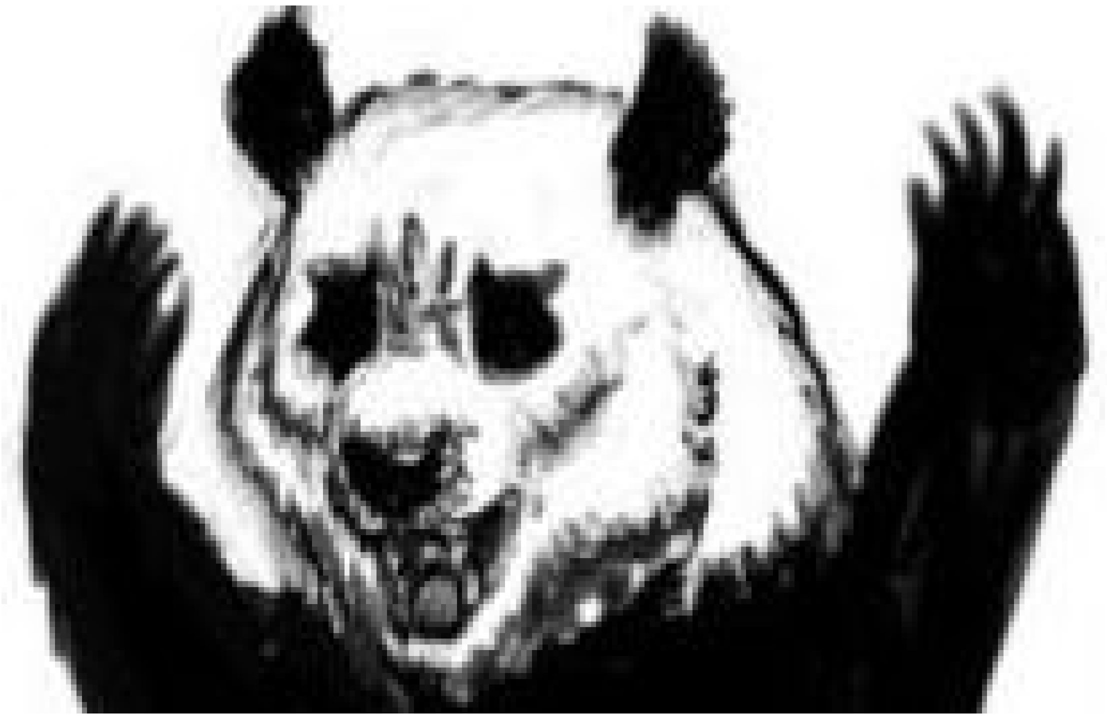
Note:when adding new part, highlighted comment should convert to the underline



Chongqing Univercity
Standard Code Library Extended

By st.Krwlng

```
//this section include the basic structure and operation in 2D-plane calculation

#define square(x) (x)*(x)
#define getvec(x, y) ((y)-(x))
#define xmul(x1, y1, x2, y2) ((x1)*(y2)-(x2)*(y1))
struct line { double a, b, c; };   //ax + by = c
struct pnt {
	double x, y;
	pnt operator+(const pnt &p) const{
		pnt ret; ret.x = x + p.x;
		ret.y = y + p.y; return ret;
	}
	pnt operator-(const pnt &p) const{
		pnt ret; ret.x = x - p.x;
		ret.y = y - p.y; return ret;
	}
	pnt operator*(const double &c) const{
		pnt ret; ret.x = x * c;
		ret.y = y * c; return ret;
	}
	pnt operator/(const double &c) const{
		pnt ret; ret.x = x / c;
		ret.y = y / c; return ret;
	}
	bool operator<(const pnt &p) const{
		if (fabs(x-p.x)<eps) return y-p.y < -eps;
		return x-p.x < -eps;
	}
};
typedef pnt vec;
const double eps = 1e-6; //1e-8, 1e-10, 1e-16
const double pi = acos(-1.0);

double nummul(const vec &v1, const vec &v2) { return v1.x * v2.x + v1.y * v2.y; }
double submul(const vec &v1, const vec &v2) { return xmul(v1.x, v1.y, v2.x, v2.y); }

int lessver(const pnt &p1, const pnt &p2) {
	if (fabs(p1.y-p2.y)<eps) return p1.x-p2.x < -eps;
	return p1.y-p2.y < -eps;
}

double getdis(const pnt &p1, const pnt &p2) { return sqrt(square(p2.x-p1.x) + square(p2.y-p1.y)); }
double getdis(const line &l, const pnt &p) { return fabs(l.a * p.x + l.b * p.y - l.c) / sqrt(square(l.a) + square(l.b)); }
double oridis(const line &l, const pnt &p) { return l.a * p.x + l.b * p.y - l.c; }

pnt getcrs(const line &l1, const line &l2) {
	pnt ret; ret.x = xmul(l1.b, l1.c, l2.b, l2.c) / xmul(l1.b, l1.a, l2.b, l2.a);
	ret.y = xmul(l1.a, l1.c, l2.a, l2.c) / xmul(l1.a, l1.b, l2.a, l2.b); return ret;
}

pnt rotate(const pnt &p, double ang) {
	vec v = { sin(ang), cos(ang) };
	pnt ret = { submul(p, v), nummul(p, v) }; return ret;
}
pnt stretch_rotate(const pnt &p, const vec &v1, const vec &v2) {
	vec v = { submul(v1, v2)/nummul(v1, v1), nummul(v1, v2)/nummul(v1, v1) };
	pnt ret = { submul(p, v), nummul(p, v) }; return ret;
} // v1 -> v2, len(v1) -> len(v2).

vec uvec(const vec &v) {
	double len = sqrt(nummul(v, v));
	vec ret; ret.x = v.x / len; ret.y = v.y / len; return ret;
}

line getline(const pnt &p1, const pnt &p2) {
	vec v = getvec(p1, p2); v = uvec(v);
	line ret; ret.a = -v.y; ret.b = v.x;
	ret.c = ret.a * p1.x + ret.b * p1.y; return ret;
} // error if p1 == p2
```

```
line getline(const pnt &p, const vec &dv) {
	vec v = uvec(v);
	line ret; ret.a = -v.y; ret.b = v.x;
	ret.c = ret.a * p.x + ret.b * p.y; return ret;
}
line getmidver(const pnt &p1, const pnt &p2) {
	vec v = getvec(p1, p2); v = v/2; pnt mid = p1 + v; v = uvec(v);
	line ret; ret.a = v.x; ret.b = v.y; ret.c = ret.a*mid.x + ret.b*mid.y; return ret;
}

//it should be garanteed that p[i] < q[i]
bool checkcrs(pnt* p, pnt* q, line* l, int i, int j, pnt &crs) {
	if (q[i] < p[j] || q[j] < p[i]) return false;
	if (submul(getvec(p[i], q[i]), getvec(p[i], p[j])) * submul(getvec(p[i], q[i]), getvec(p[i], q[j])) > eps) return false;
	if (submul(getvec(p[j], q[j]), getvec(p[j], p[i])) * submul(getvec(p[j], q[j]), getvec(p[j], q[i])) > eps) return false;
	if (fabs(submul(getvec(p[i], q[i]), getvec(p[j], q[j]))) < eps) if (q[i] < q[j]) crs = q[i]; else crs = q[j];
	else crs = getcrs(l[i], l[j]); return true;
}
```

```
//in this section, the operator+(pnt),-(pnt),*(double),/(double) should be overloaded in definition


//get the line which is the reflection of l1 through l0
//Caution: the line of l0 and l1 must be unified in construction, or it'll have some precision problem!
line lineref(const line &l0, const line &l1)
{
	pnt p; if (fabs(l0.a) > fabs(l0.b)) { p.x = l0.c / l0.a; p.y = 0; } else { p.x = 0; p.y = l0.c / l0.b; }
	vec v; v.x = l0.b; v.y = -l0.a; double len = sqrt(square(l0.a) + square(l0.b));
	double sa = v.y / len, ca = v.x / len;
	double a = l1.a * square(ca) - l1.a * square(sa) + 2 * l1.b * sa * ca;
	double b = l1.b * square(sa) - l1.b * square(ca) + 2 * l1.a * sa * ca;
	line ret; ret.a = a; ret.b = b; ret.c = l1.c + (a - l1.a) * p.x + (b - l1.b) * p.y;
	return ret;
}

//get the number of intersection of a circle and a line, as well as the point of intersection
int linexcircle(const line &l, const pnt &cen, double rad, pnt *p)
{
	double ver, hor; vec v1, v2;
	v1 = (vec){ l.a, l.b}; v1 = uvec(v1); v2 = (vec){ -v1.y, v1.x};
	ver = oridis(l, cen); if (fabs(ver)-rad > eps) return 0;
	if (fabs(fabs(ver)-rad)<eps) { p[0] = cen - v1 * ver; return 1; }
	hor = sqrt(square(rad)-square(ver));
	p[0] = cen - v1*ver + v2*hor; p[1] = cen - v1*ver - v2*hor; return 2;
}

//get the number of intersection of a circle and a segment, as well as the point of intersection
//If there exists two such points, the first one has the less value of dmul(p1->p2, p1->crs).
void segxcircle(const pnt &p1, const pnt &p2, const pnt &cen, double rad, int &top, pnt *p) {
	vec v1, v2; pnt tp; double ver, hor;
	v1 = uvec(p2-p1); v2 = (vec){ -v1.y, v1.x };
	ver = nummul(cen, v2) - nummul(p1, v2); if (fabs(ver)-rad > eps) return;
	hor = sqrt(square(rad)-square(ver));
	tp = cen - v2 * ver - v1 * hor; if (nummul(tp-p1, tp-p2) < eps) p[top++] = tp;
	if (hor > eps) { tp = tp + v1 * hor * 2; if (nummul(tp-p1, tp-p2) < eps) p[top++] = tp; }
}

//check if the circle is totaly in the given convex.
bool circleincvx(int n, pnt *p, const pnt &cen, double rad) {
	for (int i = 0; i < n; ++i) if (submul(p[i]-cen, p[i+1]-cen)<-eps) return false;
	for (int i = 0; i < n; ++i) if (fabs(oridis(getline(p[i], p[i+1]), cen))-rad<-eps) return false;
	return true;
}

//check if the point is in the circle or on the border of the circle
bool pntincircle(const pnt &p, const pnt &cen, double rad) { return getdis(p, cen)-rad < eps; }

//get the cross points of two circle.
int circlecrs(const pnt &p1, const pnt &p2, double r1, double r2, pnt *p) {
	double len = getdis(p1, p2); vec v = uvec(p2-p1);
	if (len + r1 - r2 < -eps ||  len + r2 - r1 < -eps || r1 + r2 - len < -eps) return 0;
	if (fabs(len+r1-r2) < eps) { p[0] = p1 - v * r1; return 1; }
	if (fabs(len+r2-r1) < eps || fabs(r1+r2-len) < eps) { p[0] = p1 + v * r1; return 1; }
	double ang1 = atan2(v.y, v.x), ang2 = acos((square(len)+square(r1)-square(r2))/(2*len*r1));
	v = (vec){ cos(ang1+ang2), sin(ang1+ang2) }; p[0] = p1 + v * r1;
	v = (vec){ cos(ang1-ang2), sin(ang1-ang2) }; p[1] = p1 + v * r1;
	return 2;
}
```

```
//get the intersection of a circle and a convex, return the number of the points.
//description: the array evt reflects the condition of the same postion at q. If evt[i] equals 1, it means the part of ret from q[i] to q[i+1] is an arc, otherwise it's a segment.
//caution: the length of arc exists in the ret must be larger than 0(means the condition that the ret is a entire circle can be expressed valid), but the degeneration should take extra O(n) time at last four lines of code.
int circlexcvx(int n, pnt *p, const pnt &cen, double rad, pnt *q, bool *evt) {
	int i, j, k, l, ret; pnt tp[4];
	if (circleincvx(n, p, cen, rad)) { q[0] = q[1] = (pnt){ cen.x + rad, cen.y }; evt[0] = 1; return 1; }
	for (ret = i = 0; i < n; ++i) {
		tp[0] = p[i]; l = 1; segxcircle(p[i], p[i+1], cen, rad, l, tp);
		for (k = j = 1; j < l; ++j) if (tp[j] != p[i] && tp[j] != p[i+1]) tp[k++] = tp[j];
		for (j = 0; j < k; ++j) if (incircle(tp[j], cen, rad)) { evt[ret] = 0; q[ret++] = tp[j]; }
		if (!incircle(p[i+1], cen, rad) && incircle(tp[k-1], cen, rad)) evt[ret-1] = 1;
	} q[ret] = q[0]; evt[ret] = evt[0]; l = ret; if (!ret) return 0;
	for (ret = i = 0; i < l; ++i) {
		if (!evt[i] || q[i] != q[i+1]) { q[ret] = q[i]; evt[ret] = evt[i]; ret++; }
	} if (!ret) { ret = 1; evt[0] = 0; }
	q[ret] = q[0]; evt[ret] = evt[0]; return ret;
}

//get the judgement whether ang3 is between the arc, all the ang has the constrains of [-pi, pi)
bool arcbetween(double ang1, double ang2, bool ati, double ang3)
{
	if (fabs(ang1-ang3) < eps || fabs(ang2-ang3) < eps) return true;
	return (ati ^ ((ang1-ang3) * (ang2 - ang3) < -eps));
}

//get the enclosing circle of the triangle by the given 3 points of the vertices of triangle. return the radius of the circle and the center of the circle will be assigned to the last paremeter.
double outcircle(const pnt &p1, const pnt &p2, const pnt &p3, pnt &cen)
{
	line l1 = getmidver(p1, p2), l2 = getmidver(p2, p3);
	cen = getcrs(l1, l2); return getdis(cen, p1);
}

//get the internal tangent circle of the triangle. return the radius of the circle and the center of the circle will be assigned to the last paremeter.
double incircle(const pnt &p1, const pnt &p2, const pnt &p3, pnt &cen)
{
	vec v1, v2; line l1, l2;
	v1 = uvec(p2-p1); v2 = uvec(p3-p1); l1 = getline(p1, v1+v2);
	v1 = uvec(p3-p2); v2 = uvec(p1-p2); l2 = getline(p2, v1+v2);
	cen = getcrs(l1, l2); return fabs(submul(v2, cen-p2));
}

//get the common point of three verticle line of a triangle.
pnt orthocenter(const pnt &p1, const pnt &p2, const pnt &p3)
{
	vec v1, v2, ret; line l1, l2;
	v1 = uvec(p3-p2); l1 = (line){ v1.x, v1.y, nummul(v1, p1) };
	v2 = uvec(p3-p1); l2 = (line){ v2.x, v2.y, nummul(v2, p2) };
	ret = getcrs(l1, l2); return ret;
}

//get the barycenter of a polygon given in clockwise
pnt barycenter(int n, pnt *p)
{
	int i; double tot, ta; pnt ret;
	p[n] = p[0]; ret = (pnt){ 0, 0};
	for (tot = i = 0; i < n; ++i) {
		ta = submul(p[i], p[i+1]);
		tot += ta; ret.x += (p[i].x+p[i+1].x)*ta/3; ret.y += (p[i].y+p[i+1].y)*ta/3;
	} ret.x /= tot; ret.y /= tot; return ret;
}
```

```
//get the fermat point which has the least total distance to the given n points.
#define NUM 2 //C = (NUM+1)^2
#define BD 2 //C = (BD*2+1)^2
const double ATT = 0.9, EPS = 1e-6; //when ADd = 0.9, C = log10(range/EPS) * 10

double valuate(int n, pnt *p, const pnt &tp) {
	double ret; int i;
	for (ret = i = 0; i < n; ++i) ret += getdis(tp, p[i]);
	return ret;
}
pnt fermatpoint(int n, pnt* p)
{
	double lft, rit, up, dn, stx, sty, dx, dy, lst, val, tv, ttv;
	pnt ret, vp, tp, ttp; int i, j, k, l;
	lft = dn = inf; rit = up = -inf;
	for (i = 0; i < n; ++i) {
		lft = getmin(lft, p[i].x); rit = getmax(rit, p[i].x);
		dn = getmin(dn, p[i].y); up = getmax(up, p[i].y);
	} stx = rit-lft; sty = up-dn; lst = inf;
	for (i = 0; i <= NUM; ++i) for (j = 0; j <= NUM; ++j) {
		vp = (pnt){ lft+stx/NUM*i, dn+sty/NUM*j }; val = valuate(n, p, vp);
		for (dx = stx/NUM/BD, dy = sty/NUM/BD; dx > EPS || dy > EPS; dx *= ATT, dy *= ATT) {
			for (tp = vp, tv = val, k = -BD; k <= BD; ++k) for (l = -BD; l <= BD; ++l) {
				ttp = (pnt){vp.x+dx*k, vp.y+dy*l}; ttv = valuate(n, p, ttp);
				if (ttv < tv) { tv = ttv; tp = ttp; }
			} vp = tp; val = tv;
		} if (val < lst) { lst = val; ret = vp; }
	} return vp;
}

//check whether the given point is in the polygon, no matter the order of points about polygon is clockwise or conter-clockwise
bool pntinpoly(int n, pnt *p, const pnt &p1) {
	int i, cnt; pnt p2 = { -inf, p1.y }; p[n] = p[0];
	for (cnt = i = 0; i < n; ++i) {
		if (fabs(submul(p[i+1]-p1, p[i]-p1))<eps && nummul(p[i+1]-p1, p[i]-p1)<eps) return true;
		if (fabs(p[i+1].y-p[i].y)<eps) continue;
		if (fabs(p[i+1].y-p1.y)<eps && p[i+1].x-p1.x<eps) { if (p[i+1].y-p[i].y<eps) cnt++; continue; }
		if (fabs(p[i].y-p1.y) < eps && p[i].x-p1.x < eps) { if (p[i].y-p[i+1].y<eps) cnt++; continue; }
		if (submul(p2-p1, p[i]-p1)*submul(p2-p1, p[i+1]-p1)<eps &&
			submul(p[i+1]-p[i], p1-p[i])*submul(p[i+1]-p[i], p2-p[i])<eps) cnt++;
	} return cnt % 2;
}

//get the distance of nearest pair of points.
//the ret can be saved in the global variable, in order to record the index of two points.
int seq[N], que[N];

int cmpx(const int &i, const int &j) {
	if (fabs(p[i].x-p[j].x)<eps) return p[i].y-p[j].y < -eps;
	return p[i].x-p[j].x < -eps;
}
int cmpy(const int &i, const int &j) {
	if (fabs(p[i].y-p[j].y)<eps) return p[i].x-p[j].x < -eps;
	return p[i].y-p[j].y < -eps;
}

double lstpair(pnt *p, int lft, int rit) {
	int i, j, cnt, mid; double ret;
	if (lft >= rit) return inf;
	mid = (lft+rit) / 2; ret = getmin(lstpair(p, lft, mid), lstpair(p, mid+1, rit));
	for (cnt = 0, i = lft; i <= rit; ++i) if (fabs(p[seq[i]].x-p[seq[mid]].x)-ret<eps) que[cnt++] = seq[i];
	sort(que, que+cnt, cmpy);
	for (i = 0; i < cnt; ++i) for (j = i+1; j < cnt && j < i+8; ++j) ret = getmin(ret, getdis(p[que[i]], p[que[j]]));
	return ret;
}
double nrstpair(int n, pnt *p) {
	for (int i = 0; i < n; ++i) seq[i] = i; sort(seq, seq+n, cmpx);
	return lstpair(p, 0, n-1);
}
```

```
//This section is about the incremental method


//calculate the convex of the combine of several half plane, output the number of distinct point at the border, as well as each point and line in counter-clockwise order.
//the answer of lines is in the seq array, the endpoint of line seq[i] is p[i] and p[i+1].
//Caution: the line here must be unified before calculation.
line bdr1 = { 1.0, 0.0, -inf }, bdr2 = { -1.0, 0.0, -inf }, bdr3 = { 0.0, 1.0, -inf }, bdr4 = { 0.0, -1.0, -inf };

int hfpcmp(const int &x, const int &y) {
	if (fabs(ang[x]-ang[y])<eps) return l[x].c - l[y].c > eps;
	return ang[x] - ang[y] < -eps;
}

int hfplane(int n, line *l, pnt *p, int *seq, double *ang)
{
	int i, j, top, bot, ret;
	l[n++] = bdr1; l[n++] = bdr2; l[n++] = bdr3; l[n++] = bdr4;
	for (i = 0; i < n; ++i) { ang[i] = atan2(l[i].b, l[i].a); if (fabs(ang[i]-pi)<eps) ang[i] = -pi; }
	for (i = 0; i < n; ++i) seq[i] = i; std::sort(seq, seq+n, hfpcmp);
	for (i = j = 1; i < n; ++i) if (fabs(ang[seq[i]]-ang[seq[j-1]])>eps) seq[j++] = seq[i]; n = j;
	for (i = 1, top = bot = 0; i < n; ++i) {
		while (top - bot && oridis(l[seq[i]], p[top]) < -eps) --top;
		p[top+1] = getcrs(l[seq[i]], l[seq[top]]); seq[++top] = seq[i];
		if (ang[seq[i]] > -eps) break;
	} p[bot] = (pnt){ -inf*2, -inf*2 };
	for (++i; i < n; ++i) {
		if (oridis(l[seq[i]], p[bot]) > -eps) continue;
		while (top - bot && oridis(l[seq[i]], p[top]) < -eps) --top;
		if (top == bot) break; p[top+1] = getcrs(l[seq[i]], l[seq[top]]); seq[++top] = seq[i];
		while (oridis(l[seq[top]], p[bot+1]) < -eps) ++bot; p[bot] = getcrs(l[seq[top]], l[seq[bot]]);
	} if (i < n) return 0;
	for (ret = 0, i = bot; i < top; ++i) if (fabs(p[i].x-p[i+1].x)>eps || fabs(p[i].y-p[i+1].y)>eps) {
		seq[ret] = seq[i]; p[ret] = p[i]; ret++;
	} if (!ret || fabs(p[i].x-p[0].x)>eps || fabs(p[i].y-p[0].y)>eps) {
		seq[ret] = seq[i]; p[ret] = p[i]; ret++;
	} p[ret] = p[0]; return ret;
}


//graham method to get convex
int tmp[N];
int vercmp(const int &x, const int &y) { return lessver(p[x], p[y]); }

int graham(int n, pnt* p, int *seq) {
	int i, top, bot;
	for (i = 0; i < n; ++i) tmp[i] = i; sort(tmp, tmp+n, vercmp);
	for (seq[top = bot = 0] = tmp[0], i = 1; i < n; ++i) {
		while (top > bot && submul(getvec(p[seq[top-1]], p[seq[top]]), getvec(p[seq[top]], p[tmp[i]])) < eps) --top;
		seq[++top] = tmp[i];
	} bot = top;
	for (i = n-2; i >= 0; --i) {
		while (top > bot && submul(getvec(p[seq[top-1]], p[seq[top]]), getvec(p[seq[top]], p[tmp[i]])) < eps) --top;
		seq[++top] = tmp[i];
	} return top;
}
```

```
//melkman method to get convex
int tmp[N];
int melkman(int n, pnt *p, int *seq)
{
	int i, j, bot, top;
	for (i = top = 0; i < n; ++i) if (p[i] < p[top]) top = i;
	for (j = top, i = 0; i < n; ++i, j = (j+1)%n) tmp[i] = j;
	seq[n] = tmp[0]; seq[n-1] = tmp[1]; seq[n+1] = tmp[1];
	for (i = 2; i < n; ++i) if (fabs(submul(getvec(p[seq[n]], p[seq[n-1]]), getvec(p[seq[n-1]], p[tmp[i]])))<eps) {
		if (p[seq[1]] < p[tmp[i]]) { seq[n-1] = tmp[i]; seq[n+1] = tmp[i]; }
	} else break; top = n+1; bot = n-1;
	for (; i < n; ++i) {
		if (submul(getvec(p[seq[top-1]], p[seq[top]]), getvec(p[seq[top]], p[tmp[i]])) > -eps &&
			submul(getvec(p[seq[bot+1]], p[seq[bot]]), getvec(p[seq[bot]], p[tmp[i]])) < eps) continue;
		while (submul(getvec(p[seq[top-1]], p[seq[top]]), getvec(p[seq[top]], p[tmp[i]])) < eps) --top;
		while (submul(getvec(p[seq[bot+1]], p[seq[bot]]), getvec(p[seq[bot]], p[tmp[i]])) > -eps) ++bot;
		seq[++top] = seq[--bot] = tmp[i];
	} for (i = bot; i <= top; ++i) seq[i-bot] = seq[i]; return top - bot;
}
```

```
//Randomize method to get smallest enclosing circle for N given points
int tmp[N], seq[N];
double enclosng_circle(int n, pnt *p, pnt &cen) {
	int i, j, k, l; double rad;
	for (i = 0; i < n; ++i) seq[i] = i; cen = p[0]; rad = 0.0;
	for (i = 2; i < n; ++i) {
		j = rand()%(n-i) + i; k = seq[j]; seq[j] = seq[i]; seq[i] = k;
		if (getdis(cen, p[seq[i]])-rad < eps) continue;
		tmp[0] = seq[i]; tmp[1] = seq[1]; seq[i] = seq[0];
		cen = (p[tmp[0]]+p[tmp[1]]) / 2.0; rad = getdis(cen, p[tmp[0]]);
		for (j = 2; j <= i; ++j) {
			if (getdis(cen, p[seq[j]])-rad < eps) { tmp[j] = seq[j]; continue; }
			tmp[j] = tmp[1]; tmp[1] = seq[j];
			cen = (p[tmp[0]]+p[tmp[1]]) / 2.0; rad = getdis(cen, p[tmp[0]]);
			for (k = 2; k <= j; ++k) if (getdis(cen, p[tmp[k]])-rad > eps) {
				l = tmp[k]; tmp[k] = tmp[2]; tmp[2] = l;
				rad = outcircle(p[tmp[0]], p[tmp[1]], p[tmp[2]], cen);
			}
		} for (j = 0; j <= i; ++j) seq[j] = tmp[j];
	} return rad;
}
```

```
//get the farthest point for each given point in convex, store in array ati(i's farthest point is at index ati[i])
//Caution:   1. The order of convex must be counter-clockwise
//           2. The length of array(N) should be 3 times larger than the data region.(The same for array p)
//           3. The line construct function here has no need to unify.
int pre[N], pro[N], pos[N];
bool vst[N]; line ll[N];

void farestpair(int n, pnt *p, int *ati) {
	int i, j, k, l, st, ed;
	for (i = 0; i < n; ++i) { p[i].x *= 2; p[i].y *= 2; }
	for (i = 0; i < n; ++i) { pre[i] = i-1; pro[i] = i+1; } pre[0] = n-1; pro[n-1] = 0;
	for (i = 0; i < n; ++i) p[i+n] = p[i+2*n] = p[i];
	for (l = st = i = 0; l <= n; i = pro[i], ++l) {
		ll[i] = getline(p[i], p[pro[i]]);
		while((i!=st||l==n)&&submul(p[pro[i]]-p[i],p[pos[pre[i]]]-p[i])>=0&&oridis(ll[i],p[pos[pre[i]]])+(l!=n-1?0:1)>0) {
			if (l == n && i == st) { st = pro[i]; l = n-1; }
			pro[pre[i]] = pro[i]; pre[pro[i]] = pre[i]; i = pre[i];
			ll[i] = getline(p[i], p[pro[i]]);
		} if (i == st && l != n) ed = pro[i]; else ed = pos[pre[i]];
		while (submul(p[pro[i]]-p[i], p[ed]-p[i]) < 0 || oridis(ll[i], p[ed])-(l!=n-1?1:0) < 0) ++ed; pos[i] = ed;
	} memset(vst, 0, sizeof(vst));
	for (i = st; !vst[i]; i = pro[i]) {
		j = pos[i]; k = pos[pro[i]]; vst[i] = 1; while (k < j) k += n;
		for (l = j; l < k; ++l) ati[l%n] = pro[i];
	}
}
/*Hi-light-expression-1:  (A?B:C)
                                        1.If the input is float, change the B and C to the 2*eps if its value equals 1 below.
                                        2.For the same farthest, A is diffierent according to the requirement to choose the index:
                                              1).the smallest index: A <- l==n-1
                                              2).the largest index : A <- l!=n-1
                                              3).arbitrary: A <- l==n-1, l!=n-1, 1, 0 are all ok.

 *Trick Note:
1.after erase of mid-point, the terminal of new seg's previous seg may before the endpoint pro[i], even on the left side of judge line.In this condition, the erase operation should be stopped. And when calculate terminal, the points before pro[i] should be ignored. The farthest points to them will not be determined by both ends of the current seg. using submul to judge such kind of point.

2.originally I set the l=n to l=n-1 after the erase operation. But the state l become n-1 since the st point has erased.There's a trick data in requirement of 2) if this small error is ignored.

 *Optimize Note: The operation of calculate the terminal can be optimized using dichonomy(from ed->i*k[k is least positive integer to make ed<i*k]).
*/
```

```
//this section is about the typical sweepline algorithm


//Rotate Caliper
//Be sure that the input must be a convex, and the order is counter-clockwise.
//Return value is the number of the state, the calculation means that between the angle evt[i-1] to evt[i](0 to evt[0]), the anti-polar is fst[i] and snd[i].

double next[N];

int rotate_caliper(int n, pnt* p, double *evt, int *fst, int *snd)
{
	int i, lst, mst, ret;
	double ang, ang1, ang2, dlt;
	p[n] = p[0]; p[n+1] = p[1];
	for (lst = mst = 0, i = 1; i < n; ++i) {
		if (lessver(p[i], p[lst])) lst = i;
		if (lessver(p[mst], p[i])) mst = i;
	} for (i = 0; i < n; ++i) {
		ang1 = atan2(p[i+1].y-p[i].y, p[i+1].x-p[i].x);
		ang2 = atan2(p[i+2].y-p[i+1].y, p[i+2].x-p[i+1].x);
		while (ang2-ang1<-eps) ang2 += 2*pi; next[i] = ang2 - ang1;
	} ang1 = atan2(p[lst+1].y-p[lst].y, p[lst+1].x-p[lst].x);
	ang2 = atan2(p[mst].y-p[mst+1].y, p[mst].x-p[mst+1].x);
	for (ret = 0, ang = 0; ang - pi < -eps; ) {
		dlt = getmin(ang1, ang2); ang += dlt;
		ret++; evt[ret] = ang; fst[ret] = lst; snd[ret] = mst;
		ang1 -= dlt; if (fabs(ang1)<eps) { ang1 = next[lst]; lst = (lst+1) % n; }
		ang2 -= dlt; if (fabs(ang2)<eps) { ang2 = next[mst]; mst = (mst+1) % n; }
	} return ret;
}

void get_anti_polar(int n, double *evt, int *fst, int *snd, double ang, int &ret1, int &ret2) {
	int lst = 1, mst = n, mid;
	while (mst > lst) { mid = (mst+lst)/2; if (ang-evt[mid]>eps) lst = mid+1; else mst = mid; }
	ret1 = fst[lst]; ret2 = snd[mst];
}


//Smallest Enclosing Rectangle
//get the rectangle with the least area or least prameter. return the least area and the least parameter.
//Be sure that the input must be a convex, and the orider is counter-clockwise.

void enclsng_rec(int n, pnt *p, double &area, double &pra)
{
	int i, j, k, l; vec v; double wi, hi;
	area = pra = inf; p[n] = p[0];
	for (v = uvec(getvec(p[0], p[1])), l = 0; nummul(v, getvec(p[0], p[(l+n-1)%n]))-nummul(v, getvec(p[0], p[l])) < eps; l = (l+n-1)%n);
	for (i = j = k = 0; i < n; ++i) {
		v = uvec(getvec(p[i], p[i+1]));
		while (submul(v, getvec(p[i], p[j+1]))-submul(v, getvec(p[i], p[j])) > -eps) j = (j+1) % n;
		while (nummul(v, getvec(p[i], p[k+1]))-nummul(v, getvec(p[i], p[k])) > -eps) k = (k+1) % n;
		while (nummul(v, getvec(p[i], p[l+1]))-nummul(v, getvec(p[i], p[l])) <  eps) l = (l+1) % n;
		wi = nummul(v, getvec(p[l], p[k])); hi = submul(v, getvec(p[i], p[j]));
		area = getmin(area, wi*hi); pra = getmin(pra, (wi+hi)*2);
	}
}
```

```
//3D section. including the basical operation as well as some important method which take the important role in 3D geometry solving.


#define square(x) (x)*(x)
#define xmul(x1, y1, x2, y2) ((x1)*(y2)-(x2)*(y1))
#define dmul(x1, y1, x2, y2) ((x1)*(x2)+(y1)*(y2))
const double pi = acos(-1.0), eps = 1e-8;

struct pnt {
	double x, y, z;
	pnt operator+(const pnt &p) const {
		pnt ret; ret.x = x + p.x; ret.y = y + p.y;
		ret.z = z + p.z; return ret;
	} pnt operator-(const pnt &p) const {
		pnt ret; ret.x = x - p.x; ret.y = y - p.y;
		ret.z = z - p.z; return ret;
	} pnt operator*(const double c) const {
		pnt ret; ret.x = x * c; ret.y = y * c;
		ret.z = z * c; return ret;
	} pnt operator/(const double c) const {
		pnt ret; ret.x = x / c; ret.y = y / c;
		ret.z = z / c; return ret;
	}
};
typedef pnt vec;
struct plane { vec ori; double val; };

double nummul(const vec &v1, const vec &v2) { return v1.x*v2.x + v1.y*v2.y + v1.z*v2.z; }
vec submul(const vec &v1, const vec &v2) {
	vec ret; ret.x = xmul(v1.y, v1.z, v2.y, v2.z); ret.y = xmul(v1.z, v1.x, v2.z, v2.x);
	ret.z = xmul(v1.x, v1.y, v2.x, v2.y); return ret;
}

vec uvec(vec v) {
	double len = sqrt(square(v.x)+square(v.y)+square(v.z));
	vec ret; ret.x = v.x / len; ret.y = v.y / len; ret.z = v.z / len; return ret;
}

plane getplane(const pnt &p1, const pnt &p2, const pnt &p3) {
	vec v1 = p2 - p1, v2 = p3 - p1;
	plane ret; ret.ori = uvec(submul(v1, v2));
	ret.val = nummul(ret.ori, p1); return ret;
}
plane uplane(const plane &s) {
	double len = sqrt(square(s.ori.x)+square(s.ori.y)+square(s.ori.z));
	plane ret; ret.ori = s.ori / len; ret.val = s.val / len; return ret;
}

//give the rotate angle(conter-clockwise) and the normal vector for rotate, output the position of given point after the rotate.
#define OFF(i, j) ((i) == (j) ? 1 : 0)

pnt axis_rotate(const pnt &p, const pnt &ori, double ang) {
	vec v = uvec(ori);
	double pv[3] = { p.x, p.y, p.z}, val[3] = { v.x, v.y, v.z }, off[3] = { 0, -1, 1 };
	double sa = sin(ang), ca = cos(ang), rv[3];
	for (int i = 0; i < 3; ++i) for (int j = rv[i] = 0; j < 3; ++j)
		rv[i] += pv[j] * ((1 - ca) * val[i] * val[j] + sa * val[(6-i-j)%3] * off[(3-i+j)%3] + ca * OFF(i, j));
	pnt ret = { rv[0], rv[1], rv[2] }; return ret;
}
/*  Note: the principle shows below -->
		| vx*vx, vx*vy, vx*vz |                           |  0, -vz,  vy |
	A1 =  | vy*vx, vy*vy, vy*vz |                   A2 =   | vz,   0, -vx |
		| vz*vz, vz*vy, vz*vz |                           |-vy,  vx,   0 |
	p' = ((1-cosa)*A1+sina*A2+cosa*I) * p;
	It's a simplified expression for axis_rotate2, but it doesn't always run faster than the latter.
*/
```

```
//the same as the axis_rotate. Sometimes it runs better but the code is longer.
pnt axis_rotate2(const pnt &p, const pnt &ori, double ang) {
	double ca[5], sa[5], tv1, tv2, xyv; vec v = uvec(ori); int i, j, k;
	xyv = sqrt(square(v.x)+square(v.y));
	if (xyv > eps) { ca[0] = v.x / xyv; sa[0] = -v.y / xyv; }
	else { ca[0] = 1.0; sa[0] = 0.0; }
	ca[1] = v.z; sa[1] = -xyv; ca[2] = cos(ang); sa[2] = sin(ang);
	ca[3] = ca[1]; sa[3] = -sa[1]; ca[4] = ca[0]; sa[4] = -sa[0];
	double rv[3] = { p.x, p.y, p.z };
	for (i = 0; i < 5; ++i) {
		j = (3-i%2) % 3; k = (j+1)%3;
		tv1 = xmul(rv[j], rv[k], sa[i], ca[i]); tv2 = dmul(rv[j], rv[k], sa[i], ca[i]);
		rv[j] = tv1; rv[k] = tv2;
	} pnt ret = { rv[0], rv[1], rv[2] }; return ret;
}

//Transformation with matrix
//The advantage of it is to simplify the multi transformation operation. A matrix can contain the combination of the operation. If there're n points and m operation, using matrix only takes O(n+m) time.
//Caution1: right-hand coordinate system and right-hand rule in rotation only.
//Caution2: the whole transformation matrix should initially set to I.

void matmul(double a[][4], double b[][4]) {
	int i, j, k; double c[4][4];
	for (i = 0; i < 4; ++i) for (j = 0; j < 4; ++j) { c[i][j] = a[i][j]; a[i][j] = 0; }
	for (i = 0; i < 4; ++i) for (j = 0; j < 4; ++j) for (k = 0; k < 4; ++k) a[i][j] += c[i][k] * b[k][j];
}

pnt pnttran(const pnt &p, double a[][4]) {
	double val[4] = { p.x, p.y, p.z, 1 }, rv[4] = { 0, 0, 0, 0 };
	for (int i = 0; i < 4; ++i) for (int j = 0; j < 4; ++j) rv[i] += val[j] * a[j][i];
	pnt ret = { rv[0], rv[1], rv[2] }; return ret;
}

void mattranslate(double a[][4], double dx, double dy, double dz) {
	int i; double b[4][4];
	memset(b, 0, sizeof(b)); for (i = 0; i < 4; ++i) b[i][i] = 1;
	b[3][0] = dx; b[3][1] = dy; b[3][2] = dz; matmul(a, b);
}

void matrotate(double a[][4], const vec &axis, double dlt) {
	int i, j; double b[4][4]; vec v = uvec(axis);
	double sind = sin(dlt), cosd = cos(dlt), val[3] = { v.x, v.y, v.z }, off[3] = { 0, 1, -1 };
	memset(b, 0, sizeof(b)); for (i = 0; i < 4; ++i) b[i][i] = 1;
	for (i = 0; i < 3; ++i) for (j = 0; j < 3; ++j)
		b[i][j] = (1-cosd) * val[i] * val[j] + sind * off[(3-i+j)%3] * val[(6-i-j)%3] + cosd * OFF(i, j);
	matmul(a, b);
}

void matscale(double a[][4], double cx, double cy, double cz) {
	int i; double b[4][4];
	memset(b, 0, sizeof(b)); for (i = 0; i < 4; ++i) b[i][i] = 1;
	b[0][0] = cx; b[1][1] = cy; b[2][2] = cz; matmul(a, b);
}

/*
                         |  1,  0,  0,  0 |                     | cx,  0,  0,  0 |
TRANSLATE(dx,dy,dz) =    |  0,  1,  0,  0 |  SCALE(cx,cy,cz) =  |  0, cy,  0,  0 |  ROTATE(vx,vy,vz,delta) is decribed above.
                         |  0,  0,  1,  0 |                     |  0,  0, cz,  0 |
                         | dx, dy, dz,  1 |                     |  0,  0,  0,  1 |
*/
```

```
//2-Sat
//The meaning of the edge(u->v) means u must be chosen before v is chosen.
struct node { int des, next; };
struct edge { int st, ed; };
#define N 10010
#define M 1000010
node way[M];
edge e[M];
int rd[N], seq[N], col[N], pre[N], cpn[N];
bool vst[N], use[M], chs[N];
int n, cnt, cc, rc;

void dfs(int x)
{
	int i;
	vst[x] = true;
	for (i = rd[x]; i; i = way[i].next) if (!vst[way[i].des]) dfs(way[i].des);
	seq[++cnt] = x;
}

void dfs2(int x, int c)
{
	int i;
	vst[x] = true; col[x] = c;
	for (i = rd[cpn[x]]; i; i = way[i].next) if (!vst[cpn[way[i].des]]) dfs2(cpn[way[i].des], c);
}

void release(int x)
{
	int i;
	vst[x] = true; vst[cpn[x]] = true; chs[x] = true; chs[cpn[x]] = false; cc += 2;
	for (i = rd[x]; i; i = way[i].next) pre[way[i].des]--;
}

void twosat()
{
	int i;
	//judge section
	memset(vst, false, sizeof(vst)); cnt = 0;
	for (i = 1; i <= n; ++i) if (!vst[i]) dfs(i);
	memset(vst, false, sizeof(vst)); cnt = 0;
	for (i = n; i > 0; --i) if (!vst[seq[i]]) dfs2(seq[i], ++cnt);
	for (i = 1; i <= n; ++i) if (col[i] == col[cpn[i]]) { printf("NO\n"); return; } printf("YES\n");
	//output section
	for (i = 1; i <= n; ++i) seq[col[i]] = col[cpn[i]]; for (i = 1; i <= cnt; ++i) cpn[i] = seq[i];
	memset(use, true, sizeof(use));
	for (i = 1; i <= rc; ++i) if (col[e[i].st] == col[e[i].ed]) use[i] = false;
	else { e[i].st = col[e[i].st]; e[i].ed = col[e[i].ed]; }
	memset(rd, 0, sizeof(rd)); memset(pre, 0, sizeof(pre));
	for (i = 1; i <= rc; ++i) if (use[i]) { way[i].des = e[i].ed; way[i].next = rd[e[i].st]; rd[e[i].st] = i; pre[e[i].ed]++; }
	memset(vst, false, sizeof(vst)); cc = 0;
	while (cc < cnt) for (i = 1; i <= cnt; ++i) if (!vst[i] && !pre[i]) release(i);
	for (i = 1; i <= n; ++i) if (chs[col[i]]) printf("%d ", i); printf("\n");
}
```