

```

1  #include <bits/stdc++.h>
2  #define FOR(i,a,b) for(int i=(a),_b=(b); i<=_b; ++i)
3  #define FORD(i,a,b) for(int i=(a),_b=(b); i>=_b; --i)
4  #define REP(i,a) for(int i=0,_a=(a); i < _a; ++i)
5
6  #define DEBUG(X) { cout << #X << " = " << X << endl; }
7  #define PR(A,n) { cout << #A << " = "; FOR(_,1,n) cout << A[_] << ' '; cout << endl; }
8  #define PR0(A,n) { cout << #A << " = "; REP(_,n) cout << A[_] << ' '; cout << endl; }
9
10 #define ll long long
11 #define SZ(x) ((int) (x).size())
12 #define sqr(x) ((x) * (x))
13 #define double long double
14 using namespace std;
15
16 const double PI = acos((double) -1.0);
17 const double EPS = 1e-10;
18
19 struct Point {
20     double x, y;
21     Point() {}
22     Point(double x, double y) : x(x), y(y) {}
23     Point operator - (const Point& a) const {
24         return Point(x-a.x, y-a.y);
25     }
26     double len() {
27         return hypot(x, y);
28     }
29 };
30
31 struct Circle : Point {
32     double r;
33 };
34
35 double cir_area_solve(double a, double b, double c) {
36     return acos((a*a + b*b - c*c) / 2 / a / b);
37 }
38 double cir_area_cut(double a, double r) {
39     double s1 = a * r * r / 2;
40     double s2 = sin(a) * r * r / 2;
41     return s1 - s2;
42 }
43
44 double commonCircleArea(Circle c1, Circle c2) { //return the common area of two circle
45     if (c1.r < c2.r) swap(c1, c2);
46     double d = (c1 - c2).len();
47     if (d + c2.r <= c1.r + EPS) return c2.r*c2.r*acos(-1.0);
48     if (d >= c1.r + c2.r - EPS) return 0.0;
49     double a1 = cir_area_solve(d, c1.r, c2.r);
50     double a2 = cir_area_solve(d, c2.r, c1.r);
51     return cir_area_cut(a1*2, c1.r) + cir_area_cut(a2*2, c2.r);
52 }
53
54 int main() {
55     cout << (fixed) << setprecision(12);
56     ll x1, y1, r1, x2, y2, r2;

```

```
57     while (cin >> x1 >> y1 >> r1 >> x2 >> y2 >> r2) {
58         Circle c1; c1.x = x1; c1.y = y1; c1.r = r1;
59         Circle c2; c2.x = x2; c2.y = y2; c2.r = r2;
60         cout << commonCircleArea(c1, c2) << endl;
61     }
62 }
```

```

1  #include <cstdio>
2  #include <cstdlib>
3  #include <cstring>
4  #include <cassert>
5  #include <ctime>
6  #include <cmath>
7  #include <algorithm>
8  #include <string>
9  #include <vector>
10 #include <deque>
11 #include <queue>
12 #include <list>
13 #include <set>
14 #include <map>
15
16 #define pb push_back
17 #define mp make_pair
18 #define TASKNAME ""
19
20 #ifdef LOCAL
21 #define eprintf(...) fprintf(stderr,__VA_ARGS__)
22 #else
23 #define eprintf(...)
24 #endif
25
26 #define TIMESTAMP(x) eprintf("[ " #x " ] Time = %.3lfs\n",clock()*1.0/
CLOCKS_PER_SEC)
27
28 #ifdef linux
29 #define LLD "%lld"
30 #else
31 #define LLD "%I64d"
32 #endif
33
34 #define sz(x) ((int)(x).size())
35
36 using namespace std;
37
38 typedef double ld;
39 typedef long long ll;
40 typedef vector<ll> vll;
41 typedef vector<int> vi;
42 typedef vector<vi> vvi;
43 typedef vector<bool> vb;
44 typedef vector<vb> vvb;
45 typedef pair<int, int> pii;
46
47 const int inf = 1e9;
48 const double eps = 1e-9;
49 const double INF = inf;
50 const double EPS = eps;
51
52 struct point{
53     ld x,y,z;
54     point(ld x,ld y, ld z):x(x),y(y),z(z){}
55     point(){ x = y = z = 0;}
56     bool operator==(const point& a) const{
57         return fabs(x - a.x) < eps && fabs(y - a.y) < eps && fabs(z - a.z) < eps;
58     }

```

```

59     bool load(){
60         double _x,_y,_z;
61         if (scanf("%lf%lf%lf",&_x,&_y,&_z) != 3) return 0;
62         x = _x, y = _y, z = _z;
63         return 1;
64     }
65     void print(){
66         printf("%lf %lf %lf\n",(double)x, (double)y, (double)z);
67     }
68     void eprint(){
69         eprintf("%lf %lf %lf\n",(double)x, (double)y, (double)z);
70     }
71     ld dist2() const {
72         return x*x+y*y+z*z;
73     }
74     ld dist() const{
75         return sqrt(dist2());
76     }
77 };
78
79 inline point operator+(const point& a,const point& b){
80     return point(a.x+b.x,a.y+b.y,a.z+b.z);
81 }
82
83 inline point operator-(const point& a,const point& b){
84     return point(a.x-b.x,a.y-b.y,a.z-b.z);
85 }
86
87 inline point operator*(const point& a,ld t){
88     return point(a.x*t, a.y*t, a.z*t);
89 }
90
91 inline ld det(ld a,ld b,ld c,ld d){
92     return a*d - b*c;
93 }
94
95 inline ld det(ld a11,ld a12,ld a13,ld a21,ld a22, ld a23, ld a31, ld a32,ld
a33){
96     return a11*det(a22,a23,a32,a33) - a12*det(a21,a23,a31,a33) + a13*det
(a21,a22,a31,a32);
97 }
98
99 int sgn(ld x){
100     return (x > eps) - (x < -eps);
101 }
102
103 inline ld det(const point& a,const point& b,const point& c){
104     return det(a.x,a.y,a.z,b.x,b.y,b.z,c.x,c.y,c.z);
105 }
106
107 point vp(const point& a,const point& b){
108     return point(det(a.y,a.z,b.y,b.z),-det(a.x,a.z,b.x,b.z),det
(a.x,a.y,b.x,b.y));
109 }
110
111 ld sp(const point& a,const point& b){
112     return a.x * b.x + a.y * b.y + a.z * b.z;
113 }
114

```

```

115 // BEGIN ALGO
116
117 // vp is vector product (point)
118 // sp is scalar product (ld)
119
120 struct line{
121     point p,v;
122     line(){}; /*BOXNEXT*/
123     line(const point& p,const point& v):p(p),v(v){
124         assert(!(v == point()));
125     }
126     bool on(const point& pt) const{
127         return vp(pt - p, v) == point();
128     }
129 };
130 struct plane {
131     point n;
132     ld d;
133     plane() : d(0) {}
134     plane(const point &p1, const point &p2,
135           const point &p3) {
136         n = vp(p2 - p1, p3 - p1);
137         d = -sp(n, p1);
138         assert(side(p1) == 0);
139         assert(side(p2) == 0);
140         assert(side(p3) == 0);
141     }
142     int side(const point &p) const {
143         return sgn(sp(n, p) + d);
144     }
145 };
146
147 int intersec(const line& l1, const line& l2,
148              point& res){
149     assert(!(l1.v == point()));
150     assert(!(l2.v == point()));
151     if (vp(l1.v,l2.v) == point()){
152         if (vp(l1.v, l1.p - l2.p) == point())
153             return 2; // same
154         return 0; // parallel
155     }
156     point n = vp(l1.v,l2.v);
157     point p = l2.p - l1.p;
158     if (sgn(sp(n,p)))
159         return 0; // skew
160     ld t;
161     if (sgn(n.x))
162         t = (p.y * l2.v.z - p.z * l2.v.y) / n.x;
163     else if (sgn(n.y))
164         t = (p.z * l2.v.x - p.x * l2.v.z) / n.y;
165     else if (sgn(n.z))
166         t = (p.x * l2.v.y - p.y * l2.v.x) / n.z;
167     else
168         assert(false);
169     res = l1.p + l1.v * t;
170     assert(l1.on(res)); assert(l2.on(res));
171     return 1; // intersects
172 }
173

```

```

174 ld dist(const line& l1,const line& l2){
175     point ret = l1.p - l2.p; /*BOXNEXT*/
176     ret = ret - l1.v * (sp(l1.v,ret) / l1.v.dist2());
177     point tmp = l2.v - l1.v *
178         (sp(l1.v,l2.v) / l1.v.dist2());
179     if (sgn(tmp.dist2())) /*BOXNEXT*/
180         ret = ret - tmp * (sp(tmp,ret) / tmp.dist2());
181     assert(fabs(sp(ret,l1.v)) < eps);
182     assert(fabs(sp(ret,tmp)) < eps);
183     assert(fabs(sp(ret,l2.v)) < eps);
184     return ret.dist();
185 }
186
187 void closest(const line& l1,const line& l2,
188             point& p1,point& p2){
189     if (vp(l1.v,l2.v) == point()){
190         p1 = l1.p;
191         p2 = l2.p - l1.v * /*BOXNEXT*/
192             (sp(l1.v,l2.p - l1.p) / l1.v.dist2());
193         return;
194     }
195     point p = l2.p - l1.p;
196     ld t1 = (
197         sp(l1.v,p) * l2.v.dist2() -
198         sp(l1.v,l2.v) * sp(l2.v,p)
199         ) / vp(l1.v,l2.v).dist2();
200     ld t2 = (
201         sp(l2.v,l1.v) * sp(l1.v,p) -
202         sp(l2.v,p) * l1.v.dist2()
203         ) / vp(l2.v,l1.v).dist2();
204     p1 = l1.p + l1.v * t1;
205     p2 = l2.p + l2.v * t2;
206     assert(l1.on(p1));
207     assert(l2.on(p2));
208 }
209
210 int cross(const line &l, const plane &pl,
211           point &res) {
212     ld d = sp(pl.n, l.v);
213     if (sgn(d) == 0) {
214         return (pl.side(l.p) == 0) ? 2 : 0;
215     }
216     ld t = (-sp(pl.n, l.p) - pl.d) / d;
217     res = l.p + l.v * t;
218     #ifdef DEBUG
219     assert(pl.side(res) == 0);
220     #endif
221     return 1;
222 }
223
224 bool cross(const plane& p1,const plane& p2,
225           const plane& p3, point& res){
226     ld d = det(p1.n,p2.n,p3.n);
227     if (sgn(d) == 0) {
228         return false;
229     }
230     point px(p1.n.x, p2.n.x, p3.n.x);
231     point py(p1.n.y, p2.n.y, p3.n.y);
232     point pz(p1.n.z, p2.n.z, p3.n.z);

```

```

233     point p(-p1.d, -p2.d, -p3.d);
234     res = point(
235         det(p, py, pz)/d,
236         det(px, p, pz)/d,
237         det(px, py, p)/d
238     );
239     #ifdef DEBUG
240     assert(p1.side(res) == 0);
241     assert(p2.side(res) == 0);
242     assert(p3.side(res) == 0);
243     #endif
244     return true;
245 }
246
247 int cross(const plane &p1, const plane &p2,
248     line &res) {
249     res.v = vp(p1.n, p2.n);
250     if (res.v == point()) {
251         if ( (p1.n * (p1.d / p1.n.dist2())) ==
252             (p2.n * (p2.d / p2.n.dist2())) )
253             return 2;
254         else
255             return 0;
256     }
257     plane p3;
258     p3.n = res.v;
259     p3.d = 0;
260     bool ret = cross(p1, p2, p3, res.p);
261     assert(ret);
262     assert(p1.side(res.p) == 0);
263     assert(p2.side(res.p) == 0);
264     return 1;
265 }
266 // END ALGO
267
268
269
270 int main(){
271     freopen(TASKNAME".in", "r", stdin);
272     #ifdef LOCAL
273     freopen(TASKNAME".out", "w", stdout);
274     #endif
275
276     {
277         line l;
278         l.p = point(1, 1, 1);
279         l.v = point(1, 0, -1);
280         plane p(point(10, 11, 12), point(9, 8, 7), point(1, 3, 2));
281         point res;
282         assert(cross(l, p, res) == 1);
283     }
284     {
285         plane p1(point(1, 2, 3), point(4, 5, 6), point(-1, 5, -4));
286         plane p2(point(3, 2, 1), point(6, 5, 4), point(239, 17, -42));
287         line l;
288         assert(cross(p1, p2, l) == 1);
289     }
290     {
291         plane p1(point(1, 2, 3), point(4, 5, 6), point(-1, 5, -4));

```

```
292     plane p2(point(1, 2, 3), point(7, 8, 9), point(3, -1, 10));
293     line l;
294     assert(cross(p1, p2, l) == 2);
295 }
296 {
297     plane p1(point(1, 2, 3), point(4, 5, 6), point(-1, 5, -4));
298     plane p2(point(1, 2, 4), point(4, 5, 7), point(-1, 5, -3));
299     line l;
300     assert(cross(p1, p2, l) == 0);
301 }
302
303 line l1,l2;
304 while (l1.p.load()){
305     l1.v.load(); l1.v = l1.v - l1.p;
306     l2.p.load();
307     l2.v.load(); l2.v = l2.v - l2.p;
308     if (l1.v == point() || l2.v == point()) continue;
309     point res;
310     int cnt = intersec(l1,l2,res);
311     ld d = dist(l1,l2);
312     if (fabs(d) < eps)
313         assert(cnt >= 1);
314     else
315         assert(cnt == 0);
316     point p1,p2;
317     closest(l1,l2,p1,p2);
318     assert(fabs((p1-p2).dist() - d) < eps);
319 }
320 plane a(point(1,0,0),point(0,1,0),point(0,0,1));
321 plane b(point(-1,0,0),point(0,-1,0),point(0,0,-1));
322 line l;
323 assert((cross(a,b,l))==0);
324 TIMESTAMP(end);
325 return 0;
326 }
```



```

1 // Two-phase simplex algorithm for solving linear programs of the form
2 //
3 //      maximize      c^T x
4 //      subject to    Ax <= b
5 //                   x >= 0
6 //
7 // INPUT: A -- an m x n matrix
8 //         b -- an m-dimensional vector
9 //         c -- an n-dimensional vector
10 //         x -- a vector where the optimal solution will be stored
11 //
12 // OUTPUT: value of the optimal solution (infinity if unbounded
13 //         above, nan if infeasible)
14 //
15 // To use this code, create an LPSolver object with A, b, and c as
16 // arguments. Then, call Solve(x).
17
18 #include <iostream>
19 #include <iomanip>
20 #include <vector>
21 #include <cmath>
22 #include <limits>
23
24 using namespace std;
25
26 typedef long double DOUBLE;
27 typedef vector<DOUBLE> VD;
28 typedef vector<VD> VVD;
29 typedef vector<int> VI;
30
31 const DOUBLE EPS = 1e-9;
32
33 struct LPSolver {
34     int m, n;
35     VI B, N;
36     VVD D;
37
38     LPSolver(const VVD &A, const VD &b, const VD &c): m(b.size()), n(c.size(
39     )), N(n + 1), B(m), D(m + 2, VD(n + 2))
40     {
41         for (int i = 0; i < m; i++)
42             for (int j = 0; j < n; j++)
43                 D[i][j] = A[i][j];
44         for (int i = 0; i < m; i++)
45         {
46             B[i] = n + i;
47             D[i][n] = -1;
48             D[i][n + 1] = b[i];
49         }
50         for (int j = 0; j < n; j++)
51         {
52             N[j] = j;
53             D[m][j] = -c[j];
54         }
55         N[n] = -1;
56         D[m + 1][n] = 1;
57     }
58
59     void Pivot(int r, int s)

```

```

59         {
60             for (int i = 0; i < m + 2; i++)
61                 if (i != r)
62                     for (int j = 0; j < n + 2; j++)
63                         if (j != s)
64                             D[i][j] -= D[r][j] * D[i][s] /
D[r][s];
65             for (int j = 0; j < n + 2; j++)
66                 if (j != s)
67                     D[r][j] /= D[r][s];
68             for (int i = 0; i < m + 2; i++)
69                 if (i != r)
70                     D[i][s] /= -D[r][s];
71             D[r][s] = 1.0 / D[r][s];
72             swap(B[r], N[s]);
73         }
74
75     bool Simplex(int phase)
76     {
77         int x = phase == 1 ? m + 1 : m;
78         while (true)
79         {
80             int s = -1;
81             for (int j = 0; j <= n; j++)
82             {
83                 if (phase == 2 && N[j] == -1)
84                     continue;
85                 if (s == -1 || D[x][j] < D[x][s] || D[x][j] ==
D[x][s] && N[j] < N[s])
86                     s = j;
87             }
88             if (D[x][s] > -EPS)
89                 return true;
90             int r = -1;
91             for (int i = 0; i < m; i++)
92             {
93                 if (D[i][s] < EPS)
94                     continue;
95                 if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n
+ 1] / D[r][s] ||
96                     (D[i][n + 1] / D[i][s]) == (D[r][n +
1] / D[r][s]) && B[i] < B[r])
97                     r = i;
98             }
99             if (r == -1)
100                 return false;
101             Pivot(r, s);
102         }
103     }
104
105     DOUBLE Solve(VD &x)
106     {
107         int r = 0;
108         for (int i = 1; i < m; i++)
109             if (D[i][n + 1] < D[r][n + 1])
110                 r = i;
111         if (D[r][n + 1] < -EPS)
112         {
113             Pivot(r, n);

```

```

114         if (!Simplex(1) || D[m + 1][n + 1] < -EPS)
115             return -numeric_limits<DOUBLE>::infinity();
116         for (int i = 0; i < m; i++)
117             if (B[i] == -1)
118             {
119                 int s = -1;
120                 for (int j = 0; j <= n; j++)
121                     if (s == -1 || D[i][j] < D[i]
[s] || D[i][j] == D[i][s] && N[j] < N[s])
122                         s = j;
123                 Pivot(i, s);
124             }
125     }
126     if (!Simplex(2))
127         return numeric_limits<DOUBLE>::infinity();
128     x = VD(n);
129     for (int i = 0; i < m; i++)
130         if (B[i] < n)
131             x[B[i]] = D[i][n + 1];
132     return D[m][n + 1];
133 }
134 };
135
136 int main()
137 {
138     //      Example:
139     //      maximize      x1 - x2
140     //      subject to
141     //          6x1 - x2      <= 10
142     //          -x1 - 5x2     <= -4
143     //          x1 + 5x2 + x3 <= 5
144     //          -x1 - 5x2 - x3 <= -5
145     //
146     //          x1 >= 0
147     //          x2 >= 0
148     //          x3 >= 0
149
150     const int n = 4;
151     const int m = 3;
152     DOUBLE _A[n][m] =
153     {
154         {6, -1, 0},
155         {-1, -5, 0},
156         {1, 5, 1},
157         {-1, -5, -1}
158     };
159     DOUBLE _b[n] = {10, -4, 5, -5};
160     DOUBLE _c[m] = {1, -1, 0};
161
162     VVD A(n);
163     VD b(_b, _b + n);
164     VD c(_c, _c + m);
165     for (int i = 0; i < n; i++)
166         A[i] = VD(_A[i], _A[i] + m);
167
168     LPSolver solver(A, b, c);
169     VD x;
170     DOUBLE value = solver.Solve(x);
171

```

```
172     cerr << "VALUE: " << value << endl; // VALUE: 1.29032
173     cerr << "SOLUTION:"; // SOLUTION: 1.74194 0.451613 1
174     for (size_t i = 0; i < x.size(); i++)
175         cerr << " " << x[i];
176     cerr << endl;
177     return 0;
178 }
```

```

1  // C++ routines for computational geometry.
2
3  #include <iostream>
4  #include <vector>
5  #include <cmath>
6  #include <cassert>
7
8  using namespace std;
9
10 double INF = 1e100;
11 double EPS = 1e-12;
12
13 struct PT {
14     double x, y;
15     PT() {}
16     PT(double x, double y) : x(x), y(y) {}
17     PT(const PT &p) : x(p.x), y(p.y) {}
18     PT operator + (const PT &p) const { return PT(x+p.x, y+p.y); }
19     PT operator - (const PT &p) const { return PT(x-p.x, y-p.y); }
20     PT operator * (double c) const { return PT(x*c, y*c); }
21     PT operator / (double c) const { return PT(x/c, y/c); }
22 };
23
24 double dot(PT p, PT q) { return p.x*q.x+p.y*q.y; }
25 double dist2(PT p, PT q) { return dot(p-q,p-q); }
26 double cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
27 ostream &operator<<(ostream &os, const PT &p) {
28     os << "(" << p.x << "," << p.y << ")";
29 }
30
31 // rotate a point CCW or CW around the origin
32 PT RotateCCW90(PT p) { return PT(-p.y,p.x); }
33 PT RotateCW90(PT p) { return PT(p.y,-p.x); }
34 PT RotateCCW(PT p, double t) {
35     return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
36 }
37
38 // project point c onto line through a and b
39 // assuming a != b
40 PT ProjectPointLine(PT a, PT b, PT c) {
41     return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
42 }
43
44 // project point c onto line segment through a and b
45 PT ProjectPointSegment(PT a, PT b, PT c) {
46     double r = dot(b-a,b-a);
47     if (fabs(r) < EPS) return a;
48     r = dot(c-a, b-a)/r;
49     if (r < 0) return a;
50     if (r > 1) return b;
51     return a + (b-a)*r;
52 }
53
54 // compute distance from c to segment between a and b
55 double DistancePointSegment(PT a, PT b, PT c) {
56     return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
57 }
58
59 // compute distance between point (x,y,z) and plane ax+by+cz=d

```

```

60 double DistancePointPlane(double x, double y, double z,
61                             double a, double b, double c, double d)
62 {
63     return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
64 }
65
66 // determine if lines from a to b and c to d are parallel or collinear
67 bool LinesParallel(PT a, PT b, PT c, PT d) {
68     return fabs(cross(b-a, c-d)) < EPS;
69 }
70
71 bool LinesCollinear(PT a, PT b, PT c, PT d) {
72     return LinesParallel(a, b, c, d)
73         && fabs(cross(a-b, a-c)) < EPS
74         && fabs(cross(c-d, c-a)) < EPS;
75 }
76
77 // determine if line segment from a to b intersects with
78 // line segment from c to d
79 bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
80     if (LinesCollinear(a, b, c, d)) {
81         if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
82             dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
83         if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
84             return false;
85         return true;
86     }
87     if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
88     if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
89     return true;
90 }
91
92 // compute intersection of line passing through a and b
93 // with line passing through c and d, assuming that unique
94 // intersection exists; for segment intersection, check if
95 // segments intersect first
96 PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
97     b=b-a; d=c-d; c=c-a;
98     assert(dot(b, b) > EPS && dot(d, d) > EPS);
99     return a + b*cross(c, d)/cross(b, d);
100 }
101
102 // compute center of circle given three points
103 PT ComputeCircleCenter(PT a, PT b, PT c) {
104     b=(a+b)/2;
105     c=(a+c)/2;
106     return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(a-c));
107 }
108
109 // determine if point is in a possibly non-convex polygon (by William
110 // Randolph Franklin); returns 1 for strictly interior points, 0 for
111 // strictly exterior points, and 0 or 1 for the remaining points.
112 // Note that it is possible to convert this into an *exact* test using
113 // integer arithmetic by taking care of the division appropriately
114 // (making sure to deal with signs properly) and then by writing exact
115 // tests for checking point on polygon boundary
116 bool PointInPolygon(const vector<PT> &p, PT q) {
117     bool c = 0;
118     for (int i = 0; i < p.size(); i++){

```

```

119     int j = (i+1)%p.size();
120     if ((p[i].y <= q.y && q.y < p[j].y ||
121         p[j].y <= q.y && q.y < p[i].y) &&
122         q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
123         c = !c;
124 }
125 return c;
126 }
127
128 // determine if point is on the boundary of a polygon
129 bool PointOnPolygon(const vector<PT> &p, PT q) {
130     for (int i = 0; i < p.size(); i++)
131         if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
132             return true;
133     return false;
134 }
135
136 // compute intersection of line through points a and b with
137 // circle centered at c with radius r > 0
138 vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
139     vector<PT> ret;
140     b = b-a;
141     a = a-c;
142     double A = dot(b, b);
143     double B = dot(a, b);
144     double C = dot(a, a) - r*r;
145     double D = B*B - A*C;
146     if (D < -EPS) return ret;
147     ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
148     if (D > EPS)
149         ret.push_back(c+a+b*(-B-sqrt(D))/A);
150     return ret;
151 }
152
153 // compute intersection of circle centered at a with radius r
154 // with circle centered at b with radius R
155 vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {
156     vector<PT> ret;
157     double d = sqrt(dist2(a, b));
158     if (d > r+R || d+min(r, R) < max(r, R)) return ret;
159     double x = (d*d-R*R+r*r)/(2*d);
160     double y = sqrt(r*r-x*x);
161     PT v = (b-a)/d;
162     ret.push_back(a+v*x + RotateCCW90(v)*y);
163     if (y > 0)
164         ret.push_back(a+v*x - RotateCCW90(v)*y);
165     return ret;
166 }
167
168 // This code computes the area or centroid of a (possibly nonconvex)
169 // polygon, assuming that the coordinates are listed in a clockwise or
170 // counterclockwise fashion. Note that the centroid is often known as
171 // the "center of gravity" or "center of mass".
172 double ComputeSignedArea(const vector<PT> &p) {
173     double area = 0;
174     for(int i = 0; i < p.size(); i++) {
175         int j = (i+1) % p.size();
176         area += p[i].x*p[j].y - p[j].x*p[i].y;
177     }

```

```

178     return area / 2.0;
179 }
180
181 double ComputeArea(const vector<PT> &p) {
182     return fabs(ComputeSignedArea(p));
183 }
184
185 PT ComputeCentroid(const vector<PT> &p) {
186     PT c(0,0);
187     double scale = 6.0 * ComputeSignedArea(p);
188     for (int i = 0; i < p.size(); i++){
189         int j = (i+1) % p.size();
190         c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
191     }
192     return c / scale;
193 }
194
195 // tests whether or not a given polygon (in CW or CCW order) is simple
196 bool IsSimple(const vector<PT> &p) {
197     for (int i = 0; i < p.size(); i++) {
198         for (int k = i+1; k < p.size(); k++) {
199             int j = (i+1) % p.size();
200             int l = (k+1) % p.size();
201             if (i == l || j == k) continue;
202             if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
203                 return false;
204         }
205     }
206     return true;
207 }
208
209 int main() {
210
211     // expected: (-5,2)
212     cerr << RotateCCW90(PT(2,5)) << endl;
213
214     // expected: (5,-2)
215     cerr << RotateCW90(PT(2,5)) << endl;
216
217     // expected: (-5,2)
218     cerr << RotateCCW(PT(2,5),M_PI/2) << endl;
219
220     // expected: (5,2)
221     cerr << ProjectPointLine(PT(-5,-2), PT(10,4), PT(3,7)) << endl;
222
223     // expected: (5,2) (7.5,3) (2.5,1)
224     cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7)) << " "
225         << ProjectPointSegment(PT(7.5,3), PT(10,4), PT(3,7)) << " "
226         << ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT(3,7)) << endl;
227
228     // expected: 6.78903
229     cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;
230
231     // expected: 1 0 1
232     cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
233         << LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
234         << LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;
235
236     // expected: 0 0 1

```



```

237     cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
238         << LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
239         << LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;
240
241 // expected: 1 1 1 0
242 cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << " "
243     << SegmentsIntersect(PT(0,0), PT(2,4), PT(4,3), PT(0,5)) << " "
244     << SegmentsIntersect(PT(0,0), PT(2,4), PT(2,-1), PT(-2,1)) << " "
245     << SegmentsIntersect(PT(0,0), PT(2,4), PT(5,5), PT(1,7)) << endl;
246
247 // expected: (1,2)
248 cerr << ComputeLineIntersection(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << endl;
249
250 // expected: (1,1)
251 cerr << ComputeCircleCenter(PT(-3,4), PT(6,1), PT(4,5)) << endl;
252
253 vector<PT> v;
254 v.push_back(PT(0,0));
255 v.push_back(PT(5,0));
256 v.push_back(PT(5,5));
257 v.push_back(PT(0,5));
258
259 // expected: 1 1 1 0 0
260 cerr << PointInPolygon(v, PT(2,2)) << " "
261     << PointInPolygon(v, PT(2,0)) << " "
262     << PointInPolygon(v, PT(0,2)) << " "
263     << PointInPolygon(v, PT(5,2)) << " "
264     << PointInPolygon(v, PT(2,5)) << endl;
265
266 // expected: 0 1 1 1 1
267 cerr << PointOnPolygon(v, PT(2,2)) << " "
268     << PointOnPolygon(v, PT(2,0)) << " "
269     << PointOnPolygon(v, PT(0,2)) << " "
270     << PointOnPolygon(v, PT(5,2)) << " "
271     << PointOnPolygon(v, PT(2,5)) << endl;
272
273 // expected: (1,6)
274 //           (5,4) (4,5)
275 //           blank line
276 //           (4,5) (5,4)
277 //           blank line
278 //           (4,5) (5,4)
279 vector<PT> u = CircleLineIntersection(PT(0,6), PT(2,6), PT(1,1), 5);
280 for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
281 u = CircleLineIntersection(PT(0,9), PT(9,0), PT(1,1), 5);
282 for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
283 u = CircleCircleIntersection(PT(1,1), PT(10,10), 5, 5);
284 for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
285 u = CircleCircleIntersection(PT(1,1), PT(8,8), 5, 5);
286 for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
287 u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 10, sqrt(2.0)/2.0);
288 for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
289 u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 5, sqrt(2.0)/2.0);
290 for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
291
292 // area should be 5.0
293 // centroid should be (1.1666666, 1.166666)
294 PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
295 vector<PT> p(pa, pa+4);

```

```
296     PT c = ComputeCentroid(p);
297     cerr << "Area: " << ComputeArea(p) << endl;
298     cerr << "Centroid: " << c << endl;
299
300     return 0;
301 }
```

```
1  @ECHO OFF
2
3  SET generator=%1
4  SET checkProgram=%2
5  SET correctProgram=%3
6
7  for /l %%x in (1, 0, 1) do (
8  %generator% > test.in
9  if %errorlevel% neq 0 exit
10 %checkProgram% < test.in > test1.out
11 if %errorlevel% neq 0 exit
12 %correctProgram% < test.in > test2.out
13 if %errorlevel% neq 0 exit
14 FC test1.out test2.out
15 if %errorlevel% neq 0 exit
16 del test.in test1.out test2.out
17 timeout 1 /NOBREAK
18 )
```