多边形相关

平面多边形

两向量构成的平面四边形有向面积

```
template<class T> T areaEx(Point<T> p1, Point<T> p2, Point<T> p3) {
    return cross(b, c, a);
}
```

判断四个点能否组成矩形/正方形

可以处理浮点数、共点的情况。返回分为三种情况：2 代表构成正方形；1 代表构成矩形；0 代表其他情况。

```
template<class T> int isSquare(vector<Pt> x) {
    sort(x.begin(), x.end());
    if (equal(dis(x[0], x[1]), dis(x[2], x[3])) && sign(dis(x[0], x[1])) &&
        equal(dis(x[0], x[2]), dis(x[1], x[3])) && sign(dis(x[0], x[2])) &&
        lineParallel(Lt{x[0], x[1]}, Lt{x[2], x[3]}) &&
        lineParallel(Lt{x[0], x[2]}, Lt{x[1], x[3]}) &&
        lineVertical(Lt{x[0], x[1]}, Lt{x[0], x[2]})) {
        return equal(dis(x[0], x[1]), dis(x[0], x[2])) ? 2 : 1;
    }
    return 0;
}
```

点是否在任意多边形内

射线法判定，t 为穿越次数，当其为奇数时即代表点在多边形内部；返回 2 代表点在多边形边界上。

```
template<class T> int pointInPolygon(Point<T> a, vector<Point<T>> p) {
    int n = p.size();
    for (int i = 0; i < n; i++) {
        if (pointOnSegment(a, Line{p[i], p[(i + 1) % n]})) {
            return 2;
        }
    }
    int t = 0;
    for (int i = 0; i < n; i++) {
        auto u = p[i], v = p[(i + 1) % n];
        if (u.x < a.x && v.x >= a.x && pointOnLineLeft(a, Line{v, u})) {
            t ^= 1;
        }
        if (u.x >= a.x && v.x < a.x && pointOnLineLeft(a, Line{u, v})) {
            t ^= 1;
        }
    }
    return t == 1;
}
```

线段是否在任意多边形内部

```cpp
template<class T>
bool segmentInPolygon(Line<T> l, vector<Point<T>> p) {
// 线段与多边形边界不相交且两端点都在多边形内部
#define L(x, y) pointOnLineLeft(x, y)
    int n = p.size();
    if (!pointInPolygon(l.a, p)) return false;
    if (!pointInPolygon(l.b, p)) return false;
    for (int i = 0; i < n; i++) {
        auto u = p[i];
        auto v = p[(i + 1) % n];
        auto w = p[(i + 2) % n];
        auto [t, p1, p2] = segmentIntersection(l, Line(u, v));
        if (t == 1) return false;
        if (t == 0) continue;
        if (t == 2) {
            if (pointOnSegment(v, l) && v != l.a && v != l.b) {
                if (cross(v - u, w - v) > 0) {
                    return false;
                }
            }
        } else {
            if (p1 != u && p1 != v) {
                if (L(l.a, Line(v, u)) || L(l.b, Line(v, u))) {
                    return false;
                }
            } else if (p1 == v) {
                if (l.a == v) {
                    if (L(u, l)) {
                        if (L(w, l) && L(w, Line(u, v))) {
                            return false;
                        }
                    } else {
                        if (L(w, l) || L(w, Line(u, v))) {
                            return false;
                        }
                    }
                } else if (l.b == v) {
                    if (L(u, Line(l.b, l.a))) {
                        if (L(w, Line(l.b, l.a)) && L(w, Line(u, v))) {
                            return false;
                        }
                    } else {
                        if (L(w, Line(l.b, l.a)) || L(w, Line(u, v))) {
                            return false;
                        }
                    }
                } else {
                    if (L(u, l)) {
                        if (L(w, Line(l.b, l.a)) || L(w, Line(u, v))) {
                            return false;
                        }
```

```
                    } else {
                        if (L(w, l) || L(w, Line(u, v))) {
                            return false;
                        }
                    }
                }
            }
        }
    }
    return true;
}
```

任意多边形的面积

```
template<class T> ld area(vector<Point<T>> P) {
    int n = P.size();
    ld ans = 0;
    for (int i = 0; i < n; i++) {
        ans += cross(P[i], P[(i + 1) % n]);
    }
    return ans / 2.0;
}
```

皮克定理

绘制在方格纸上的多边形面积公式可以表示为 $S = n + \frac{s}{2} - 1$，其中 n 表示多边形内部的点数、s 表示多边形边界上的点数。一条线段上的点数为 $\gcd(|x_1 - x_2|, |y_1 - y_2|) + 1$。

任意多边形上/内的网格点个数（仅能处理整数）

皮克定理用。

```
int onPolygonGrid(vector<Point<int>> p) { // 多边形上
    int n = p.size(), ans = 0;
    for (int i = 0; i < n; i++) {
        auto a = p[i], b = p[(i + 1) % n];
        ans += gcd(abs(a.x - b.x), abs(a.y - b.y));
    }
    return ans;
}
int inPolygonGrid(vector<Point<int>> p) { // 多边形内
    int n = p.size(), ans = 0;
    for (int i = 0; i < n; i++) {
        auto a = p[i], b = p[(i + 1) % n], c = p[(i + 2) % n];
        ans += b.y * (a.x - c.x);
    }
    ans = abs(ans);
    return (ans - onPolygonGrid(p)) / 2 + 1;
}
```

二维凸包

获取二维静态凸包（Andrew算法）

`flag` 用于判定凸包边上的点、重复的顶点是否要加入到凸包中，为 0 时代表加入凸包（不严格）；为 1 时不加入凸包（严格）。时间复杂度为 $\mathcal{O}(N\log N)$ 。

```cpp
template<class T> vector<Point<T>> staticConvexHull(vector<Point<T>> A, int flag = 1) {
    int n = A.size();
    if (n <= 2) { // 特判
        return A;
    }
    vector<Point<T>> ans(n * 2);
    sort(A.begin(), A.end());
    int now = -1;
    for (int i = 0; i < n; i++) { // 维护下凸包
        while (now > 0 && cross(A[i], ans[now], ans[now - 1]) <= 0) {
            now--;
        }
        ans[++now] = A[i];
    }
    int pre = now;
    for (int i = n - 2; i >= 0; i--) { // 维护上凸包
        while (now > pre && cross(A[i], ans[now], ans[now - 1]) <= 0) {
            now--;
        }
        ans[++now] = A[i];
    }
    ans.resize(now);
    return ans;
}
```

二维动态凸包

固定为 `int` 型，需要重新书写 `Line` 函数，`cmp` 用于判定边界情况。可以处理如下两个要求：

- 动态插入点 (x,y) 到当前凸包中；
- 判断点 (x,y) 是否在凸包上或是在内部（包括边界）。

```cpp
template<class T> bool turnRight(Pt a, Pt b) {
    return cross(a, b) < 0 || (cross(a, b) == 0 && dot(a, b) < 0);
}
struct Line {
    static int cmp;
    mutable Point<int> a, b;
    friend bool operator<(Line x, Line y) {
        return cmp ? x.a < y.a : turnRight(x.b, y.b);
    }
    friend auto &operator<<(ostream &os, Line l) {
        return os << "<" << l.a << ", " << l.b << ">";
    }
};
```

```

int Line::cmp = 1;
struct UpperConvexHull : set<Line> {
    bool contains(const Point<int> &p) const {
        auto it = lower_bound({p, 0});
        if (it != end() && it->a == p) return true;
        if (it != begin() && it != end() && cross(prev(it)->b, p - prev(it)->a) <= 0) {
            return true;
        }
        return false;
    }
    void add(const Point<int> &p) {
        if (contains(p)) return;
        auto it = lower_bound({p, 0});
        for (; it != end(); it = erase(it)) {
            if (turnRight(it->a - p, it->b)) {
                break;
            }
        }
        for (; it != begin() && prev(it) != begin(); erase(prev(it))) {
            if (turnRight(prev(prev(it))->b, p - prev(prev(it))->a)) {
                break;
            }
        }
        if (it != begin()) {
            prev(it)->b = p - prev(it)->a;
        }
        if (it == end()) {
            insert({p, {0, -1}});
        } else {
            insert({p, it->a - p});
        }
    }
};
struct ConvexHull {
    UpperConvexHull up, low;
    bool empty() const {
        return up.empty();
    }
    bool contains(const Point<int> &p) const {
        Line::cmp = 1;
        return up.contains(p) && low.contains(-p);
    }
    void add(const Point<int> &p) {
        Line::cmp = 1;
        up.add(p);
        low.add(-p);
    }
    bool isIntersect(int A, int B, int C) const {
        Line::cmp = 0;
        if (empty()) return false;
        Point<int> k = {-B, A};
        if (k.x < 0) k = -k;
```

```
        if (k.x == 0 && k.y < 0) k.y = -k.y;
        Point<int> P = up.upper_bound({{0, 0}, k})->a;
        Point<int> Q = -low.upper_bound({{0, 0}, k})->a;
        return sign(A * P.x + B * P.y - C) * sign(A * Q.x + B * Q.y - C) > 0;
    }
    friend ostream &operator<<(ostream &out, const ConvexHull &ch) {
        for (const auto &line : ch.up) out << "(" << line.a.x << "," << line.a.y << ")";
        cout << "/";
        for (const auto &line : ch.low) out << "(" << -line.a.x << "," << -line.a.y << ")";
        return out;
    }
};
```

点与凸包的位置关系

0 代表点在凸包外面；1 代表在凸壳上；2 代表在凸包内部。

```
template<class T> int contains(Point<T> p, vector<Point<T>> A) {
    int n = A.size();
    bool in = false;
    for (int i = 0; i < n; i++) {
        Point<T> a = A[i] - p, b = A[(i + 1) % n] - p;
        if (a.y > b.y) {
            swap(a, b);
        }
        if (a.y <= 0 && 0 < b.y && cross(a, b) < 0) {
            in = !in;
        }
        if (cross(a, b) == 0 && dot(a, b) <= 0) {
            return 1;
        }
    }
    return in ? 2 : 0;
}
```

闵可夫斯基和

计算两个凸包合成的大凸包。

```
template<class T> vector<Point<T>> mincowski(vector<Point<T>> P1, vector<Point<T>> P2) {
    int n = P1.size(), m = P2.size();
    vector<Point<T>> V1(n), V2(m);
    for (int i = 0; i < n; i++) {
        V1[i] = P1[(i + 1) % n] - P1[i];
    }
    for (int i = 0; i < m; i++) {
        V2[i] = P2[(i + 1) % m] - P2[i];
    }
    vector<Point<T>> ans = {P1.front() + P2.front()};
```

```
    int t = 0, i = 0, j = 0;
    while (i < n && j < m) {
        Point<T> val = sign(cross(v1[i], v2[j])) > 0 ? v1[i++] : v2[j++];
        ans.push_back(ans.back() + val);
    }
    while (i < n) ans.push_back(ans.back() + v1[i++]);
    while (j < m) ans.push_back(ans.back() + v2[j++]);
    return ans;
}
```

半平面交

计算多条直线左边平面部分的交集。

```
template<class T> vector<Point<T>> halfcut(vector<Line<T>> lines) {
    sort(lines.begin(), lines.end(), [&](auto l1, auto l2) {
        auto d1 = l1.b - l1.a;
        auto d2 = l2.b - l2.a;
        if (sign(d1) != sign(d2)) {
            return sign(d1) == 1;
        }
        return cross(d1, d2) > 0;
    });
    deque<Line<T>> ls;
    deque<Point<T>> ps;
    for (auto l : lines) {
        if (ls.empty()) {
            ls.push_back(l);
            continue;
        }
        while (!ps.empty() && !pointOnLineLeft(ps.back(), l)) {
            ps.pop_back();
            ls.pop_back();
        }
        while (!ps.empty() && !pointOnLineLeft(ps[0], l)) {
            ps.pop_front();
            ls.pop_front();
        }
        if (cross(l.b - l.a, ls.back().b - ls.back().a) == 0) {
            if (dot(l.b - l.a, ls.back().b - ls.back().a) > 0) {
                if (!pointOnLineLeft(ls.back().a, l)) {
                    assert(ls.size() == 1);
                    ls[0] = l;
                }
                continue;
            }
            return {};
        }
        ps.push_back(lineIntersection(ls.back(), l));
        ls.push_back(l);
    }
    while (!ps.empty() && !pointOnLineLeft(ps.back(), ls[0])) {
        ps.pop_back();
```

```
        ls.pop_back();
    }
    if (ls.size() <= 2) {
        return {};
    }
    ps.push_back(lineIntersection(ls[0], ls.back()));
    return vector(ps.begin(), ps.end());
}
```