# Thư viện hình học

**int** **cmp**(**double** a, **double** b) {

**if** (**abs**(a - b) < EPS) {

**return** 0;

}

**return** (a > b ? 1 : -1);

} **struct** Point {

**double** x, y;

**Point**(**double** \_x = 0, **double** \_y = 0) {

x = \_x, y = \_y;

}

**bool** **operator ==** (**const** Point& that) **const** {

**return** (cmp(x, that.x) == 0 && cmp(y, that.y) == 0);

}

**bool** **operator <** (**const** Point& that) **const** {

**if** (cmp(x, that.x) != 0) {

**return** cmp(x, that.x) < 0;

}

**return** cmp(y, that.y) < 0;

}

};

Kiểm tra khi đi từ điểm p0 qua điểm p1 rồi đến điểm p2 thì góc rẽ ở p1 có ngược chiều kim đồng hồ hay không. Trả về:  
 0: Nếu ba điểm thẳng hàng  
 1: Nếu góc rẽ ở điểm p1 ngược chiều kim đồng hồ (rẽ trái)  
 -1: Nếu góc rẽ ở điểm p1 xuôi chiều kim đồng hồ (rẽ phải)

**int** **ccw**(Point p0, Point p1, Point p2) {

**double** dx1 = p1.x - p0.x, dy1 = p1.y - p0.y;

**double** dx2 = p2.x - p0.x, dy2 = p2.y - p0.y;

**return** cmp(dx1 \* dy2 - dx2 \* dy1, 0);

}

Trả về vị trí của điểm p0 so với đoạn thẳng có 2 đầu mút p1, p2:  
 -2: Không xác định được đoạn thẳng vì điểm p1 và p2 trùng nhau  
 -1: Không nằm trên đoạn thẳng nối bởi 2 điểm p1, p2  
 0: Nằm trên đoạn thẳng  
 1:Nằm trên đường thẳng nhưng ngoài đoạn, ở gần điểm nhỏ hơn trong 2 điểm p1, p2  
 2:Nằm trên đường thẳng nhưng ngoài đoạn, ở gần điểm lớn hơn trong 2 điểm p1, p2

**int** **segmentPos**(Point p0, Point p1, Point p2) {

**if** (p1 == p2) {

**return** -2;

} **else** **if** (ccw(p0, p1, p2) != 0) {

**return** -1;

} **else** **if** (p2 < p1) {

swap(p1, p2);

}

**if** (p0 < p1) {

**return** 1;

} **else** **if** (p2 < p0) {

**return** 2;

}

**return** 0;

}

**bool** **linePos**(Point p0, Point p1, Point p2) {

**return** (ccw(p0, p1, p2) == 0);

}

**bool** **getLine**(Point p0, Point p1, **double**& a, **double**& b, **double**& c) {

**if** (p0 == p1) {

**return** **false**;

}

a = p1.y - p0.y;

b = p0.x - p1.x;

c = -(a \* p0.x + b \* p0.y);

**return** **true**;

}

Tìm giao điểm của 2 đường thẳng p0.p1 và p2.p3, giao điểm được trả về tại p4 (nếu có). Hàm trả về:  
 -1: Nếu 2 đường thẳng đã cho trùng nhau  
 0: Nếu chúng song song  
 1: Nếu chúng cắt nhau tại điểm p4

**int** **getIntersection**(Point p0, Point p1, Point p2, Point p3, Point& p4) {

**double** a0, b0, c0, a1, b1, c1;

getLine(p0, p1, a0, b0, c0);

getLine(p2, p3, a1, b1, c1);

**double** d = a0 \* b1 - a1 \* b0;

**double** dx = b0 \* c1 - b1 \* c0;

**double** dy = -(c1 \* a0 - c0 \* a1);

**if** (cmp(d, 0) == 0) {

**return** (cmp(dx, 0) == 0 && cmp(dy, 0) == 0 ? -1 : 0);

}

p4.x = dx / d;

p4.y = dy / d;

**return** 1;

}

**double** **dist**(Point p0, Point p1) {

**double** dx = p1.x - p0.x, dy = p1.y - p0.y;

**return** **sqrt**(dx \* dx + dy \* dy);

}

**bool** **isSegmentCut**(Point p0, Point p1, Point p2, Point p3) {

**if** (ccw(p0, p1, p2) \* ccw(p0, p1, p3) < 0 &&

ccw(p2, p3, p0) \* ccw(p2, p3, p1) < 0) {

**return** **true**;

}

**return** (segmentPos(p0, p2, p3) == 0 ||

segmentPos(p1, p2, p3) == 0 ||

segmentPos(p2, p0, p1) == 0 ||

segmentPos(p3, p0, p1) == 0 ? **true** : **false**);

}

Point p0;

**bool** **degreeCmp**(Point p1, Point p2) {

**int** d = ccw(p0, p1, p2);

**if** (d != 0) {

**return** (d > 0);

}

**return** cmp(dist(p0, p1), dist(p0, p2)) < 0;

}

Tìm bao lồi của tập điểm p[] sử dụng thuật toán Graham Scan, O(N logN).  
Sắp xếp lại tập điểm p[] và trả về số điểm thuộc bao lồi tại n, khi ấy các điểm thuộc bao lồi sẽ là n điểm đầu tiên (0..n - 1) của p[].

**void** **grahamScan**(Point p[], **int**& n) {

**if** (n == 1) {

**return**;

}

**int** j = 0;

For(i, 1, n - 1) {

**if** (cmp(p[j].y, p[i].y) > 0 ||

(cmp(p[j].y, p[i].y) == 0 && cmp(p[j].x, p[i].x) < 0)) {

j = i;

}

}

swap(p[0], p[j]);

p0 = p[0];

sort(p + 1, p + n, degreeCmp);

**int** k = 2;

For(i, 2, n - 1) {

**while** (k > 1 && ccw(p[i], p[k - 1], p[k - 2]) >= 0) {

k--;

}

swap(p[k++], p[i]);

}

n = k;

}

Kiểm tra điểm p0 có nằm trong đa giác lồi hay không, O(N).

**bool** **insideConvexPolygon**(Point p0, Point p[], **int** n) {

**int** k, l;

**if** (n == 1) {

**return** **false**;

}

**if** (n == 2) {

**if** (ccw(p0, p[0], p[1]) == 0) {

**return** (segmentPos(p0, p[0], p[1]) == 0);

} **else** {

**return** **false**;

}

}

k = l = 0;

Rep(i, n) {

**int** j = ccw(p0, p[i], p[(i + 1) % n]);

k = (j < 0) ? 1 : k;

l = (j > 0) ? 1 : l;

}

**return** (k + l != 2);

}

**bool** **notHaveCommonPointBetween2ConvexPolygons**(Point p1[], **int** n1, Point p2[], **int** n2) {

Rep(i, n1) {

**if** (insideConvexPolygon(p1[i], p2, n2)) {

**return** **false**;

}

}

Rep(i, n2) {

**if** (insideConvexPolygon(p2[i], p1, n1)) {

**return** **false**;

}

}

Rep(i, n1) {

Rep(j, n2) {

**if** (isSegmentCut(p1[i], p1[(i + 1) % n1], p2[j], p2[(j + 1) % n2])) {

**return** **false**;

}

}

}

**return** **true**;

}

Tính diện tích đa giác bị giới hạn bởi {đường chéo nối từ đỉnh i đến đỉnh j} và {các cạnh của đa giác nhận được khi đi từ đỉnh i đến đỉnh j, theo chiều tăng dần của chỉ số các đỉnh}.

**double** **diagonalLineArea**(Point p[], **int** n, **int** i, **int** j) {

**double** s = 0;

**for** (**int** k = i; ; k = (k + 1) % n) {

**int** l = (k == j) ? i : (k + 1) % n;

s += (p[l].x - p[k].x) \* (p[l].y + p[k].y);

**if** (k == j) {

**break**;

}

}

**return** abs(s) / 2.0;

}

Kiểm tra hai đoạn thẳng có thực sự cắt nhau hay không (giao điểm không trùng với 4 điểm đầu mút).

**bool** **isRealCut**(Point p0, Point p1, Point p2, Point p3) {

**return** ccw(p0, p1, p2) \* ccw(p0, p1, p3) < 0

&& ccw(p2, p3, p0) \* ccw(p2, p3, p1) < 0;

}

Kiểm tra đường chéo của một đa giác (không nhất thiết là đa giác lồi) có cắt cạnh nào của chính đa giác đó hay không.

**bool** **isDiagonalLineCutPolygon**(Point p[], **int** n, Point p0, Point p1) {

Rep(i, n) {

**int** j = (i + 1) % n;

**if** (isRealCut(p0, p1, p[i], p[j])) {

**return** **true**;

}

}

**return** **false**;

}

Kiểm tra đường chéo của một đa giác (không nhất thiết là đa giác lồi) có nằm trong đa giác đó hay không.

**bool** **isDiagonalLineInsidePolygon**(Point p[], **int** n, **int** i, **int** j) {

**if** (isDiagonalLineCutPolygon(p, n, p[i], p[j])) {

**return** **false**;

}

**return** cmp(-diagonalLineArea(p, n, 1, 0) + diagonalLineArea(p, n, i, j)

+ diagonalLineArea(p, n, j, i), 0) == 0;

}

Kiểm tra điểm có nằm trong đa giác (không nhất thiết là đa giác lồi) hay không, O(N).

**bool** **insidePolygon**(Point p[], **int** n, Point p0) {

Rep(i, n) {

**int** j = (i + 1) % n;

**if** (segmentPos(p0, p[i], p[j]) == 0) {

**return** **true**;

}

}

p0.y += EPS;

**int** k = 0;

Rep(i, n) {

**int** j = (i + 1) % n;

**if** ((p0.y - p[i].y) \* (p0.y - p[j].y) < 0) {

**double** a, b, c;

getLine(p[i], p[j], a, b, c);

**double** x1 = -(c + b \* p0.y) / a;

k += (cmp(x1, p0.x) >= 0);

}

}

**return** (k % 2 > 0);

}

**void** **add**(Point p3[], **int**& n3, Point p0) {

**if** (n3 > 0 && p0 == p3[n3 - 1])

**return**;

**if** (n3 > 1 && ccw(p3[n3 - 2], p3[n3 - 1], p0) == 0) {

p3[n3 - 1] = p0;

**return**;

}

p3[n3++] = p0;

}

Cho một đa giác không tự cắt (không nhất thiết là đa giác lồi). Dùng đường thẳng p1.p2 để phân chia đa giác đó thành nhiều phần. Sử dụng p[] và n cho cả 2 mục đích in/out. Kết quả trả về là n điểm đầu tiên (0..n-1) của p[].  
*Lưu ý: Đường thẳng p1.p2 không được đi qua p0.*

**void** **divide**(Point p[], **int**& n, Point p0, Point p1, Point p2) {

**double** a, b, c;

getLine(p1, p2, a, b, c);

**int** k;

Rep(i, n) {

**if** ((p[i].x \* a + p[i].y \* b + c) \* (p0.x \* a + p0.y \* b + c) > 0) {

k = i;

**break**;

}

}

Point p3[1010];

**int** n3 = 0;

add(p3, n3, p[k]);

For(l, 1, n) {

**int** i = (k + l) % n;

**int** j = (i + n - 1) % n;

**if** ((p[j].x \* a + p[j].y \* b + c) \* (p[i].x \* a + p[i].y \* b + c) <= 0) {

Point p4;

**if** (getIntersection(p[j], p[i], p1, p2, p4) == 1) {

add(p3, n3, p4);

}

}

**if** (i != k) {

**if** ((p[i].x \* a + p[i].y \* b + c) \* (p0.x \* a + p0.y \* b + c) > 0) {

add(p3, n3, p[i]);

}

}

}

n = n3;

Rep(i, n) {

p[i] = p3[i];

}

}

Tìm giao của 2 đa giác lồi.

**void** **getIntersectionOf2ConvexPolygons**(Point p1[], **int** n1, Point p2[], **int** n2,

Point p0[], **int**& n0) {

n0 = 0;

Rep(i, n1) {

**if** (insideConvexPolygon(p1[i], p2, n2)) {

p0[n0++] = p1[i];

}

}

Rep(i, n2) {

**if** (insideConvexPolygon(p2[i], p1, n1)) {

p0[n0++] = p2[i];

}

}

Rep(i, n1) {

Rep(j, n2) {

**int** k = (i + 1) % n1;

**int** l = (j + 1) % n2;

Point p3;

getIntersection(p1[i], p1[k], p2[j], p2[l], p3);

**if** (segmentPos(p3, p1[i], p1[k]) == 0) {

**if** (segmentPos(p3, p2[j], p2[l]) == 0) {

p0[n0++] = p3;

}

}

}

}

grahamScan(p0, n0);

}

Tính góc giữa 2 vector khác 0 p0.p1 và p0.p2 (tính theo Radian)

**double** **getAngle**(Point p0, Point p1, Point p2) {

**double** d1 = dist(p0, p1);

**double** d2 = dist(p0, p2);

**return** **acos**(((p1.x - p0.x) \* (p2.x - p0.x) + (p1.y - p0.y) \* (p2.y - p0.y))

/ (d1 \* d2));

}

Kiểm tra điểm p0 có nằm trong góc tạo bởi 2 tia p1.p2 và p1.p3 hay không.

**bool** **insideAngle**(Point p0, Point p1, Point p2, Point p3) {

**if** (p0 == p1)

**return** **true**;

**if** (ccw(p0, p1, p2) \* ccw(p0, p1, p3) > 0)

**return** **false**;

**return** (getAngle(p1, p0, p2) < PI + EPS && getAngle(p1, p0, p3) < PI + EPS);

}

Kiểm tra điểm p0 có nằm trong tam giác p1.p2.p3 hay không.

**bool** **insideTriangle**(Point p0, Point p1, Point p2, Point p3) {

**if** (segmentPos(p0, p1, p2) == 0) {

**return** **true**;

}

**if** (segmentPos(p0, p2, p3) == 0) {

**return** **true**;

}

**if** (segmentPos(p0, p1, p3) == 0) {

**return** **true**;

}

**return** (ccw(p0, p1, p2) \* ccw(p0, p1, p3) < 0

&& ccw(p0, p2, p1) \* ccw(p0, p2, p3) < 0);

}

Kiểm tra điểm p0 có nằm trong đa giác lồi p[] hay không, O(logN).

**bool** **insideConvexPolygonLogN**(Point p0, Point p[], **int** n) {

**if** (!insideAngle(p0, p[0], p[1], p[n - 1])) {

**return** **false**;

}

**int** i = 1;

**int** j = n - 1;

**while** (j - i > 1) {

**int** k = (i + j) >> 1;

**if** (insideAngle(p0, p[0], p[i], p[k])) {

j = k;

} **else** {

i = k;

}

}

**return** insideTriangle(p0, p[0], p[i], p[j]);

}

Tìm giao của 2 đường tròn, các giao điểm được trả về trong vector vp. Ngoài ra hàm còn trả về giá trị là số lượng giao điểm (-1 nếu có vô số giao điểm – tức 2 đường tròn trùng nhau).

**int** **intersectionsOf2Circles**(Point p0, **double** r0, Point p1, **double** r1, vector<Point>& vp) {

vp.clear();

**double** d = dist(p0, p1);

**if** (d > r0 + r1) {

**return** 0;

}

**if** (d < abs(r0 - r1)) {

**return** 0;

}

**if** (cmp(r0 - r1, 0) == 0 && cmp(p0.x - p1.x, 0) == 0 && cmp(p0.y - p1.y, 0) == 0) {

**return** -1;

}

**double** a = (r0 \* r0 - r1 \* r1 + d \* d) / 2 / d;

**double** h = **sqrt**(r0 \* r0 - a \* a);

Point p2(p0.x + a \* (p1.x - p0.x) / d, p0.y + a \* (p1.y - p0.y) / d);

vp.PB(Point(p2.x + h \* (p1.y - p0.y) / d, p2.y - h \* (p1.x - p0.x) / d));

**if** (cmp(h, 0)) {

**return** 1;

}

vp.PB(Point(p2.x - h \* (p1.y - p0.y) / d, p2.y + h \* (p1.x - p0.x) / d));

**return** 2;

}

Tìm giao của đường thẳng và đường tròn, các giao điểm được trả về trong vector p, ngoài ra hàm trả về giá trị là số lượng giao điểm.

**int** **intersectionOfCircleAndLine**(Point p0, **double** r0, Point p1, Point p2,

vector<Point>& p) {

p.clear();

Point u = p2 - p1;

Point proj = projection(p0 - p1, u) + p1;

**if** ((proj - p0).length() < r0 - eps) {

ld add = sqrt(r0 \* r0 - (proj - p0) \* (proj - p0));

u = u \* (add / u.length());

p.pb(proj + u);

p.pb(proj - u);

**return** 2;

} **else** **if** (abs((proj - p0).length() - r0) <= eps) {

p.pb(proj);

**return** 1;

} **else**

**return** 0;

}

Tìm tiếp điểm khi kẻ tiếp tuyến từ điểm p1 đến đường tròn (p0, r), các tiếp điểm được trả về trong vector p. Ngoài ra hàm còn trả về giá trị là số lượng tiếp điểm:  
 0: Nếu p1 nằm trong đường tròn  
 1: Nếu p1 nằm trên đường tròn  
 2: Nếu p1 nằm ngoài – tức có thể vẽ 2 tiếp tuyến tới đường tròn

**int** **getLineFromPointToCircle**(Point p0, **double** r, Point p1, vector<Point>& p) {

**double** d = dist(p0, p1);

**if** (cmp(d, r) < 0) {

**return** 0;

}

**if** (cmp(d, r) == 0) {

**return** 1;

}

**double** t1 = **atan2**(p1.y - p0.y, p1.x - p0.x);

**double** t0 = **acos**(r / d);

p.PB(Point(p0.x + r \* **cos**(t1 - t0), p0.y + r \* **sin**(t1 - t0)));

p.PB(Point(p0.x + r \* **cos**(t1 + t0), p0.y + r \* **sin**(t1 + t0)));

**return** 2;

}

Lấy đường trung trực của đoạn thẳng p0.p1.  
Nếu p0 và p1 trùng nhau thì không tồn tại đường trung trực, trả về FALSE, ngược lại trả về TRUE.

**bool** **getCenterLine**(Point p0, Point p1, **double**& a, **double**& b, **double**& c) {

**if** (p0 == p1) {

**return** **false**;

}

a = p1.x - p0.x;

b = p1.y - p0.y;

Point p2;

p2.x = (p0.x + p1.x) / 2;

p2.y = (p0.y + p1.y) / 2;

c = -(p2.x \* a + p2.y \* b);

**return** **true**;

}

Lấy đường tròn đi qua 3 điểm. Trả về FALSE nếu không tồn tại – tức 3 điểm thẳng hàng, ngược lại trả về TRUE.

**bool** **getCircle**(Point p0, Point p1, Point p2, Point& p3, **double**& r) {

**if** (ccw(p0, p1, p2) == 0) {

**return** **false**;

}

**double** a0, b0, c0, a1, b1, c1;

getCenterLine(p0, p1, a0, b0, c0);

getCenterLine(p0, p2, a1, b1, c1);

**double** d = a0 \* b1 - a1 \* b0;

**double** dx = b0 \* c1 - b1 \* c0;

**double** dy = -(c1 \* a0 - c0 \* a1);

p3.x = dx / d;

p3.y = dy / d;

r = dist(p3, p0);

**return** **true**;

}

Tìm khoảng cách giữa 2 điểm gần nhau nhất trong tập điểm.

**struct** YComparator {

**bool** **operator ()** (**const** Point p0, **const** Point p1) **const** {

**if** (p0.y != p1.y) {

**return** p0.y < p1.y;

}

**return** p0.x < p1.x;

}

};  
 **double** **closestPairDist**(Point a[], **int** n) {

sort(a, a + n);

set <Point, YComparator> b;

**int** j = 0;

**double** ret = 1E100;

Rep(i, n) {

**while** (a[i].x - a[j].x > ret) {

b.erase(a[j++]);

}

Point c = a[i];

c.y -= ret;

VAR(lowerIt, b.lower\_bound(c));

c.y += 2 \* ret;

VAR(upperIt, b.upper\_bound(c));

**for** (VAR(it, lowerIt); it != upperIt; it++) {

ret = min(ret, dist(a[i], \*it));

}

b.insert(a[i]);

}

**return** ret;

}

3D convex hull

/\*

V - E + F = 2

E <= 3V - 6

F <= 2V - 4

\*/

**inline** **double** **det**(**double** a, **double** b, **double** c, **double** d) {

**return** a \* d - b \* c;

}

**struct** Point {

**double** x, y, z;

**Point**() {

}

**Point**(**double** x, **double** y, **double** z) :

x(x), y(y), z(z) {

}

Point **operator -**(**const** Point &op) **const** {

**return** Point(x - op.x, y - op.y, z - op.z);

}

**double** **operator \***(**const** Point &op) **const** {

**return** x \* op.x + y \* op.y + z \* op.z;

}

**double** **length**() {

**return** **sqrt**(x \* x + y \* y + z \* z);

}

Point **operator %**(**const** Point &op) **const** {

**return** Point(det(y, z, op.y, op.z), -det(x, z, op.x, op.z), det(x, y,

op.x, op.y));

}

};

**typedef** vector<**int**> Side;

**inline** **int** **sign**(**double** x) {

**return** x < -eps ? -1 : x > eps ? 1 : 0;

}

vector<Point> arr;

vector<**int**> rnd;

set<**int**> used;

Side **getFirstSide**(vector<Point> &p) {

**int** i1 = 0;

Rep(i,sz(p)) {

**if** (p[i].z < p[i1].z || (p[i].z == p[i1].z && p[i].x < p[i1].x)

|| (p[i].z == p[i1].z && p[i].x == p[i1].x && p[i].y < p[i1].y)) {

i1 = i;

}

}

**int** i2 = i1 == 0 ? 1 : 0;

Rep(i,sz(p)) {

**if** (i != i1) {

Point zDir(0, 0, 1);

**double** curCos = (p[i] - p[i1]) \* zDir / (p[i] - p[i1]).length();

**double** bestCos = (p[i2] - p[i1]) \* zDir / (p[i2] - p[i1]).length();

**if** (curCos < bestCos) {

i2 = i;

}

}

}

**int** i3 = -1;

**int** n = sz(p);

Rep(ri,n) {

**int** i = rnd[ri];

**if** (i != i1 && i != i2) {

Point norm = (p[i1] - p[i]) % (p[i2] - p[i]);

**bool** sg[] = { 0, 0, 0 };

Rep(t,n) {

**int** j = rnd[t];

sg[1 + sign((p[j] - p[i]) \* norm)] = **true**;

**if** (sg[0] && sg[2]) {

**break**;

}

}

**if** (sg[0] ^ sg[2]) {

i3 = i;

**if** (!sg[0]) {

swap(i3, i2);

}

**break**;

}

}

}

vector<**int**> res;

res.pb(i1);

res.pb(i2);

res.pb(i3);

**return** res;

}

**inline** **int** **getSideKey**(**int** i, **int** j, **int** k) {

**int** key = (i \* 1000 + j) \* 1000 + k;

**return** key;

}

**inline** **bool** **isUsed**(**int** i, **int** j, **int** k) {

**return** used.find(getSideKey(i, j, k)) != used.end();

}

**inline** **double** **getAngle**(**const** Point &n1, **const** Point &n2) {

**return** **atan2**((n1 % n2).length(), n1 \* n2);

}

**inline** **double** **getNormsAngle**(**int** i, **int** j, **int** k, **int** t, vector<Point> &p) {

Point n1 = (p[j] - p[i]) % (p[k] - p[i]);

Point n2 = (p[t] - p[i]) % (p[j] - p[i]);

**return** getAngle(n1, n2);

}

**void** **dfs**(**int** i, **int** j, **int** k, vector<Point> &p, vector<Side> &sides) {

**if** (i < j && i < k) {

vector<**int**> side(3);

side[0] = i;

side[1] = j;

side[2] = k;

sides.pb(side);

}

**int** key = getSideKey(i, j, k);

used.insert(key);

**int** n = sz(p);

**if** (!isUsed(j, k, i))

dfs(j, k, i, p, sides);

**if** (!isUsed(k, i, j))

dfs(k, i, j, p, sides);

**int** bestT = -1;

**double** bestAngle = 1e20;

Point curNorm = (p[j] - p[i]) % (p[k] - p[i]);

Point dir = p[j] - p[i];

Rep(t,n) {

**if** (t != i && t != j && t != k) {

Point newNorm = (p[t] - p[i]) % dir;

**double** curAng = curNorm \* newNorm / newNorm.length();

**if** (bestT == -1 || curAng > bestAngle) {

bestT = t;

bestAngle = curAng;

}

}

}

**if** (!isUsed(i, bestT, j)) {

dfs(i, bestT, j, p, sides);

}

}

vector<Side> **convexHull3d**(vector<Point> p) {

used.clear();

rnd.resize(sz(p));

Rep(i,sz(p))

rnd[i] = i;

random\_shuffle(rnd.begin(), rnd.end());

Side side0 = getFirstSide(p);

vector<Side> sides;

dfs(side0[0], side0[1], side0[2], p, sides);

**return** sides;

}

/\* eliminate conflict sides \*/

**inline** **bool** **isEmpty**(Point x, Point y, Point z) {

**return** abs(x \* Point(y.y \* z.z - y.z \* z.y, y.z \* z.x - y.x \* z.z, y.x

\* z.y - y.y \* z.x)) <= eps;

}

**inline** **bool** **conflict**(Side a, Side b) {

Point x = arr[a[0]], y = arr[a[1]], z = arr[a[2]];

Rep(i,3) {

Point t = arr[b[i]];

**if** (!isEmpty(x - t, y - t, z - t))

**return** **false**;

}

**return** **true**;

}

vector<Side> **eliminate**(vector<Side> p) {

vector<Side> res;

vector<**bool**> fre;

fre.resize(sz(p), **true**);

Rep(i,sz(p)) {

**if** (!fre[i])

**continue**;

res.pb(p[i]);

For(j,i+1,sz(p) - 1)

**if** (fre[j]) {

**if** (conflict(p[i], p[j])) {

fre[j] = **false**;

res.back().insert(res.back().end(), p[j].begin(),

p[j].end());

}

}

}

Rep(i,sz(res)) {

sort(res[i].begin(), res[i].end());

res[i].resize(unique(res[i].begin(), res[i].end()) - res[i].begin());

}

**return** res;

}

Some useful Functions

**inline** Point **projection**(Point v, Point u) {

**long** **double** scalar = (v \* u) / (u \* u);

u.x \*= scalar;

u.y \*= scalar;

u.z \*= scalar;

**return** u;

}

**inline** Point **projection**(Point p, Point a, Point b, Point c) {

Point u = (b - a) % (c - a);

Point v = p - a;

**long** **double** scalar = (v \* u) / (u \* u);

u.x \*= scalar;

u.y \*= scalar;

u.z \*= scalar;

**return** p - u;

}

**inline** **long** **double** **dist**(Point p, Point a, Point b) {

p = p - a;

Point proj = projection(p, b - a);

**return** sqrt(p \* p - proj \* proj);

}

**inline** **long** **double** **area**(Point a, Point b, Point c) {

**long** **double** h = dist(a, b, c);

**return** (h \* (b - c).length()) / 2;

}

**inline** **long** **double** **volume**(Point x, Point y, Point z) {

Point base = Point(y.y \* z.z - y.z \* z.y, y.z \* z.x - y.x \* z.z, y.x \* z.y

- y.y \* z.x);

**return** abs(x.x \* base.x + x.y \* base.y + x.z \* base.z) / 3;

}

# Số học

Dùng Extended Euclid để tìm nghiệm của phương trình ax + by = gcd(a, b). Giả sử kết quả trả về là (x0, y0), họ nghiệm của phương trình sẽ là với . Phương trình tổng quát ax + by = d chỉ có nghiệm khi d chia hết cho gcd(a, b).  
Hàm positiveEE() tìm nghiệm với a, b nguyên không âm, hàm extendedEuclid() tìm nghiệm với a, b nguyên tùy ý.

pll **positiveEE**(ll a, ll b) {

**if** (b == 0) {

**return** MP(1, 0);

}

pll ret = positiveEE(b, a % b);

**return** MP(ret.S, ret.F - ret.S \* (a / b));

}

pll **extendedEuclid**(ll a, ll b) {

pll ret = positiveEE(abs(a), abs(b));

**return** MP((a < 0 ? -ret.F : ret.F), (b < 0 ? -ret.S : ret.S));

}

Tính baseexp theo module MOD trong O(log exp).

ll **fastPow**(ll base, ll exp, ll MOD = MODULO){

**if** (base == 0) {

**return** 0;

}

ll ret = 1, bb = base;

**while** (exp > 0) {

**if** (exp & 1) {

ret = mod(ret \* bb, MOD);

}

exp >>= 1, bb = mod(bb \* bb, MOD);

}

**return** ret;

}

Kiểm tra số nguyên tố bằng thuật toán Rabin-Miller, độ chính xác (1 – 0.25numTry), numTry tối đa 20.

Trick: Với N vào khoảng <= 4E9 chỉ cần test với 3 số **2** **7** **61** là đạt được độ chính xác 100%.

**const** **int** PRIME[] = {2, 3, 5, 7, 11, 13, 17, 19,

23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71};

**bool** **rabinMiller**(ll x, **int** numTry = 15) {

**if** (x <= 2 || (x & 1) == 0) {

**return** (x == 2 ? **true** : **false**);

}

ll k = x - 1;

**int** m = 0;

**for** (; (k & 1) == 0; k >>= 1, m++);

**for** (**int** i = 0; i < numTry && PRIME[i] < x; i++) {

ll t = fastPow(PRIME[i], k, x);

**if** (t == 1 || t == x - 1) {

**continue**;

}

**for** (**int** j = 0; j < m && t != x - 1; j++) {

t = mod(t \* t, x);

}

**if** (t != x - 1) {

**return** **false**;

}

}

**return** **true**;

}

Euler Totient Function

**int** **eulerTotient**(**int** x) {

**int** ret = x, bound = sqrt(x);

For(i, 2, bound) {

**if** (x % i == 0) {

ret = ret / i \* (i - 1);

**for** (; x % i == 0; x /= i);

}

}

**if** (x != 1) {

ret = ret / x \* (x - 1);

}

**return** ret;

}

Tổng các ước dương của 1 số

Phân tích N thành tích của các thừa số nguyên tố:

Thì khi đó, tổng các ước dương của N là:

Giải phương trình đồng dư:

a1x1 + a2x2 + … + anxn đồng dư với b (mod m).

Trong đó, a1, a2, …, an, b và m là các số nguyên dương.

Nếu vô nghiệm, hàm trả về FALSE, ngược lại trả về TRUE và bộ nghiệm (x1, x2, …, xn) được lưu trong vector ret.

**const** **int** MAXN = 1005;

**int** g[MAXN], x[MAXN];

**bool** **congruenceEquation**(vector <**int**> a, **int** b, **int** m, vector <**int**> &ret) {

**int** n = SIZE(a);

a.PB(m);

g[0] = a[0];

For(i, 1, n) g[i] = gcd(g[i - 1], a[i]);

ret.clear();

**if** (b % g[n]) **return** **false**;

**int** val = b / g[n];

Ford(i, n, 1) {

pll p = extendedEuclid(g[i - 1], a[i]);

x[i] = p.S \* val % m;

val = p.F \* val % m;

}

x[0] = val;

For(i, 0, n) x[i] = (x[i] + m) % m;

Rep(i, n) ret.PB(x[i]);

**return** **true**;

}

# Công thức hình học