

二维计算几何集合

2D Computational Geometry

注意事项

1. 输出陷阱：小心是否会输出 -0.00 ，此时应该加上 eps 后再输出；
2. 函数越界：使用 $\text{asin}(a)$, $\text{acos}(a)$, $\text{sqrt}(a)$ 等函数时，应先校准 a （避免 $\text{asin}(1.000001)$ 情形）；
3. 所有比较用 cmp 和 sgn 函数！
4. 如果数据较小，可以调低 eps ；
5. 重载的叉乘运算符 \wedge 优先级较低，注意加括号。

模板

```
1  #include<algorithm>
2  #include<cstring>
3  #include<vector>
4  #include<cstdio>
5  #include<cmath>
6
7  using namespace std;
8
9  const double eps = 1e-8;
10 const double PI = 4 * atan2(1, 1);
11 const double INF = 1e16;
12 const int N = 100005;
13 inline int sgn(double x){
14     if(fabs(x) < eps) return 0;
15     else if(x > 0) return 1;
16     else return -1;
17 }
18 inline int cmp(double x, double y){ return sgn(x-y); }
19 double rand01(){ return rand() / (double)RAND_MAX; }
20 double randeps(){ return (rand01() - 0.5) * eps; }
21
22 //----- Vector & Point -----//
23 struct Vector{
24     double x, y;
25     Vector() {}
26     Vector(double x, double y):x(x), y(y){}
27     void read(){ scanf("%lf%lf", &x, &y); }
28 };
29 typedef Vector Point;
30 Vector operator + (Vector A, Vector B){ return Vector(A.x + B.x, A.y + B.y); }
31 Vector operator - (Vector A, Vector B){ return Vector(A.x - B.x, A.y - B.y); }
32 Vector operator * (double k, Vector A){ return Vector(k * A.x, k * A.y); }
33 Vector operator * (Vector A, double k){ return k * A; }
34 Vector operator / (Vector A, double k){ return Vector(A.x / k, A.y / k); }
35 bool operator < (const Vector &A, const Vector &B){
36     return cmp(A.x, B.x) == 0 ? cmp(A.y, B.y) < 0 : cmp(A.x, B.x) < 0;
37 }
38 bool operator > (const Vector &A, const Vector &B){ return B < A; }
39 bool operator == (const Vector &A, const Vector &B){ return (cmp(A.x, B.x) == 0) && (cmp(A.y, B.y) == 0); }
40 bool operator != (const Vector &A, const Vector &B){ return !(A == B); }
41 // dot product
42 double operator * (Vector A, Vector B){ return A.x * B.x + A.y * B.y; }
43 // cross product
44 double operator ^ (Vector A, Vector B){ return A.x * B.y - A.y * B.x; }
45 double Length(Vector A){ return sqrt(A * A); }
46 // polar angle of vector A, in  $(-\text{PI}, \text{PI}]$ 
47 double Angle(Vector A){ return atan2(A.y, A.x); }
48 // angle between two vectors, in  $[0, \text{PI}]$ 
49 double Angle(Vector A, Vector B){ return atan2(fabs(A ^ B), A * B); }
50 // angle between two vectors, in  $(-\text{PI}, \text{PI}]$ 
51 double signedAngle(Vector A, Vector B){
52     double ang = Angle(A, B); if(sgn(A ^ B) < 0) ang *= -1; return ang;
53 }
54 // check which half plane is vector A in (up / down)
```

```

55 bool quad(Vector A){ return sgn(A.y) == 1 || (sgn(A.y) == 0 && sgn(A.x) <= 0); }
56 // cmpAngle() for sort/lower_bound by polar angle
57 bool cmpAngle(const Vector &A, const Vector &B){
58     if(quad(A) != quad(B)) return quad(A) < quad(B);
59     return sgn(A ^ B) > 0 || (sgn(A ^ B) == 0 && Length(A) < Length(B));
60 }
61 // the signed area of the parallelogram formed by vector(AB) and vector(AC)
62 double ParallelogramArea(Point A, Point B, Point C){ return (B - A) ^ (C - A); }
63 // the signed area of the parallelogram formed by vector v1 and v2
64 double ParallelogramArea(Vector v1, Vector v2){ return v1 ^ v2; }
65 // the signed area of the triangle ABC
66 double TriangleArea(Point A, Point B, Point C){ return ((B - A) ^ (C - A)) / 2; }
67 // rotate rad counterclockwise
68 Vector Rotate(Vector A, double rad){
69     double co = cos(rad), si = sin(rad);
70     return Vector(A.x * co - A.y * si, A.x * si + A.y * co);
71 }
72 // get the normal vector of A
73 Vector Normal(Vector A){ double L = Length(A); return Vector(-A.y/L, A.x/L); }
74 // get the symmetry vector of A about B
75 Vector Symmetry(Vector A, Vector B){ return 2 * B * (A * B / (B * B)) - A; }
76 // test if vector(bc) is to the left of (ab)
77 bool ToTheLeft(Point A, Point B, Point C){ return sgn((B - A) ^ (C - B)) > 0; }
78 // test if vector B is to the left of vector A
79 bool ToTheLeft(Vector A, Vector B){ return sgn(A ^ B) > 0; }
80 double DistancePointToPoint(Point A, Point B){ return Length(A-B); }
81 //-----//
82
83 //----- Line -----//
84 struct Line{
85     Point p;
86     Vector v;
87     double ang; // angle of inclination (-PI, PI]
88     Line() {}
89     Line(Point p, Vector v):p(p), v(v){ ang = atan2(v.y, v.x); }
90     Line(double a, double b, double c){ // ax + by + c = 0
91         if(sgn(a) == 0) p = Point(0, -c/b), v = Vector(1, 0);
92         else if(sgn(b) == 0) p = Point(-c/a, 0), v = Vector(0, 1);
93         else p = Point(0, -c/b), v = Vector(-b, a);
94     }
95     Point getPoint(double t){ return p + v * t; }
96     bool operator < (const Line &L) const{ return ang < L.ang; }
97 };
98 bool PointOnLine(Point p, Line l){ return sgn(l.v ^ (p-l.p)) == 0; }
99 bool PointOnRight(Point p, Line l){ return sgn(l.v ^ (p-l.p)) < 0; }
100 bool LineParallel(Line l1, Line l2){ return sgn(l1.v ^ l2.v) == 0; }
101 bool LineSame(Line l1, Line l2){ return LineParallel(l1, l2) && sgn((l1.p-l2.p) ^ l1.v) == 0; }
102 Point GetLineIntersection(Line l1, Line l2){
103     Vector u = l1.p - l2.p;
104     double t = (l2.v ^ u) / (l1.v ^ l2.v);
105     return l1.p + l1.v * t;
106 }
107 double DistancePointToLine(Point p, Line l){ return fabs(((p - l.p) ^ l.v) / Length(l.v)); }
108 double DistancePointToLine(Point p, Point A, Point B){ return fabs(((B - A) ^ (p - A)) / Length(B - A)); }
109 double DistancePointToSegment(Point p, Point A, Point B){
110     if(A == B) return DistancePointToPoint(p, A);
111     Vector v1 = p - A, v2 = p - B, v3 = A - B; // v1:vector(Ap), v2:vector(Bp), v3:vector(BA)
112     if(sgn(v1 * v3) > 0) return DistancePointToPoint(p, A);
113     if(sgn(v2 * v3) < 0) return DistancePointToPoint(p, B);
114     return DistancePointToLine(p, A, B);
115 }
116 Point PointLineProjection(Point p, Line l){ return l.p + l.v * ((l.v * (p - l.p)) / (l.v * l.v)); }
117 bool PointOnSegment(Point p, Point A, Point B){
118     return sgn((p - A) * (p - B)) <= 0 && sgn((p - A) ^ (p - B)) == 0;
119 }
120 bool PointOnSegmentEndExcluded(Point p, Point A, Point B){
121     return sgn((p - A) * (p - B)) < 0 && sgn((p - A) ^ (p - B)) == 0;
122 }
123 bool SegmentIntersectedEndExcluded(Point A1, Point A2, Point B1, Point B2){
124     return (sgn((A1 - B1) ^ (B1 - B2)) * sgn((A2 - B1) ^ (B1 - B2)) < 0)
125         && (sgn((B1 - A1) ^ (A1 - A2)) * sgn((B2 - A1) ^ (A1 - A2)) < 0);
126 }
127 bool SegmentIntersected(Point A1, Point A2, Point B1, Point B2){
128     if(SegmentIntersectedEndExcluded(A1, A2, B1, B2)) return true;
129     return PointOnSegment(A1, B1, B2) || PointOnSegment(A2, B1, B2) ||
130         PointOnSegment(B1, A1, A2) || PointOnSegment(B2, A1, A2) ? true : false;
131 }
132 bool LineSegmentIntersected(Line L, Point A, Point B){

```

```

133     Point p_1 = L.p, p_2 = L.getPoint(1);
134     return sgn(((p_2 - p_1) ^ (A - p_1))) * sgn(((p_2 - p_1) ^ (B - p_1))) <= 0;
135 }
136 bool LineSegmentIntersectedEndExcluded(Line L, Point A, Point B){
137     Point p_1 = L.p, p_2 = L.getPoint(1);
138     return sgn(((p_2 - p_1) ^ (A - p_1))) * sgn(((p_2 - p_1) ^ (B - p_1))) < 0;
139 }
140 //-----//
141
142 //----- Polygon -----//
143 typedef vector<Point> Polygon;
144 double PolygonArea(int n, Point p[]){
145     double S = 0;
146     for(int i = 2; i < n; i++){
147         S += ((p[i] - p[1]) ^ (p[i+1] - p[1])) / 2;
148     }
149     return S;
150 }
151 double PolygonArea(Polygon poly){
152     double S = 0;
153     for(int i = 1; i < poly.size() - 1; i++){
154         S += ((poly[i] - poly[0]) ^ (poly[i+1] - poly[0])) / 2;
155     }
156     return S;
157 }
158 int PointInPolygon(Point A, int n, Point p[]){ // 0: outside; 1: inside; -1: on edge
159     int wn = 0; // winding number
160     for(int i = 1; i <= n; i++){
161         int nxt = i + 1 > n ? 1 : i + 1;
162         if(PointOnSegment(A, p[i], p[nxt])) return -1;
163         int k = sgn((p[nxt] - p[i]) ^ (A - p[i]));
164         int d1 = sgn(p[i].y - A.y);
165         int d2 = sgn(p[nxt].y - A.y);
166         if(k > 0 && d1 <= 0 && d2 > 0) wn++;
167         if(k < 0 && d2 <= 0 && d1 > 0) wn--;
168     }
169     if(wn != 0) return 1;
170     return 0;
171 }
172 int PointInPolygon(Point A, Polygon poly){ // 0: outside; 1: inside; -1: on edge
173     int wn = 0; // winding number
174     int n = poly.size();
175     for(int i = 0; i < n; i++){
176         int nxt = (i + 1) % n;
177         if(PointOnSegment(A, poly[i], poly[nxt])) return -1;
178         int k = sgn((poly[nxt] - poly[i]) ^ (A - poly[i]));
179         int d1 = sgn(poly[i].y - A.y);
180         int d2 = sgn(poly[nxt].y - A.y);
181         if(k > 0 && d1 <= 0 && d2 > 0) wn++;
182         if(k < 0 && d2 <= 0 && d1 > 0) wn--;
183     }
184     if(wn != 0) return 1;
185     return 0;
186 }
187 Point getPolygonCenter(int n, Point p[]){
188     Point res(0, 0);
189     double S = 0;
190     for(int i = 2; i < n; i++){
191         double area = ((p[i] - p[1]) ^ (p[i+1] - p[1]));
192         res = res + (p[1] + p[i] + p[i+1]) * area;
193         S += area;
194     }
195     return res / S / 3;
196 }
197 // the left part of l and poly form a new polygon
198 Polygon cutPolygon(Line l, Polygon poly){
199     Polygon newpoly;
200     int n = poly.size();
201     for(int i = 0; i < n; i++){
202         Point C = poly[i], D = poly[(i+1)%n];
203         if(sgn(l.v ^ (C - l.p)) >= 0) newpoly.push_back(C);
204         if(sgn(l.v ^ (C - D)) != 0){
205             Point q = GetLineIntersection(l, Line(C, C - D));
206             if(PointOnSegmentEndExcluded(q, C, D)) newpoly.push_back(q);
207         }
208     }
209     return newpoly;
210 }
211 //-----//

```

```

211 //----- Circle -----//
212 struct Circle{
213     Point p;
214     double r;
215     Circle() {}
216     Circle(Point p, double r):p(p), r(r) {}
217     Point getPoint(double alpha){
218         return Point(p.x + cos(alpha) * r, p.y + sin(alpha) * r);
219     }
220 };
221 void getLineCircleIntersection(Line L, Circle C, Point res[], int &resn){
222     // resn is the number of intersection points
223     // intersection points are stored in res[]
224     resn = 0;
225     Point q = PointLineProjection(C.p, L);
226     double d = DistancePointToPoint(C.p, q);
227     if(cmp(d, C.r) > 0) return; // separated
228     else if(cmp(d, C.r) == 0){ res[++resn] = q; return; } // tangent
229     Vector u = L.v / Length(L.v);
230     double dd = sqrt(C.r * C.r - d * d);
231     res[++resn] = q - dd * u, res[++resn] = q + dd * u; // intersected
232 }
233 void getCircleCircleIntersection(Circle C1, Circle C2, Point res[], int &resn){
234     // resn is the number of intersection points (-1 if two circles coincide)
235     // intersection points are stored in res[]
236     resn = 0;
237     double d = DistancePointToPoint(C1.p, C2.p);
238     if(sgn(d) == 0) {
239         if (cmp(C1.r, C2.r) == 0) resn = -1; // two circles are the same
240         return; // or concentric
241     }
242     if(cmp(C1.r + C2.r, d) < 0) return; // separated
243     if(cmp(fabs(C1.r - C2.r), d) > 0) return; // contained
244     double a = Angle(C2.p - C1.p);
245     double da = acos((d * d + C1.r * C1.r - C2.r * C2.r) / (2 * d * C1.r));
246     Point p1 = C1.getPoint(a - da), p2 = C1.getPoint(a + da);
247     res[++resn] = p1;
248     if(p1 != p2) res[++resn] = p2; // tangent or intersected
249 }
250 void getTangents(Point p, Circle C, Line L[], int &lid){
251     // lid is the number of tangent lines
252     // tangent lines are stored in L[]
253     lid = 0;
254     Vector u = C.p - p;
255     double d = Length(u);
256     if(cmp(d, C.r) < 0) return;
257     else if(cmp(d, C.r) == 0) L[++lid] = Line(p, Rotate(u, PI / 2));
258     else if(cmp(d, C.r) > 0){
259         double ang = asin(C.r / d);
260         L[++lid] = Line(p, Rotate(u, -ang));
261         L[++lid] = Line(p, Rotate(u, ang));
262     }
263 }
264 void getTangents(Point p, Circle C, Point P[], int &pid){
265     // pid is the number of tangent points
266     // tangent points are stored in P[]
267     pid = 0;
268     Vector u = p - C.p;
269     double d = Length(u);
270     if(cmp(d, C.r) < 0) return;
271     else if(cmp(d, C.r) == 0) P[++pid] = p;
272     else if(cmp(d, C.r) > 0){
273         double ang = acos(C.r / d);
274         P[++pid] = C.p + Rotate(u, -ang) / d * C.r;
275         P[++pid] = C.p + Rotate(u, ang) / d * C.r;
276     }
277 }
278 void getTangents(Circle C1, Circle C2, Point c1[], Point c2[], int &resn){
279     // resn is the number of tangent lines
280     // c1[] and c2[] are relevant points on C1 and C2
281     resn = 0;
282     if(cmp(C1.r, C2.r) < 0) swap(C1, C2), swap(c1, c2);
283     double d = DistancePointToPoint(C1.p, C2.p);
284     if(sgn(d) == 0 && cmp(C1.r, C2.r) == 0){ resn = -1; return; } // two circles are the same
285     if(cmp(C1.r - C2.r, d) > 0) return; // contained
286     double base = Angle(C2.p - C1.p);
287     if(cmp(C1.r - C2.r, d) == 0){ // internally tangent
288         c1[++resn] = C1.getPoint(base), c2[resn] = C2.getPoint(base);

```

```

289         return;
290     }
291     double ang = acos((C1.r - C2.r) / d);
292     c1[++resn] = C1.getPoint(base - ang), c2[resn] = C2.getPoint(base - ang);
293     c1[++resn] = C1.getPoint(base + ang), c2[resn] = C2.getPoint(base + ang);
294     if(cmp(C1.r + C2.r, d) == 0) // externally tangent
295         c1[++resn] = C1.getPoint(base), c2[resn] = C2.getPoint(base + PI);
296     else if(cmp(C1.r + C2.r, d) < 0) // separated
297         ang = acos((C1.r + C2.r) / d);
298         c1[++resn] = C1.getPoint(base - ang), c2[resn] = C2.getPoint(base - ang + PI);
299         c1[++resn] = C1.getPoint(base + ang), c2[resn] = C2.getPoint(base + ang + PI);
300     }
301 }
302 double TriangleCircleIntersectionArea(Point A, Point B, double r){
303     // Circle's center is O(0, 0), radius is r
304     // The triangle is OAB
305     double ra = sqrt(A*A), rb = sqrt(B*B);
306     Point O(0, 0);
307     Line AB = Line(A, B-A);
308     if(cmp(ra, r) <= 0 && cmp(rb, r) <= 0) return (A ^ B) / 2;
309     else if(cmp(ra, r) >= 0 && cmp(rb, r) >= 0){
310         double d = DistancePointToLine(O, AB);
311         double theta = signedAngle(A, B);
312         if(cmp(d, r) >= 0) return theta * r * r / 2;
313         else{
314             Point H = GetLineIntersection(AB, Line(O, Normal(B-A)));
315             if(PointOnSegment(H, A, B)){
316                 Point t[3]; int _t;
317                 getLineCircleIntersection(AB, Circle(O, r), t, _t);
318                 double phi = signedAngle(t[1], t[2]);
319                 return (theta + sin(phi) - phi) * r * r / 2;
320             }
321             else return theta * r * r / 2;
322         }
323     }
324     else{
325         Point t[3]; int _t;
326         getLineCircleIntersection(AB, Circle(O, r), t, _t);
327         if(PointOnSegment(t[2], A, B)) t[1] = t[2];
328         if(cmp(ra, r) <= 0)
329             return signedAngle(t[1], B) * r * r / 2 + (A ^ t[1]) / 2;
330         else
331             return signedAngle(A, t[1]) * r * r / 2 + (t[1] ^ B) / 2;
332     }
333 }
334 double PolygonCircleIntersectionArea(int n, Point p[], Circle C){
335     // p[] is a simple polygon
336     // ATT: result might be negative
337     double res = 0;
338     for(int i = 1; i <= n; i++){
339         res += TriangleCircleIntersectionArea(p[i]-C.p, p[i%n+1]-C.p, C.r);
340     }
341     return res;
342 }
343 // get inversion from a circle to a circle
344 // ensure that point O is not on circle A beforehand
345 Circle getInversionC2C(Point O, double R, Circle A){
346     double OA = Length(A.p - O);
347     double rB = R * R / 2 * (1 / (OA - A.r) - 1 / (OA + A.r));
348     double xB = O.x + rB / A.r * (A.p.x - O.x);
349     double yB = O.y + rB / A.r * (A.p.y - O.y);
350     return Circle(Point(xB, yB), rB);
351 }
352 // get inversion from a line to a circle
353 // ensure that point O is not on line L beforehand
354 // point O is on the result circle
355 Circle getInversionL2C(Point O, double R, Line L){
356     Point P = PointLineProjection(O, L);
357     double d = Length(P - O);
358     double r = R * R / d / 2;
359     Vector v = (P - O) / Length(P - O) * r;
360     return Circle(O + v, r);
361 }
362 // get inversion from a circle to a line
363 // ensure that point O is on circle A
364 Line getInversionC2L(Point O, double R, Circle A){
365     Point P = (A.p - O) / Length(A.p - O) * R * R / A.r / 2;
366     Vector v = Normal(A.p - O);
367     return Line(P, v);

```

```

367 }
368 //-----//
369 //#####//
370
371 //----- Convex Hull -----//
372 void ConvexHull(int n, Point p[], Point sta[], int &staid){
373     // there're n points stored in p[], the points on convex hull will be saved in sta[]
374     sort(p+1, p+n+1);
375     n = unique(p+1, p+n+1) - (p+1);
376     staid = 0;
377     for(int i = 1; i <= n; i++){
378         // points on edge
379         // while(staid > 1 && sgn((sta[staid]-sta[staid-1]) ^ (p[i]-sta[staid-1])) < 0) staid--;
380         // no points on edge
381         while(staid > 1 && sgn((sta[staid]-sta[staid-1]) ^ (p[i]-sta[staid-1])) <= 0) staid--;
382         sta[++staid] = p[i];
383     }
384     int k = staid;
385     for(int i = n-1; i >= 1; i--){
386         // points on edge
387         // while(staid > k && sgn((sta[staid]-sta[staid-1]) ^ (p[i]-sta[staid-1])) < 0) staid--;
388         // no points on edge
389         while(staid > k && sgn((sta[staid]-sta[staid-1]) ^ (p[i]-sta[staid-1])) <= 0) staid--;
390         sta[++staid] = p[i];
391     }
392     if(n > 1) staid--;
393 }
394
395 // check if point A is in ConvexHull p[]
396 // note that p[1] must be the original point
397 bool PointInConvexHull(Point A, int n, Point p[]){
398     if(sgn(A ^ p[2]) > 0 || sgn(A ^ p[n]) < 0) return false;
399     int pos = lower_bound(p + 1, p + n + 1, A, cmpAngle) - p - 1;
400     return sgn((A - p[pos]) ^ (p[pos%n+1] - p[pos])) <= 0;
401 }
402
403 void Minkowski(int n1, Point p1[], int n2, Point p2[], Point tmp[], Point res[], int &resn){
404     // tmp[] is an auxiliary array
405     // p1[] is a convex hull consist of n1 points
406     // p2[] is a convex hull consist of n2 points
407     // res[] is the Minkowski tmp of these two convex hull consist of resn points
408     p1[n1+1] = p1[1], p2[n2+1] = p2[1];
409     vector<Vector> v1, v2;
410     for(int i = 1; i <= n1; i++) v1.emplace_back(p1[i+1] - p1[i]);
411     for(int i = 1; i <= n2; i++) v2.emplace_back(p2[i+1] - p2[i]);
412     int pt1 = 0, pt2 = 0, tid = 1;
413     tmp[1] = p1[1] + p2[1];
414     while(pt1 < n1 && pt2 < n2){
415         tid++;
416         if(sgn(v1[pt1] ^ v2[pt2]) >= 0) tmp[tid] = tmp[tid-1] + v1[pt1++];
417         else tmp[tid] = tmp[tid-1] + v2[pt2++];
418     }
419     while(pt1 < n1) tid++, tmp[tid] = tmp[tid-1] + v1[pt1++];
420     while(pt2 < n2) tid++, tmp[tid] = tmp[tid-1] + v2[pt2++];
421     ConvexHull(tid, tmp, res, resn);
422 }
423
424 //-----//
425
426 //----- Rotating Calipers -----//
427 void RotatingCalipers(int m, Point p[]){
428     // p[] = sta[], m = staid in ConvexHull()
429     if(m == 2){
430         // do something
431         return;
432     }
433     p[m+1] = p[1];
434     int ver = 2;
435     for(int i = 1; i <= m; i++){ // enumerate edge: p[i] ~ p[i+1]
436         while(TriangleArea(p[i], p[i+1], p[ver]) < TriangleArea(p[i], p[i+1], p[ver+1])){
437             ver++;
438             if(ver == m+1) ver = 1; // find the corresponding point: ver
439             // do something
440         }
441     }
442 }
443 // calculate the diameter of a convex hull
444 double DiameterConvexHull(int m, Point p[]){

```

```

445 // p[] = sta[], m = staid in ConvexHull()
446 double ans = 0;
447 if(m == 2){
448     ans = (p[1] - p[2]) * (p[1] - p[2]);
449     return sqrt(ans);
450 }
451 p[m+1] = p[1];
452 int ver = 2;
453 for(int i = 1; i <= m; i++){
454     while(TriangleArea(p[i], p[i+1], p[ver]) < TriangleArea(p[i], p[i+1], p[ver+1])){
455         ver++;
456         if(ver == m+1) ver = 1;
457         ans = max(ans, max((p[ver] - p[i]) * (p[ver] - p[i]), (p[ver] - p[i+1]) * (p[ver] - p[i+1]))));
458     }
459 }
460 return sqrt(ans);
461 }
462
463 // solve min-area-rectangle cover OR min-perimeter-rectangle cover problem
464 struct MinRectangleCover{
465
466     double minArea, minPeri;
467     Point minAreaPoints[10], minPeriPoints[10];
468
469     void cal(int i, int nxti, int ver, int j, int k, Point p[]){
470         Point t[4];
471         Vector v = p[nxti] - p[i], u = Normal(v);
472         t[0] = GetLineIntersection(Line(p[i], v), Line(p[j], u));
473         t[1] = GetLineIntersection(Line(p[j], u), Line(p[ver], v));
474         t[2] = GetLineIntersection(Line(p[ver], v), Line(p[k], u));
475         t[3] = GetLineIntersection(Line(p[k], u), Line(p[i], v));
476         double area = fabs((t[1] - t[0]) ^ (t[0] - t[3]));
477         if(cmp(area, minArea) < 0){
478             minArea = area;
479             minAreaPoints[0] = t[0], minAreaPoints[1] = t[1];
480             minAreaPoints[2] = t[2], minAreaPoints[3] = t[3];
481         }
482         double peri = Length(t[1]-t[0]) + Length(t[0]-t[3]); peri *= 2;
483         if(cmp(peri, minPeri) < 0){
484             minPeri = peri;
485             minPeriPoints[0] = t[0], minPeriPoints[1] = t[1];
486             minPeriPoints[2] = t[2], minPeriPoints[3] = t[3];
487         }
488     }
489     inline void Norm(int &x, int m){ ((x %= m) += m) %= m; if(x == 0) x = m; }
490     inline double func(int mid, int i, int nxti, Point p[], int m, int kind){
491         Norm(mid, m);
492         if(kind == 1)
493             return (p[nxti]-p[i]) * (p[mid]-p[i]) / Length(p[nxti]-p[i]);
494         else
495             return (p[i]-p[nxti]) * (p[mid]-p[nxti]) / Length(p[i]-p[nxti]);
496     }
497     int tripartition(int l, int r, int i, int nxti, Point p[], int m, int kind){
498         while(r < l) r += m;
499         int mid1 = l, mid2 = r;
500         while(mid1 < mid2){
501             mid1 = l + (r - l) / 3;
502             mid2 = r - (r - l) / 3;
503             // func(x) is a unimodal function
504             if(func(mid1, i, nxti, p, m, kind) < func(mid2, i, nxti, p, m, kind))
505                 l = mid1 + 1;
506             else
507                 r = mid2 - 1;
508         }
509         return l;
510     }
511     // minimum rectangle covering the points p[]
512     void solve(int m, Point p[]){
513         minArea = minPeri = INF;
514         int ver = 2;
515         for(int i = 1; i <= m; i++){
516             int nxti = i + 1; Norm(nxti, m);
517             while(TriangleArea(p[i], p[nxti], p[ver]) < TriangleArea(p[i], p[nxti], p[ver+1]))
518                 ver++, Norm(ver, m);
519             int l = nxti, r = ver;
520             int j = tripartition(l, r, i, nxti, p, m, 1);
521             l = ver, r = i;
522             int k = tripartition(l, r, i, nxti, p, m, 2);
523             Norm(k, m), Norm(j, m);

```

```

523         cal(i, nexti, ver, j, k, p);
524     }
525 }
526
527 };
528
529 //-----//
530
531 //----- HalfplaneIntersection -----//
532 struct Halfplane{
533     Point P[N]; // P[i] is the intersection of line Q[i] and Q[i+1]
534     Line Q[N]; // deque
535     void HalfplaneIntersection(Line L[], int n, Point res[], int &m){
536         // L[] are the lines, n is the number of lines, res[] stores the result of the intersection (a polygon)
537         // m is the number of points of the intersection (which is a polygon)
538         sort(L + 1, L + n + 1);
539         int head, tail;
540         Q[head = tail = 0] = L[1];
541         for(int i = 2; i <= n; i++){
542             while(head < tail && PointOnRight(P[tail - 1], L[i])) tail--;
543             while(head < tail && PointOnRight(P[head], L[i])) head++;
544             Q[++tail] = L[i];
545             if(sgn(Q[tail].v ^ Q[tail - 1].v) == 0){ // parallel, the inner one remains
546                 tail--;
547                 if(!PointOnRight(L[i].p, Q[tail])) Q[tail] = L[i];
548             }
549             if(head < tail) P[tail - 1] = GetLineIntersection(Q[tail-1], Q[tail]);
550         }
551         while(head < tail && PointOnRight(P[tail - 1], Q[head])) tail--; // delete useless plane
552         P[tail] = GetLineIntersection(Q[tail], Q[head]);
553
554         m = 0;
555         for(int i = head; i <= tail; i++) res[++m] = P[i];
556     }
557 };
558 //-----//
559
560 int main(){
561     ;
562 }

```