```cpp
bool fge(double x, double y) { return x >= y - eps; }
double fsqrt(double x) { return feq(x, 0) ? 0 : sqrt(x); }

// polygon

struct pt_t {
  double x, y;
  pt_t operator+(const pt_t &p) const { return { x + p.x, y + p.y }; }
  pt_t operator-(const pt_t &p) const { return { x - p.x, y - p.y }; }
  pt_t operator*(const double &c) const { return { x * c, y * c }; }
  bool operator<(const pt_t &another) const {
    return (x != another.x ? x < another.x : y < another.y);
  }
};

// aX + bY + c = 0
struct line_t {
  double a, b, c;
};

// (X - x)^2 + (Y - y)^2 = r^2
struct circle_t {
  double x, y, r;
};

// normal vector = (a, b), passing p
line_t solve_line(double a, double b, pt_t p) {
  return { a, b, -a * p.x - b * p.y };
}

// passing p, q
line_t solve_line(pt_t p, pt_t q) {
  return solve_line(q.y - p.y, -q.x + p.x, p);
}

// t should be radius
pt_t rot(pt_t p, double r) {
  return {
    cos(r) * p.x - sin(r) * p.y,
    sin(r) * p.x + cos(r) * p.y
  };
}

double norm2(pt_t p) {
  return p.x * p.x + p.y * p.y;
}

double norm(pt_t p) {
  return sqrt(norm2(p));
}

double dist(line_t l, pt_t p) {
  return abs(l.a * p.x + l.b * p.y + l.c)
    / sqrt(l.a * l.a + l.b * l.b);
}

bool on_same_line(pt_t s, pt_t t, pt_t p) {
  line_t l = solve_line(s, t);
  if (feq(dist(l, p), 0)) return true;
  else return false;
```

```cpp
}

bool in_segment(pt_t s, pt_t t, pt_t p) {
  line_t l = solve_line(s, t);
  if (feq(dist(l, p), 0)
    && fge(p.x, min(s.x, t.x))
    && fge(max(s.x, t.x), p.x)
    && fge(p.y, min(s.y, t.y))
    && fge(max(s.y, t.y), p.y)) return true;
  else return false;
}

// (NAN, NAN) if lines coincide with each other
// (INF, INF) if lines are parallel but not coincide
pt_t cross_point(line_t l, line_t m) {
  double d = l.a * m.b - l.b * m.a;
  if (feq(d, 0)) {
    if (feq(l.a * m.c - l.c * m.a, 0)) return { INF, INF };
    else return { NAN, NAN };
  }
  else {
    double x = l.b * m.c - m.b * l.c;
    double y = l.a * m.c - m.a * l.c;
    return { x / d, y / -d };
  }
}

// if size is 0, then not crossed
vector<pt_t> cross_point(circle_t f, line_t l) {
  double d = dist(l, { f.x, f.y });
  if (!fge(f.r, d)) return {};
  line_t m = solve_line(l.b, -l.a, { f.x, f.y });
  pt_t p = cross_point(l, m);
  if (feq(d, f.r)) return { p };
  else {
    pt_t u = { l.b, -l.a };
    pt_t v = u * (sqrt(pow(f.r, 2) - pow(d, 2)) / norm(u));
    return { p + v, p - v };
  }
}

// if size is 0, then not crossed
vector<pt_t> cross_point(circle_t f, circle_t g) {
  line_t l = {
    -2 * f.x + 2 * g.x,
    -2 * f.y + 2 * g.y,
    (f.x * f.x + f.y * f.y - f.r * f.r) - (g.x * g.x + g.y * g.y - g.r * g.r)
  };
  return cross_point(f, l);
}

// tangent points of f through p
// if size is 0, then p is strictly contained in f
// if size is 1, then p is on f
// otherwise size is 2
vector<pt_t> tangent_point(circle_t f, pt_t p) {
  vector<pt_t> ret;
  double d2 = norm2(pt_t({ f.x, f.y }) - p);
  double r2 = d2 - f.r * f.r;
  if (fge(r2, 0)) {
```

```
    circle_t g = { p.x, p.y, fsqrt(r2) };
    ret = cross_point(f, g);
  }
  return ret;
}

// tangent lines of f through p
// if size is 0, then p is strictly contained in f
// if size is 1, then p is on f
// otherwise size is 2
vector<line_t> tangent_line(circle_t f, pt_t p) {
  vector<pt_t> qs = tangent_point(f, p);
  vector<line_t> ret(qs.size());
  Loop(i, ret.size()) {
    ret[i] = solve_line(qs[i].x - f.x, qs[i].y - f.y, qs[i]);
  }
  return ret;
}

// tangent points on f through which there is a line tangent to g
// if size is 0, then one is strictly contained in the other
// if size is 1, then they are touched inside
// if size is 2, then they are crossed
// if size is 3, then they are touched outside
// otherwise size is 4
vector<pt_t> tangent_point(circle_t f, circle_t g) {
  vector<pt_t> ret;
  double d2 = norm2({ g.x - f.x, g.y - f.y });
  vector<double> r2(2);
  r2[0] = d2 - f.r * f.r + 2 * f.r * g.r;
  r2[1] = d2 - f.r * f.r - 2 * f.r * g.r;
  Loop(k, 2) {
    if (fge(r2[k], 0)) {
      circle_t g2 = { g.x, g.y, fsqrt(r2[k]) };
      vector<pt_t> buf = cross_point(f, g2);
      Loop(i, buf.size()) ret.push_back(buf[i]);
    }
  }
  return ret;
}

// common tangent lines between two circles
// if size is 0, then one is strictly contained in the other
// if size is 1, then they are touched inside
// if size is 2, then they are crossed
// if size is 3, then they are touched outside
// otherwise size is 4
vector<line_t> tangent_line(circle_t f, circle_t g) {
  vector<pt_t> qs = tangent_point(f, g);
  vector<line_t> ret(qs.size());
  Loop(i, ret.size()) {
    ret[i] = tangent_line(f, qs[i]).front();
  }
  return ret;
}

// inner product
double dot(pt_t p, pt_t q) {
  return p.x * q.x + p.y * q.y;
}
```

```

// outer product
double cross(pt_t p, pt_t q) {
  return p.x * q.y - p.y * q.x;
}

// suppose a is counterclockwise, a.size() >= 3
double polygon_area(vector<pt_t> a) {
  double ret = 0;
  Loop(i, a.size()) {
    int j = (i + 1 < a.size() ? i + 1 : 0);
    ret += cross(a[i], a[j]);
  }
  ret = abs(ret) / 2;
  return ret;
}

class Triangulate {
private:
  vvi tri_ids;
  vector<vector<pt_t>> tri_pts;
  vector<pt_t> a;
  bool enable(pt_t p, pt_t q, pt_t r) {
    line_t l = solve_line(q, r);
    if (feq(dist(l, p), 0)) return false;
    if (fge(cross(q - p, r - p), 0)) return true;
    else return false;
  }
  void contraction(vi &ids) {
    int n = ids.size();
    if (n < 3) return;
    Loop(i, n) {
      int id_p = (i - 1 + n) % n;
      int id_q = i;
      int id_r = (i + 1) % n;
      pt_t p = a[ids[id_p]];
      pt_t q = a[ids[id_q]];
      pt_t r = a[ids[id_r]];
      line_t l = solve_line(p, r);
      if (feq(dist(l, q), 0)) {
        ids.erase(ids.begin() + i);
        contraction(ids);
        return;
      }
    }
  }
  void divide(vi &ids) {
    contraction(ids);
    int n = ids.size();
    if (n < 3) return;
    Loop(i, n) {
      int id_p = (i - 1 + n) % n;
      int id_q = i;
      int id_r = (i + 1) % n;
      pt_t p = a[ids[id_p]];
      pt_t q = a[ids[id_q]];
      pt_t r = a[ids[id_r]];
      if (enable(p, q, r)) {
        line_t l = solve_line(p, r);
        bool judge = true;
```

```
        Loop(j, n) {
          if (j == id_p || j == id_q || j == id_r) continue;
          pt_t xp = a[ids[j]];
          if (in_triangle({ p,q,r }, xp)) judge = false;
        }
        if (judge) {
          tri_ids.push_back({ id_p, id_q, id_r });
          tri_pts.push_back({ p, q, r });
          ids.erase(ids.begin() + i);
          divide(ids);
          return;
        }
      }
    }
  }
  int in_triangle(const vector<pt_t> &a, pt_t p) {
    int ret = 2;
    Loop(i, 3) {
      int j = (i + 1) % 3;
      line_t l = solve_line(a[i], a[j]);
      double d = dist(l, p);
      if (feq(d, 0)) ret = 1;
      else if (fge(M_PI, angle(a[j] - a[i], p - a[i])));
      else return 0;
    }
    return ret;
  }
public:
  // each triangle will be represented counterclockwisely
  Triangulate(const vector<pt_t> &a) {
    this->a = a;
    vi ids(a.size());
    Loop(i, ids.size()) ids[i] = i;
    divide(ids);
  }
  vvi get_ids() {
    return tri_ids;
  }
  vector<vector<pt_t>> get_pts() {
    return tri_pts;
  }
  // suppose a is counterclockwise, a.size() >= 3
  // return 0 if not, return 1 if on line, return 2 if strictly included
  int in_polygon(pt_t p) {
    int ret = 0;
    Loop(i, tri_pts.size()) {
      if (in_triangle(tri_pts[i], p)) {
        ret = 2;
      }
    }
    if (ret != 0) {
      Loop(i, a.size()) {
        int j = (i + 1) % a.size();
        if (in_segment(a[i], a[j], p)) ret = 1;
      }
    }
    return ret;
  }
};
```

```cpp
vector<pt_t> convex_hull(vector<pt_t> ps) {
  int n = ps.size();
  sort(ps.begin(), ps.end());
  Loop(i, n - 1) ps.push_back(ps[n - 2 - i]);
  vector<pt_t> ret;
  int m = 2;
  Loop(i, n * 2 - 1) {
    if (i == n) m = ret.size() + 1;
    while (ret.size() >= m) {
      int k = ret.size();
      if (in_segment(ret[k - 2], ps[i], ret[k - 1])) break;
      else if (fge(cross(ret[k - 1] - ret[k - 2], ps[i] - ret[k - 2]), 0)) ret.pop_back();
      else break;
    }
    ret.push_back(ps[i]);
  }
  ret.pop_back();
  reverse(ret.begin(), ret.end());
  return ret;
}

// angle [0, 2PI) of vector p to vector q
double angle(pt_t p, pt_t q) {
  p = p * (1.0 / norm(p));
  q = q * (1.0 / norm(q));
  double r0 = acos(max(min(dot(p, q), 1.0), -1.0));
  double r1 = asin(max(min(dot(p, q), 1.0), -1.0));
  if (r1 >= 0) return r0;
  else return 2 * M_PI - r0;
}
```