][(u + 1) %
                            p[j].size()];
                        int sc = sign(cross(b - a, c - a)),
                            sd = sign(cross(b - a, d - a));
                        if(!sc && !sd) {
                            if(sign(dot(b - a, d - c)) > 0 &&
                                i > j) {
                                segs.emplace_back(rat(a, b,
                                    c), 1),
                                segs.emplace_back(rat(a,
                                    b, d), -1);
                            }
                        } else {
                            double sa = cross(d - c, a - c),
                                sb = cross(d - c, b - c);
                            if(sc >= 0 && sd < 0)
                                segs.emplace_back(sa / (sa -
                                    sb), 1);
                            else if(sc < 0 && sd >= 0)
                                segs.emplace_back(sa / (sa -
                                    sb), -1);
                        }
                    }
                }
            }
            sort(segs.begin(), segs.end());
            double pre = min(max(segs[0].first, 0.0), 1.0),
                now, sum = 0;
            int cnt = segs[0].second;
            for(int j = 1; j < segs.size(); ++j) {
                now = min(max(segs[j].first, 0.0), 1.0);
                if (!cnt) sum += now - pre;
                cnt += segs[j].second;
                pre = now;
            }
            ans += cross(a, b) * sum;
        }
        return ans * 0.5;
    }
}

```

3.29 PolygonWidth

```

// Maximum distance between 2 points IN the polygon
double width(vector<PT> &p) {

```

```

    int n = (int)p.size();
    if (n <= 2) return 0;
    double ans = inf;
    int i = 0, j = 1;
    while (i < n) {
        while ((cross(p[(i + 1) % n] - p[i], p[(j + 1) % n] -
            p[j]) >= 0) j = (j + 1) % n;
        ans = min(ans, dist_from_point_to_line(p[i], p[(i +
            1) % n], p[j]));
        i++;
    }
    return ans;
}

```

3.30 Ray

```

// minimum distance from point c to ray (starting point a
// and direction vector b)
double dist_from_point_to_ray(PT a, PT b, PT c) {
    b = a + b;
    double r = dot(c - a, b - a);
    if (r < 0.0) return dist(c, a);
    return dist_from_point_to_line(a, b, c);
}

// starting point as and direction vector ad
bool ray_ray_intersection(PT as, PT ad, PT bs, PT bd) {
    double dx = bs.x - as.x, dy = bs.y - as.y;
    double det = bd.x * ad.y - bd.y * ad.x;
    if (fabs(det) < eps) return 0;
    double u = (dy * bd.x - dx * bd.y) / det;
    double v = (dy * ad.x - dx * ad.y) / det;
    if (sign(u) >= 0 && sign(v) >= 0) return 1;
    else return 0;
}

double ray_ray_distance(PT as, PT ad, PT bs, PT bd) {
    if (ray_ray_intersection(as, ad, bs, bd)) return 0.0;
    double ans = dist_from_point_to_ray(as, ad, bs);
    ans = min(ans, dist_from_point_to_ray(bs, bd, as));
    return ans;
}

```

3.31 Segment

```

// returns true if point p is on line segment ab
bool is_point_on_seg(PT a, PT b, PT p) {
    if (fabs(cross(p - b, a - b)) < eps) {
        if (p.x < min(a.x, b.x) || p.x > max(a.x, b.x))
            return false;
        if (p.y < min(a.y, b.y) || p.y > max(a.y, b.y))
            return false;
        return true;
    }
    return false;
}

// minimum distance point from point c to segment ab that
// lies on segment ab

```

```

PT project_from_point_to_seg(PT a, PT b, PT c) {
    double r = dist2(a, b);
    if (sign(r) == 0) return a;
    r = dot(c - a, b - a) / r;
    if (r < 0) return a;
    if (r > 1) return b;
    return a + (b - a) * r;
}

// minimum distance from point c to segment ab
double dist_from_point_to_seg(PT a, PT b, PT c) {
    return dist(c, project_from_point_to_seg(a, b, c));
}

// intersection point between segment ab and segment cd
// assuming unique intersection exists
bool seg_seg_intersection(PT a, PT b, PT c, PT d, PT &ans)
{
    double oa = cross2(c, d, a), ob = cross2(c, d, b);
    double oc = cross2(a, b, c), od = cross2(a, b, d);
    if (oa * ob < 0 && oc * od < 0){
        ans = (a * ob - b * oa) / (ob - oa);
        return 1;
    }
    else return 0;
}

// intersection point between segment ab and segment cd
// assuming unique intersection may not exists
// se.size()==0 means no intersection
// se.size()==1 means one intersection
// se.size()==2 means range intersection
set<PT> seg_seg_intersection_inside(PT a, PT b, PT c, PT
    d) {
    PT ans;
    if (seg_seg_intersection(a, b, c, d, ans)) return {ans};
    set<PT> se;
    if (is_point_on_seg(c, d, a)) se.insert(a);
    if (is_point_on_seg(c, d, b)) se.insert(b);
    if (is_point_on_seg(a, b, c)) se.insert(c);
    if (is_point_on_seg(a, b, d)) se.insert(d);
    return se;
}

// intersection between segment ab and line cd
// 0 if do not intersect, 1 if proper intersect, 2 if
// segment intersect
int seg_line_relation(PT a, PT b, PT c, PT d) {
    double p = cross2(c, d, a);
    double q = cross2(c, d, b);
    if (sign(p) == 0 && sign(q) == 0) return 2;
    else if (p * q < 0) return 1;
    else return 0;
}

// intersection between segment ab and line cd assuming
// unique intersection exists
bool seg_line_intersection(PT a, PT b, PT c, PT d, PT
    &ans) {
    bool k = seg_line_relation(a, b, c, d);
    assert(k != 2);
    if (k) line_line_intersection(a, b, c, d, ans);
    return k;
}

// minimum distance from segment ab to segment cd
double dist_from_seg_to_seg(PT a, PT b, PT c, PT d) {
    PT dummy;
}

```

```

    if (seg_seg_intersection(a, b, c, d, dummy)) return 0.0;
    else return min({dist_from_point_to_seg(a, b, c),
                    dist_from_point_to_seg(a, b, d),
                    dist_from_point_to_seg(c, d, a),
                    dist_from_point_to_seg(c, d, b)});
}

```

3.32 SmallestEnclosingCircle

```

double eps = 1e-9;
using Point = complex<double>;
struct Circle{ Point p; double r; };
double dist(Point p, Point q){ return abs(p-q); }
double area2(Point p, Point q){ return (conj(p)*q).imag(); }
bool in(const Circle& c, Point p){ return dist(c.p, p) <
    c.r + eps; }
Circle INVAL = Circle{Point(0, 0), -1};
Circle mCC(Point a, Point b, Point c){
    b -= a; c -= a;
    double d = 2*(conj(b)*c).imag(); if(abs(d)<eps)
        return INVAL;
    Point ans = (c*norm(b) - b*norm(c)) * Point(0, -1)
        / d;
    return Circle{a + ans, abs(ans)};
}
Circle solve(vector<Point> p) {
    mt19937 gen(0x94949);
    shuffle(p.begin(), p.end(),
        gen);
    Circle c = INVAL;
    for(int i=0; i<p.size(); ++i) if(c.r<0 || !in(c,
        p[i])){
        c = Circle{p[i], 0};
        for(int j=0; j<i; ++j) if(!in(c, p[j])){
            Circle ans{(p[i]+p[j])*0.5,
                dist(p[i], p[j])*0.5};
            if(c.r == 0) {c = ans; continue;}
            Circle l, r; l = r = INVAL;
            Point pq = p[j]-p[i];
            for(int k=0; k<=j; ++k) if(!in(ans,
                p[k])){
                double a2 = area2(pq,
                    p[k]-p[i]);
                Circle c = mCC(p[i], p[j],
                    p[k]);
                if(c.r<0) continue;
                else if(a2 > 0 &&
                    (l.r<0||area2(pq,
                        c.p-p[i]) > area2(pq,
                        l.p-p[i]))) l = c;
                else if(a2 < 0 &&
                    (r.r<0||area2(pq,
                        c.p-p[i]) < area2(pq,
                        r.p-p[i]))) r = c;
            }
            if(l.r<0&&r.r<0) c = ans;
            else if(l.r<0) c = r;
            else if(r.r<0) c = l;
            else c = l.r<=r.r?l:r;
        }
    }
}

```

```

    }
    return c;
}

```

3.33 TriangleCircleIntersection

```

// system should be translated from circle center
double triangle_circle_intersection(PT c, double r, PT a,
    PT b) {
    double sd1 = dist2(c, a), sd2 = dist2(c, b);
    if(sd1 > sd2) swap(a, b), swap(sd1, sd2);
    double sd = dist2(a, b);
    double d1 = sqrtl(sd1), d2 = sqrtl(sd2), d = sqrt(sd);
    double x = abs(sd2 - sd - sd1) / (2 * d);
    double h = sqrtl(sd1 - x * x);
    if(r >= d2) return h * d / 2;
    double area = 0;
    if(sd + sd1 < sd2) {
        if(r < d1) area = r * r * (acos(h / d2) - acos(h /
            d1)) / 2;
        else {
            area = r * r * (acos(h / d2) - acos(h / r)) /
            2;
            double y = sqrtl(r * r - h * h);
            area += h * (y - x) / 2;
        }
    } else {
        if(r < h) area = r * r * (acos(h / d2) + acos(h /
            d1)) / 2;
        else {
            area += r * r * (acos(h / d2) - acos(h / r)) /
            2;
            double y = sqrtl(r * r - h * h);
            area += h * y / 2;
            if(r < d1) {
                area += r * r * (acos(h / d1) - acos(h / r))
                / 2;
                area += h * y / 2;
            }
            else area += h * x / 2;
        }
    }
    return area;
}

```

3.34 Utilities

```

double perimeter(vector<PT> &p) {
    double ans=0; int n = p.size();
    for (int i = 0; i < n; i++) ans += dist(p[i], p[(i + 1)
        % n]);
    return ans;
}

double area(vector<PT> &p) {
    double ans = 0; int n = p.size();

```

```

for (int i = 0; i < n; i++) ans += cross(p[i], p[(i + 1) % n]);
return fabs(ans) * 0.5;
}
double area_of_triangle(PT a, PT b, PT c) {
    return fabs(cross(b - a, c - a) * 0.5);
}
// 0 if cw, 1 if ccw
bool get_direction(vector<PT> &p) {
    double ans = 0; int n = p.size();
    for (int i = 0; i < n; i++) ans += cross(p[i], p[(i + 1) % n]);
    if (sign(ans) > 0) return 1;
    return 0;
}
// find a point from a through b with distance d
PT point_along_line(PT a, PT b, double d) {
    assert(a != b);
    return a + ((b - a) / (b - a).norm()) * d;
}
// projection point c onto line through a and b assuming a != b
PT project_from_point_to_line(PT a, PT b, PT c) {
    return a + (b - a) * dot(c - a, b - a) / (b - a).norm2();
}
// reflection point c onto line through a and b assuming a != b
PT reflection_from_point_to_line(PT a, PT b, PT c) {
    PT p = project_from_point_to_line(a,b,c);
    return p + p - c;
}
// minimum distance from point c to line through a and b
double dist_from_point_to_line(PT a, PT b, PT c) {
    return fabs(cross(b - a, c - a) / (b - a).norm());
}
// 0 if not parallel, 1 if parallel, 2 if collinear
int is_parallel(PT a, PT b, PT c, PT d) {
    double k = fabs(cross(b - a, d - c));
    if (k < eps){
        if (fabs(cross(a - b, a - c)) < eps && fabs(cross(c - d, c - a)) < eps) return 2;
        else return 1;
    }
    else return 0;
}
// check if two lines are same
bool are_lines_same(PT a, PT b, PT c, PT d) {
    if (fabs(cross(a - c, c - d)) < eps && fabs(cross(b - c, c - d)) < eps) return true;
    return false;
}

// 1 if point is ccw to the line, 2 if point is cw to the line, 3 if point is on the line
int point_line_relation(PT a, PT b, PT p) {
    int c = sign(cross(p - a, b - a));
    if (c < 0) return 1;
    if (c > 0) return 2;
    return 3;
}

```

4 Graph

4.1 2SAT

```

struct TwoSatSolver {
    int n_vars;
    int n_vertices;
    vector<vector<int>> adj, adj_t;
    vector<bool> used;
    vector<int> order, comp;
    vector<bool> assignment;
    TwoSatSolver(int _n_vars)
        : n_vars(_n_vars),
        n_vertices(2 * n_vars),
        adj(n_vertices),
        adj_t(n_vertices),
        used(n_vertices),
        order(),
        comp(n_vertices, -1),
        assignment(n_vars) {
            order.reserve(n_vertices);
        }
    void dfs1(int v) {
        used[v] = true;
        for (int u : adj[v]) {
            if (!used[u]) dfs1(u);
        }
        order.push_back(v);
    }
    void dfs2(int v, int cl) {
        comp[v] = cl;
        for (int u : adj_t[v]) {
            if (comp[u] == -1) dfs2(u, cl);
        }
    }
    bool solve_2SAT() {
        order.clear();
        used.assign(n_vertices, false);
        for (int i = 0; i < n_vertices; ++i) {
            if (!used[i]) dfs1(i);
        }

        comp.assign(n_vertices, -1);
        for (int i = 0, j = 0; i < n_vertices; ++i) {
            int v = order[n_vertices - i - 1];
            if (comp[v] == -1) dfs2(v, j++);
        }

        assignment.assign(n_vars, false);
        for (int i = 0; i < n_vertices; i += 2) {
            if (comp[i] == comp[i + 1]) return false;
            assignment[i / 2] = comp[i] > comp[i + 1];
        }
        return true;
    }
    void add_disjunction(int a, bool na, int b, bool nb) {
        // na and nb signify whether a and b are to be negated
        a = 2 * a ^ na;
        b = 2 * b ^ nb;
        int neg_a = a ^ 1;
        int neg_b = b ^ 1;
        adj[neg_a].push_back(b);
        adj[neg_b].push_back(a);
        adj_t[b].push_back(neg_a);
        adj_t[a].push_back(neg_b);
    }
    static void example_usage() {
        TwoSatSolver solver(3); // a, b, c
        solver.add_disjunction(0, false, 1, true); // a v not b
        solver.add_disjunction(0, true, 1, true); // not a v not b
        solver.add_disjunction(1, false, 2, false); // b v c
        solver.add_disjunction(0, false, 0, false); // a v a
        assert(solver.solve_2SAT() == true);
        auto expected = vector<bool>(True, False, True);
        assert(solver.assignment == expected);
    }
};

```

4.2 Dinic

```

// Worst-case time complexity: O(V^2 * E)
// Unit-capacity networks (all capacities = 1):
// O(min(V^{2/3}, sqrt(E)) * E).
// Bipartite matching (as a special case): O(sqrt(V) * E).
// (often near O(E * sqrt(V)) or O(E * V) in practice
// depending on structure).
#define int long long
const int inf = 1e18;
struct dinic {
    struct edge {
        int v, oth, fl, cap;
    };
    vector<vector<edge>> adj;
    vector<int> ptr, lv;
    dinic() {}
    int st, en;
    dinic(int n) {
        st = n + 1;
        en = n + 2;
        adj.assign(n + 7, vector<edge>());
        ptr.assign(n + 7, 0);
        lv.assign(n + 7, 0);
    }
    void add(int u, int v, int cap, int undirected = 0) {
        adj[u].push_back({v, (int)adj[v].size(), 0, cap});
        adj[v].push_back({u, (int)adj[u].size() - 1, 0,
                         undirected * cap});
    }
    bool bfs() {
        fill(lv.begin(), lv.end(), -1);
        lv[st] = 0;
        queue<int> q;
        q.push(st);
        while (q.size() && lv[en] == -1) {
            int u = q.front();
            q.pop();

```

```

    for (edge& x : adj[u]) {
        if (lv[x.v] == -1 && x.cap > x.fl) {
            q.push(x.v);
            lv[x.v] = lv[u] + 1;
        }
    }
    return lv[en] != -1;
}
int dfs(int u, int flowin) {
    if (u == en) return flowin;
    int flout = 0;
    for (; ptr[u] < (int)adj[u].size(); ptr[u]++) {
        edge& x = adj[u][ptr[u]];
        if (x.cap > x.fl && lv[x.v] == lv[u] + 1) {
            int tmp = dfs(x.v, min(flowin, x.cap - x.fl));
            x.fl += tmp;
            adj[x.v][x.oth].fl -= tmp;
            flout += tmp;
            if (flowin == 0) break;
        }
    }
    return flout;
}
int max_flow() {
    int res = 0;
    while (bfs()) {
        fill(ptr.begin(), ptr.end(), 0);
        res += dfs(st, inf);
    }
    return res;
}
};
```

4.3 EulerPath

```

struct EulerUndirected {
    EulerUndirected(int _n) : n(_n), m(0), adj(_n), deg(_n, 0) {}
    void add_edge(int u, int v) {
        adj[u].push_front(Edge(v));
        auto it1 = adj[u].begin();
        adj[v].push_front(Edge(u));
        auto it2 = adj[v].begin();
        it1->rev = it2;
        it2->rev = it1;
        ++deg[u];
        ++deg[v];
        ++m;
    }
    pair<bool, vector<int>> solve() {
        int cntOdd = 0;
        int start = -1;
        for (int i = 0; i < n; i++) {
            if (deg[i] % 2) {
                ++cntOdd;
```

```

                if (cntOdd > 2) return {false, {}};
                if (start < 0) start = i;
            }
            // no odd vertex -> start from any vertex with positive degree
            if (start < 0) {
                for (int i = 0; i < n; i++) {
                    if (deg[i]) {
                        start = i;
                        break;
                    }
                }
                if (start < 0) {
                    // no edge -> empty path
                    return {true, {}};
                }
            }
            vector<int> path;
            find_path(start, path);

            if (m + 1 != static_cast<int>(path.size())) {
                return {false, {}};
            }

            return {true, path};
        }
        struct Edge {
            int to;
            list<Edge>::iterator rev;

            Edge(int _to) : to(_to) {}

            // private:
            int n, m;
            vector<list<Edge>> adj;
            vector<int> deg;
        };
        void find_path(int v, vector<int>& path) {
            while (adj[v].size() > 0) {
                int next = adj[v].front().to;
                adj[next].erase(adj[v].front().rev);
                adj[v].pop_front();
                find_path(next, path);
            }
            path.push_back(v);
        }
    };
};
```

4.4 EulerPathDirected

```

struct EulerDirected {
    EulerDirected(int _n) : n(_n), adj(n), in_deg(n, 0),
                        out_deg(n, 0) {}
    void add_edge(int u, int v) { // directed edge
        assert(0 <= u && u < n);
        assert(0 <= v && v < n);
```

```

        adj[u].push_front(v);
        in_deg[v]++;
        out_deg[u]++;
    }
    pair<bool, vector<int>> solve() {
        int start = -1, last = -1;
        for (int i = 0; i < n; i++) {
            // for all u, |in_deg(u) - out_deg(u)| <= 1
            if (abs(in_deg[i] - out_deg[i]) > 1) return {false, {}};

            if (out_deg[i] > in_deg[i]) {
                // At most 1 vertex with out_deg[u] - in_deg[u] = 1 (start vertex)
                if (start >= 0) return {false, {}};
                start = i;
            }

            if (in_deg[i] > out_deg[i]) {
                // At most 1 vertex with in_deg[u] - out_deg[u] = 1 (last vertex)
                if (last >= 0) return {false, {}};
                last = i;
            }
        }

        // can start at any vertex with degree > 0
        if (start < 0) {
            for (int i = 0; i < n; i++) {
                if (in_deg[i]) {
                    start = i;
                    break;
                }
            }
            // no start vertex --> all vertices have degree == 0
            if (start < 0) return {true, {}};
        }
        vector<int> path;
        find_path(start, path);
        reverse(path.begin(), path.end());

        // check that we visited all vertices with degree > 0
        vector<bool> visited(n, false);
        for (int u : path) visited[u] = true;

        for (int u = 0; u < n; u++) {
            if (in_deg[u] && !visited[u]) {
                return {false, {}};
            }
        }
        return {true, path};
    }
    private:
    int n;
    vector<list<int>> adj;
    vector<int> in_deg, out_deg;

    void find_path(int v, vector<int>& path) {
        while (adj[v].size() > 0) {
            int next = adj[v].front();
            adj[v].pop_front();
            adj[next].push_front(v);
```

```

    find_path(next, path);
}
path.push_back(v);
}
};
```

4.5 GeneralMatching

```

const int MAXN = 2020 + 1;
struct GM { // 1-based Vertex index
    int vis[MAXN], par[MAXN], orig[MAXN], match[MAXN],
        aux[MAXN], t, N;
    vector<int> conn[MAXN];
    queue<int> Q;
    void addEdge(int u, int v) {
        conn[u].push_back(v);
        conn[v].push_back(u);
    }
    void init(int n) {
        N = n;
        t = 0;
        for (int i = 0; i <= n; ++i) {
            conn[i].clear();
            match[i] = aux[i] = par[i] = 0;
        }
    }
    void augment(int u, int v) {
        int pv = v, nv;
        do {
            pv = par[v];
            nv = match[pv];
            match[v] = pv;
            match[pv] = v;
            v = nv;
        } while (u != pv);
    }
    int lca(int v, int w) {
        ++t;
        while (true) {
            if (v) {
                if (aux[v] == t) return v;
                aux[v] = t;
                v = orig[par[match[v]]];
            }
            swap(v, w);
        }
    }
    void blossom(int v, int w, int a) {
        while (orig[v] != a) {
            par[v] = w;
            w = match[v];
            if (vis[w] == 1) Q.push(w), vis[w] = 0;
            orig[v] = orig[w] = a;
            v = par[w];
        }
    }
    bool bfs(int u) {
        fill(vis + 1, vis + 1 + N, -1);
        iota(orig + 1, orig + N + 1, 1);
```

```

    Q = queue<int>();
    Q.push(u);
    vis[u] = 0;
    while (!Q.empty()) {
        int v = Q.front();
        Q.pop();
        for (int x : conn[v]) {
            if (vis[x] == -1) {
                par[x] = v;
                vis[x] = 1;
                if (!match[x]) return augment(u, x), true;
                Q.push(match[x]);
                vis[match[x]] = 0;
            } else if (vis[x] == 0 && orig[v] != orig[x]) {
                int a = lca(orig[v], orig[x]);
                blossom(x, v, a);
                blossom(v, x, a);
            }
        }
        return false;
    }
    int Match() {
        int ans = 0;
        // find random matching (not necessary, constant
        // improvement)
        vector<int> V(N - 1);
        iota(V.begin(), V.end(), 1);
        shuffle(V.begin(), V.end(), mt19937(0x94949));
        for (auto x : V)
            if (!match[x]) {
                for (auto y : conn[x])
                    if (!match[y]) {
                        match[x] = y, match[y] = x;
                        ++ans;
                        break;
                    }
                for (int i = 1; i <= N; ++i)
                    if (!match[i] && bfs(i)) ++ans;
            }
        return ans;
    }
};
```

4.6 GlobalMinCut

```

pair<int, vi> GetMinCut(vector<vi>& weights) {
    int N = sz(weights);
    vi used(N), cut, best_cut;
    int best_weight = -1;
    for (int phase = N - 1; phase >= 0; phase--) {
        vi w = weights[0], added = used;
        int prev, k = 0;
        rep(i, 0, phase) {
            prev = k;
            k = -1;
            rep(j, 1, N) if (!added[j] && (k == -1 || w[j] >
                w[k])) k = j;
            if (i == phase - 1) {
```

```

                rep(j, 0, N) weights[prev][j] += weights[k][j];
                rep(j, 0, N) weights[j][prev] = weights[prev][j];
                used[k] = true;
                cut.push_back(k);
                if (best_weight == -1 || w[k] < best_weight) {
                    best_cut = cut;
                    best_weight = w[k];
                }
            } else {
                rep(j, 0, N) w[j] += weights[k][j];
                added[k] = true;
            }
        }
        return {best_weight, best_cut};
    }
}
```

4.7 HopcroftKarp

```

struct maximum_bipartite_matching {
    int n, m;
    vector<int> matchX, matchY, dist;
    vector<vector<int>> g;
    int matched;
    maximum_bipartite_matching(int _n, int _m)
        : n(_n), m(_m), matchX(n + 1, -1), matchY(m + 1, -1),
        dist(n + 1, -1), g(n + 1), matched(0) {}

    void add_edge(int u, int v) { g[u].push_back(v); }
    void bfs() {
        queue<int> q;
        q.push(0);
        dist[0] = 0;
        for (int i = 1; i < n; ++i) {
            if (matchX[i] == -1)
                q.push(i), dist[i] = 0;
            else
                dist[i] = -1;
        }
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int v : g[u])
                if (matchY[v] == -1)
                    dist[matchY[v]] = dist[u] + 1;
                q.push(matchY[v]);
        }
    }
    bool dfs(int u) {
        for (int v : g[u])
            if (matchY[v] == -1) {
                matchX[u] = v, matchY[v] = u;
                return true;
            }
        for (int v : g[u])
            if (dist[matchY[v]] == dist[u] + 1)
                dfs(matchY[v]);
        matchX[u] = v, matchY[v] = u;
```

```

        return 1;
    }
    return 0;
}

void match() {
    while (1) {
        bfs();
        int augment = 0;
        for (int i = 0; i < n; ++i)
            if (matchX[i] == -1) augment += dfs(i);
        if (!augment) break;
        matched += augment;
    }
}

vector<pii> get_edges() {
    vector<pii> res;
    for (int i = 0; i < n; ++i)
        if (matchX[i] != -1) res.push_back({i, matchX[i]});
    return res;
}

```

4.8 JointsnBridges

```

const int maxN = 10010;
int n, m;
bool joint[maxN];
int timeDfs = 0, bridge = 0;
int low[maxN], num[maxN];
vector<int> g[maxN];
void dfs(int u, int pre) {
    int child = 0;
    num[u] = low[u] = ++timeDfs;
    for (int v : g[u]) {
        if (v == pre) continue;
        if (!num[v]) {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] == num[v]) bridge++;
            child++;
            if (u == pre) { // if u is root of DFS tree
                if (child > 1) joint[u] = true;
            }
        } else if (low[v] >= num[u]) joint[u] = true;
        else low[u] = min(low[u], num[v]);
    }
}
int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int u, v;
        cin >> u >> v;
        g[u].push_back(v);
        g[v].push_back(u);
    }
    for (int i = 1; i <= n; i++)

```

```

        if (!num[i]) dfs(i, i);
        int cntJoint = 0;
        for (int i = 1; i <= n; i++) cntJoint += joint[i];
        cout << cntJoint << ' ' << bridge;
    }

```

4.9 MCMF slow

```

// Time Complexity: O(F * (V * E))
template <typename T, typename C>
class mcmf {
public:
    static constexpr T eps = (T) 1e-9;

    struct edge {
        int from;
        int to;
        T c;
        T f;
        C cost;
    };

    vector<vector<int>> g;
    vector<edge> edges;
    vector<C> d;
    vector<int> q;
    vector<bool> in_queue;
    vector<int> pe;
    int n;
    int st, fin;
    T flow;
    C cost;

    mcmf(int _n, int _st, int _fin) : n(_n), st(_st),
                                         fin(_fin) {
        assert(st <= _n && st < n && 0 <= fin && fin < n && st
               != fin);
        g.resize(n);
        d.resize(n);
        in_queue.resize(n);
        pe.resize(n);
        flow = 0;
        cost = 0;
    }

    void clear_flow() {
        for (const edge &e : edges) {
            e.f = 0;
        }
        flow = 0;
    }

    void add(int from, int to, T forward_cap, T
             backward_cap, C cost) {
        assert(0 <= from && from < n && 0 <= to && to < n);
        g[from].push_back((int) edges.size());
        edges.push_back({from, to, forward_cap, 0, cost});
        g[to].push_back((int) edges.size());
        edges.push_back({to, from, backward_cap, 0, -cost});
    }

```

```

    }

    bool expath() {
        fill(d.begin(), d.end(), numeric_limits<C>::max());
        q.clear();
        q.push_back(st);
        d[st] = 0;
        in_queue[st] = true;
        int beg = 0;
        bool found = false;
        while (beg < (int) q.size()) {
            int i = q[beg++];
            if (i == fin) {
                found = true;
            }
            in_queue[i] = false;
            for (int id : g[i]) {
                const edge &e = edges[id];
                if (e.c - e.f > eps && d[i] + e.cost < d[e.to]) {
                    d[e.to] = d[i] + e.cost;
                    pe[e.to] = id;
                    if (!in_queue[e.to]) {
                        q.push_back(e.to);
                        in_queue[e.to] = true;
                    }
                }
            }
        }
        if (found) {
            T push = numeric_limits<T>::max();
            int v = fin;
            while (v != st) {
                const edge &e = edges[pe[v]];
                push = min(push, e.c - e.f);
                v = e.from;
            }
            v = fin;
            while (v != st) {
                edge &e = edges[pe[v]];
                e.f += push;
                edge &back = edges[pe[v] ^ 1];
                back.f -= push;
                v = e.from;
            }
            flow += push;
            cost += push * d[fin];
        }
        return found;
    }

    pair<T, C> max_flow_min_cost() {
        while (expath()) {}
        return {flow, cost};
    }
}

```

4.10 MCMF

```
// Time Complexity: O(F * (E * log V))
```

```

template <typename T, typename C>
class MCMF {
public:
    static constexpr T eps = (T) 1e-9;

    struct edge {
        int from;
        int to;
        T c;
        T f;
        C cost;
    };

    int n;
    vector<vector<int>> g;
    vector<edge> edges;
    vector<C> d;
    vector<C> pot;
    --gnu_pbds::priority_queue<pair<C, int>> q;
    vector<typename decltype(q)::point_iterator> its;
    vector<int> pe;
    const C INF_C = numeric_limits<C>::max() / 2;

    explicit MCMF(int n_) : n(n_), g(n), d(n), pot(n, 0),
        its(n), pe(n) {}

    int add(int from, int to, T forward_cap, T backward_cap,
        C edge_cost) {
        assert(0 <= from && from < n && 0 <= to && to < n);
        assert(forward_cap >= 0 && backward_cap >= 0);
        int id = static_cast<int>(edges.size());
        g[from].push_back(id);
        edges.push_back({from, to, forward_cap, 0, edge_cost});
        g[to].push_back(id + 1);
        edges.push_back({to, from, backward_cap, 0,
            -edge_cost});
        return id;
    }

    void expath(int st) {
        fill(d.begin(), d.end(), INF_C);
        q.clear();
        fill(its.begin(), its.end(), q.end());
        its[st] = q.push({pot[st], st});
        d[st] = 0;
        while (!q.empty()) {
            int i = q.top().second;
            q.pop();
            its[i] = q.end();
            for (int id : g[i]) {
                const edge &e = edges[id];
                int j = e.to;
                if (e.c - e.f > eps && d[i] + e.cost < d[j]) {
                    d[j] = d[i] + e.cost;
                    pe[j] = id;
                    if (its[j] == q.end()) {
                        its[j] = q.push({pot[j] - d[j], j});
                    } else {
                        q.modify(its[j], {pot[j] - d[j], j});
                    }
                }
            }
        }
    }
};

pair<T, C> max_flow_min_cost(int st, int fin) {
    T flow = 0;
    C cost = 0;
    bool ok = true;
    for (auto& e : edges) {
        if (e.c - e.f > eps && e.cost + pot[e.from] -
            pot[e.to] < 0) {
            ok = false;
            break;
        }
    }
    if (ok) {
        expath(st);
    } else {
        vector<int> deg(n, 0);
        for (int i = 0; i < n; i++) {
            for (int eid : g[i]) {
                auto& e = edges[eid];
                if (e.c - e.f > eps) {
                    deg[e.to] += 1;
                }
            }
        }
        vector<int> que;
        for (int i = 0; i < n; i++) {
            if (deg[i] == 0) {
                que.push_back(i);
            }
        }
        for (int b = 0; b < (int) que.size(); b++) {
            for (int eid : g[que[b]]) {
                auto& e = edges[eid];
                if (e.c - e.f > eps) {
                    deg[e.to] -= 1;
                    if (deg[e.to] == 0) {
                        que.push_back(e.to);
                    }
                }
            }
        }
        fill(pot.begin(), pot.end(), INF_C);
        pot[st] = 0;
        if (static_cast<int>(que.size()) == n) {
            for (int v : que) {
                if (pot[v] < INF_C) {
                    for (int eid : g[v]) {
                        auto& e = edges[eid];
                        if (e.c - e.f > eps) {
                            if (pot[v] + e.cost < pot[e.to]) {
                                pot[e.to] = pot[v] + e.cost;
                                pe[e.to] = eid;
                            }
                        }
                    }
                }
            }
        } else {
            que.assign(1, st);
        }
    }
}

```

```

    }
    swap(d, pot);
}

pair<T, C> max_flow_min_cost(int st, int fin) {
    T flow = 0;
    C cost = 0;
    bool ok = true;
    for (auto& e : edges) {
        if (e.c - e.f > eps && e.cost + pot[e.from] -
            pot[e.to] < 0) {
            ok = false;
            break;
        }
    }
    if (ok) {
        expath(st);
    } else {
        vector<int> deg(n, 0);
        for (int i = 0; i < n; i++) {
            for (int eid : g[i]) {
                auto& e = edges[eid];
                if (e.c - e.f > eps) {
                    deg[e.to] += 1;
                }
            }
        }
        vector<int> que;
        for (int i = 0; i < n; i++) {
            if (deg[i] == 0) {
                que.push_back(i);
            }
        }
        for (int b = 0; b < (int) que.size(); b++) {
            for (int eid : g[que[b]]) {
                auto& e = edges[eid];
                if (e.c - e.f > eps) {
                    deg[e.to] -= 1;
                    if (deg[e.to] == 0) {
                        que.push_back(e.to);
                    }
                }
            }
        }
        fill(pot.begin(), pot.end(), INF_C);
        pot[st] = 0;
        if (static_cast<int>(que.size()) == n) {
            for (int v : que) {
                if (pot[v] < INF_C) {
                    for (int eid : g[v]) {
                        auto& e = edges[eid];
                        if (e.c - e.f > eps) {
                            if (pot[v] + e.cost < pot[e.to]) {
                                pot[e.to] = pot[v] + e.cost;
                                pe[e.to] = eid;
                            }
                        }
                    }
                }
            }
        } else {
            que.assign(1, st);
        }
    }
}

```

```

vector<bool> in_queue(n, false);
in_queue[st] = true;
for (int b = 0; b < (int) que.size(); b++) {
    int i = que[b];
    in_queue[i] = false;
    for (int id : g[i]) {
        const edge &e = edges[id];
        if (e.c - e.f > eps && pot[i] + e.cost <
            pot[e.to]) {
            pot[e.to] = pot[i] + e.cost;
            pe[e.to] = id;
            if (!in_queue[e.to]) {
                que.push_back(e.to);
                in_queue[e.to] = true;
            }
        }
    }
}
while (pot[fin] < INF_C) {
    T push = numeric_limits<T>::max();
    int v = fin;
    while (v != st) {
        const edge &e = edges[pe[v]];
        push = min(push, e.c - e.f);
        v = e.from;
    }
    v = fin;
    while (v != st) {
        edge &e = edges[pe[v]];
        e.f += push;
        edge &back = edges[pe[v] ^ 1];
        back.f -= push;
        v = e.from;
    }
    flow += push;
    cost += push * pot[fin];
    expath(st);
}
return {flow, cost};
}

```

4.11 SCC

```

struct DirectedDfs {
    vector<vector<int>> g;
    int n;
    vector<int> num, low, current, S;
    int counter;
    vector<int> comp_ids;
    vector<vector<int>> scc;

DirectedDfs(const vector<vector<int>>& _g)
: g(_g),
n(g.size()),
num(n, -1),
low(n, 0),

```

```

        current(n, 0),
        counter(0),
        comp_ids(n, -1) {
    for (int i = 0; i < n; i++) {
        if (num[i] == -1) dfs(i);
    }
}

void dfs(int u) {
    low[u] = num[u] = counter++;
    S.push_back(u);
    current[u] = 1;
    for (auto v : g[u]) {
        if (num[v] == -1) dfs(v);
        if (current[v]) low[u] = min(low[u], low[v]);
    }
    if (low[u] == num[u]) {
        scc.push_back(vector<int>());
        while (1) {
            int v = S.back();
            S.pop_back();
            current[v] = 0;
            scc.back().push_back(v);
            comp_ids[v] = ((int)scc.size()) - 1;
            if (u == v) break;
        }
    }
}

// build DAG of strongly connected components
// Returns: adjacency list of DAG
std::vector<std::vector<int>> build_scc_dag() {
    std::vector<std::vector<int>> dag(scc.size());
    for (int u = 0; u < n; u++) {
        int x = comp_ids[u];
        for (int v : g[u]) {
            int y = comp_ids[v];
            if (x != y) {
                dag[x].push_back(y);
            }
        }
    }
    return dag;
}

```

4.12 TopoSort

```

pair<bool, vector<int>> topo_sort(const
    vector<vector<int>>& g) {
    int n = g.size();
    // init in_deg
    vector<int> in_deg(n, 0);
    for (int u = 0; u < n; u++) {
        for (int v : g[u]) {
            in_deg[v]++;
        }
    }
}

```

```

// find topo order
vector<int> res;
queue<int> qu;
for (int u = 0; u < n; u++) {
    if (in_deg[u] == 0) {
        qu.push(u);
    }
}

while (!qu.empty()) {
    int u = qu.front();
    qu.pop();
    res.push_back(u);
    for (int v : g[u]) {
        in_deg[v]--;
        if (in_deg[v] == 0) {
            qu.push(v);
        }
    }
}

if ((int)res.size() < n) {
    return {false, {}};
}
return {true, res};
}

```

4.13 spfa

```

#include<bits/stdc++.h>
typedef pair<int, int> ii;
const int MaxN = 1e5 + 5;
const int Inf = 1e9;
vector<vector<ii>> AdjList;
int Dist[MaxN];
int Cnt[MaxN];
bool inqueue[MaxN];
int S;
int N;
queue<int> q;

bool spfa() {
    for(int i = 1 ; i <= N ; i++) {
        Dist[i] = Inf;
        Cnt[i] = 0;
        inqueue[i] = false;
    }
    Dist[S] = 0;
    q.push(S);
    inqueue[S] = true;
    while(!q.empty()) {
        int u = q.front();
        q.pop();
        inqueue[u] = false;
        for (ii tmp: AdjList[u]) {
            int v = tmp.first;
            int w = tmp.second;

```

```

            if (Dist[u] + w < Dist[v]) {
                Dist[v] = Dist[u] + w;
                if (!inqueue[v]) {
                    q.push(v);
                    inqueue[v] = true;
                    Cnt[v]++;
                    if (Cnt[v] > N)
                        return false;
                }
            }
        }
        return true;
}

```

5 Math

5.1 Euclid

```

// x, y such that ax + by = gcd(a, b)
ll gcd(ll a, ll b) { return __gcd(a, b); }
ll euclid(ll a, ll b, ll &x, ll &y) {
    if (b) {
        ll d = euclid(b, a % b, y, x);
        return y -= a / b * x, d;
    }
    return x = 1, y = 0, a;
}

```

5.2 FFT

```

using ld = double;
// Can use std::complex<ld> instead to make code shorter
// (but it will be slightly slower)
struct Complex {
    ld x[2];

    Complex() { x[0] = x[1] = 0.0; }
    Complex(ld a) { x[0] = a; }
    Complex(ld a, ld b) {
        x[0] = a;
        x[1] = b;
    }
    Complex(const std::complex<ld>& c) {
        x[0] = c.real();
        x[1] = c.imag();
    }

    Complex conj() const { return Complex(x[0], -x[1]); }

    Complex operator+(const Complex& c) const {
        return Complex{
            x[0] + c.x[0],
            x[1] + c.x[1],
        };
    }
}

```

```

}

Complex operator-(const Complex& c) const {
    return Complex{
        x[0] - c.x[0],
        x[1] - c.x[1],
    };
}

Complex operator*(const Complex& c) const { return
    Complex(x[0] * c.x[0] - x[1] * c.x[1], x[0] *
    c.x[1] + x[1] * c.x[0]); }

Complex& operator+=(const Complex& c) { return *this =
    *this + c; }

Complex& operator=(const Complex& c) { return *this =
    *this - c; }

Complex& operator*=(const Complex& c) { return *this =
    *this * c; }

};

void fft(vector<Complex>& a) {
    int n = a.size();
    int L = 31 - __builtin_clz(n);
    static vector<Complex> R(2, 1);
    static vector<Complex> rt(2, 1);
    for (static int k = 2; k < n; k *= 2) {
        R.resize(n);
        rt.resize(n);
        auto x = Complex(polar(ld(1.0), acos(ld(-1.0)) / 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 = (ld*)&rt[j + k].x, y = (ld*)&a[i + j +
                    k].x;
                Complex 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<ld> multiply(const vector<ld>& a, const vector<ld>&
        b) {
        if (a.empty() || b.empty()) return {};
        vector<ld> res(a.size() + b.size() - 1);
        int L = 32 - __builtin_clz(res.size()), n = 1 << L;
        vector<Complex> in(n), out(n);

        for (size_t i = 0; i < a.size(); ++i) in[i].x[0] = a[i];
        for (size_t i = 0; i < b.size(); ++i) in[i].x[1] = b[i];

        fft(in);
        for (Complex& x : in) x *= x;
    }
}

```

```

        for (int i = 0; i < n; ++i) out[i] = in[-i & (n - 1)] -
            in[i].conj();
        fft(out);

        for (size_t i = 0; i < res.size(); ++i) res[i] =
            out[i].x[1] / (4 * n);
        return res;
    }

    long long my_round(ld x) {
        if (x < 0) return -my_round(-x);
        return (long long)(x + 1e-2);
    }

    vector<long long> multiply(const vector<int>& a, const
        vector<int>& b) {
        vector<ld> ad(a.begin(), a.end());
        vector<ld> bd(b.begin(), b.end());
        auto rd = multiply(ad, bd);
        vector<long long> res(rd.size());
        for (int i = 0; i < (int)res.size(); ++i) {
            res[i] = my_round(rd[i]);
        }
        return res;
    }
}

```

5.3 Factorization

```

inline long long qpow(long long a, int b) {
    long long ans = 1;
    while (b) {
        if (b & 1) ans = ans * a % mod;
        a = a * a % mod;
        b >>= 1;
    }
    return ans;
}

inline long long rv(int x) { return qpow(x, mod - 2) %
    mod; }

bool is_prime(long long n) {
    if (n <= 1) return false;
    for (int a : {2, 3, 5, 13, 19, 73, 193, 407521,
        299210837}) {
        if (n == a) return true;
        if (n % a == 0) return false;
    }
    long long d = n - 1;
    while (!(d & 1)) d >>= 1;
    for (int a : {2, 325, 9375, 28178, 450775, 9780504,
        1795265022}) {
        long long t = d, y = ipow(a, t, n);
        while (t != n - 1 && y != 1 && y != n - 1) y = mul(y,
            y, n), t <<= 1;
        if (y != n - 1 && !(t & 1)) return false;
    }
    return true;
}

long long pollard(long n) {
    auto f = [n](long x) { return mul(x, x, n) + 1; };

```

```

    long long x = 0, y = 0, t = 0, prd = 2, i = 1, q;
    while (t++ % 40 || gcd(prd, n) == 1) {
        if (x == y) x = ++i, y = f(x);
        if ((q = mul(prd, max(x, y) - min(x, y), n))) prd = q;
        x = f(x), y = f(f(y));
    }
    return gcd(prd, n);
}

vector<long long> factor(long n) {
    if (n == 1) return {};
    if (is_prime(n)) return {n};
    long x = pollard(n);
    auto l = factor(x), r = factor(n / x);
    l.insert(l.end(), r.begin(), r.end());
    return l;
}

```

5.4 FastSubsetTransform

```

// fast and/or/xor convolution
void FST(vi& a, bool inv) {
    for (int n = sz(a), step = 1; step < n; step *= 2) {
        for (int i = 0; i < n; i += 2 * step) rep(j, i, i +
            step) {
            int &u = a[j], &v = a[j + step];
            tie(u, v) = inv ? pii(v - u, u) : pii(v, u + v); // AND
            inv ? pii(v, u - v) : pii(u + v, u); // OR
            pii(u + v, u - v); // XOR
        }
        if (inv)
            for (int& x : a) x /= sz(a); // XOR only
    }
    vi conv(vi a, vi b) {
        FST(a, 0);
        FST(b, 0);
        rep(i, 0, sz(a)) a[i] *= b[i];
        FST(a, 1);
        return a;
    }
}

```

5.5 Interpolate

```

const int mod = 1e9 + 7;
const int N = 1e6 + 6;

long long inv[N], po[N], pre[N], suf[N], dakdak[N];
long long ans, num;

inline long long qpow(long long a, int b) {
    long long ans = 1;
    while (b) {
        if (b & 1) ans = ans * a % mod;
        a = a * a % mod;
        b >>= 1;
    }
    return ans;
}

```

```

    }
    return ans;
}
inline long long rv(int x) { return qpow(x, mod - 2) %
    mod; }
void prec() {
    inv[0] = 1;
    for (int i = 1; i <= k + 1; ++i) {
        inv[i] = (1LL * inv[i - 1] * rv(i)) % mod;
        po[i] = (po[i - 1] + qpow(i, k)) % mod;
    }
    for (int i = 1; i <= k + 1; ++i) {
        dakdak[i] = (inv[i] * inv[k + 1 - i]) % mod;
    }
}
inline long long interpolate(int x, int k, bool bf =
    false) {
    if (k == 0) return x;
    if (x <= k + 1 || bf) {
        return po[x];
    }
    pre[0] = x;
    suf[k + 1] = x - (k + 1);
    for (int i = 1; i <= k; i++) pre[i] = (pre[i - 1] * (x -
        i)) % mod;
    for (int i = k; i >= 1; i--) suf[i] = (suf[i + 1] * (x -
        i)) % mod;
    ans = 0;
    for (int i = 0; i <= k + 1; i++) {
        if (i == 0)
            num = suf[1];
        else if (i == k + 1)
            num = pre[k];
        else
            num = (pre[i - 1] * suf[i + 1]) % mod; // numerator
        if ((i + k) & 1)
            ans = (ans + ((po[i] * num % mod) * dakdak[i])) % mod;
        else
            ans = (ans - ((po[i] * num % mod) * dakdak[i])) % mod;
        ans = (ans + mod) % mod;
    }
    return ans;
}

```

5.6 Lucas

```

// computing nCk mod p in O(p + log_p n)
// if p isn't prime -> prime factorize p = p1^a1 * p2^a2 *
... * pr^ar
// -> calculate nCk mod p1, mod p2, ..., mod pr
// -> use chinese remainder theorem to recover nCk mod p
ll lucas(ll n, ll k, int p, vi& fact, vi& invfact) {
    ll c = 1;
    while (n || k) {
        ll a = n % p, b = k % p;
        if (a < b) return 0;

```

```

        c = c * fact[a] % p * invfact[b] % p * invfact[a - b] %
            p;
        n /= p;
        k /= p;
    }
    return c;
}

```

5.7 Matrix

```

template <class T, int N>
struct Matrix {
    typedef Matrix M;
    array<array<T, N>, N> d{};
    M operator*(const M& m) const {
        M a;
        rep(i, 0, N) rep(j, 0, N) rep(k, 0, N) a.d[i][j] +=
            d[i][k] * m.d[k][j];
        return a;
    }
    vector<T> operator*(const vector<T>& vec) const {
        vector<T> ret(N);
        rep(i, 0, N) rep(j, 0, N) ret[i] += d[i][j] * vec[j];
        return ret;
    }
    M operator(1l p) const {
        assert(p >= 0);
        M a, b(*this);
        rep(i, 0, N) a.d[i][i] = 1;
        while (p) {
            if (p & 1) a = a * b;
            b = b * b;
            p >>= 1;
        }
        return a;
    }
};

```

5.8 MillerRabin

```

inline uint64_t mod_mult64(uint64_t a, uint64_t b,
    uint64_t m) { return __int128_t(a) * b % m; }
uint64_t mod_pow64(uint64_t a, uint64_t b, uint64_t m) {
    uint64_t ret = (m > 1);
    for (;;) {
        if (b & 1) ret = mod_mult64(ret, a, m);
        if (!(b >= 1)) return ret;
        a = mod_mult64(a, a, m);
    }
}

// Works for all primes p < 2^64
bool is_prime(uint64_t n) {
    if (n <= 3) return (n >= 2);
    static const uint64_t small[] = {

```

```

        2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
        41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,
        89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139,
        149, 151, 157, 163, 167, 173, 179, 181, 191,
        193, 197, 199,
    };
    for (size_t i = 0; i < sizeof(small) / sizeof(uint64_t); ++
        i) {
        if (n % small[i] == 0) return n == small[i];
    }

    // Makes use of the known bounds for Miller-Rabin
    // pseudoprimes.
    static const uint64_t millerrabin[] = {
        2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
    };
    static const uint64_t A014233[] = {
        // From OEIS.
        2047LL, 1373653LL, 25326001LL, 3215031751LL,
        2152302898747LL, 3474749660383LL,
        341550071728321LL, 341550071728321LL,
        3825123056546413051LL, 3825123056546413051LL,
        3825123056546413051LL, 0,
    };
    uint64_t s = n - 1, r = 0;
    while (s % 2 == 0) {
        s /= 2;
        r++;
    }
    for (size_t i = 0, j; i < sizeof(millerrabin) /
        sizeof(uint64_t); i++) {
        uint64_t md = mod_pow64(millerrabin[i], s, n);
        if (md != 1) {
            for (j = 1; j < r; j++) {
                if (md == n - 1) break;
                md = mod_mult64(md, md, n);
            }
            if (md != n - 1) return false;
        }
        if (n < A014233[i]) return true;
    }
    return true;
}

```

5.9 Mobius

```

mobius[1] = 1;
for (int i = 2; i < N; ++i) {
    --mobius[i];
    for (int j = i + 1; j < N; j += i) mobius[j] -=
        mobius[i];
}

```

5.10 ModInverse

```

const ll mod = 1000000007, LIM = 200000;
```

```
ll* inv = new ll[LIM] - 1; inv[1] = 1;
for(ll i = 2; i < LIM; ++i) inv[i] = mod - (mod / i) *
    inv[mod % i] % mod;
```

5.11 ModMulLL

```
typedef unsigned long long ull;
const int bits = 10; // if a 11 numbers are less than
                    2^k , set bits = 64k
const ull po = 1 << bits;
ull mod_mul(ull a, ull b, ull &c) {
    ull x = a * (b & (po - 1)) % c;
    while ((b >>= bits) > 0) {
        a = (a << bits) % c;
        x += (a * (b & (po - 1))) % c;
    }
    return x % c;
}
ull mod_pow(ull a, ull b, ull mod) {
    if (b == 0) return 1;
    ull res = mod_pow(a, b / 2, mod);
    res = mod_mul(res, res, mod);
    if (b & 1) return mod_mul(res, a, mod);
    return res;
}
```

5.12 Modular Arithmetic

```
const ll mod = 17; // change to something else
struct Mod {
    ll x;
    Mod(ll xx) : x(xx) {}
    Mod operator+(Mod b) { return Mod((x + b.x) % mod); }
    Mod operator-(Mod b) { return Mod((x - b.x + mod) %
        mod); }
    Mod operator*(Mod b) { return Mod((x * b.x) % mod); }
    Mod operator/(Mod b) { return *this * invert(b); }
    Mod invert(Mod a) {
        ll x, y, g = euclid(a.x, mod, x, y);
        assert(g == 1);
        return Mod((x + mod) % mod);
    }
    Mod operator(ll e) {
        if (!e) return Mod(1);
        Mod r = *this (e / 2);
        r = r * r;
        return e & 1 ? *this * r : r;
    }
};
```

5.13 NTT

```
/* NTT with modulo 998244353
```

```
notes:
NTT with mod m
g is any primitive root modulo m (g = 3 works well for
998244353)
n divides m - 1 evenly
wn = g^{(m - 1) / n}
https://codeforces.com/blog/entry/75326
 */

const int N = 1 << 21;
const ll mod = 998244353;
const ll g = 3;

int rev[N];
ll w[N], iw[N], wt[N], inv_n;

ll binpow(ll a, ll b) {
    ll res = 1;
    for (; b; b >>= 1, a = (1ll * a * a) % mod)
        if (b & 1) res = (1ll * res * a) % mod;
    return res;
}

void precalc(int lg) {
    int n = 1 << lg;
    inv_n = binpow(n, mod - 2);
    for (int i = 0; i < n; ++i) {
        rev[i] = 0;
        for (int j = 0; j < lg; ++j)
            if (i & (1 << j)) rev[i] |= (1 << (lg - j - 1));
    }
    ll wn = binpow(g, (mod - 1) / n);
    w[0] = 1;
    for (int i = 1; i < n; ++i) w[i] = (1ll * w[i - 1] * wn) %
        mod;
    ll iwn = binpow(wn, mod - 2);
    iw[0] = 1;
    for (int i = 1; i < n; ++i) iw[i] = (1ll * iw[i - 1] * iwn) %
        mod;
}

void ntt(vector<ll> &a, int lg, bool inv = 0) {
    int n = (1 << lg);
    for (int i = 0; i < n; ++i)
        if (i < rev[i]) swap(a[i], a[rev[i]]);
    for (int len = 2; len <= n; len <= n; len <= 1) {
        int d = n / len;
        for (int j = 0; j < (len >> 1); ++j) wt[j] = (inv ?
            iw[d * j] : w[d * j]);
        for (int i = 0; i < n; i += len) {
            for (int j = 0; j < (len >> 1); ++j) {
                ll x = a[i + j], y = (1ll * a[i + j + (len >> 1)] *
                    wt[j]) % mod;
                a[i + j] = (x + y) % mod;
                a[i + j + (len >> 1)] = (x - y + mod) % mod;
            }
        }
        if (inv)
            for (int i = 0; i < n; ++i) a[i] = (1ll * a[i] * inv_n) %
                mod;
    }
}
```

```
}

vector<ll> multiply(vector<ll> a, vector<ll> b) {
    int n = 1, lg = 0;
    int na = sz(a), nb = sz(b);
    while (n < na + nb) n <= 1, ++lg;
    precalc(lg);
    a.resize(n);
    b.resize(n);
    ntt(a, lg);
    ntt(b, lg);
    for (int i = 0; i < n; ++i) a[i] = (1ll * a[i] * b[i]) %
        mod;
    ntt(a, lg, 1);
    vector<ll> c;
    for (int i = 0; i < na + nb - 1; ++i) c.push_back(a[i]);
    // while(!c.empty() and c.back() == 0)
    //     c.pop_back();
    return c;
}
```

5.14 Notes

5.14.1 Cycles

Let $g_S(n)$ be the number of n -permutations whose cycle lengths all belong to the set S . Then

$$\sum_{n=0}^{\infty} g_S(n) \frac{x^n}{n!} = \exp \left(\sum_{n \in S} \frac{x^n}{n} \right)$$

5.14.2 Derangements

Permutations of a set such that none of the elements appear in their original position.

$$D(n) = (n-1)(D(n-1) + D(n-2)) = nD(n-1) + (-1)^n = \left\lfloor \frac{n!}{e} \right\rfloor$$

5.14.3 Burnside's lemma

Given a group G of symmetries and a set X , the number of elements of X up to symmetry equals

$$\frac{1}{|G|} \sum_{g \in G} |X^g|,$$

where X^g are the elements fixed by g ($g \cdot x = x$).

If $f(n)$ counts “configurations” (of some sort) of length n , we can ignore rotational symmetry using $G = Z_n$ to get

$$g(n) = \frac{1}{n} \sum_{k=0}^{n-1} f(\gcd(n, k)) = \frac{1}{n} \sum_{k|n} f(k) \phi(n/k).$$

5.14.4 Partition function

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

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

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

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	$\sim 2e5$	$\sim 2e8$

5.14.5 Lucas' Theorem

Let n, m be non-negative integers and p a prime. Write $n = n_k p^k + \dots + n_1 p + n_0$ and $m = m_k p^k + \dots + m_1 p + m_0$. Then $\binom{n}{m} \equiv \prod_{i=0}^k \binom{n_i}{m_i} \pmod{p}$.

5.14.6 Bernoulli numbers

EGF of Bernoulli numbers is $B(t) = \frac{t}{e^t - 1}$ (FFT-able). $B[0, \dots] = [1, -\frac{1}{2}, \frac{1}{6}, 0, -\frac{1}{30}, 0, \frac{1}{42}, \dots]$
Sums of powers:

$$\sum_{i=1}^n n^m = \frac{1}{m+1} \sum_{k=0}^m \binom{m+1}{k} B_k \cdot (n+1)^{m+1-k}$$

Euler-Maclaurin formula for infinite sums:

$$\sum_{i=m}^{\infty} f(i) = \int_m^{\infty} f(x) dx - \sum_{k=1}^{\infty} \frac{B_k}{k!} f^{(k-1)}(m)$$

$$\approx \int_m^{\infty} f(x) dx + \frac{f(m)}{2} - \frac{f'(m)}{12} + \frac{f''(m)}{720} + O(f^{(5)}(m))$$

5.14.7 Stirling numbers of the first kind

Number of permutations on n items with k cycles.

$$c(n, k) = c(n-1, k-1) + (n-1)c(n-1, k), \quad c(0, 0) = 1$$

$$\sum_{k=0}^n c(n, k)x^k = x(x+1)\dots(x+n-1)$$

$$c(8, k) = 8, 0, 5040, 13068, 13132, 6769, 1960, 322, 28, 1$$

$$c(n, 2) = 0, 0, 1, 3, 11, 50, 274, 1764, 13068, 109584, \dots$$

5.14.8 Eulerian numbers

Number of permutations $\pi \in S_n$ in which exactly k elements are greater than the previous element. k j:s s.t. $\pi(j) > \pi(j+1)$, $k+1$ j:s s.t. $\pi(j) \geq j$, k j:s s.t. $\pi(j) > j$.

$$E(n, k) = (n-k)E(n-1, k-1) + (k+1)E(n-1, k)$$

$$E(n, 0) = E(n, n-1) = 1$$

$$E(n, k) = \sum_{j=0}^k (-1)^j \binom{n+1}{j} (k+1-j)^n$$

5.14.9 Stirling numbers of the second kind

Partitions of n distinct elements into exactly k groups.

$$S(n, k) = S(n-1, k-1) + kS(n-1, k)$$

$$S(n, 1) = S(n, n) = 1$$

$$S(n, k) = \frac{1}{k!} \sum_{j=0}^k (-1)^{k-j} \binom{k}{j} j^n$$

5.14.10 Bell numbers

Total number of partitions of n distinct elements. $B(n) = 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, \dots$ For p prime,

$$B(p^m + n) \equiv mB(n) + B(n+1) \pmod{p}$$

5.14.11 Labeled unrooted trees

on n vertices: n^{n-2}
on k existing trees of size n_i : $n_1 n_2 \dots n_k n^{k-2}$
with degrees d_i : $(n-2)! / ((d_1-1)! \dots (d_n-1)!)$

5.14.12 Catalan numbers

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1} = \frac{(2n)!}{(n+1)!n!}$$

$$C_0 = 1, \quad C_{n+1} = \frac{2(2n+1)}{n+2} C_n, \quad C_{n+1} = \sum C_i C_{n-i}$$

$$C_n = 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \dots$$

- sub-diagonal monotone paths in an $n \times n$ grid.
- strings with n pairs of parenthesis, correctly nested.
- binary trees with $n+1$ leaves (0 or 2 children).
- ordered trees with $n+1$ vertices.
- ways a convex polygon with $n+2$ sides can be cut into triangles by connecting vertices with straight lines.
- permutations of $[n]$ with no 3-term increasing subseq.

5.14.13 Hockey Stick Identity

$$\sum_{k=r}^n \binom{k}{r} = \binom{n+1}{r+1}$$

5.15 PhiFunction

```
const int LIM = 5000000;
int phi[LIM];
void calculatePhi() {
    rep(i, 0, LIM) phi[i] = i & 1 ? i : i / 2;
    for (int i = 3; i < LIM; i += 2)
        if (phi[i] == i)
            for (int j = i; j < LIM; j += i) (phi[j] /= i) *= i - 1;
```

5.16 PollardFactorize

```
using ll = long long;
using ull = unsigned long long;
using ld = long double;
ll mult(ll x, ll y, ll md) {
    ull q = (ld)x * y / md;
    ll res = ((ull)x * y - q * md);
    if (res >= md) res -= md;
    if (res < 0) res += md;
    return res;
}
ll powMod(ll x, ll p, ll md) {
    if (p == 0) return 1;
    if (p & 1) return mult(x, powMod(x, p - 1, md), md);
    return powMod(mult(x, x, md), p / 2, md);
}
bool checkMillerRabin(ll x, ll md, ll s, int k) {
    x = powMod(x, s, md);
    if (x == 1) return true;
    while (k--) {
        if (x == md - 1) return true;
        x = mult(x, x, md);
        if (x == 1) return false;
    }
    return false;
}
bool isPrime(ll x) {
    if (x == 2 || x == 3 || x == 5 || x == 7) return true;
    if (x % 2 == 0 || x % 3 == 0 || x % 5 == 0 || x % 7 == 0) return false;
    if (x < 121) return x > 1;
    ll s = x - 1;
    int k = 0;
    while (s % 2 == 0) {
        s >>= 1;
        k++;
    }
    if (x < 1LL << 32) {
        for (ll z : {2, 7, 61}) {
            if (!checkMillerRabin(z, x, s, k)) return false;
        }
    } else {
        for (ll z : {2, 325, 9375, 28178, 450775, 9780504, 1795265022}) {
```

```

    if (!checkMillerRabin(z, x, s, k)) return false;
}
return true;
}

ll gcd(ll x, ll y) { return y == 0 ? x : gcd(y, x % y); }

void pollard(ll x, vector<ll> &ans) {
if (isPrime(x)) {
    ans.push_back(x);
    return;
}
ll c = 1;
while (true) {
    c = 1 + get_rand(x - 1);
    auto f = [&](ll y) {
        ll res = mult(y, y, x) + c;
        if (res >= x) res -= x;
        return res;
    };
    ll y = 2;
    int B = 100;
    int len = 1;
    ll g = 1;
    while (g == 1) {
        ll z = y;
        for (int i = 0; i < len; i++) {
            z = f(z);
        }
        ll zs = -1;
        int lft = len;
        while (g == 1 && lft > 0) {
            zs = z;
            ll p = 1;
            for (int i = 0; i < B && i < lft; i++) {
                p = mult(p, abs(z - y), x);
                z = f(z);
            }
            g = gcd(p, x);
            lft -= B;
        }
        if (g == 1) {
            y = z;
            len <= 1;
            continue;
        }
        if (g == x) {
            g = 1;
            z = zs;
            while (g == 1) {
                g = gcd(abs(z - y), x);
                z = f(z);
            }
        }
        if (g == x) break;
        assert(g != 1);
        pollard(g, ans);
        pollard(x / g, ans);
        return;
    }
}

```

5.17 PrimitiveRoot

```

}

// return list of all prime factors of x (can have
// duplicates)
vector<ll> factorize(ll x) {
    vector<ll> ans;
    for (ll p : {2, 3, 5, 7, 11, 13, 17, 19}) {
        while (x % p == 0) {
            x /= p;
            ans.push_back(p);
        }
    }
    if (x != 1) {
        pollard(x, ans);
    }
    sort(ans.begin(), ans.end());
    return ans;
}

// return pairs of (p, k) where x = product(p^k)
vector<pair<ll, int>> factorize_pk(ll x) {
    auto ps = factorize(x);
    ll last = -1, cnt = 0;
    vector<pair<ll, int>> res;
    for (auto p : ps) {
        if (p == last)
            ++cnt;
        else {
            if (last > 0) res.emplace_back(last, cnt);
            last = p;
            cnt = 1;
        }
    }
    if (cnt > 0) {
        res.emplace_back(last, cnt);
    }
    return res;
}

vector<ll> get_all_divisors(ll n) {
    auto pks = factorize_pk(n);

    vector<ll> res;
    function<void(int, ll)> gen = [&](int i, ll prod) {
        if (i == static_cast<int>(pks.size())) {
            res.push_back(prod);
            return;
        }

        ll cur_power = 1;
        for (int cur = 0; cur <= pks[i].second; ++cur) {
            gen(i + 1, prod * cur_power);
            cur_power *= pks[i].first;
        }
    };

    gen(0, 1LL);
    sort(res.begin(), res.end());
    return res;
}

```

```

// Primitive root of modulo n is integer g iff for all a <
//   n & gcd(a, n) == 1, there exist k: g^k = a mod n
// k is called discrete log of a (in case P is prime, can
//   find in O(sqrt(P)) by noting that (P-1) is divisible
//   by k)
//
// Exist if:
// - n is 1, 2, 4
// - n = p^k for odd prime p
// - n = 2*p^k for odd prime p
int powmod (int a, int b, int p) {
    int res = 1;
    while (b) {
        if (b & 1)
            res = int (res * 1LL * a % p), --b;
        else
            a = int (a * 1LL * a % p), b >>= 1;
    }
    return res;
}

int generator (int p) {
    vector<int> fact;
    int phi = p-1, n = phi;
    for (int i=2; i*i<=n; ++i)
        if (n % i == 0) {
            fact.push_back (i);
            while (n % i == 0)
                n /= i;
        }
    if (n > 1)
        fact.push_back (n);

    for (int res=2; res<=p; ++res) {
        bool ok = true;
        for (size_t i=0; i<fact.size() && ok; ++i)
            ok &= powmod (res, phi / fact[i], p) != 1;
        if (ok) return res;
    }
    return -1;
}

```

5.18 TernarySearch

```

// Find the smallest i in [a; b] that maximizes f(i),
// assuming
// that f(a) < ... < f(i) >= ... >= f(b)
// Usage: int ind = ternSearch(0,n-1,[&](int i){return
//   a[i];});
template <class F>
int ternSearch(int a, int b, F f) {
    assert(a <= b);
    while (b - a >= 5) {
        int mid = (a + b) / 2;
        if (f(mid) < f(mid + 1))
            a = mid; // (A)
        else
            b = mid + 1;
    }
}

```

```
rep(i, a + 1, b + 1) if (f(a) < f(i)) a = i; // (B)
return a;
}
```

5.19 XorBasis

```
struct XorBasis {
    vector<T> basis;
    bool add(T mask) {
        for (auto u : basis) mask = min(mask, mask ^ u);
        for (auto& u : basis) u = min(u, u ^ mask);
        if (mask) {
            basis.push_back(mask);
            return true;
        }
        return false;
    }
    T getMax() {
        T res = 0;
        for (auto u : basis) res = max(res, res ^ u);
        return res;
    }
    void init() { sort(basis.begin(), basis.end()); }
    bool InSpan(T mask) {
        for (auto u : basis) mask = min(mask, mask ^ u);
        return (mask == 0);
    }
    T getK(T k) // 0-indexed
    {
        T res = 0;
        for (int i = sz(basis) - 1; i >= 0; i--)
            if (k >> i & 1) res ^= basis[i];
        return res;
    }
};
```

6 Miscellaneous

6.1 FastInput

```
inline char gc() { // like getchar()
    static char buf[1 << 16];
    static size_t bc, be;
    if (bc >= be) {
        buf[0] = 0, bc = 0;
        be = fread(buf, 1, sizeof(buf), stdin);
    }
    return buf[bc++]; // returns 0 on EOF
}
int readInt() {
    int a, c;
    while ((a = gc()) < 40)
        ;
    if (a == '-') return -readInt();
    while ((c = gc()) >= 48) a = a * 10 + c - 480;
```

```
    return a - 48;
}
```

6.2 template

```
#include <bits/stdc++.h>
using namespace std;
mt19937_64
    rng(chrono::steady_clock::now().time_since_epoch().count());
ll get_rand(ll l, ll r) {
    return uniform_int_distribution<ll>(l, r)(rng);
}
int32_t main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    cerr << "\n[Time]: " << 1000.0 * clock() /
        CLOCKS_PER_SEC << " ms.\n";
    return 0;
}
```

7 String

7.1 AhoCorasick

```
template <int MAXC = 26>
struct AhoCorasick {
    vector<array<int, MAXC>> C;
    vector<int> F;
    vector<vector<int>> FG;
    vector<bool> E;

    int node() {
        int r = C.size();
        E.push_back(0);
        F.push_back(-1);
        C.emplace_back();
        fill(C.back().begin(), C.back().end(), -1);
        return r;
    }
    int ctrans(int n, int c) {
        if (C[n][c] == -1) C[n][c] = node();
        return C[n][c];
    }
    int ftrans(int n, int c) const {
        while (n && C[n][c] == -1) n = F[n];
        return C[n][c] != -1 ? C[n][c] : 0;
    }
    AhoCorasick(vector<vector<int>> P) {
        node();
        for (int i = 0; i < (int)P.size(); i++) {
            int n = 0;
            for (int c : P[i]) n = ctrans(n, c);
            E[n] = 1;
        }
        queue<int> Q;
```

```
F[0] = 0;
for (int c : C[0])
    if (c != -1) Q.push(c), F[c] = 0;
while (!Q.empty()) {
    int n = Q.front();
    Q.pop();
    for (int c = 0; c < MAXC; ++c)
        if (C[n][c] != -1) {
            int f = F[n];
            while (f && C[f][c] == -1) f = F[f];
            F[C[n][c]] = C[f][c] != -1 ? C[f][c] : 0;
            Q.emplace(C[n][c]);
        }
}
FG.resize(F.size());
for (int i = 1; i < (int)F.size(); i++) {
    FG[F[i]].push_back(i);
    if (E[i]) Q.push(i);
}
while (!Q.empty()) {
    int n = Q.front();
    Q.pop();
    for (int f : FG[n]) E[f] = 1, Q.push(f);
}
bool check(vector<int> V) {
    if (E[0]) return 1;
    int n = 0;
    for (int c : V) {
        n = ftrans(n, c);
        if (E[n]) return 1;
    }
    return 0;
}
```

7.2 KMP

```
// prefix function: *length* of longest prefix which is
// also suffix:
// pi[i] = max(k: s[0..k-1] == s[i-(k-1)..i]
//
// KMP {{{
template <typename Container>
std::vector<int> prefix_function(const Container& s) {
    int n = s.size();
    std::vector<int> pi(n);
    for (int i = 1; i < n; ++i) {
        int j = pi[i - 1];
        while (j > 0 && s[i] != s[j]) j = pi[j - 1];
        if (s[i] == s[j]) ++j;
        pi[i] = j;
    }
    return pi;
}

// Tested: https://oj.vnoi.info/problem/substr
// Return all positions (0-based) that pattern 'pat'
// appears in 'text'
```

```

std::vector<int> kmp(const std::string& pat, const
    std::string& text) {
    auto pi = prefix_function(pat + '\0' + text);
    std::vector<int> res;
    for (size_t i = pi.size() - text.size(); i < pi.size();
        ++i) {
        if (pi[i] == (int)pat.size()) {
            res.push_back(i - 2 * pat.size());
        }
    }
    return res;
}

// Tested: https://oj.vnoi.info/problem/icpc22_mt_b
// Returns cnt[i] = # occurrences of prefix of length-i
// NOTE: cnt[0] = n+1 (0-length prefix appears n+1 times)
std::vector<int> prefix_occurrences(const string& s) {
    int n = s.size();
    auto pi = prefix_function(s);
    std::vector<int> res(n + 1);
    for (int i = 0; i < n; ++i) res[pi[i]]++;
    for (int i = n - 1; i > 0; --i) res[pi[i - 1]] += res[i];
    for (int i = 0; i <= n; ++i) res[i]++;
    return res;
}

```

7.3 Manacher

```

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

```

```

    string t;
    for (auto c : s) {
        t += string("#") + c;
    }
    auto res = manacher_odd(t + "#");
    return vector<int>(begin(res) + 1, end(res) - 1);
}

```

7.4 StringHashing

```

const int MOD1 = 127657753, MOD2 = 987654319;
const int p1 = 137, p2 = 277;

```

7.5 SuffixArray

```

/**
 * Author:         , chilli
 * Date: 2019-04-11
 * License: Unknown
 * Source: Suffix array - a powerful tool for dealing with
 *          strings
 * (Chinese IOI National team training paper, 2009)
 * Description: Builds suffix array for a string.
 * \texttt{sa[i]} is the starting index of the suffix which
 * is $i$'th in the sorted suffix array.
 * The returned vector is of size $n+1$, and \texttt{sa[0]
 * = n}.
 * The \texttt{lcp} array contains longest common prefixes
 * for
 * neighbouring strings in the suffix array:
 * \texttt{lcp[i] = lcp(sa[i], sa[i-1])}, \texttt{lcp[0] =
 * 0}.
 * The input string must not contain any zero bytes.
 * Time: O(n \log n)
 * Status: stress-tested
 */
struct SuffixArray {
    vi sa, lcp;
    SuffixArray(string& s, int lim=256) { // or
        basic_string<int>
        int n = sz(s) + 1, k = 0, a, b;
        vi x(all(s)), y(n), ws(max(n, lim));

```

```

        x.push_back(0), sa = lcp = y, iota(all(sa),
            0);
        for (int j = 0, p = 0; p < n; j = max(1, j *
            2), lim = p) {
            p = j, iota(all(y), n - j);
            rep(i,0,n) if (sa[i] >= j) y[p++] =
                sa[i] - j;
            fill(all(ws), 0);
            rep(i,0,n) ws[x[i]]++;
            rep(i,1,lim) ws[i] += ws[i - 1];
            for (int i = n; i--;) {
                sa[--ws[x[y[i]]]] = y[i];
                swap(x, y), p = 1, x[sa[0]] = 0;
                rep(i,1,n) a = sa[i - 1], b = sa[i],
                    x[b] =
                        (y[a] == y[b] && y[a + j] ==
                            y[b + j]) ? p - 1 : p++;
            }
            for (int i = 0, j; i < n - 1; lcp[x[i++]] =
                k)
                for (k && k--, j = sa[x[i] - 1];
                    s[i + k] == s[j + k];
                    k++);
        }
    };
    int64_t cnt_distinct_substrings(const std::string& s) {
        auto lcp = LCP(s, suffix_array(s, 0, 255));
        return s.size() * (int64_t)(s.size() + 1) / 2
            - std::accumulate(lcp.begin(), lcp.end(), 0LL);
    }
}

```

7.6 Z

```

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

```