```cpp
#include <vector>
#include <algorithm>
#include <cstdlib>
#include <complex>
#include <iostream>
using namespace std;

typedef complex<long double> point;
#define sz(a) ((int)(a).size())
#define all(n) (n).begin(),(n).end()
#define EPS 1e-9
#define OO 1e9
#define X real()
#define Y imag()
#define vec(a,b) ((b)-(a))
#define polar(r,t) ((r)*exp(point(0,(t))))
#define angle(v) (atan2((v).Y,(v).X))
#define length(v) ((long double)hypot((v).Y,(v).X))
#define lengthSqr(v) (dot(v,v))
#define dot(a,b) ((conj(a)*(b)).real())
#define cross(a,b) ((conj(a)*(b)).imag())
#define rotate(v,t) (polar(v,t))
#define rotateabout(v,t,a)  (rotate(vec(a,v),t)+(a))
#define reflect(p,m) ((conj((p)/(m)))*(m))
#define normalize(p) ((p)/length(p))
#define same(a,b) (lengthSqr(vec(a,b))<EPS)
#define mid(a,b) (((a)+(b))/point(2,0))
#define perp(a) (point(-(a).Y,(a).X))
#define colliner pointOnLine

enum STATE {
        IN, OUT, BOUNDRY
};

bool intersect(const point &a, const point &b, const point &p, const point &q,
              point &ret) {

    //handle degenerate cases

    double d1 = cross(p - a, b - a);
    double d2 = cross(q - a, b - a);
    ret = (d1 * q - d2 * p) / (d1 - d2);
    if(fabs(d1 - d2) > EPS) return 1;
    return 0;
}

bool pointOnLine(const point& a, const point& b, const point& p) {
    if(same(a,b)) return same(a,p);
    return fabs(cross(vec(a,b),vec(a,p))) < EPS;
}

bool pointOnRay(const point& a, const point& b, const point& p) {
    if(same(a,b)) return same(a,p);
    point v1 = normalize(vec(a,b));
    point v2 = normalize(vec(a,p));
    return same(v1,v2);
}

bool pointOnSegment(const point& a, const point& b, const point& p) {
        if (same(a, b))
                return same(a,p);

        return pointOnRay(a, b, p) && pointOnRay(b, a, p);
}

long double pointLineDist(const point& a, const point& b, const point& p) {
        if (same(a,b))
                return hypot(a.X - p.X, a.Y - p.Y);

        return fabs(cross(vec(a,b),vec(a,p)) / length(vec(a,b)));
}

long double pointSegmentDist(const point& a, const point& b, const point& p) {
        if (dot(vec(a,b),vec(a,p)) < EPS)
                return length(vec(a,p));
        if (dot(vec(b,a),vec(b,p)) < EPS)
                return length(vec(b,p));
        return pointLineDist(a, b, p);
}

int lineLatticePointsCount(int x1, int y1, int x2, int y2) {
        return abs(gcd(x1 - x2, y1 - y2)) + 1;
}

long double triangleAreaBH(long double b, long double h) {
        return b * h / 2;
}

long double triangleArea2sidesAngle(long double a, long double b, long double t) {
        return fabs(a * b * sin(t) / 2);
}

long double triangleArea2anglesSide(long double t1, long double t2,
                long double s) {
        return fabs(s * s * sin(t1) * sin(t2) / (2 * sin(t1 + t2)));
}

long double triangleArea3sides(long double a, long double b, long double c) {
        long double s = ((a + b + c) / 2);
        return sqrt(s * (s - a) * (s - b) * (s - c));
}

long double triangleArea3points(const point& a, const point& b, const point& c) {
        return fabs(cross(a,b) + cross(b,c) + cross(c,a)) / 2;
}

//count interior
int picksTheorm(int a, int b) {
        return a - b / 2 + 1;
}

//get angle opposite to side a
long double cosRule(long double a, long double b, long double c) {
        // Handle denom = 0
        long double res = (b * b + c * c - a * a) / (2 * b * c);
        if (res > 1)
                res = 1;
        if (res < -1)
                res = -1;
        return acos(res);
}

long double sinRuleAngle(long double s1, long double s2, long double a1) {
        // Handle denom = 0
        long double res = s2 * sin(a1) / s1;
        if (res > 1)
                res = 1;
        if (res < -1)
                res = -1;
        return asin(res);
}

long double sinRuleSide(long double s1, long double a1, long double a2) {
        // Handle denom = 0
        long double res = s1 * sin(a2) / sin(a1);
        return fabs(res);
```

```
}

int circleLineIntersection(const point& p0, const point& p1, const point& cen,
                long double rad, point& r1, point & r2) {

        if (same(p0,p1)){
                if(fabs(lengthSqr(vec(p0,cen))-(rad*rad)) < EPS)
                {
                    r1 = r2 = p0;
                    return 1;
                }
                return 0;
        }
        long double a, b, c, t1, t2;
        a = dot(p1-p0,p1-p0);
        b = 2 * dot(p1-p0,p0-cen);
        c = dot(p0-cen,p0-cen) - rad * rad;
        double det = b * b - 4 * a * c;
        int res;
        if (fabs(det) < EPS)
                det = 0, res = 1;
        else if (det < 0)
                res = 0;
        else
                res = 2;
        det = sqrt(det);
        t1 = (-b + det) / (2 * a);
        t2 = (-b - det) / (2 * a);
        r1 = p0 + t1 * (p1 - p0);
        r2 = p0 + t2 * (p1 - p0);
        return res;
}

int circleCircleIntersection(const point &c1, const long double&r1,
                const point &c2, const long double&r2, point &res1, point
&res2) {
        if (same(c1,c2) && fabs(r1 - r2) < EPS) {
                res1 = res2 = c1;
                return fabs(r1) < EPS ? 1 : OO;
        }
        long double len = length(vec(c1,c2));
        if (fabs(len - (r1 + r2)) < EPS || fabs(fabs(r1 - r2) - len) < EPS) {
                point d, c;
                long double r;
                if (r1 > r2)
                        d = vec(c1,c2), c = c1, r = r1;
                else
                        d = vec(c2,c1), c = c2, r = r2;
                res1 = res2 = normalize(d) * r + c;
                return 1;
        }
        if (len > r1 + r2 || len < fabs(r1 - r2))
                return 0;
        long double a = cosRule(r2, r1, len);
        point c1c2 = normalize(vec(c1,c2)) * r1;
        res1 = rotate(c1c2,a) + c1;
        res2 = rotate(c1c2,-a) + c1;
        return 2;
}

void circle2(const point& p1, const point& p2, point& cen, long double& r) {
        cen = mid(p1,p2);
        r = length(vec(p1,p2)) / 2;
}

bool circle3(const point& p1, const point& p2, const point& p3, point& cen,
                long double& r) {
        point m1 = mid(p1,p2);
        point m2 = mid(p2,p3);
        point perp1 = perp(vec(p1,p2));
        point perp2 = perp(vec(p2,p3));
        bool res = intersect(m1, m1 + perp1, m2, m2 + perp2, cen);
        r = length(vec(cen,p1));
        return res;
}

STATE circlePoint(const point & cen, const long double & r, const point& p) {
        long double lensqr = lengthSqr(vec(cen,p));
        if (fabs(lensqr - r * r) < EPS)
                return BOUNDRY;
        if (lensqr < r * r)
                return IN;
        return OUT;
}

int tangentPoints(const point & cen, const long double & r, const point& p,
                point &r1, point &r2) {
        STATE s = circlePoint(cen, r, p);
        if (s != OUT) {
                r1 = r2 = p;
                return s == BOUNDRY;
        }
        point cp = vec(cen,p);
        long double h = length(cp);
        long double a = acos(r / h);
        cp = normalize(cp) * r;
        r1 = rotate(cp,a) + cen;
        r2 = rotate(cp,-a) + cen;
        return 2;
}

// minimum enclosing circle
//init p array with the points and ps with the number of points
//cen and rad are result circle
//you must call random_shuffle(p,p+ps); before you call mec
#define MAXPOINTS 100000
point p[MAXPOINTS], r[3], cen;
int ps, rs;
long double rad;

void mec() {
        if (rs == 3) {
                circle3(r[0], r[1], r[2], cen, rad);
                return;
        }
        if (rs == 2 && ps == 0) {
                circle2(r[0], r[1], cen, rad);
                return;
        }
        if (!ps) {
                cen = r[0];
                rad = 0;
                return;
        }
        ps--;
        mec();
        if (circlePoint(cen, rad, p[ps]) == OUT) {
                r[rs++] = p[ps];
                mec();
                rs--;
        }
        ps++;

}

//to check if the points are sorted anti-clockwise or clockwise
//remove the fabs at the end and it will return -ve value if clockwise
long double polygonArea(const vector<point>&p) {
        long double res = 0;
        for (int i = 0; i < sz(p); i++) {
                int j = (i + 1) % sz(p);
                res += cross(p[i],p[j]);
        }
        return fabs(res) / 2;
}

// return the centroid point of the polygon
// The centroid is also known as the "centre of gravity" or the "center of mass". The position of the centroid
// assuming the polygon to be made of a material of uniform density.
point polyginCentroid(vector<point> &polygon) {
        point res(0, 0);
        long double a = 0;

        for (int i = 0; i < (int) polygon.size(); i++) {
                int j = (i + 1) % polygon.size();

                res.X += (polygon[i].X + polygon[j].X) * (polygon[i].X * polygon[j].Y
                                - polygon[j].X * polygon[i].Y);

                res.Y += (polygon[i].Y + polygon[j].Y) * (polygon[i].X * polygon[j].Y
                                - polygon[j].X * polygon[i].Y);

                a += polygon[i].X * polygon[j].Y - polygon[i].Y * polygon[j].X;
        }

        a *= 0.5;
        res.X /= 6 * a;
        res.Y /= 6 * a;

        return res;
}

int picksTheorm(vector<point>& p) {
        long double area = 0;
        int bound = 0;
        for (int i = 0; i < sz(p); i++) {
                int j = (i + 1) % sz(p);
                area += cross(p[i],p[j]);
                point v = vec(p[i],p[j]);
                bound += abs(__gcd((int) v.X, (int) v.Y));
        }
        area /= 2;
        area = fabs(area);
        return round(area - bound / 2 + 1);
}

void polygonCut(const vector<point>& p, const point&a, const point&b, vector<
                point>& res) {
        res.clear();
        for (int i = 0; i < sz(p); i++) {
                int j = (i + 1) % sz(p);
                bool in1 = cross(vec(a,b),vec(a,p[i])) > EPS;
                bool in2 = cross(vec(a,b),vec(a,p[j])) > EPS;
                if (in1)
                        res.push_back(p[i]);
                if (in1 ^ in2) {
                        point r;
                        intersect(a, b, p[i], p[j], r);
                        res.push_back(r);
                }
        }
}

```

```
//assume that both are anti-clockwise
void convexPolygonIntersect(const vector<point>& p, const vector<point>& q,
                vector<point>& res) {
        res = q;
        for (int i = 0; i < sz(p); i++) {
                int j = (i + 1) % sz(p);
                vector<point> temp;
                polygonCut(res, p[i], p[j], temp);
                res = temp;
                if (res.empty())
                        return;
        }
}

void voronoi(const vector<point> &pnts, const vector<point>& rect, vector<
                vector<point> > &res) {
        res.clear();
        for (int i = 0; i < sz(pnts); i++) {
                res.push_back(rect);
                for (int j = 0; j < sz(pnts); j++) {
                        if (j == i)
                                continue;
                        point p = perp(vec(pnts[i],pnts[j]));
                        point m = mid(pnts[i],pnts[j]);
                        vector<point> temp;
                        polygonCut(res.back(), m, m + p, temp);
                        res.back() = temp;
                }
        }
}

STATE pointInPolygon(const vector<point>& p, const point &pnt) {
        point p2 = pnt + point(1, 0);
        int cnt = 0;
        for (int i = 0; i < sz(p); i++) {
                int j = (i + 1) % sz(p);
                if (pointOnSegment(p[i], p[j], pnt))
                        return BOUNDRY;
                point r;
                intersect(pnt, p2, p[i], p[j], r);
                if (!pointOnRay(pnt, p2, r))
                        continue;
                if (same(r,p[i]) || same(r,p[j]))
                        if (fabs(r.Y - min(p[i].Y, p[j].Y)) < EPS)
                                continue;
                if (!pointOnSegment(p[i], p[j], r))
                        continue;
                cnt++;
        }
        return cnt & 1 ? IN : OUT;
}

struct cmp {
        point about;
        cmp(point c) {
                about = c;
        }
        bool operator()(const point& p, const point& q) const {
                double cr = cross(vec(about,p),vec(about,q));
                if (fabs(cr) < EPS)
                        return make_pair(p.Y, p.X) < make_pair(q.Y, q.X);
                return cr > 0;
        }
};

void sortAntiClockWise(vector<point>& pnts){
        point mn(1 / 0.0, 1 / 0.0);
        for (int i = 0; i < sz(pnts); i++)
            if (make_pair(pnts[i].Y, pnts[i].X) < make_pair(mn.Y, mn.X))
```

```
                mn = pnts[i];

        sort(all(pnts),cmp(mn));
}

void convexHull(vector<point> pnts, vector<point> &convex) {
        sortAntiClockWise(pnts);
        convex.clear();
        convex.push_back(pnts[0]);
        if (sz(pnts) == 1)
                return;
        convex.push_back(pnts[1]);
        for (int i = 2; i <= sz(pnts); i++) {
                point c = pnts[i % sz(pnts)];
                while (sz(convex) > 1) {
                        point b = convex.back();
                        point a = convex[sz(convex) - 2];
                        if (cross(vec(b,a),vec(b,c)) < -EPS)
                                break;
                        convex.pop_back();
                }
                if (i < sz(pnts))
                        convex.push_back(pnts[i]);
        }
}
```