```
   heap_l.pop();
   heap_l.push (tmp); }
  int x = tree[rt].l, y = tree[rt].r;
  if (~x && ~y && sqrdist (d, tree[x].p) > sqrdist (d, tree[y].p)) std::swap (x, y);
  if (~x && ((int)heap_l.size() < m || tree[x].min_dist (d, k) < heap_l.top().dist))
   _min_kth ((depth + 1) % k, x, m, d);
  if (~y && ((int)heap_l.size() < m || tree[y].min_dist (d, k) < heap_l.top().dist))
   _min_kth ((depth + 1) % k, y, m, d); }
 void _max_kth (const int &depth, const int &rt, const int &m, const point &d) {
  result tmp = result (sqrdist (tree[rt].p, d), tree[rt].p);
  if ((int)heap_r.size() < m) heap_r.push (tmp);
  else if (tmp > heap_r.top()) {
   heap_r.pop();
   heap_r.push (tmp); }
  int x = tree[rt].l, y = tree[rt].r;
  if (~x && ~y && sqrdist (d, tree[x].p) < sqrdist (d, tree[y].p)) std::swap (x, y);
  if (~x && ((int)heap_r.size() < m || tree[x].max_dist (d, k) >= heap_r.top().dist))
   _max_kth ((depth + 1) % k, x, m, d);
  if (~y && ((int)heap_r.size() < m || tree[y].max_dist (d, k) >= heap_r.top().dist))
   _max_kth ((depth + 1) % k, y, m, d); }
 void init (int n, int k) { this -> k = k; size = 0; int rt = 0; build (0, rt, 0, n - 1); }
 result min_kth (const point &d, const int &m) { heap_l = decltype (heap_l) (); _min_kth (0, 0, m, d); return heap_l.top (); }
 result max_kth (const point &d, const int &m) { heap_r = decltype (heap_r) (); _max_kth (0, 0, m, d); return heap_r.top (); } };
```

2.3 RMQ

```
for (int st = 1; st < 20; ++st) for (int i = 0; i < N; ++i)
 if (i + (1 << st - 1) < N) rmq[st][i] = std::min (rmq[st - 1][i], rmq[st - 1][i + (1 << st - 1)]);

int len = 31 - __builtin_clz (r - l + 1);
return std::min (rmq[len][l], rmq[len][r - (1 << len) + 1]);
```

3 Geometry

Generally ϵ should be less than $\frac{1}{xy}$.

```
#define cd const double &
const double EPS = 1E-8, PI = acos (-1);
int sgn (cd x) { return x < -EPS ? -1 : x > EPS; }
int cmp (cd x, cd y) { return sgn (x - y); }
double sqr (cd x) { return x * x; }
double msqrt (cd x) { return sgn (x) <= 0 ? 0 : sqrt (x); }
```

3.1 2D geometry

1. `point::rot90`: Counter-clockwise rotation.
2. `line_circle_intersect`: In order of the direction of a.
3. `circle_intersect`: Counter-clockwise with respect of O_a.
4. `tangent`: Counter-clockwise with respect of a.
5. `extangent`: Counter-clockwise with respect of O_a.
6. `intangent`: Counter-clockwise with respect of O_a.

```
#define cp const point &
struct point {
 double x, y;
 explicit point (cd x = 0, cd y = 0) : x (x), y (y) {}
 int dim () const { return sgn (y) == 0 ? sgn (x) > 0 : sgn (y) > 0; }
 point unit () const { double l = msqrt (x * x + y * y); return point (x / l, y / l); }
 point rot90 () const { return point (-y, x); }
 point _rot90 () const { return point (y, -x); }
 point rot (cd t) const {
  double c = cos (t), s = sin (t);
  return point (x * c - y * s, x * s + y * c); } };
bool operator == (cp a, cp b) { return cmp (a.x, b.x) == 0 && cmp (a.y, b.y) == 0; }
bool operator != (cp a, cp b) { return cmp (a.x, b.x) != 0 || cmp (a.y, b.y) != 0; }
bool operator < (cp a, cp b) { return cmp (a.x, b.x) == 0 ? cmp (a.y, b.y) < 0 : cmp (a.x, b.x) < 0; }
point operator - (cp a) { return point (-a.x, -a.y); }
point operator + (cp a, cp b) { return point (a.x + b.x, a.y + b.y); }
point operator - (cp a, cp b) { return point (a.x - b.x, a.y - b.y); }
point operator * (cp a, cd b) { return point (a.x * b, a.y * b); }
point operator / (cp a, cd b) { return point (a.x / b, a.y / b); }
double dot (cp a, cp b) { return a.x * b.x + a.y * b.y; }
double det (cp a, cp b) { return a.x * b.y - a.y * b.x; }
double dis2 (cp a, cp b = point ()) { return sqr (a.x - b.x) + sqr (a.y - b.y); }
double dis (cp a, cp b = point ()) { return msqrt (dis2 (a, b)); }
#define cl const line &
struct line {
 point s, t;
 explicit line (cp s = point (), cp t = point ()) : s (s), t (t) {} };
bool point_on_segment (cp a, cl b) { return sgn (det (a - b.s, b.t - b.s)) == 0 && sgn (dot (b.s - a, b.t - a)) <= 0; }
bool two_side (cp a, cp b, cl c) { return sgn (det (a - c.s, c.t - c.s)) * sgn (det (b - c.s, c.t - c.s)) < 0; }
bool intersect_judgment (cl a, cl b) {
 if (point_on_segment (b.s, a) || point_on_segment (b.t, a)) return true;
 if (point_on_segment (a.s, b) || point_on_segment (a.t, b)) return true;
 return two_side (a.s, a.t, b) && two_side (b.s, b.t, a); }
point line_intersect (cl a, cl b) {
 double s1 = det (a.t - a.s, b.s - a.s), s2 = det (a.t - a.s, b.t - a.s);
 return (b.s * s2 - b.t * s1) / (s2 - s1); }
double point_to_line (cp a, cl b) { return std::abs (det (b.t - b.s, a - b.s)) / dis (b.s, b.t); }
point project_to_line (cp a, cl b) { return b.s + (b.t - b.s) * (dot (a - b.s, b.t - b.s) / dis2 (b.t, b.s)); }
double point_to_segment (cp a, cl b) {
 if (sgn (dot (b.s - a, b.t - b.s) * dot (b.t - a, b.t - b.s)) <= 0) return std::abs (det (b.t - b.s, a - b.s)) / dis (b.s, b.t);
 return std::min (dis (a, b.s), dis (a, b.t)); }
bool in_polygon (cp p, const std::vector <point> & po) {
 int n = (int) po.size (), counter = 0;
 for (int i = 0; i < n; ++i) {
  point a = po[i], b = po[(i + 1) % n];
  // Modify the next line if necessary.
  if (point_on_segment (p, line (a, b))) return true;
  int x = sgn (det (p - a, b - a)), y = sgn (a.y - p.y), z = sgn (b.y - p.y);
  if (x > 0 && y <= 0 && z > 0) counter++;
  if (x < 0 && z <= 0 && y > 0) counter--; }
 return counter != 0; }
double polygon_area (const std::vector <point> &a) {
 double ans = 0.0;
 for (int i = 0; i < (int) a.size (); ++i) ans += det (a[i], a[ (i + 1) % a.size ()]) / 2.0;
 return ans; }
#define cc const circle &
struct circle {
 point c; double r;
 explicit circle (point c = point (), double r = 0) : c (c), r (r) {} };
bool operator == (cc a, cc b) { return a.c == b.c && cmp (a.r, b.r) == 0; }
bool operator != (cc a, cc b) { return !(a == b); }
bool in_circle (cp a, cc b) { return cmp (dis (a, b.c), b.r) <= 0; }
circle make_circle (cp a, cp b) { return circle ((a + b) / 2, dis (a, b) / 2); }
circle make_circle (cp a, cp b, cp c) { point p = circumcenter (a, b, c); return circle (p, dis (p, a)); }
std::vector <point> line_circle_intersect (cl a, cc b) {
 if (cmp (point_to_line (b.c, a), b.r) > 0) return std::vector <point> ();
 double x = msqrt (sqr (b.r) - sqr (point_to_line (b.c, a)));
 point s = project_to_line (b.c, a), u = (a.t - a.s).unit ();
 if (sgn (x) == 0) return std::vector <point> ({s});
 return std::vector <point> ({s - u * x, s + u * x}); }
double circle_intersect_area (cc a, cc b) {
 double d = dis (a.c, b.c);
 if (sgn (d - (a.r + b.r)) >= 0) return 0;
 if (sgn (d - std::abs(a.r - b.r)) <= 0) {
  double r = std::min (a.r, b.r); return r * r * PI; }
 double x = (d * d + a.r * a.r - b.r * b.r) / (2 * d), t1 = acos (std::min (1., std::max (-1., x / a.r))), t2 = acos (std::min (1., std::max (-1., (d - x) / b.r)));
 return a.r * a.r * t1 + b.r * b.r * t2 - d * a.r * sin (t1); }
std::vector <point> circle_intersect (cc a, cc b) {
 if (a.c == b.c || cmp (dis (a.c, b.c), a.r + b.r) > 0 || cmp (dis (a.c, b.c), std::abs (a.r - b.r)) < 0) return std::vector <point> ();
 point r = (b.c - a.c).unit (); double d = dis (a.c, b.c);
 double x = ((sqr (a.r) - sqr (b.r)) / d + d) / 2, h = msqrt (sqr (a.r) - sqr (x));
 if (sgn (h) == 0) return std::vector <point> ({a.c + r * x});
 return std::vector <point> ({a.c + r * x - r.rot90 () * h, a.c + r * x + r.rot90 () * h}); }
std::vector <point> tangent (cp a, cc b) { circle p = make_circle (a, b.c); return circle_intersect (p, b); }
std::vector <line> extangent (cc a, cc b) {
 std::vector <line> ret;
 if (cmp (dis (a.c, b.c), std::abs (a.r - b.r)) <= 0) return ret;
 if (sgn (a.r - b.r) == 0) {
  point dir = b.c - a.c; dir = (dir * a.r / dis (dir)).rot90 ();
  ret.push_back (line (a.c - dir, b.c - dir));
  ret.push_back (line (a.c + dir, b.c + dir));
 } else {
  point p = (b.c * a.r - a.c * b.r) / (a.r - b.r);
  std::vector <point> pp = tangent (p, a), qq = tangent (p, b);
  if (pp.size () == 2 && qq.size () == 2) {
   if (cmp (a.r, b.r) < 0) std::swap (pp[0], pp[1]), std::swap (qq[0], qq[1]);
   ret.push_back (line (pp[0], qq[0]));
   ret.push_back (line (pp[1], qq[1])); } }
 return ret; }
std::vector <line> intangent (cc a, cc b) {
 std::vector <line> ret;
 point p = (b.c * a.r + a.c * b.r) / (a.r + b.r);
 std::vector <point> pp = tangent (p, a), qq = tangent (p, b);
 if (pp.size () == 2 && qq.size () == 2) {
  ret.push_back (line (pp[0], qq[0]));
  ret.push_back (line (pp[1], qq[1])); }
 return ret; }
```

3.1.1 Convex hull

Counter-clockwise, starting with the smallest point, and with the minimum number of points. Modify `!= -s` to `== s` in `turn` to conserve all points on the hull.

`convex_tan` finds the covering `[s..t]` of a certain point.

```
bool turn (cp a, cp b, cp c, int s) { return sgn (det (b - a, c - a)) != -s; }
std::pair <std::vector <point>, int> convex_hull (std::vector <point> a) {
 int cnt = 0; std::sort (a.begin (), a.end ());
 static std::vector <point> ret; ret.resize (a.size () << 1);
 for (int i = 0; i < (int) a.size (); ++i) {
  while (cnt > 1 && turn (ret[cnt - 2], a[i], ret[cnt - 1], 1)) --cnt;
  ret[cnt++] = a[i]; }
 int fixed = cnt;
 for (int i = (int) a.size () - 1; i >= 0; --i) {
  while (cnt > fixed && turn (ret[cnt - 2], a[i], ret[cnt - 1], 1)) --cnt;
  ret[cnt++] = a[i]; }
 return std::make_pair (std::vector <point> (ret.begin (), ret.begin () + cnt - 1), fixed - 1); }

int lb (cp x, const std::vector <point> &v, int l, int r, int s) {
 if (l > r) l = r;
 while (l != r) {
  int m = (l + r) / 2;
  if (sgn (det (v[m % v.size ()] - x, v[(m + 1) % v.size ()] - x)) == s) r = m; else l = m + 1; }
 return r % v.size (); }
std::pair <int, int> convex_tan (cp x, const std::vector <point> &v, int rp) {
 if (cmp (x.x, v[0].x) < 0) return std::make_pair (lb (x, v, rp, v.size (), -1), lb (x, v, 0, rp, 1));
 else if (cmp (x.x, v[rp].x) > 0) return std::make_pair (lb (x, v, 0, rp, -1), lb (x, v, rp, v.size (), 1));
 else {
  int id = std::lower_bound (v.begin (), v.begin () + rp, x) - v.begin ();
  if (id == 0 || sgn (det (v[id - 1] - x, v[id] - x)) < 0)
   return std::make_pair (lb (x, v, 0, id, -1), lb (x, v, id, rp, 1));
  id = std::lower_bound (v.begin () + rp, v.end (), x, std::greater <point> ()) - v.begin ();
  if (id == rp || sgn (det (v[id - 1] - x, v[id % v.size ()] - x)) < 0)
   return std::make_pair (lb (x, v, rp, id, -1), lb (x, v, id, v.size (), 1));
  return std::make_pair (-1, -1); } }
```

3.1.2 Delaunay triangulation

In mathematics and computational geometry, a Delaunay triangulation (also known as a Delone triangulation) for a given set P of discrete points in a plane is a triangulation $DT(P)$ such that no point in P is inside the circumcircle of any triangle in $DT(P)$. Delaunay triangulations maximize the minimum angle of all the angles of the triangles in the triangulation; they tend to avoid sliver triangles.

The Delaunay triangulation of a discrete point set P in general position corresponds to the dual graph of the Voronoi diagram for P. Special cases include the existence of three points on a line and four points on circle.

Properties: Let n be the number of points.

1. The union of all triangles in the triangulation is the convex hull of the points.
2. The Delaunay triangulation contains $O(n)$ triangles.
3. If there are b vertices on the convex hull, then any triangulation of the points has at most $2n - 2 - b$ triangles, plus one exterior face.
4. If points are distributed according to a Poisson process in the plane with constant intensity, then each vertex has on average six surrounding triangles.
5. In the plane, the Delaunay triangulation maximizes the minimum angle. Compared to any other triangulation of the points, the smallest angle in the Delaunay triangulation is at least as large as the smallest angle in any other. However, the Delaunay triangulation does not necessarily minimize the maximum angle. The Delaunay triangulation also does not necessarily minimize the length of the edges.
6. A circle circumscribing any Delaunay triangle does not contain any other input points in its interior.
7. If a circle passing through two of the input points doesn't contain any other of them in its interior, then the segment connecting the two points is an edge of a Delaunay triangulation of the given points.
8. Each triangle of the Delaunay triangulation of a set of points in d-dimensional spaces corresponds to a facet of convex hull of the projection of the points onto a $(d+1)$-dimensional paraboloid, and vice versa.
9. The closest neighbor b to any point p is on an edge bp in the Delaunay triangulation since the nearest neighbor graph is a subgraph of the Delaunay triangulation.
10. The Delaunay triangulation is a geometric spanner: the shortest path between two vertices, along Delaunay edges, is known to be no longer than $\frac{4\pi}{3\sqrt{3}} \approx 2.418$ times the Euclidean distance between them.
11. The Euclidean minimum spanning tree of a set of points is a subset of the Delaunay triangulation of the same points, and this can be exploited to compute it efficiently.

Usage:

1. Initialize the coordinate range with `trig::LOTS`.
2. `trig::find`: Find the triangle that contains the given point.
3. `trig::add_point`: Add the point to the triangulation.
4. One certain triangle is in the triangulation if `tri::has_child () == 0`.
5. To find the neighbouring triangles of `u`, check `u.e[i].tri`, with vertice of the corresponding edge `u.p[(i + 1) % 3]` and `u.p[(i + 2) % 3]`.

```
const int N = 100000 + 5, MAX_TRIS = N * 6;
bool in_circumcircle (cp p1, cp p2, cp p3, cp p4) {
 double u11 = p1.x - p4.x, u21 = p2.x - p4.x, u31 = p3.x - p4.x;
 double u12 = p1.y - p4.y, u22 = p2.y - p4.y, u32 = p3.y - p4.y;
 double u13 = sqr (p1.x) - sqr (p4.x) + sqr (p1.y) - sqr (p4.y);
 double u23 = sqr (p2.x) - sqr (p4.x) + sqr (p2.y) - sqr (p4.y);
 double u33 = sqr (p3.x) - sqr (p4.x) + sqr (p3.y) - sqr (p4.y);
 double det = -u13 * u22 * u31 + u12 * u23 * u31 + u13 * u21 * u32 - u11 * u23 * u32 - u12 * u21 * u33 + u11 * u22 * u33;
 return sgn (det) > 0; }
double side (cp a, cp b, cp p) { return (b.x - a.x) * (p.y - a.y) - (b.y - a.y) * (p.x - a.x); }
typedef int side_t; struct tri; typedef tri* tri_r;
struct edge {
 tri_r t; side_t side;
 edge (tri_r t = 0, side_t side = 0) : t(t), side(side) {} };
struct tri {
 point p[3]; edge e[3]; tri_r child[3]; tri () {}
 tri (cp p0, cp p1, cp p2) { p[0] = p0; p[1] = p1; p[2] = p2;
  child[0] = child[1] = child[2] = 0; }
 bool has_child() const { return child[0] != 0; }
 int num_child() const { return child[0] == 0 ? 0 : child[1] == 0 ? 1 : child[2] == 0 ? 2 : 3; }
 bool contains (cp q) const {
  double a = side (p[0], p[1], q), b = side(p[1], p[2], q), c = side(p[2], p[0], q);
  return sgn (a) >= 0 && sgn (b) >= 0 && sgn (c) >= 0; } };
void set_edge (edge a, edge b) {
 if (a.t) a.t -> e[a.side] = b;
 if (b.t) b.t -> e[b.side] = a; }
class trig {
 public:
  tri tpool[MAX_TRIS], *tot;
  trig() { const double LOTS = 1E6;
   the_root = new (tot++) tri (point (-LOTS, -LOTS), point (LOTS, -LOTS), point (0, LOTS)); }
  tri_r find (cp p) const { return find (the_root, p); }
  void add_point (cp p) { add_point (find (the_root, p), p); }
 private:
  tri_r the_root;
  static tri_r find (tri_r root, cp p) {
   for(; ; ) { if (!root -> has_child ()) return root;
    else for (int i = 0; i < 3 && root -> child[i]; ++i)
     if (root -> child[i] -> contains (p))
      { root = root->child[i]; break; } } }
  void add_point (tri_r root, cp p) {
   tri_r tab, tbc, tca;
   tab = new (tot++) tri (root -> p[0], root -> p[1], p);
   tbc = new (tot++) tri (root -> p[1], root -> p[2], p);
  tca = new (tot++) tri (root -> p[2], root -> p[0], p);
  set_edge (edge (tab, 0), edge (tbc, 1)); set_edge (edge (tbc, 0), edge (tca, 1));
  set_edge (edge (tca, 0), edge (tab, 1)); set_edge (edge (tab, 2), root -> e[2]);
  set_edge (edge (tbc, 2), root -> e[0]); set_edge (edge (tca, 2), root -> e[1]);
  root -> child[0] = tab; root -> child[1] = tbc; root -> child[2] = tca;
  flip (tab, 2); flip (tbc, 2); flip (tca, 2); }
 void flip (tri_r t, side_t pi) {
  tri_r trj = t -> e[pi].t; int pj = t -> e[pi].side;
  if(!trj || !in_circumcircle (t -> p[0], t -> p[1], t -> p[2], trj -> p[pj])) return;
  tri_r trk = new (tot++) tri (t -> p[(pi + 1) % 3], trj -> p[pj], t -> p[pi]);
  tri_r trl = new (tot++) tri (trj -> p[(pj + 1) % 3], t -> p[pi], trj -> p[pj]);
  set_edge (edge (trk, 0), edge (trl, 0));
  set_edge (edge (trk, 1), t -> e[(pi + 2) % 3]); set_edge (edge (trk, 2), trj -> e[(pj + 1) % 3]);
  set_edge (edge (trl, 1), trj -> e[(pj + 2) % 3]); set_edge (edge (trl, 2), t -> e[(pi + 1) % 3]);
  t -> child[0] = trk; t -> child[1] = trl; t -> child[2] = 0;
  trj -> child[0] = trk; trj -> child[1] = trl; trj -> child[2] = 0;
  flip (trk, 1); flip (trk, 2); flip (trl, 1); flip (trl, 2); } };
void build (std::vector <point> ps, trig &t) {
 t.tot = t.tpool; std::random_shuffle (ps.begin (), ps.end ());
 for (point &p : ps) t.add_point (p); }
```

3.1.3 Fermat point

Find a point P that minimizes $|PA| + |PB| + |PC|$.

```
point fermat_point (cp a, cp b, cp c) {
 if (a == b) return a; if (b == c) return b; if (c == a) return c;
 double ab = dis (a, b), bc = dis (b, c), ca = dis (c, a);
 double cosa = dot (b - a, c - a) / ab / ca;
 double cosb = dot (a - b, c - b) / ab / bc;
 double cosc = dot (b - c, a - c) / ca / bc;
 double sq3 = PI / 3.0; point mid;
 if (sgn (cosa + 0.5) < 0) mid = a;
 else if (sgn (cosb + 0.5) < 0) mid = b;
 else if (sgn (cosc + 0.5) < 0) mid = c;
 else if (sgn (det (b - a, c - a)) < 0) mid = line_intersect (line (a, b + (c - b).rot (sq3)), line (b, c + (a - c).rot (sq3)));
 else mid = line_intersect (line (a, c + (b - c).rot (sq3)), line (c, b + (a - b).rot (sq3)));
 return mid; }
```

3.1.4 Half plane intersection

1. cut: Online in $O(n^2)$.
2. half_plane_intersect: Offline in $O(mlogm)$.

```
std::vector <point> cut (const std::vector<point> &c, line p) {
 std::vector <point> ret;
 if (c.empty ()) return ret;
 for (int i = 0; i < (int) c.size (); ++i) {
  int j = (i + 1) % (int) c.size ();
  if (turn_left (p.s, p.t, c[i])) ret.push_back (c[i]);
  if (two_side (c[i], c[j], p)) ret.push_back (line_intersect (p, line (c[i], c[j]))); }
 return ret; }
bool turn_left (cl l, cp p) { return sgn (det (l.t - l.s, p - l.s)) >= 0; }
int cmp (cp a, cp b) { return a.dim () != b.dim () ? (a.dim () < b.dim () ? -1 : 1) : -sgn (det (a, b)); }
std::vector <point> half_plane_intersect (std::vector <line> h) {
 typedef std::pair <point, line> polar;
 std::vector <polar> g; g.resize (h.size ());
 for (int i = 0; i < (int) h.size (); ++i) g[i] = std::make_pair (h[i].t - h[i].s, h[i]);
 sort (g.begin (), g.end (), [&] (const polar &a, const polar &b) {
  if (cmp (a.first, b.first) == 0) return sgn (det (a.second.t - a.second.s, b.second.t - a.second.s)) < 0;
  else return cmp (a.first, b.first) < 0; });
 h.resize (std::unique (g.begin (), g.end (), [] (const polar &a, const polar &b) { return cmp (a.first, b.first) == 0; }) - g.begin ());
 for (int i = 0; i < (int) h.size (); ++i) h[i] = g[i].second;
 int fore = 0, rear = -1; std::vector <line> ret (h.size (), line ());
 for (int i = 0; i < (int) h.size (); ++i) {
  while (fore < rear && !turn_left (h[i], line_intersect (ret[rear - 1], ret[rear]))) --rear;
  while (fore < rear && !turn_left (h[i], line_intersect (ret[fore], ret[fore + 1]))) ++fore;
  ret[++rear] = h[i]; }
 while (rear - fore > 1 && !turn_left (ret[fore], line_intersect (ret[rear - 1], ret[rear]))) --rear;
 while (rear - fore > 1 && !turn_left (ret[rear], line_intersect (ret[fore], ret[fore + 1]))) ++fore;
 if (rear - fore < 2) return std::vector <point> ();
 std::vector <point> ans; ans.resize (rear + 1);
 for (int i = 0; i < rear + 1; ++i) ans[i] = line_intersect (ret[i], ret[(i + 1) % (rear + 1)]);
 return ans; }
```

3.1.5 Intersection of a polygon and a circle

```
struct polygon_circle_intersect {
 double sector_area (cp a, cp b, const double &r) {
  double c = (2.0 * r * r - dis2 (a, b)) / (2.0 * r * r);
  return r * r * acos (c) / 2.0; }
 double area (cp a, cp b, const double &r) {
  double dA = dot (a, a), dB = dot (b, b), dC = point_to_segment (point (), line (a, b));
  if (sgn (dA - r * r) <= 0 && sgn (dB - r * r) <= 0) return det (a, b) / 2.0;
  point tA = a.unit () * r, tB = b.unit () * r;
  if (sgn (dC - r) > 0) return sector_area (tA, tB, r);
  std::vector <point> ret = line_circle_intersect (line (a, b), circle (point (), r));
  if (sgn (dA - r * r) > 0 && sgn (dB - r * r) > 0)
   return sector_area (tA, ret[0], r) + det (ret[0], ret[1]) / 2.0 + sector_area (ret[1], tB, r);
  if (sgn (dA - r * r) > 0) return det (ret[0], b) / 2.0 + sector_area (tA, ret[0], r);
  else return det (a, ret[1]) / 2.0 + sector_area (ret[1], tB, r); }
 double solve (const std::vector <point> &p, cc c) {
  double ret = 0.0;
  for (int i = 0; i < (int) p.size (); ++i) {
   int s = sgn (det (p[i] - c.c, p[(i + 1) % p.size ()] - c.c));
   if (s > 0) ret += area (p[i] - c.c, p[(i + 1) % p.size ()] - c.c, c.r);
   else ret -= area (p[(i + 1) % p.size ()] - c.c, p[i] - c.c, c.r); }
  return std::abs (ret); } };
```

3.1.6 Minimum circle

```
circle minimum_circle (std::vector <point> p) {
 circle ret; std::random_shuffle (p.begin (), p.end ());
 for (int i = 0; i < (int) p.size (); ++i) if (!in_circle (p[i], ret)) {
  ret = circle (p[i], 0); for (int j = 0; j < i; ++j) if (!in_circle (p[j], ret)) {
   ret = make_circle (p[j], p[i]); for (int k = 0; k < j; ++k)
    if (!in_circle (p[k], ret)) ret = make_circle (p[i], p[j], p[k]); } }
 return ret; }
```

3.1.7 Minkowski addition

```
// Two clockwise convex hull with the first point equal to the last each.
int i[2] = {0, 0}, len[2] = { (int) cv[0].size () - 1, (int) cv[1].size () - 1 };
std::vector <point> mnk;
mnk.push_back (cv[0][0] + cv[1][0]);
do {
 int d = dot (cv[0][i[0] + 1] - cv[0][i[0]], cv[1][i[1] + 1] - cv[1][i[1]]) >= 0;
 mnk.push_back (cv[d][i[d] + 1] - cv[d][i[d]] + mnk.back ());
 i[d] = (i[d] + 1) % len[d];
} while (i[0] || i[1]);
```

3.1.8 Nearest pair of points

Solve in range $[l, r)$. Necessary to sort p[] first. Complexity $O(n \log n)$.

```
double solve (std::vector <point> &p, int l, int r) {
 if (l + 1 >= r) return INF;
 int m = (l + r) / 2; double mx = p[m].x; std::vector <point> v;
 double ret = std::min (solve(p, l, m), solve(p, m, r));
 for (int i = l; i < r; ++i)
  if (sqr (p[i].x - mx) < ret) v.push_back (p[i]);
 sort (v.begin (), v.end (), [&] (cp a, cp b) { return a.y < b.y; } );
 for (int i = 0; i < v.size (); ++i)
  for (int j = i + 1; j < v.size (); ++j) {
   if (sqr (v[i].y - v[j].y) > ret) break;
   ret = min (ret, dis2 (v[i] - v[j])); }
 return ret; }
```

3.1.9 Triangle center

Trilinear coordinates:

1. incenter: $1:1:1$.
2. centroid: $bc:ca:ab$.
3. circumcenter: $\cos A : \cos B : \cos C$.
4. orthocenter: $\sec A : \sec B : \sec C$.
5. Non-trival Fermat point: $\csc(A+\pi/3) : \csc(B+\pi/3) : \csc(C+\pi/3)$.

```
point incenter (cp a, cp b, cp c) {
 double p = dis (a, b) + dis (b, c) + dis (c, a);
 return (a * dis (b, c) + b * dis (c, a) + c * dis (a, b)) / p; }
point circumcenter (cp a, cp b, cp c) {
 point p = b - a, q = c - a, s (dot (p, p) / 2, dot (q, q) / 2);
 return a + point (det (s, point (p.y, q.y)), det (point (p.x, q.x), s)) / det (p, q); }
point orthocenter (cp a, cp b, cp c) { return a + b + c - circumcenter (a, b, c) * 2; }
```

3.1.10 Union of circles

```
template <int MAXN = 500> struct union_circle {
 int C; circle c[MAXN]; double area[MAXN];
 struct event {
  point p; double ang; int delta;
  event (cp p = point (), double ang = 0, int delta = 0) : p(p), ang(ang), delta(delta) {}
  bool operator < (const event &a) { return ang < a.ang; } };
 void addevent(cc a, cc b, std::vector <event> &evt, int &cnt) {
  double d2 = dis2 (a.c, b.c), d_ratio = ((a.r - b.r) * (a.r + b.r) / d2 + 1) / 2,
   p_ratio = msqrt (std::max (0., -(d2 - sqr(a.r - b.r)) * (d2 - sqr(a.r + b.r)) / (d2 * d2 * 4)));
  point d = b.c - a.c, p = d.rot(PI / 2), q0 = a.c + d * d_ratio + p * p_ratio, q1 = a.c + d * d_ratio - p * p_ratio;
  double ang0 = atan2 ((q0 - a.c).y, (q0 - a.c).x), ang1 = atan2 ((q1 - a.c).x, (q1 - a.c).y);
  evt.emplace_back(q1, ang1, 1); evt.emplace_back(q0, ang0, -1); cnt += ang1 > ang0; }
 bool same(cc a, cc b) { return sgn (dis (a.c, b.c)) == 0 && sgn (a.r - b.r) == 0; }
 bool overlap(cc a, cc b) {  return sgn (a.r - b.r - dis (a.c, b.c)) >= 0;  }
 bool intersect(cc a, cc b) { return sgn (dis (a.c, b.c) - a.r - b.r) < 0; }
 void solve() {
  std::fill (area, area + C + 2, 0);
  for (int i = 0; i < C; ++i) {
   int cnt = 1; std::vector <event> evt;
   for (int j = 0; j < i; ++j) if (same (c[i], c[j])) ++cnt;
   for (int j = 0; j < C; ++j) if (j != i && !same (c[i], c[j]) && overlap (c[j], c[i])) ++cnt;
   for (int j = 0; j < C; ++j) if (j != i && !overlap (c[j], c[i]) && !overlap (c[i], c[j]) && intersect (c[i], c[j]))
    addevent (c[i], c[j], evt, cnt);
   if (evt.empty ()) area[cnt] += PI * c[i].r * c[i].r;
   else {
    std::sort (evt.begin (), evt.end ());
    evt.push_back (evt.front ());
    for (int j = 0; j + 1 < (int) evt.size (); ++j) {
     cnt += evt[j].delta; area[cnt] += det(evt[j].p, evt[j + 1].p) / 2;
     double ang = evt[j + 1].ang - evt[j].ang; if (ang < 0) ang += PI * 2;
     area[cnt] += ang * c[i].r * c[i].r / 2 - sin(ang) * c[i].r * c[i].r / 2; } } } } };
```

3.2 3D geometry

3.2.1 3D point

`rotate`: Right-hand rule with right-handed coordinates.

```
#define cp3 const point3 &
struct point3 {
 double x, y, z;
 explicit point3 (cd x = 0, cd y = 0, cd z = 0) : x (x), y (y), z (z) {} };
point3 operator + (cp3 a, cp3 b) { return point3 (a.x + b.x, a.y + b.y, a.z + b.z); }
point3 operator - (cp3 a, cp3 b) { return point3 (a.x - b.x, a.y - b.y, a.z - b.z); }
point3 operator * (cp3 a, cd b) { return point3 (a.x * b, a.y * b, a.z * b); }
point3 operator / (cp3 a, cd b) { return point3 (a.x / b, a.y / b, a.z / b); }
double dot (cp3 a, cp3 b) { return a.x * b.x + a.y * b.y + a.z * b.z; }
point3 det (cp3 a, cp3 b) { return point3 (a.y * b.z - a.z * b.y, -a.x * b.z + a.z * b.x, a.x * b.y - a.y * b.x); }
double dis2 (cp3 a, cp3 b = point3 ()) { return sqr (a.x - b.x) + sqr (a.y - b.y) + sqr (a.z - b.z); }
double dis (cp3 a, cp3 b = point3 ()) { return msqrt ( dis2 (a, b)); }
point3 rotate(cp3 p, cp3 axis, double w) {
 double x = axis.x, y = axis.y, z = axis.z;
 double s = x * x + y * y + z * z, ss = msqrt(s), cosw = cos(w), sinw = sin(w);
 double a[4][4]; memset(a, 0, sizeof (a));
 a[3][3] = 1;
 a[0][0] = ((y * y + z * z) * cosw + x * x) / s;
 a[0][1] = x * y * (1 - cosw) / s + z * sinw / ss;
 a[0][2] = x * z * (1 - cosw) / s - y * sinw / ss;
 a[1][0] = x * y * (1 - cosw) / s - z * sinw / ss;
 a[1][1] = ((x * x + z * z) * cosw + y * y) / s;
 a[1][2] = y * z * (1 - cosw) / s + x * sinw / ss;
 a[2][0] = x * z * (1 - cosw) / s + y * sinw / ss;
 a[2][1] = y * z * (1 - cosw) / s - x * sinw / ss;
 a[2][2] = ((x * x + y * y) * cos(w) + z * z) / s;
 double ans[4] = {0, 0, 0, 0}, c[4] = {p.x, p.y, p.z, 1};
 for (int i = 0; i < 4; ++i) for (int j = 0; j < 4; ++j)
  ans[i] += a[j][i] * c[j];
 return point3 (ans[0], ans[1], ans[2]);
}
```

3.2.2 3D line

```
#define cl3 const line3 &
struct line3 {
 point3 s, t;
 explicit line3 (cp3 s = point3 (), cp3 t = point3 ()) : s (s), t (t) {} };
point3 line_plane_intersection (cl3 a, cl3 b) { return a.s + (a.t - a.s) * dot (b.s - a.s, b.t - b.s) / dot (a.t - a.s, b.t - b.s); }
line3 plane_intersection (cl3 a, cl3 b) {
 point3 p = det (a.t - a.s, b.t - b.s), q = det (a.t - a.s, p), s = line_plane_intersection (line3 (a.s, a.s + q), b);
 return line3 (s, s + p); }
point3 project_to_plane (cp3 a, cl3 b) { return a + (b.t - b.s) * dot (b.t - b.s, b.s - a) / dis2 (b.t - b.s); }
```

3.2.3 3D convex hull

Input n and p. Return $face$.
Note: Remove coincide points first.

```
template <int MAXN = 500>
struct convex_hull3 {
 double mix (cp3 a, cp3 b, cp3 c) { return dot (det (a, b), c); }
 double volume (cp3 a, cp3 b, cp3 c, cp3 d) { return mix (b - a, c - a, d - a); }
 struct tri {
  int a, b, c; tri() {}
  tri(int _a, int _b, int _c): a(_a), b(_b), c(_c) {}
  double area() const { return dis (det (p[b] - p[a], p[c] - p[a])) / 2; }
  point3 normal() const { return det (p[b] - p[a], p[c] - p[a]).unit (); }
  double dis (cp3 p0) const { return dot (normal (), p0 - p[a]); } };
 int n; std::vector <point3> p;
 std::vector <tri> face, tmp;
 int mark[MAXN][MAXN], time;
 void add (int v) {
  ++time; tmp.clear ();
  for (int i = 0; i < (int) face.size (); ++i) {
   int a = face[i].a, b = face[i].b, c = face[i].c;
   if (sgn (volume (p[v], p[a], p[b], p[c])) > 0)
    mark[a][b] = mark[b][a] = mark[a][c] = mark[c][a] = mark[b][c] = mark[c][b] = time;
   else tmp.push_back (face[i]); }
  face.clear (); face = tmp;
  for (int i = 0; i < (int) tmp.size (); ++i) {
   int a = face[i].a, b = face[i].b, c = face[i].c;
   if (mark[a][b] == time) face.emplace_back (v, b, a);
   if (mark[b][c] == time) face.emplace_back (v, c, b);
   if (mark[c][a] == time) face.emplace_back (v, a, c); } }
 void reorder () {
  for (int i = 2; i < n; ++i) {
   point3 tmp = det (p[i] - p[0], p[i] - p[1]);
   if (sgn (dis (tmp))) {
    std::swap (p[i], p[2]);
    for (int j = 3; j < n; ++j)
     if (sgn (volume (p[0], p[1], p[2], p[j]))) {
      std::swap (p[j], p[3]); return; } } } }
 void build_convex () {
  reorder (); face.clear ();
  face.emplace_back (0, 1, 2);
  face.emplace_back (0, 2, 1);
  for (int i = 3; i < n; ++i) add(i); } };
```

4 Graph

```
template <int MAXN = 100000, int MAXM = 100000>
struct edge_list {
 int size, begin[MAXN], dest[MAXM], next[MAXM];
 void clear (int n) { size = 0; std::fill (begin, begin + n, -1); }
 edge_list (int n = MAXN) { clear (n); }
```

Birthday Cake

Kattis - birthdaycake

On his birthday, John's parents made him a huge birthday cake! Everyone had a wonderful dinner, and now it's time to eat the cake. There are n candles on the cake. John wants to divide the cake into n pieces so that each piece has exactly one candle on it, and there are no left-over pieces. For that, he made m cuts across the cake. Could you help check if John's cuts successfully divide the candles on the cake?

Formally, the cake is a circle of radius r centered at $(0, 0)$. The candles are n distinct points located strictly inside the circle. Each cut is a straight line $ax + by + c = 0$, described by three coefficients a, b, and c.

Input

Input starts with three integers n $(1 \le n \le 50)$, m $(1 \le m \le 15)$, and r $(1 \le r \le 100)$ on the first line.

The next n lines give the locations of the candles. Each line has two integers x and y giving the coordinates of one candle $(0 \le \sqrt{x^2 + y^2} < r)$.

The next m lines give the coefficients of the cutting lines. Each line has three integers a, b, and c $(0 \le |a|, |b| \le 100, 0 \le |c| \le 20\,000)$ describing a line of the form $ax + by + c = 0$. The values a and b are not both zero.

All candles and lines are distinct. No candle is on a cut line. No line is completely outside or tangent to the cake. The input guarantees that the number of cake pieces remains the same if any cut line is shifted by at most 10^{-4} in any direction. The input also guarantees that each candle remains in the interior of the same piece of cake if its position is shifted by at most 10^{-4} in any direction.

Output

Output "`yes`" if John's cuts successfully divide the cake so that each piece he obtains has exactly one candle on it. Otherwise, output "`no`".

Sample 1

Input copy

```
4 2 3
0 1
1 0
-1 0
0 -1
-1 1 0
2 1 0
```

Output copy

```
yes
```

Sample 2

Input copy

```
4 3 3
0 1
1 2
-1 2
0 -1
-1 1 -2
-1 -1 2
0 -1 0
```

Output copy

```
no
```

Sample 3

Input copy

```
3 2 3
2 1
0 0
-1 -2
1 1 -2
3 6 12
```

Output copy

```
yes
```

Sample 4

Input	Output

```
3 1 2
0 0
-1 1
1 -1
-2 2 1
```
	`no`

Largest Triangle

Kattis - largesttriangle

Given N points on a 2-dimensional space, determine the area of the largest triangle that can be formed using 3 of those N points. If there is no triangle that can be formed, the answer is 0.

Input

The first line contains an integer N ($3 \le N \le 5\,000$) denoting the number of points. Each of the next N lines contains two integers x and y ($0 \le x, y \le 4 \cdot 10^7$). There are **no** specific constraints on these N points, i.e. the points are not necessarily distinct, the points are not given in specific order, there may be 3 or more collinear points, etc.

Output

Print the answer in one line. Your answer should have an absolute error of at most 10^{-5}.

Sample 1

Input copy

```
7
0 0
0 5
7 7
0 10
0 0
20 0
10 10
```

Output copy

```
100.00000
```

Covered Walkway

Kattis - coveredwalkway

Your university wants to build a new walkway, and they want at least part of it to be covered. There are certain points which must be covered. It doesn't matter if other points along the walkway are covered or not.

The building contractor has an interesting pricing scheme. To cover the walkway from a point at x to a point at y, they will charge $c + (x - y)^2$, where c is a constant. Note that it is possible for $x = y$. If so, then the contractor would simply charge c.

Given the points along the walkway and the constant c, what is the minimum cost to cover the walkway?

Input

Input consists of a single test case. The test case will begin with a line with two integers, n ($1 \leq n \leq 1\,000\,000$) and c ($1 \leq c \leq 10^9$), where n is the number of points which must be covered, and c is the contractor's constant. Each of the following n lines will contain a single integer, representing a point along the walkway that must be covered. The points will be in order, from smallest to largest. All of the points will be in the range from 1 to 10^9, inclusive.

Output

Output a single integer, representing the minimum cost to cover all of the specified points. All possible inputs yield answers which will fit in a signed 64-bit integer.

Sample 1

Input copy	Output copy

10 5000
1
23
45
67
101
124
560
789
990
1019
	30726