



Al.Cash's blog

Geometry: Polygon algorithms

By **Al.Cash**, 4 years ago,

I decided to share my implementations for the basic polygon algorithms. I see almost no problems on this topic and I hope this will change in the future.

First, let's remind the definitions we will use:

1. **Polygon** is a plane figure that is bounded by a finite chain of straight line segments closing in a loop to form a closed chain or circuit. These segments are called its edges or sides, and the points where two edges meet are the polygon's vertices or corners ([wiki](#)).
2. Polygon is **convex** if a line segment connecting any two points on its boundary lies inside the polygon. Equivalently, all its interior angles are less than or equal to 180 degrees.
3. Polygon is **strictly convex** if in addition no three vertices lie on the same line. Equivalently, all its interior angles are less than 180 degrees.
4. Polygon is **simple** if its boundary doesn't cross itself.

I will present two algorithms for each problem: one for **arbitrary simple polygon**, and one for **strictly convex polygon**, that has better complexity. The priorities in implementation design were as follows:

- handle all the corner cases, **except for degenerate polygons** with zero area;
- perform all computations in integers;
- optimize performance;
- write as concise and clear code as possible.

Please, let me know if you find a way to improve on any of these goals in any algorithm listed. First, some references.

1. The code in this article will make no sense unless you read [my previous one](#). I will use structures and functions defined there.
2. I highly recommend reading [Computational Geometry in C \(2nd Edition\)](#) by Joseph O'Rourke (google for pdf version). This is truly fundamental work on this topic! While the code in there is far from perfect, the ideas are described well in my opinion. I will indicate specific pages when I reference this book.
3. Some algorithms can be found on [geomalgorithms.com](#), but I dislike the implementations. I took only one from there.

I define `Polygon` as a `vector` of `Point`s. Polygon size is denoted as N in the complexity formulas. Since polygon is a chain, the next two functions help in jumping from the last vertex to the first and vice versa when necessary. After writing all the algorithms I realized it would be better to implement custom iterators to do that. Maybe one day...

```
template <class F> using Polygon = vector<Point<F>>;
inline int prev(int i, int n) { return i == 0 ? n-1 : i-1; }
inline int next(int i, int n) { return i == n-1 ? 0 : i+1; }
template <class T> inline int sgn(const T& x) { return (T(0) < x) - (x < T(0)); }
}
```

1. Area: $O(N)$

Probably, everyone knows how to compute polygon area as a sum of triangle or trapezoid areas. If not, you can read [\[geomalgorithms\]](#) or [O'Rourke, p.16] for more detailed explanation. However, as it's shown in `area2D_Polygon` on *geomalgorithms*, this computation can be optimized, resulting in the following code:

```
template <class F>
F area(const Polygon<F>& poly) {
    int n = static_cast<int>(poly.size());
    F area = F(0);
    for (int i = 0; i < n; ++i)
        area += poly[i].x * (poly[next(i, n)].y - poly[prev(i, n)].y);
    return area;
}
```

→ **Pay attention**

Before contest


Codeforces Round #650 (Div. 3).

08:01:50


[Register now »](#)

Like 74 people like this. Be the first of your friends.

→ **nemish1999**



Rating: **956**
Contribution: **0**



nemish1999

- [Settings](#)
- [Blog](#)
- [Teams](#)
- [Submissions](#)
- [Favourites](#)
- [Talks](#)
- [Contests](#)

→ **Top rated**

#	User	Rating
1	MiFaFaOvO	3681
2	Um_nik	3567
3	tourist	3520
4	maroonrk	3421
5	apiadu	3397
6	300iq	3317
7	ecnerwala	3309
8	Benq	3283
9	LHiC	3229
10	TLE	3223

<u>Countries</u>	<u>Cities</u>	<u>Organizations</u>
Australia	Brisbane	University of Queensland
Canada	Ottawa	University of Ottawa
France	Paris	University of Paris
Germany	Munich	Ludwig-Maximilians-Universität München
Greece	Athens	National Technical University of Athens
India	New Delhi	Indian Institute of Technology
Italy	Rome	Sapienza University of Rome
Japan	Tokyo	The University of Tokyo
South Korea	Seoul	Korea Advanced Institute of Science and Technology
Spain	Madrid	Complutense University of Madrid
Sweden	Stockholm	Uppsala University
Switzerland	Zürich	ETH Zurich
Taiwan	Taipei	National Taiwan University
United Kingdom	London	Imperial College London
USA	Washington D.C.	Georgetown University

[View all →](#)

→ **Top contributors**

#	User	Contrib.
1	Errichto	194
1	antontrygubO_o	194
3	pikmike	180
4	vovuh	177
5	Ashishgup	170
6	Radewoosh	169
7	Um_nik	166
7	tourist	166
9	ko_osaga	163
10	McDic	162

[View all →](#)

→ **Favourite groups**

#	Name
1	<u>Egyptian Olympiad in Informatics (EOI) - Open Contests Archive</u>
2	<u>JCPC Training 2018 - Level 1</u>
3	<u>Join to win ACM ICPC 2019</u>
4	<u>JCPC Training 2018 - Level 3</u>
5	<u>JCPC Training 2018 - Level 2</u>
6	<u>JCPC Training 2018 - Level 4</u>
7	<u>TAP (Argentinian programming tournament) contests</u>
8	<u>UW-Madison Competitive Programmers</u>
9	<u>Camp de Varzea do ICMC</u>

Note that this function returns **doubled oriented area**. Doubled, because for any polygon with integer coordinated the area may be not integer but half-integer. Oriented means that it's positive if polygon vertices are listed in counter-clockwise (ccw) order and negative otherwise.

2. Orientation

Polygon vertices are given in either counter-clockwise or clockwise order. **All the algorithms below assume that orientation is counter-clockwise.** Because of that, we need a way to verify this condition. Common solution is to check the sign of `area` return value. However, it's not necessary to compute the area.

2.1 Simple polygon: $O(N)$

As noted in [O'Rourke, p.12], it's enough to check orientation at one of extreme polygon vertices, for example lower-left. Complexity remains the same, but operations are faster: comparisons instead of multiplication and summation.

```
// True if orientation of a simple polygon is counter-clockwise.
template <class F>
bool orientation(const Polygon<F>& poly) {
    int n = static_cast<int>(poly.size());
    int i = static_cast<int>(min_element(begin(poly), end(poly)) - begin(poly));
    return ccw(poly[prev(i, n)], poly[next(i, n)], poly[i]) > 0;
}
```

2.2 Convex polygon: $O(1)$

All vertices of the convex polygon are extreme, so we can check orientation at an arbitrary vertex.

3. Convex hull: $O(N\log N)$

We will be using convex polygons, so first let's learn to build one from a set of points. There are various algorithms with the same complexity, so I chose the one with the smallest constant: Andrew's monotone chain algorithm [geomalgorithms]. It's faster, because points are sorted lexicographically, not using more complex angle comparison as in `sortByAngle` in my previous article. The implementation was taken from [here](#). The difference is that I build right and left chains, not upper and lower. Also, I handle the degenerate case when all the points coincide. In such case the original code outputs 2 points instead of one.

```
template <class F>
Polygon<F> convexHull(Polygon<F> points) {
    sort(begin(points), end(points));
    Polygon<F> hull;
    hull.reserve(points.size() + 1);
    for (int phase = 0; phase < 2; ++phase) {
        auto start = hull.size();
        for (auto& point : points) {
            while (hull.size() >= start+2 &&
                ccw(point, hull.back(), hull[hull.size()-2]) <= 0)
                hull.pop_back();
            hull.push_back(point);
        }
        hull.pop_back();
        reverse(begin(points), end(points));
    }
    if (hull.size() == 2 && hull[0] == hull[1]) hull.pop_back();
    return hull;
}
```

4. Inclusion tests

Given a point, sometimes we need to check whether it lies inside or outside the given polygon. All the functions below return

- -1 if the point is strictly inside the polygon;
- 0 if the point is on the boundary;
- 1 if the point is outside the polygon.

4.1 Triangle: $O(1)$

The easiest example of a polygon is triangle. It's an important object in computer graphics and deserves special treating :) All the cases are visualized in [O'Rourke, p.235], from where I derived the following conditions:

- point is inside if all the signs are equal;
- point is outside if there are two opposite signs;
- otherwise point is on the boundary (two signs are equal and the third one is 0).

10	AAST Cairo individuals 18"
11	ALCPC Training
12	Aktau228
13	Tubitak Yaz Kampi 2018
14	OEIS substitute choosing
15	sysu team7
16	UTN FRSF Training
17	FE-CU Training 2019
18	AUC Collegiate Programming Contest 2018
19	IITU
20	U of T Practice
21	INSAT ACM SC external contests group
22	KBO-1 7 grade
23	KBO-1 8-9 grades
24	KBO-1 10-11 grades
View all →	

→ Find user

Handle:

Find

→ Recent actions

sidhant → [Tutorial on FFT/NTT — The tough made simple. \(Part 1 \).](#) 🔒

Stepavly → [Codeforces Round #650 \(Div. 3\)](#) 🔒

evima → [Codeforces Round #286 Editorial \(Complete\)](#) 🔒

Una_Shem → [About Gift Distribution](#) 🔒

Nickolas → [Announcement: Microsoft Q# Coding Contest – Summer 2020](#) 🔒

jjang36524 → [Thinking about a contest](#) 🔒

vovuh → [Codeforces Round #634 \(Div. 3\) Editorial](#) 🔒

darren_yao → [An Introduction to the USA Computing Olympiad -- Book Released](#) 🔒

I_love_tigersugar → [Codeforces Round #601 Editorial](#) 🔒

dreamoon_love_AA → [Codeforces Round #631 Editorial](#) 🔒

vovuh → [Codeforces Round #544 \(Div. 3\)](#) 🔒

pikmike → [Educational Codeforces Round 89 Editorial](#) 🔒

gepardo → [Tutorial of Codeforces Round #379 \(Div. 2\)](#) 🔒

pllk → [Guide to Competitive Programming and CSES Problem Set](#) 🔒

Jellyman102 → [An Efficient \(and quite common\) Way to Navigate Grid Problems \[C++\]](#) 🔒

mohammedehab2002 → [Codeforces round #649 editorial](#) 🔒

SecondThread → [Algorithms Dead Episode 1: Division Under Mod!](#) 🔒

R2-D2_hates_lemon → [Did Codeforces change the formula for rating system?](#) 🔒

Nickolas → [Microsoft Q# Coding Contest – Summer 2020 — Warmup: editorial](#) 🔒

dolphingarlic → [\[Tutorial\] Path sum queries on a tree using a Fenwick tree](#) 🔒

Cache → [Time Complexity of lower_bound](#) 🔒

cjchirag7 → [Editorial of Virtual Farewell IIT \(ISM\) Dhanbad](#) 🔒

EbTech → [How to Compete in Rust](#) 🔒

misis → [Registration for Volga Summer Camp 2020 is opened](#) 🔒

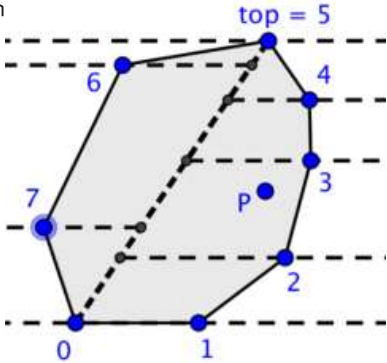
chokudai → [AtCoder Beginner Contest 170 Announcement](#) 🔒

Detailed →

```
template <class F1, class F2>
int pointVsTriangle(const Point<F1>& point, const Polygon<F2>& triangle) {
    assert(triangle.size() == 3);
    int signs[3];
    for (int i = 0; i < 3; ++i)
        signs[i] = sgn(ccw(point, triangle[next(i, 3)], triangle[i]));
    if (signs[0] == signs[1] && signs[1] == signs[2]) return -1;
    for (int i = 0; i < 3; ++i) if (signs[i] * signs[next(i, 3)] == -1) return 1;
    return 0;
}
```

4.2 Convex polygon: $O(\log N)$

The popular approach here is to triangulate the polygon by drawing diagonals from one vertex to all the others, find the angle where the given point lies using binary search, and then check if it's inside the triangle or not. However, I got the best time on the problem 166B - Многоугольники using the same approach as in convex hull construction (and I'm not the only one using fast I/O there). Also, I have less special cases to check (even if we don't need to distinguish the border from the interior).



If the point is below (or on the same horizontal line but to the right of) the lower-left vertex, it's definitely outside, similarly for upper-right. Now, let's connect the lower-left and the upper-right polygon vertices. If the given point lies on this line, it's either on the polygon boundary (if it coincides with one of the endpoints or this line is a polygon edge) or inside. If the point is to the right (analogously to the left), we need to check if it's to the left of the right chain built by the convex hull algorithm. The corresponding edge is found using binary search, again comparing points lexicographically.

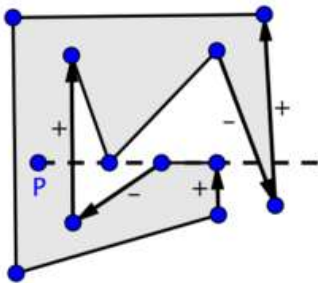
Current implementation assumes that lower-left index has index 0, and we know the index of upper-right vertex (called `top`, it can be found once in linear time). This assumption is perfectly reasonable, because it fits the output of convex hull algorithm. Picture shows points locations that will be compared with each polygon edge. For example, point *P* will be compared with edge 2-3, and it's inside the polygon because it's to the left of this edge.

```
template <class F1, class F2>
int pointVsConvexPolygon(const Point<F1>& point, const Polygon<F2>& poly, int
top) {
    if (point < poly[0] || point > poly[top]) return 1;
    auto orientation = ccw(point, poly[top], poly[0]);
    if (orientation == 0) {
        if (point == poly[0] || point == poly[top]) return 0;
        return top == 1 || top + 1 == poly.size() ? 0 : -1;
    } else if (orientation < 0) {
        auto itRight = lower_bound(begin(poly) + 1, begin(poly) + top, point);
        return sgn(ccw(itRight[0], point, itRight[-1]));
    } else {
        auto itLeft = upper_bound(poly.rbegin(), poly.rend() - top-1, point);
        return sgn(ccw(itLeft == poly.rbegin() ? poly[0] : itLeft[-1], point,
itLeft[0]));
    }
}
```

Exercise: verify that this code works correctly when horizontal edges are present.

4.3 Simple polygon: $O(N)$

We will use the winding number described for example on [geomalgorithms], the same number is the answer for the problem Timus 1599. It's defined as a number of full counter-clockwise turns around the given point we make if we follow the polygon edges. In case of a simple polygon it's either 1 when the point is inside or 0 if outside.



Winding number can be computed in the following way. We draw an arbitrary ray from the given point and each time the polygon edge crosses it we add either +1 if the crossing was from the right to the left side and -1 otherwise. Such edges are marked with arrows on the picture. For optimization purposes I chose the horizontal ray going in increasing *x* direction. The ray itself is included in the upper half-plane. For a horizontal edge, I check whether it contains the given point. If the edge endpoints are on the different sides of the line containing the ray, I check if the edge crosses this line to the right side of the given point.

```
template <class F1, class F2>
int pointVsPolygon(const Point<F1>& point, const Polygon<F2>& poly) {
    int n = static_cast<int>(poly.size()), windingNumber = 0;
    for (int i = 0; i < n; ++i) {
        if (point == poly[i]) return 0;
        int j = next(i, n);
        if (poly[i].y == point.y && poly[j].y == point.y) {
            if (min(poly[i].x, poly[j].x) <= point.x &&
                point.x <= max(poly[i].x, poly[j].x)) return 0;
        } else {
            bool below = poly[i].y < point.y;
            if (below != (poly[j].y < point.y)) {
                auto orientation = ccw(point, poly[j], poly[i]);
                if (orientation == 0) return 0;
                if (below == (orientation > 0)) windingNumber += below ? 1 : -1;
            }
        }
    }
    return windingNumber == 0 ? 1 : -1;
}
```

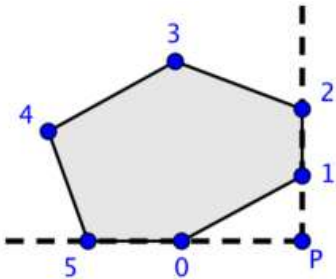
Exercise: think of an example when the `point == poly[i]` check is necessary.

5. Extreme points

I define an extreme vertex with respect to the given direction as following: if we draw a line through this vertex in the given direction, there are no polygon vertices to the right of it. This direction can be fixed for all vertices, or it can be for example direction from a given point to the vertex. In the latter case extreme vertex is the right tangent vertex (for example vertices 1 and 2 for point *P* in the picture).

5.1 Convex polygon: $O(\log N)$

The binary search algorithm was described in [O'Rourke, p.270] and implemented in functions `[polyMax_2D]` and `[tangent_PointPolyC]`. My contribution here is writing one general function instead of three similar ones. I packed all the cases handling into one `if` statement. At least for me, it's easier to understand and remember. Also, I believe the cases when direction coincides with polygon side (as in the picture) aren't handled correctly. These cases are (in O'Rourke notation):



```
if C points neither up nor down
    if A points up
        then [a, b] <- [a, c]
        else [a, b] <- [c, b]
if A points neither up nor down
    if C points up
        then [a, b] <- [c, b]
        else [a, b] <- [a, c]
```

There are at most 2 vertices where this can happen (1 and 5 in the picture) and one of them is always extreme (number 1). This implies *A* and *C* can't both point neither up nor down at the same time.

Now I'll explain my implementation a bit. `vertexCmp` returns -1 if vertex *i* is to the right of vertex *j* with respect to the given direction. `isExtreme` just follows my definition, but in case when there are two extreme vertices on the same line, it returns `true` only for the first in counter-clockwise order (`true` for 1 and 5, `false` for 2 and 0 in the picture). Once extreme vertex is encountered, it's immediately returned. For the algorithm to work correctly, the binary search isn't executed in case vertex 0 is extreme.

```
template <class F, class Function>
int extremeVertex(const Polygon<F>& poly, Function direction) {
    int n = static_cast<int>(poly.size()), left = 0, leftSgn;
    auto vertexCmp = [&poly, direction](int i, int j) {
        return sgn(ccw(direction(poly[j]), poly[j] - poly[i]));
    };
    auto isExtreme = [n, vertexCmp](int i, int& isgn) {
        return (isgn = vertexCmp(next(i, n), i)) >= 0 && vertexCmp(i, prev(i, n)) < 0;
    };
    for (int right = isExtreme(0, leftSgn) ? 1 : n; left + 1 < right; ) {
        int middle = (left + right) / 2, middleSgn;
        if (isExtreme(middle, middleSgn)) return middle;
        if (leftSgn != middleSgn ? leftSgn < middleSgn
            : leftSgn == vertexCmp(left, middle)) right = middle;
        else left = middle, leftSgn = middleSgn;
    }
}
```



```
    return left;
}
```

Probably, you have a question regarding `direction` function. Below is the code showing how this function looks like to find the right and left tangents from the given point. Fixed direction will be demonstrated in Section 6.1.

```
template <class F1, class F2>
pair<int, int> tangentsConvex(const Point<F1>& point, const Polygon<F2>& poly)
{
    return {
        extremeVertex(poly, [&point](const Point<F2>& q) { return q - point; }),
        extremeVertex(poly, [&point](const Point<F2>& q) { return point - q; })};
}
```

5.2 Simple polygon

Extreme points always lie on the convex hull. If you're processing multiple queries for such points, it's worthwhile to build the convex hull for the given points beforehand and apply the algorithm from Section 5.1. Convex hull can be built using algorithm from Section 3, but there's [more efficient solution](#) for simple polygons I have yet to analyze.

6. Polygon stabbing

The problem of finding the intersection of a geometric object with a line is often called the "stabbing" problem.

6.1 Convex polygon: $O(\log N)$

OK, here's the place where I got lazy. I just implemented an idea expressed in [O'Rourke, p.271] and didn't test it well. The result is either empty or a segment. First, I find two extreme points with respect to the line direction, and then I find intersection segment endpoints on two chains between them using binary search. Care must be taken, because one of these chains passes through 0 vertex. After the correct edge is found, I call the `intersectLines` function from my previous article. I believe `extremeVertex` can be modified to find each intersection point in one binary search, but I'll leave this idea for the future.

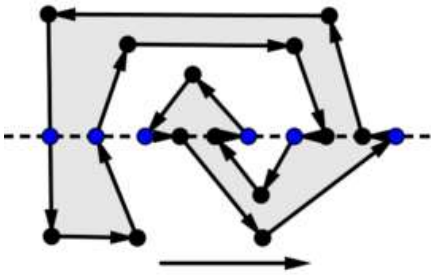
```
template <class F1, class F2, class F3>
bool stabConvexPolygon(const Line<F1>& line, const Polygon<F2>& poly, Line<F3>&
res) {
    assert(line);
    int right = extremeVertex(poly, [&line](const Point<F2>&) { return line.ab;
});
    int left = extremeVertex(poly, [&line](const Point<F2>&) { return -line.ab;
});
    auto vertexCmp = [&line](const Point<F2>& vertex) {
        return sgn(ccw(line.ab, vertex - line.a)); };
    int rightSgn = vertexCmp(poly[right]), leftSgn = vertexCmp(poly[left]);
    if (rightSgn < 0 || leftSgn > 0) return false;
    auto intersectChain = [&line, &poly, vertexCmp](int first, int last,
                                                    int firstSgn, Point<F3>& res)
    {
        int n = static_cast<int>(poly.size());
        while (next(first, n) != last) {
            int middle = (first + last + (first < last ? 0 : n)) / 2;
            if (middle >= n) middle -= n;
            if (vertexCmp(poly[middle]) == firstSgn) first = middle;
            else last = middle;
        }
        intersectLines<0, 0, 0, 0>(line, makeLine(poly[first], poly[last]), res);
    };
    intersectChain(left, right, leftSgn, res.a);
    intersectChain(right, left, rightSgn, res.ab);
    res.ab -= res.a;
    return true;
}
```

6.2 Simple polygon: $O(N)$

The problem of finding the intersection length was given at the first educational round [598F - Длина разреза](#). Surprisingly, I didn't notice a single comment mentioning that it can be solved in linear time (per line). Recall that the line is represented in parametric form $A + tAB$, then the intersection is the set of segments for the value of t . It's enough to find the total length of these segments and multiply it by $|AB|$. The solution is based on the simple fact: when traversing the polygon in counter-clockwise direction, every time we cross the line from right to left we get an end of some t segment, so this value is added to the answer, and from left to right we get a beginning, so it's subtracted from the answer. We only sort this values if we're

asked to produce all the intersection segments, not just their total length.

The problem isn't easy because some vertices and edges may lie on the line, for a total of 6 cases illustrated in the picture. Points we're interested at are marked blue. The general idea to process these cases is the following. If we're crossing from right to left, we need the furthest point in the line direction. Suppose we're processing a vertex that lies on the line, should we include it in the answer? It's not the furthest only in case one of its adjacent edges lies on the line, and the second vertex is further. This condition is checked using dot product. To determine the direction in which we're moving, it's enough to know on which side of the line previous and next vertices lie (variables `prevSgn` and `nextSgn` take values 1 on the left, 0 on the line, -1 on the right). For example, if `prevSgn == 0 && nextSgn == 1`, we're moving from right to left.



I'm not going to go through all the cases, but I'll explain how the values of t are computed. If an edge crosses the line not at a vertex, the formula from `intersectLines` applies. If vertex V lies on the line, we have $V = A + tAB$, and taking dot product of each side with AB results in $(V - A) * AB = t|AB|^2$. This is the only place in this article where I needed non-integer computations.

```
template <class F1, class F2, class F = distF<F1, F2>>
F stabPolygonLength(const Line<F1>& line, const Polygon<F2>& poly) {
    assert(line);
    F tSum = F(0);
    int n = static_cast<int>(poly.size());
    auto vertexSgn = [&line, &poly](int i) { return sgn(line.ab ^ (poly[i] -
line.a)); };
    int prevSgn = vertexSgn(n - 1), iSgn = vertexSgn(0), nextSgn;
    for (int i = 0; i < n; ++i, prevSgn = iSgn, iSgn = nextSgn) {
        nextSgn = vertexSgn(next(i, n));
        if (iSgn == 0) {
            if (prevSgn == 0) {
                if (nextSgn != 0 && nextSgn == sgn((poly[i] - poly[prev(i, n)]) *
line.ab))
                    tSum += nextSgn * static_cast<F>((poly[i] - line.a) * line.ab) /
abs(line.ab);
            } else {
                if ((nextSgn != 0 && nextSgn != prevSgn) ||
                    (nextSgn == 0 && prevSgn == sgn((poly[next(i, n)] - poly[i]) *
line.ab)))
                    tSum -= prevSgn * static_cast<F>((poly[i] - line.a) * line.ab) /
abs(line.ab);
            }
        } else if (nextSgn == -iSgn) {
            auto vect = poly[next(i, n)] - poly[i];
            tSum += nextSgn * static_cast<F>((poly[i] - line.a) ^ vect) / (line.ab ^
vect);
        }
    }
    return tSum * norm(line.ab);
}
```

7. Maximum distance (convex polygon): $O(N)$

As a final chord, I decided to include my implementation for the maximum distance (squared) between two vertices of the convex polygon. It uses the well known rotating calipers technique. If the input is not a convex polygon, call `convexHull` from Section 3 first.

```
template <class F>
F maxDist2(const Polygon<F>& poly) {
    int n = static_cast<int>(poly.size());
    F res = F(0);
    for (int i = 0, j = n < 2 ? 0 : 1; i < j; ++i)
        for (; j = next(j, n)) {
            res = max(res, dist2(poly[i], poly[j]));
            if (ccw(poly[i+1] - poly[i], poly[next(j, n)] - poly[j]) >= 0) break;
        }
    return res;
}
```

🔗 geometry, convex-polygon



Comments (13)

[Write comment?](#)

4 years ago, # | ☆

▲ +20 ▼

Regarding the maximum distance in convex polygon, what about the same algorithm with comparing distances between points only? Can anybody prove its correctness? I once tried to break it with stress testing and I failed. I also got AC on various problems online (I hoped to get WA though).



Errichto

```
int j = 0;
for(int i = 0; i < n; ++i) {
    if(j == i) j = next(i);
    while(dist(poly[i], poly[next(j)]) > dist(poly[i], poly[j]))
        j = next(j);
    res = max(res, dist(poly[i], poly[j]));
}
```

→ [Reply](#)

4 years ago, # ^ | ☆

▲ +70 ▼

Seems like it's hard to find counterexamples with random test cases, as they should satisfy many conditions to break this algorithm, and probability is almost zero for that to happen. But if you write those conditions, and play with them on paper / or in some application, it shouldn't be hard to find some.



DuX

Here is one:

```
(-0.9846, -1.53251)
(0.49946, 1.19525)
(0.79916, 0.98291)
(4.02136, -1.57843)
(3.92734, -2.37856)
(3.88558, -2.37188)
```

You can see it in geogebra: <https://ggbm.at/m2mYKEZe>

→ [Reply](#)

4 years ago, # ^ | ☆

▲ +26 ▼



Errichto

Awesome! It's indeed a convex polygon and my algorithm says 4.98 instead of 5.01. I once also tried to find a counter-test on paper but I didn't succeed. I hope I will manage to expand your test to bigger one, to break slow solutions too.

Thank you very much.

→ [Reply](#)



SHAMPINION

9 months ago, # ^ | ☆

▲ +18 ▼

Hi, have you already managed to find out how to make big tests?

→ [Reply](#)

4 years ago, # | ☆

← Rev. 3 ▲ +15 ▼

Overall a pretty good guide, thanks!!

I'm just confused regarding something, in your other post you have that ccw(l,r) returns >0 if CW, <0 if CCW or 0 if colinear.



Diego1149

In the method pointVsPolygon you have this line: auto orientation = ccw(point, poly[j], poly[i]);

But I'm confused, because if point=(0,0), poly[i]=(1,-5) and poly[j]=(1,5) then below = 1 and ccw(point, poly[j], poly[i]) = 10

But then you have: if (below == (orientation < 0)) windingNumber += below ? 1 : -1; but since below=1 and orientation is not lesser than 0, it evaluates to false, but clearly the line is perpendicular to the ray... Am I missing something here, or misunderstood the algorithm?? Or did you mean !=, or ccw(point, poly[i], poly[j])??

→ [Reply](#)



Al.Cash

4 years ago, # ^ | ☆

▲ +16 ▼

Fixed, thank you!

This mistake doesn't affect the returning value, so I didn't notice it when refactoring the code.

→ [Reply](#)



Diego1149

4 years ago, # ^ | ☆ 0

Now I see, it doesn't affect the overall output of the function because you were sending a ray to the left.
→ Reply



Fighter.human

4 years ago, # | ☆ 0

Thanks :) Learned a lot. Hope you will write more in near future.
→ Reply



bhayanak

3 years ago, # | ☆ 0

how to solve this ...?it seems difficult.
<https://www.codechef.com/COOK83/problems/ADADET>
→ Reply

3 years ago, # | ☆ 0

I have a question regarding your definition on what a extreme point is and how it behaves in the code.

For the polygon stabbing function, you find both the right and left extreme points. Let's say we have the following polygon (I have labeled the vertices starting from 0 in counterclockwise order) and the following line:



Since the line vector is horizontal pointing in the x-axis positive direction, I would expect the extreme point in this direction to be point at index 2 (coordinates: (7,3)), as there is no vertex in the polygon further to the right (or as they define it equivalently in geomalgorithms, no other point in the polygon protected onto the line is further in the direction of the line vector). For the same vector in the opposite direction, I would expect the extreme point to be point at index 0 (coordinates: (1,1)).



ivanzuki

I tried running such example using the polygon stabbing function presented here but I get index 3 for extreme point when the line vector is pointing in the x-axis positive direction, and index 0 when the line vector is pointing in the x-axis negative direction.

Could you enlighten me on what I'm misunderstanding?

Also, could we use dot product instead of cross product to check whether some vertex i is above some vertex j? (particularly in the case when the direction is fixed)
→ Reply



mathusalen

2 years ago, # | ☆ 0

Thanks, very nice tutorial. I think there may be a typo on the area function. For polygons that are ccw oriented it provides negative area. Check with the triangle (0,0) (1,0) (0, 1).
→ Reply

2 years ago, # | ☆ ← Rev. 3 0



sleepify

As per the expression for area as given in Geomalgorithms, I think, next — prev is the correct way.

```
area += poly[i].x * (poly[prev(i, n)].y - poly[next(i, n)].y); //
area < 0 for ccw
area += poly[i].x * (poly[next(i, n)].y - poly[prev(i, n)].y); //
area > 0 for ccw
→ Reply
```



xmyqsh

14 months ago, # | ☆ -9

How about this implementation?

```
#include <iostream>
#include <vector>
#include <map>
#include <list>
#include <cmath>
using namespace std;

float distToLine(const pair<float, float>& p1, const pair<float, float>& p2, const pair<float, float>& p) {
    return (p2.first - p1.first) * (p.second - p1.second) -
           (p2.second - p1.second) * (p.first - p1.first);
}
```

void convexHull(list<pair<float, float>>& ps, const pair<float, float>& p)


```

void convexHull(list<pair<float, float>>& ps, const pair<float,
float>& p1, const pair<float, float>& p2,
const pair<float,
float>& p, bool side, bool clockwise) {
    if (ps.empty()) { ps.emplace_back(p); return; }
    if (side) {
        list<pair<float, float>> ano;
        auto maxIter1 = ps.end(), maxIter2 = ps.end(); float maxDist1
= 0, maxDist2 = 0, dist;
        for (auto it = ps.begin(); it != ps.end(); ) {
            if ((dist = distToLine(p1, p, *it)) > 0) {
                if (dist > maxDist1) { maxIter1 = it; maxDist1 =
dist; }

                ++it;
            } else if ((dist = distToLine(p, p2, *it)) > 0) {
                ano.emplace_back(*it);
                it = ps.erase(it);
                if (dist > maxDist2) { maxIter2 = prev(ano.end());
maxDist2 = dist; }
            } else it = ps.erase(it);
        }
        if (maxIter1 != ps.end()) { auto p0 = *maxIter1;
ps.erase(maxIter1); convexHull(ps, p1, p, p0, true, clockwise); }
        if (maxIter2 != ps.end()) { auto p0 = *maxIter2;
ano.erase(maxIter2); convexHull(ano, p, p2, p0, true, clockwise); }
        if (clockwise) {
            ps.emplace_back(p); ps.splice(ps.end(), ano);
        } else {
            ano.emplace_back(p); ano.splice(ano.end(), ps);
            swap(ano, ps);
        }
    } else {
        list<pair<float, float>> ano;
        auto minIter = ps.end(), maxIter = ps.end(); float minDist =
0, maxDist = 0;
        for (auto it = ps.begin(); it != ps.end(); ) {
            float dist = distToLine(p1, p2, *it);
            if (dist < 0) {
                ano.emplace_back(*it);
                it = ps.erase(it);
                if (dist < minDist) { minIter = prev(ano.end());
minDist = dist; }
            } else if (dist > 0) {
                if (dist > maxDist) { maxIter = it; maxDist = dist; }
                ++it;
            } else it = ps.erase(it);
        }
        if (maxIter != ps.end()) { auto p = *maxIter;
ps.erase(maxIter); convexHull(ps, p1, p2, p, true, clockwise); }
        if (minIter != ps.end()) { auto p = *minIter;
ano.erase(minIter); convexHull(ano, p2, p1, p, true, clockwise); }
        if (clockwise) {
            ps.emplace_front(p1); ps.emplace_back(p2);
ps.splice(ps.end(), ano);
        } else {
            ano.emplace_front(p1); ano.emplace_back(p2);
ano.splice(ano.end(), ps);
            swap(ano, ps);
        }
    }
}

int main() {
    //code
    int T, N; float x, y; cin >> T;
    while (T--) {
        cin >> N; list<pair<float, float>> ps;
        cin >> x >> y; ps.emplace_back(make_pair(x, y));
        auto minP = ps.begin(), maxP = ps.begin();
        for (int i = 1; i != N; ++i) {
            cin >> x >> y; ps.emplace_back(make_pair(x, y));
            if (ps.back().first < minP->first ||
(fabs(ps.back().first - minP->first) < 1e-8 &&
ps.back().second > minP->second)) minP = prev(ps.end());
            if (ps.back().first > maxP->first ||
(fabs(ps.back().first - maxP->first) < 1e-8 &&
ps.back().second > maxP->second)) maxP = prev(ps.end());
        }
        auto p1 = *minP, p2 = *maxP;
        if (p1 == p2) { cout << -1 << endl; continue; }
        ps.erase(minP); ps.erase(maxP);
        convexHull(ps, p1, p2, p, true, clockwise);
    }
}

```

```
        ps.erase(minn), ps.erase(maxx),
    bool clockwise = false;
        convexHull(ps, p1, p2, p2, false, clockwise);
    if (ps.size() < 3) cout << -1 << endl;
    else {
        for (auto it = ps.begin(); it != prev(ps.end()); ++it)
            cout << it->first << ' ' << it->second << ", ";
        cout << ps.back().first << ' ' << ps.back().second << endl;
    }
    }
    return 0;
}
```

→ [Reply](#).

Supported by

