typedef double db;

const db EPS = 1e-9;

//由于硬件限制，浮点数运算有误差，eps用来消除误差

inline int sign(db a) { return a < -EPS ? -1 : a > EPS; }

//判断数符号，负数返回-1，0返回0，正数返回1

inline int cmp(db a, db b) { return sign(a - b); }

//比较两数大小

//点类，向量类

//因为有许多操作相似，所以并在一起

struct P {

db x, y;

//点表示坐标，向量表示向量

P() {}

P(db \_x, db \_y) : x(\_x), y(\_y) {}

//构造函数

P operator+(P p) { return {x + p.x, y + p.y}; }

P operator-(P p) { return {x - p.x, y - p.y}; }

P operator\*(db d) { return {x \* d, y \* d}; }

P operator/(db d) { return {x / d, y / d}; }

//向量加减乘除

bool operator<(P p) const {

int c = cmp(x, p.x);

if (c)

return c == -1;

return cmp(y, p.y) == -1;

}

bool operator==(P o) const {

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

}

//比较字典序

db dot(P p) { return x \* p.x + y \* p.y; } //点积 (二点之间度数a: >0是a<90度,==0度数a=90度,<0度数a>90度) |a|\*|b|\*cos(a)

//点积 (求的也是平行四边行的面积 )

db det(P p) { return x \* p.y - y \* p.x; } //叉积 (二点之间度数a: >0是0<a<180度,==0度数a=0度||a==180共线,<0度数180<a>360度) |a|\*|b|\*sin(a)

//叉积

db distTo(P p) { return (\*this - p).abs(); } // 到P点距离

//点距离

db alpha() { return atan2(y, x); } // 极角

void read() { cin >> x >> y; }

void write() { cout << "(" << x << "," << y << ")" << endl; }

db abs() { return sqrt(abs2()); } // 到原点距离

db abs2() { return x \* x + y \* y; }

P rot90() { return P(-y, x); } // 逆时针90

P unit() { return \*this / abs(); }// 单位向量

int quad() const { return sign(y) == 1 || (sign(y) == 0 && sign(x) >= 0); }

//判断点在极角坐标系上半边还是下半边，极点和极轴也算上半边

P rot(db an) { return {x \* cos(an) - y \* sin(an), x \* sin(an) + y \* cos(an)}; } // 顺时针an度

//向量旋转

};

//线类，半平面类

struct L { // ps[0] -> ps[1]

P ps[2];

P &operator[](int i) { return ps[i]; }

P dir() { return ps[1] - ps[0]; }

L(P a, P b) {

ps[0] = a;

ps[1] = b;

}

bool include(P p) { return sign((ps[1] - ps[0]).det(p - ps[0])) > 0; }

L push()

{ // push eps outward

const double eps = 1e-8;

P delta = (ps[1] - ps[0]).rot90().unit() \* eps;

return {ps[0] + delta, ps[1] + delta};

}

};

// (p1p2) 叉 (p1p3)

#define cross(p1, p2, p3) ((p2.x - p1.x) \* (p3.y - p1.y) - (p3.x - p1.x) \* (p2.y - p1.y))

// 从p1方向看过去(p2,p3) >0逆时针==0共线<0顺时针

#define crossOp(p1, p2, p3) sign(cross(p1, p2, p3))

// 直线p1p2, q1q2 是否恰有一个交点

bool chkLL(P p1, P p2, P q1, P q2)

{

db a1 = cross(q1, q2, p1), a2 = -cross(q1, q2, p2);

return sign(a1 + a2) != 0;

}

// 求直线p1p2, q1q2 的交点

P isLL(P p1, P p2, P q1, P q2)

{

db a1 = cross(q1, q2, p1), a2 = -cross(q1, q2, p2);

return (p1 \* a2 + p2 \* a1) / (a1 + a2);

}

P isLL(L l1, L l2) { return isLL(l1[0], l1[1], l2[0], l2[1]); }

//求直线交点

// 判断区间[l1, r1], [l2, r2]是否相交

bool intersect(db l1, db r1, db l2, db r2)

{

if (l1 > r1)

swap(l1, r1);

if (l2 > r2)

swap(l2, r2);

return !(cmp(r1, l2) == -1 || cmp(r2, l1) == -1);

}

// 线段相交 (可能是一条线段没有穿过另一条线段)

bool isSS(P p1, P p2, P q1, P q2)

{

return intersect(p1.x, p2.x, q1.x, q2.x) && intersect(p1.y, p2.y, q1.y, q2.y) &&

crossOp(p1, p2, q1) \* crossOp(p1, p2, q2) <= 0 && crossOp(q1, q2, p1) \* crossOp(q1, q2, p2) <= 0;

}

// 线段严格相交 (二条线段交叉穿过)

bool isSS\_strict(P p1, P p2, P q1, P q2)

{

return crossOp(p1, p2, q1) \* crossOp(p1, p2, q2) < 0 && crossOp(q1, q2, p1) \* crossOp(q1, q2, p2) < 0;

}

// 判断 m 在不在a 和 b中间

bool isMiddle(db a, db m, db b)

{

return sign(a - m) == 0 || sign(b - m) == 0 || (a < m != b < m);

}

// 判断点m 是不是横纵坐标都在a和b中间

bool isMiddle(P a, P m, P b)

{

return isMiddle(a.x, m.x, b.x) && isMiddle(a.y, m.y, b.y);

}

// 点p是不是在线段p1p2上 (精度容易出错)

bool onSeg(P p1, P p2, P q)

{

return crossOp(p1, p2, q) == 0 && isMiddle(p1, q, p2);

}

// 点 p 是不是在严格线段 p1p2 上(不包含端点) (精度容易出错)

bool onSeg\_strict(P p1, P p2, P q)

{

return crossOp(p1, p2, q) == 0 && sign((q - p1).dot(p1 - p2)) \* sign((q - p2).dot(p1 - p2)) < 0;

}

// 求 q 到p1p2的投影 （垂足） （p1 不可以等于 p2）

P proj(P p1, P p2, P q)

{

P dir = p2 - p1;

return p1 + dir \* (dir.dot(q - p1) / dir.abs2());

}

// 求 q 以 直线p1p2为轴的反射 （p1 不可以等于 p2）

P reflect(P p1, P p2, P q)

{

return proj(p1, p2, q) \* 2 - q;

}

// 求 q 到 线段p1p2 的最小距离

db nearest(P p1, P p2, P q){

if (p1 == p2)

return p1.distTo(q);

P h = proj(p1, p2, q);

if (isMiddle(p1, h, p2))

return q.distTo(h);

return min(p1.distTo(q), p2.distTo(q));

}

// 求线段p1p2 与 线段q1q2 的距离

db disSS(P p1, P p2, P q1, P q2){

if (isSS(p1, p2, q1, q2))

return 0;

return min(min(nearest(p1, p2, q1), nearest(p1, p2, q2)), min(nearest(q1, q2, p1), nearest(q1, q2, p2)));

}

//线段距离

db rad(P p1, P p2) {

return atan2l(p1.det(p2), p1.dot(p2));

}

db incircle(P p1, P p2, P p3){

db A = p1.distTo(p2);

db B = p2.distTo(p3);

db C = p3.distTo(p1);

return sqrtl(A \* B \* C / (A + B + C));

}

// polygon

//简单多边形的问题只有判断点在多边形内，和多边形面积简单，其他只做凸多边形

db area(vector<P> ps )//多边形面积

{

db ret = 0;

for(int i=0;i<ps.size();++i)

ret += ps[i].det(ps[(i + 1) % ps.size()]);

return ret / 2;

}

int contain(vector<P> ps, P p) //判断点在多边形内

{ // 2:inside,1:on\_seg,0:outside

int n = ps.size(), ret = 0;

for(int i = 0, i < n; i++)

{

P u = ps[i], v = ps[(i + 1) % n];

if (onSeg(u, v, p))

return 1;

if (cmp(u.y, v.y) <= 0)

swap(u, v);

if (cmp(p.y, u.y) > 0 || cmp(p.y, v.y) <= 0)

continue;

ret ^= crossOp(p, u, v) > 0;

}

return ret \* 2;

}

vector<P> convexHull(vector<P> ps)

{

int n = ps.size();

if (n <= 1)

return ps;

sort(ps.begin(), ps.end());

vector<P> qs(n \* 2);

int k = 0;

for (int i = 0; i < n; qs[k++] = ps[i++])

while (k > 1 && crossOp(qs[k - 2], qs[k - 1], ps[i]) <= 0)

--k;

for (int i = n - 2, t = k; i >= 0; qs[k++] = ps[i--])

while (k > t && crossOp(qs[k - 2], qs[k - 1], ps[i]) <= 0)

--k;

qs.resize(k - 1);

return qs;

}

vector<P> convexHullNonStrict(vector<P> ps)

{

// caution: need to unique the Ps first

int n = ps.size();

if (n <= 1)

return ps;

sort(ps.begin(), ps.end());

vector<P> qs(n \* 2);

int k = 0;

for (int i = 0; i < n; qs[k++] = ps[i++])

while (k > 1 && crossOp(qs[k - 2], qs[k - 1], ps[i]) < 0)

--k;

for (int i = n - 2, t = k; i >= 0; qs[k++] = ps[i--])

while (k > t && crossOp(qs[k - 2], qs[k - 1], ps[i]) < 0)

--k;

qs.resize(k - 1);

return qs;

}

//凸包

db convexDiameter(vector<P> ps) //凸包直径

{

int n = ps.size();

if (n <= 1)

return 0;

int is = 0, js = 0;

rep(k, 1, n) is = ps[k] < ps[is] ? k : is, js = ps[js] < ps[k] ? k : js;

int i = is, j = js;

db ret = ps[i].distTo(ps[j]);

do

{

if ((ps[(i + 1) % n] - ps[i]).det(ps[(j + 1) % n] - ps[j]) >= 0)

(++j) %= n;

else

(++i) %= n;

ret = max(ret, ps[i].distTo(ps[j]));

} while (i != is || j != js);

return ret;

}

vector<P> convexCut(const vector<P> &ps, P q1, P q2) //直线切割凸包，返回直线左边凸包的点

{

vector<P> qs;

int n = ps.size();

for (int i = 0; i < n; i++)

{

P p1 = ps[i], p2 = ps[(i + 1) % n];

int d1 = crossOp(q1, q2, p1), d2 = crossOp(q1, q2, p2);

if (d1 >= 0)

qs.push\_back(p1);

if (d1 \* d2 < 0)

qs.pb(isLL(p1, p2, q1, q2));

}

return qs;

}

db min\_dist(vector<P> &ps, int l, int r)

{

if (r - l <= 5)

{

db ret = 1e18;

for(int i=l;i<r;++i)

for(int j=l;j<i;++j)

ret = min(ret, ps[i].distTo(ps[j]));

return ret;

}

int m = (l + r) >> 1;

db ret = min(min\_dist(ps, l, m), min\_dist(ps, m, r));

vector<P> qs;

for(int i=l;i<r;++i)

if (abs(ps[i].x - ps[m].x) <= ret)

qs.push\_back(ps[i]);

sort(qs.begin(), qs.end(), [](P a, P b) -> bool

{ return a.y < b.y; });

for(int i=1;i<qs.size();++i)

for (int j = i - 1; j >= 0 && qs[j].y >= qs[i].y - ret; --j)

ret = min(ret, qs[i].distTo(qs[j]));

return ret;

}

**叉积求多边形面积**

inline int sign(db a) { return a < -EPS ? -1 : a > EPS; }

inline int cmp(db a, db b) { return sign(a - b); }

struct P {

db x, y;

P() {}

P(db \_x, db \_y) : x(\_x), y(\_y) {}

P operator+(P p) { return {x + p.x, y + p.y}; }

P operator-(P p) { return {x - p.x, y - p.y}; }

P operator\*(db d) { return {x \* d, y \* d}; }

P operator/(db d) { return {x / d, y / d}; }

//向量加减乘除

bool operator<(P p) const {

int c = cmp(x, p.x);

if (c)

return c == -1;

return cmp(y, p.y) == -1;

}

bool operator==(P o) const {

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

}

db dot(P p) { return x \* p.x + y \* p.y; } //点积 (二点之间度数a: >0是a<90度,==0度数a=90度,<0度数a>90度) |a|\*|b|\*cos(a)

db det(P p) { return x \* p.y - y \* p.x; } //叉积 (二点之间度数a: >0是0<a<180度,==0度数a=0度||a==180共线,<0度数180<a>360度) |a|\*|b|\*sin(a)

db distTo(P p) { return (\*this - p).abs(); } // 到P点距离

db abs() { return sqrt(abs2()); } // 到原点距离

db abs2() { return x \* x + y \* y; }

int quad() const { return sign(y) == 1 || (sign(y) == 0 && sign(x) >= 0); }

}p[110];

int n;

double area(vector<P> ps) {

db ret = 0;

for(int i=0;i<ps.size();++i)

ret += ps[i].det(ps[(i + 1) % ps.size()]);

return ret / 2.0;

}

int main() {

cin >> n;

vector<P> t;

for (int i = 1; i <= n; i++) {

db x, y;

cin >> x >> y;

t.push\_back(P(x, y));

}

reverse(t.begin(), t.end());

sort(t.begin(), t.end(), [&](P a, P b) {

if (a.quad() != b.quad()) {

return a.quad() > b.quad();

}

return sign(a.det(b)) > 0;

});

cout << fabs(area(t)) << endl;

return 0;

}

**极角序**

struct Node {

double len;

int x, y, id, type;

bool operator< (const Node &t) const {

if (t.type != type) {

return type < t.type;

}

return t.x \* y - t.y \* x > 0; // 叉积

}

};

int T, n, m;

int x[N], y[N], z[N];

vector<Node> a;

double calc(int x, int y) {

int dx = x - 0, dy = y - 0;

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

}

void solve() {

cin >> n >> m;

for (int i = 1; i <= n; i++) {

cin >> x[i] >> y[i];

if (x >= 0 && y >= 0) {

a.push\_back({calc(x[i], y[i]), x[i], y[i], i, 1});

} else if (x >= 0 && y <= 0) {

a.push\_back({calc(x[i], y[i]), x[i], y[i], i, 2});

} else if (x <= 0 && y <= 0) {

a.push\_back({calc(x[i], y[i]), x[i], y[i], i, 3});

} else if (x <= 0 && y >= 0) {

a.push\_back({calc(x[i], y[i]), x[i], y[i], i, 4});

}

}

sort(a.begin(), a.end());

}