```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
#include <math.h>
#include <complex>
using namespace std;
#define rep(i,n) for (int i=0; i < (n); i++)

// 主に以下の資料を参考に作成した。
// - http://www.prefield.com/algorithm
// - http://www.deqnotes.net/acmicpc/2d_geometry/
// - https://github.com/infnty/acm/tree/master/lib/geometry
// - サークルの先輩が作ったライブラリ

/* 基本要素 */

typedef double D;       // 座標値の型。doubleかlong doubleを想定
typedef complex<D> P;   // Point
typedef pair<P, P> L;   // Line
typedef vector<P> VP;
const D EPS = 1e-9;     // 許容誤差。問題によって変える
#define X real()
#define Y imag()
#define LE(n,m) ((n) < (m) + EPS)
#define GE(n,m) ((n) + EPS > (m))
#define EQ(n,m) (abs((n)-(m)) < EPS)

// 内積  dot(a,b) = |a||b|cosθ
D dot(P a, P b) {
  return (conj(a)*b).X;
}
// 外積  cross(a,b) = |a||b|sinθ
D cross(P a, P b) {
  return (conj(a)*b).Y;
}

// 点の進行方向
int ccw(P a, P b, P c) {
  b -= a;  c -= a;
  if (cross(b,c) >  EPS) return +1;  // counter clockwise
  if (cross(b,c) < -EPS) return -1;  // clockwise
  if (dot(b,c)   < -EPS) return +2;  // c--a--b on line
  if (norm(b) < norm(c)) return -2;  // a--b--c on line or a==b
  return 0;                          // a--c--b on line or a==c or b==c
}

/* 交差判定  直線・線分は縮退してはならない。接する場合は交差するとみなす。isecはintersectの略 */

// 直線と点
bool isecLP(P a1, P a2, P b) {
  return abs(ccw(a1, a2, b)) != 1;  // return EQ(cross(a2-a1, b-a1), 0); と等価
}

// 直線と直線
bool isecLL(P a1, P a2, P b1, P b2) {
  return !isecLP(a2-a1, b2-b1, 0) || isecLP(a1, b1, b2);
}

// 直線と線分
bool isecLS(P a1, P a2, P b1, P b2) {
  return cross(a2-a1, b1-a1) * cross(a2-a1, b2-a1) < EPS;
}

// 線分と線分
bool isecSS(P a1, P a2, P b1, P b2) {
  return ccw(a1, a2, b1)*ccw(a1, a2, b2) <= 0 &&
         ccw(b1, b2, a1)*ccw(b1, b2, a2) <= 0;
}

// 線分と点
bool isecSP(P a1, P a2, P b) {
```

```
  return !ccw(a1, a2, b);
}


/* 距離　各直線・線分は縮退してはならない */

// 点pの直線aへの射影点を返す
P proj(P a1, P a2, P p) {
  return a1 + dot(a2-a1, p-a1)/norm(a2-a1) * (a2-a1);
}

// 点pの直線aへの反射点を返す
P reflection(P a1, P a2, P p) {
  return 2.0*proj(a1, a2, p) - p;
}

D distLP(P a1, P a2, P p) {
  return abs(proj(a1, a2, p) - p);
}

D distLL(P a1, P a2, P b1, P b2) {
  return isecLL(a1, a2, b1, b2) ? 0 : distLP(a1, a2, b1);
}

D distLS(P a1, P a2, P b1, P b2) {
  return isecLS(a1, a2, b1, b2) ? 0 : min(distLP(a1, a2, b1), distLP(a1, a2, b2));
}

D distSP(P a1, P a2, P p) {
  P r = proj(a1, a2, p);
  if (isecSP(a1, a2, r)) return abs(r-p);
  return min(abs(a1-p), abs(a2-p));
}

D distSS(P a1, P a2, P b1, P b2) {
  if (isecSS(a1, a2, b1, b2)) return 0;
  return min(min(distSP(a1, a2, b1), distSP(a1, a2, b2)),
             min(distSP(b1, b2, a1), distSP(b1, b2, a2)));
}

// 2直線の交点
P crosspointLL(P a1, P a2, P b1, P b2) {
  D d1 = cross(b2-b1, b1-a1);
  D d2 = cross(b2-b1, a2-a1);
  if (EQ(d1, 0) && EQ(d2, 0)) return a1;  // same line
  if (EQ(d2, 0)) throw "kouten ga nai";   // 交点がない
  return a1 + d1/d2 * (a2-a1);
}


/* 円 */

D distLC(P a1, P a2, P c, D r) {
  return max(distLP(a1, a2, c) - r, 0.0);
}

D distSC(P a1, P a2, P c, D r) {
  D dSqr1 = norm(c-a1), dSqr2 = norm(c-a2);
  if (dSqr1 < r*r ^ dSqr2 < r*r) return 0;  // 円が線分を包含するとき距離0ならここをORに変える
  if (dSqr1 < r*r & dSqr2 < r*r) return r - sqrt(max(dSqr1, dSqr2));
  return max(distSP(a1, a2, c) - r, 0.0);
}

VP crosspointLC(P a1, P a2, P c, D r) {
  VP ps;
  P ft = proj(a1, a2, c);
  if (!GE(r*r, norm(ft-c))) return ps;

  P dir = sqrt(max(r*r - norm(ft-c), 0.0)) / abs(a2-a1) * (a2-a1);
  ps.push_back(ft + dir);
  if (!EQ(r*r, norm(ft-c))) ps.push_back(ft - dir);
  return ps;
}

```

```
D distCC(P a, D ar, P b, D br) {
  D d = abs(a-b);
  return GE(d, abs(ar-br)) ? max(d-ar-br, 0.0) : abs(ar-br) - d;
}

// 2円の交点
VP crosspointCC(P a, D ar, P b, D br) {
  VP ps;
  P ab = b-a;
  D d = abs(ab);
  D crL = (norm(ab) + ar*ar - br*br) / (2*d);
  if (EQ(d, 0) || ar < abs(crL)) return ps;

  P abN = ab * P(0, sqrt(ar*ar - crL*crL) / d);
  P cp = a + crL/d * ab;
  ps.push_back(cp + abN);
  if (!EQ(norm(abN), 0)) ps.push_back(cp - abN);
  return ps;
}

// 点pから円aへの接線の接点
VP tangentPoints(P a, D ar, P p) {
  VP ps;
  D sin = ar / abs(p-a);
  if (!LE(sin, 1)) return ps;  // ここでNaNも弾かれる
  D t = M_PI_2 - asin(min(sin, 1.0));
  ps.push_back(                a + (p-a)*polar(sin, t));
  if (!EQ(sin, 1)) ps.push_back(a + (p-a)*polar(sin, -t));
  return ps;
}

// 2円の共通接線。返される各直線に含まれる頂点は円との接点となる
vector<L> tangentLines(P a, D ar, P b, D br) {
  vector<L> ls;
  D d = abs(b-a);
  rep (i,2) {
    D sin = (ar - (1-i*2)*br) / d;
    if (!LE(sin*sin, 1)) break;
    D cos = sqrt(max(1 - sin*sin, 0.0));
    rep (j,2) {
      P n = (b-a) * P(sin, (1-j*2)*cos) / d;
      ls.push_back(L(a + ar*n, b + (1-i*2)*br*n));
      if (cos < EPS) break;  // 重複する接線を無視（重複していいならこの行不要）
    }
  }
  return ls;
}

// 三角形の外心。点a,b,cは同一線上にあってはならない
P circumcenter(P a, P b, P c) {
  a = (a-c)*0.5;
  b = (b-c)*0.5;
  return c + crosspointLL(a, a*P(1,1), b, b*P(1,1));
}

// 点aと点bを通り、半径がrの円の中心を返す
VP circlesPointsRadius(P a, P b, D r) {
  VP cs;
  P abH = (b-a)*0.5;
  D d = abs(abH);
  if (d == 0 || d > r) return cs;  // 必要なら !LE(d,r) として円1つになる側へ丸める
  D dN = sqrt(r*r - d*d);          // 必要なら max(r*r - d*d, 0) とする
  P n = abH * P(0,1) * (dN / d);
  cs.push_back(a + abH + n);
  if (dN > 0) cs.push_back(a + abH - n);
  return cs;
}

// 点aと点bを通り、直線lに接する円の中心
VP circlesPointsTangent(P a, P b, P l1, P l2) {
  P n = (l2-l1) * P(0,1);
  P m = (b-a) * P(0,0.5);
  D rC = dot((a+b)*0.5-l1, n);
  D qa = norm(n)*norm(m) - dot(n,m)*dot(n,m);
```

```
  D qb = -rC * dot(n,m);
  D qc = norm(n)*norm(m) - rC*rC;
  D qd = qb*qb - qa*qc;  // qa*k^2 + 2*qb*k + qc = 0

  VP cs;
  if (qd < -EPS) return cs;
  if (EQ(qa, 0)) {
    if (!EQ(qb, 0)) cs.push_back((a+b)*0.5 - m * (qc/qb/2));
    return cs;
  }
  D t = -qb/qa;
  cs.push_back(              (a+b)*0.5 + m * (t + sqrt(max(qd, 0.0))/qa));
  if (qd > EPS) cs.push_back((a+b)*0.5 + m * (t - sqrt(max(qd, 0.0))/qa));
  return cs;
}

// 点集合を含む最小の円の中心
P minEnclosingCircle(const VP& ps) {
  P c;
  double move = 0.5;
  rep(i,39) {  // 2^(-39-1) \approx 0.9e-12
    rep(t,50) {
      D max = 0;
      int k = 0;
      rep (j, ps.size()) if (max < norm(ps[j]-c)) {
        max = norm(ps[j]-c);
        k = j;
      }
      c += (ps[k]-c) * move;
    }
    move /= 2;
  }
  return c;
}


/* 多角形 */

// 頂点の順序（sortやmax_elementに必要）
namespace std {
  bool operator<(const P a, const P b) {
    return a.X != b.X ? a.X < b.X : a.Y < b.Y;
  }
}

// 凸包
VP convexHull(VP ps) {  // 元の点集合がソートされていいならVP&に
  int n = ps.size(), k = 0;
  sort(ps.begin(), ps.end());
  VP ch(2*n);
  for (int i = 0; i < n; ch[k++] = ps[i++]) // lower-hull
    while (k >= 2 && ccw(ch[k-2], ch[k-1], ps[i]) <= 0) --k;  // 余計な点も含むなら == -1 とする
  for (int i = n-2, t = k+1; i >= 0; ch[k++] = ps[i--]) // upper-hull
    while (k >= t && ccw(ch[k-2], ch[k-1], ps[i]) <= 0) --k;
  ch.resize(k-1);
  return ch;
}

// 凸判定。縮退を認めないならccwの判定部分を != 1 とする
bool isCcwConvex(const VP& ps) {
  int n = ps.size();
  rep (i, n) if (ccw(ps[i], ps[(i+1) % n], ps[(i+2) % n]) == -1) return false;
  return true;
}

// 凸多角形の内部判定  O(n)
// 点が領域内部なら1、境界上なら2、外部なら0を返す
int inConvex(P p, const VP& ps) {
  int n = ps.size();
  int dir = ccw(ps[0], ps[1], p);
  rep (i, n) {
    int ccwc = ccw(ps[i], ps[(i + 1) % n], p);
    if (!ccwc) return 2;  // 線分上に存在
    if (ccwc != dir) return 0;
```

```
  }
  return 1;
}

// 凸多角形の内部判定  O(logn)
// 点が領域内部なら1、境界上なら2、外部なら0を返す
int inCcwConvex(const VP& ps, P p) {
  int n = ps.size();
  P g = (ps[0] + ps[n / 3] + ps[n*2 / 3]) / 3.0;
  if (g == p) return 1;
  P gp = p - g;

  int l = 0, r = n;
  while (l + 1 < r) {
    int mid = (l + r) / 2;
    P gl = ps[l] - g;
    P gm = ps[mid] - g;
    if (cross(gl, gm) > 0) {
      if (cross(gl, gp) >= 0 && cross(gm, gp) <= 0) r = mid;
      else l = mid;
    }
    else {
      if (cross(gl, gp) <= 0 && cross(gm, gp) >= 0) l = mid;
      else r = mid;
    }
  }
  r %= n;
  D cr = cross(ps[l] - p, ps[r] - p);
  return EQ(cr, 0) ? 2 : cr < 0 ? 0 : 1;
}

// 多角形の内部判定
// 点が領域内部なら1、境界上なら2、外部なら0を返す
int inPolygon(const VP& ps, P p) {
  int n = ps.size();
  bool in = false;
  rep (i, n) {
    P a = ps[i] - p;
    P b = ps[(i + 1) % n] - p;
    if (EQ(cross(a,b), 0) && LE(dot(a,b), 0)) return 2;
    if (a.Y > b.Y) swap(a,b);
    if ((a.Y*b.Y < 0 || (a.Y*b.Y < EPS && b.Y > EPS)) && LE(cross(a, b), 0)) in = !in;
  }
  return in;
}

// 凸多角形クリッピング
VP convexCut(const VP& ps, P a1, P a2) {
  int n = ps.size();
  VP ret;
  rep(i,n) {
    int ccwc = ccw(a1, a2, ps[i]);
    if (ccwc != -1) ret.push_back(ps[i]);
    int ccwn = ccw(a1, a2, ps[(i + 1) % n]);
    if (ccwc * ccwn == -1) ret.push_back(crosspointLL(a1, a2, ps[i], ps[(i + 1) % n]));
  }
  return ret;
}

// 凸多角形の直径（最遠点対）
pair<int, int> convexDiameter(const VP& ps) {
  int n = ps.size();
  int i = min_element(ps.begin(), ps.end()) - ps.begin();
  int j = max_element(ps.begin(), ps.end()) - ps.begin();
  int maxI, maxJ;
  D maxD = 0;
  rep(_, 2*n) {
    if (maxD < norm(ps[i]-ps[j])) {
      maxD = norm(ps[i]-ps[j]);
      maxI = i;
      maxJ = j;
    }
    if (cross(ps[i]-ps[(i+1) % n], ps[(j+1) % n]-ps[j]) <= 0) j = (j+1) % n;
    else                                                      i = (i+1) % n;
```

```
  }
  return make_pair(maxI, maxJ);
}

// 多角形の符号付面積
D area(const VP& ps) {
  D a = 0;
  rep (i, ps.size()) a += cross(ps[i], ps[(i+1) % ps.size()]);
  return a / 2;
}

// 多角形の幾何学的重心
P centroid(const VP& ps) {
  int n = ps.size();
  D aSum = 0;
  P c;
  rep (i, n) {
    D a = cross(ps[i], ps[(i+1) % n]);
    aSum += a;
    c += (ps[i] + ps[(i+1) % n]) * a;
  }
  return 1 / aSum / 3 * c;
}

// ボロノイ領域
VP voronoiCell(P p, const VP& ps, const VP& outer) {
  VP cl = outer;
  rep (i, ps.size()) {
    if (EQ(norm(ps[i]-p), 0)) continue;
    P h = (p+ps[i])*0.5;
    cl = convexCut(cl, h, h + (ps[i]-h)*P(0,1) );
  }
  return cl;
}

/* 幾何グラフ */

struct Edge {
  int from, to;
  D cost;
  Edge(int from, int to, D cost) : from(from), to(to), cost(cost) {}
};
struct Graph {
  int n;
  vector<vector<Edge> > edges;
  Graph(int n) : n(n), edges(n) {}
  void addEdge(Edge e) {
    edges[e.from].push_back(e);
    edges[e.to].push_back(Edge(e.to, e.from, e.cost));
  }
};

// 線分アレンジメント（線分の位置関係からグラフを作成）
Graph segmentArrangement(const vector<L>& segs, VP& ps) {
  int n = segs.size();
  rep (i, n) {
    ps.push_back(segs[i].first);
    ps.push_back(segs[i].second);
    rep (j, i) {
      if (isecSS(                 segs[i].first, segs[i].second, segs[j].first, segs[j].second))
        ps.push_back(crosspointLL(segs[i].first, segs[i].second, segs[j].first, segs[j].second));
    }
  }
  sort(ps.begin(), ps.end());
  ps.erase(unique(ps.begin(), ps.end()), ps.end());

  int m = ps.size();
  Graph gr(m);
  vector<pair<D, int> > list;
  rep (i, n) {
    list.clear();
    rep (j, m) {
      if (isecSP(segs[i].first, segs[i].second, ps[j]))
        list.push_back(make_pair(norm(segs[i].first-ps[j]), j));
```

```
    }
    sort(list.begin(), list.end());
    rep (j, list.size() - 1) {
      int a = list[j  ].second;
      int b = list[j+1].second;
      gr.addEdge(Edge(a, b, abs(ps[a]-ps[b])));
    }
  }
  return gr;
}

// 可視グラフ（点集合から見える位置へ辺を張ったグラフ）
Graph visibilityGraph(const VP& ps, const vector<VP>& objs) {
  int n = ps.size();
  Graph gr(n);
  rep (i,n) rep (j,i) {
    P a = ps[i], b = ps[j];
    if (!EQ(norm(a-b), 0)) rep (k, objs.size()) {
      const VP& obj = objs[k];
      int inStA = inConvex(a, obj);
      int inStB = inConvex(b, obj);
      if ((inStA ^ inStB) % 2 || inStA * inStB != 1 && inConvex((a+b)*0.5, obj) == 1) goto skip;
      rep (l, obj.size()) {
        P cur = obj[l];
        P next = obj[(l + 1) % obj.size()];
        if (isecSS(a, b, cur, next) && !isecSP(cur, next, a) && !isecSP(cur, next, b)) goto skip;
      }
    }
    gr.addEdge( Edge(i, j, abs(a-b)) );
    skip: {}
  }
  return gr;
}


/* その他 */

// 重複する線分を併合する
vector<L> mergeSegments(vector<L> segs) {
  int n = segs.size();
  rep (i,n) if (segs[i].second < segs[i].first) swap(segs[i].second, segs[i].first);

  rep (i,n) rep (j,i) {
    L &l1 = segs[i], &l2 = segs[j];
    if (EQ(cross(l1.second-l1.first, l2.second-l2.first), 0)
        && isecLP(l1.first, l1.second, l2.first)
        && ccw   (l1.first, l1.second, l2.second) != 2
        && ccw   (l2.first, l2.second, l1.second) != 2) {
      segs[j] = L(min(l1.first, l2.first), max(l1.second, l2.second));
      segs[i--] = segs[--n];
      break;
    }
  }
  segs.resize(n);
  return segs;
}


// この辺にコードを載せるほどでもないが重要な定理とか図とか書いておくとよい気がします

// 余弦定理
// △ABC において、a = BC, b = CA, c = AB としたとき
// a^2 = b^2 + c^2 ? 2bc cos ∠CAB

// ヘロンの公式
// 3辺の長さがa,b,cである三角形の面積T
// T = sqrt{ s(s-a)(s-b)(s-c) }, s = (a+b+c)/2

// ピックの定理
// 多角形の頂点が全て格子点上にあり、内部に穴がないとき
// S = i + b/2 - 1 (S:多角形の面積, i: 多角形の内部にある格子点の数, b: 辺上の格子点の数)
```