Chapter 14

Computational Geometry

Summary of Chapter 14- from the book
Programming Challenges: The Programming Contest Training Manual
By: Steven S. Skiena , and Miguel A. Revilla
2003 Springer-Verlag New York, Inc.
ISBN: 0-387-00163-8

Computational Geometry:

Geometric computing has been become increasingly important in applications such as computer graphics, robotics, and computer-aided design, because shape is an inherent property of real objects. But most real-world objects are not made of lines which go to infinity. Instead, most computer programs represent geometry as arrangements of line segments. Arbitrary closed curves or shapes can be represented by ordered collections of line segments or *polygons*.

Computational geometry can be defined (for our purposes) as the geometry of discrete line segments and polygons. It is a fun and interesting subject, but one not typically taught in required college courses. This gives the ambitious student who learns a little computational geometry a leg up on the competition, and a window into a fascinating area of algorithms still under active research today.

Line Segments and Intersection:

A line segment s is the portion of a line l which lies between two given points inclusive. Thus line segments are most naturally represented by pairs of endpoints.

The most important geometric primitive on segments, testing whether a given pair of them intersect, proves surprisingly complicated because of tricky special cases that arise. Two segments may lie on parallel lines, meaning they do not intersect at all. One segment may intersect at another's endpoint, or the two segments may lie on top of each other so they intersect in a segment instead of a single point.

This problem of geometric special cases, or degeneracy, seriously complicates the problem of building robust implementations of computational geometry algorithms. Degeneracy can be a real pain in the neck to deal with. Read any problem specification carefully to see if it promises no parallel lines or overlapping segments. Without such guarantees, however, you had better program defensively and deal with them.

Polygons and Angle Computations:

Polygons are closed chains of non-intersecting line segments. That they are closed means the first vertex of the chain is the same as the last. That they are non-intersecting means that pairs of segments meet only at endpoints.

Polygons are the basic structure to describe shapes in the plane. Instead of explicitly listing the segments (or edges) of polygon, we can implicitly represent them by listing the n vertices $v_0, v_1, \dots, v_{n-1}$ in order around the boundary of the polygon. Thus a segment exists between the i^{th} and $(i + 1)^{st}$ points in the chain for $0 \leq i \leq n - 1$. These indices are taken mod n to ensure there is an edge between the first and last point, i.e.: v_0 follow v_{n-1}, since $(n-1) + 1 \equiv n \equiv 0 \pmod{n}$.

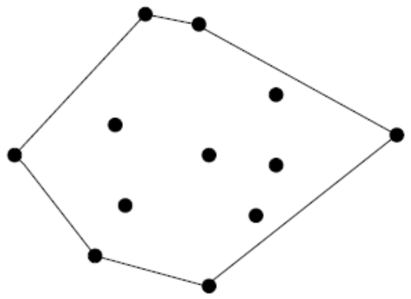
A polygon P is convex if any line segment defined by two points within P lies entirely within P; i.e., there are no notches or bumps such that the segment can exit and reenter P. This implies that all internal angles in a convex polygon must be acute; i.e., at most 180o or π radians.

Actually computing the angle defined between three ordered points is a tricky problem. We can avoid the need to know actual angles in most geometric algorithms by using the *counterclockwise predicate ccw*(a, b, c). This routine tests whether point c lies to the *left* of the directed line which goes from point a to point b. If so, the angle formed by sweeping from a to c in a counterclockwise manner around b is acute, hence the name of the predicate. If not, the point either lies to the *right* of $\overrightarrow{ab}$ or the three points are collinear.

These predicates can be computed using the *signed_triangle_area*( ) formula. Negative area results if point c is to the *right* of $\overrightarrow{ab}$. Zero area results if all three points are collinear. For robustness in the face of floating point errors, we compare it to a tiny constant ε instead of zero.

The above stated functions exist in the files "*geometry.h*" and "*geometry.cpp*" included in chapter 13.

Convex Hulls:
Convex hull is to computational geometry what sorting is to other algorithmic problems, a first step to apply to unstructured data so we can do more interesting things with it. The *convex hull C*(*S*) of a set of points *S* is the smallest convex polygon containing *S*, as shown in the following figure:



There are almost as many different algorithms for convex hull as there are for sorting. The Graham's scan algorithm for convex hull which we will implement first sorts the points in either angular or left-right order, and then incrementally inserts the points into the hull in this sorted order. Previous hull points rendered obsolete by the last insertion are then deleted.
Our implementation is based on the Gries and Stojmenovi´c version of Graham scan, which sorts the vertices by *angle* around the leftmost-lowest point. Observe that both the leftmost and lowest points *must* lie on the hull, because they cannot lie within some other triangle of points. We use the second criteria to break ties for the first, since there might be many different but equally leftmost points. Such considerations are necessary to achieve robustness with degenerate input.
The main loop of the algorithm inserts the points in increasing angular order around this initial point. Because of this ordering, the newly inserted point must sit on the hull of the thus-far-inserted points. This new insertion may form a triangle containing former hull points which now must be deleted. These points-to-be-deleted will sit at the end of the chain as the most recent surviving insertions. The deletion criteria is whether the new insertion makes an obtuse angle with the last two points on the chain – recall that only acute angles appear in convex polygons. If the angle is too large, the last point on the chain has to go. We repeat until a small enough angle is created or we run out of points. We can use our *ccw*( ) predicate to test whether the angle is too big:

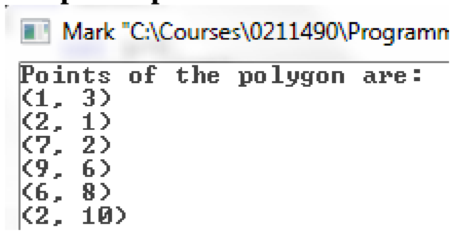
The following program includes functions to form a convex hull from a set of points.

```
// convexHull.cpp

#include <stdlib.h>
#include "geometry.cpp"

point first_point;

int leftlower(const void * a, const void * b)
{
        point *p1 = (point*) a, *p2 = (point*) b;
        if ((*p1)[X] < (*p2)[X]) return (-1);
        if ((*p1)[X] > (*p2)[X]) return (1);
        if ((*p1)[Y] < (*p2)[Y]) return (-1);
        if ((*p1)[Y] > (*p2)[Y]) return (1);

        return 0;
}

int smaller_angle(const void * a, const void * b)
{
        point *p1 = (point*) a, *p2 = (point*) b;
        if (collinear(first_point,*p1,*p2))
        {
                if (distance(first_point,*p1) <= distance(first_point,*p2))
                        return (-1);
                else
                        return (1);
        }
        if (ccw(first_point,*p1,*p2))
                return (-1);
        else
                return (1);
}

void sort_and_remove_duplicates(point in[], int *n)
{
        int i;
        int oldn;
        int hole;
        qsort(in, *n, sizeof(point), leftlower);
        oldn = *n;
        hole = 1;
        for (i=1; i<(oldn-1); i++)
        {
                if ((in[hole-1][X]==in[i][X]) && (in[hole-1][Y]==in[i][Y]))
                        (*n)--;
                else
                {
                        copy_point(in[i],in[hole]);
                        hole = hole + 1;
                }
        }

        copy_point(in[oldn-1],in[hole]);
}

void convex_hull(point in[], int n, polygon *hull)
{
        int i;
        int top;
        if (n <= 3)
        {
                for (i=0; i<n; i++)
                        copy_point(in[i],hull->p[i]);
                hull->n = n;
                return;
        }

        sort_and_remove_duplicates(in,&n);
        copy_point(in[0], first_point);
        qsort(&in[1], n-1, sizeof(point), smaller_angle);
        copy_point(first_point,hull->p[0]);
        copy_point(in[1],hull->p[1]);
        copy_point(first_point,in[n]);

        top = 1;
        i = 2;
        while (i <= n)
        {
                if (!ccw(hull->p[top-1], hull->p[top], in[i]))
                        top = top-1;
                else
                {
                        top = top+1;
                        copy_point(in[i],hull->p[top]);
                        i = i+1;
                }
        }

        hull->n = top;
}
```

The following driver produces the output below:

```
#include "convexHull.cpp"

void main()
{
        int n=8;
        point s[10];
        polygon t, *Poly=&t;

        s[0][X]=5;  s[0][Y]=5;
        s[1][X]=7;  s[1][Y]=2;
        s[2][X]=6;  s[2][Y]=8;
        s[3][X]=2;  s[3][Y]=1;
        s[4][X]=1;  s[4][Y]=3;
        s[5][X]=9;  s[5][Y]=6;
        s[6][X]=3;   s[6][Y]=3;
        s[7][X]=2;   s[7][Y]=10;

        convex_hull(s, n, Poly);
        print_polygon(Poly);
}
```

Sample output:



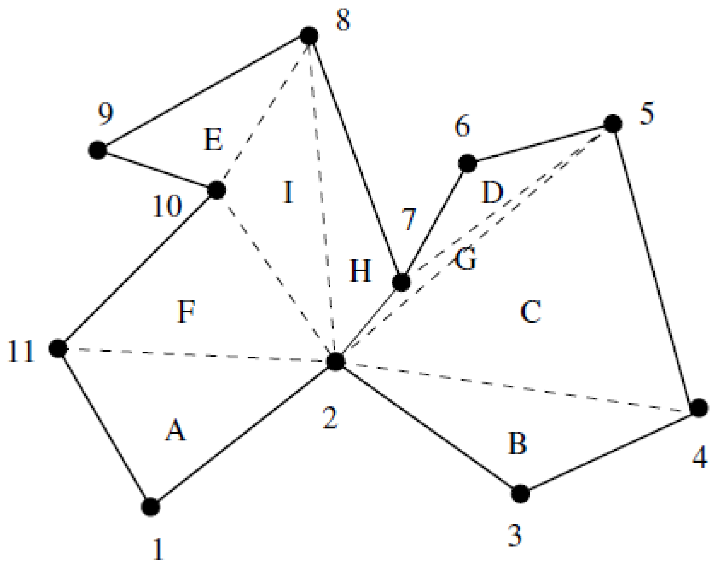
```
Mark "C:\Courses\0211490\Programn
Points of the polygon are:
(1, 3)
(2, 1)
(7, 2)
(9, 6)
(6, 8)
(2, 10)
```

Triangulation: Algorithms and Related Problems

Finding the perimeter of a polygon is easy; just compute the length of each edge using the Euclidean distance formula and add them together. Computing the area of irregular blobs is somewhat harder. The most straightforward approach is to divide the polygon into non-overlapping triangles and then sum the area of each triangle. The operation of partitioning a polygon into triangles is called *triangulation*.

Triangulating a convex polygon is easy, for we can just connect a given vertex *v* to all *n*−1 other vertices like a fan. This doesn't work for general polygons, however, because the edges might go outside the polygon. We must carve up a polygon *P* into triangles using non-intersecting chords which lie completely within *P*.

We can represent the triangulation either by listing the chords or, as we do here, with an explicit list of the vertex indices in each triangle.



Triangulating a polygon via the van Gogh (ear-cutting) algorithm, with triangles labeled in order of insertion (*A* – *I*)

Van Gogh's Algorithm:

Several polygon triangulation algorithms are known, the most efficient of which run in time linear in the number of vertices. But perhaps the simplest algorithm to program is based on ear-cutting. An ear of a polygon *P* is a triangle defined by a vertex *v* and its left and right neighbors (*l* and *r*), such that the triangle (*v*, *l*, *r*) lies completely within *P*.

Since $\overrightarrow{lv}$ and $\overrightarrow{vr}$ are boundary segments of P, the chord defining the ear is $\overrightarrow{rl}$. Under what conditions can this chord be in the triangulation? First, $\overrightarrow{rl}$ must lie completely within the interior of P. To have a chance, *lvr* must define an acute angle. Second, no other segment of the polygon can be cut by this chord, for if so a bite will be taken out of the triangle.

The important fact is that every polygon always contains an ear; in fact at least two of them for $n \geq 3$. This suggests the following algorithm. Test each one of the vertices until we find an ear. Adding the associated chord cuts the ear off, thus reducing the number of vertices by one. The remaining polygon must also have an ear, so we can keep cutting and recurring until only three vertices remain, leaving a triangle.

Testing whether a vertex defines an ear has two parts. For the angle test, we can trot out our *ccw*/*cw* predicates again. We must take care that our expectations are consistent with the vertex order of the polygon. We assume the vertices of the polygon to be labeled in counterclockwise order around the virtual center, as in the above figure. Reversing the order of the polygon would require flipping the sign on our angle test.

For the segment-cutting test, it suffices to test whether there exist any vertex which lies within the induced triangle. If the triangle is empty of points, the polygon must be empty of segments because *P* does not self-intersect. Testing whether a given point lies within a triangle will be discussed later.

Our main triangulation routine is thus limited to testing the earness of vertices, and clipping them off once we find them. A nice property of our array-of-points polygon representation is that the two immediate neighbors of vertex *i* are easily found, namely, in the $(i-1)^{st}$ and $(i+1)^{st}$ positions in the array. This data structure does not cleanly support vertex deletion, however. To solve this problem, we define auxiliary arrays *l* and *r* that point to the current left and right neighbors of every point remaining in the polygon.

Area Computations:

The area $A(T)$ of a triangle T can be calculated using the coordinates of its three vertices $v_0(x_0, y_0)$, $v_1(x_1, y_1)$, $v_2(x_2, y_2)$ as in the formula:

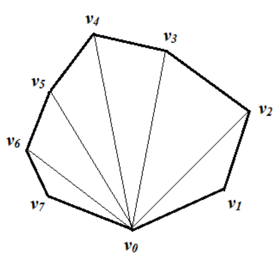
$$A(v_0, v_1, v_2) = \tfrac{1}{2}[(x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)]$$

or

$$A(v_0, v_1, v_2) = \tfrac{1}{2}[(x_0 y_1 - x_1 y_0) + (x_1 y_2 - x_2 y_1) + (x_2 y_0 - x_0 y_2)]$$

We can compute the area of any triangulated polygon by summing the area of all triangles. This is easy to implement using the routines we have already developed. The area of the convex polygon *P* defined by its vertices $\{v_0, v_1, \dots, v_{n-1}\}$ labeled counterclockwise is given by:

$$A(P) = A(v_0, v_1, v_2) + A(v_0, v_2, v_3) + A(v_0, v_3, v_4) + \dots + A(v_0, v_{n-2}, v_{n-1})$$

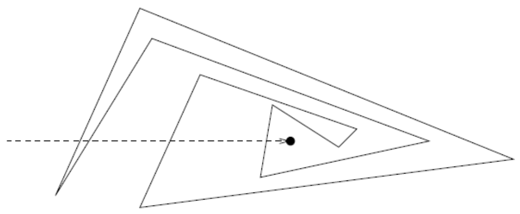


However, there is an even slicker algorithm based on the notion of signed areas for triangles, which we used as the basis for our *ccw* routine. By properly summing the signed areas of the triangles defined by an arbitrary point *p* with each segment of polygon *P* we get the area of *P*, because the negatively signed triangles cancel the area outside the polygon. This computation simplifies to the equation

$$A(P) = \frac{1}{2}\sum_{i=0}^{n-1}(x_i\, y_{i+1} - x_{i+1}\, y_i)$$

Since the index calculations are done mod *n*, then $x_n = x0$ and $y_n = y0$. The above formula holds for convex and non-convex polygons, provided that the vertices are labeled counterclockwise.

Point Location:
Our triangulation algorithm defined a vertex as an ear only when the associated triangle contained no other points. Thus ear testing requires us to test whether a given point *p* lies within the interior of a triangle *t*.



The odd/even parity of the number of boundary crossings determines whether a given point is inside or outside a given polygon.
Triangles are always convex polygons, because three vertices do not give the freedom to create bumps and notches. A point lies within a convex polygon if it is to the left of each of the directed lines $\overrightarrow{p_i p_{i+1}}$, where the vertices of the polygon are represented in counterclockwise order. The function *point_in_triangle*( ) enables us to easily make such left-of decisions together with the *ccw* predicate.
This algorithm works to decide point location (in *P* or out?) for convex polygons. But it breaks down for general polygons. Imagine the task of deciding whether a point is inside or outside the center of a complex spiral-shaped polygon. There is a straightforward solution for general polygons using the code we have already developed. Ear-clipping required us to test whether a given point lies within a given triangle. Thus we can use triangulate to divide the polygon into triangular cells and then test each of the cells to see whether they contain the point. If one of them does, the point is in the polygon.
Triangulation is a heavyweight solution for this problem, however, just as it was for area. There is a much simpler algorithm based on the *Jordan curve theorem*, which states that every polygon or other closed figure has an inside and an outside. You can't get from one to the other without crossing the boundary.
This gives the following algorithm, illustrated in the above figure. Suppose we draw a line *l* that starts from outside the polygon *P* and goes through point *q*. If this line crosses the polygon boundary an even number of times before reaching *q*, it must lie outside P. Why? If we start outside the polygon, then every pair of boundary crossings leaves us outside. Thus an odd number of boundary crossings put us inside *P*.
Important subtleties occur at degenerate cases. Cutting through a vertex of *p* crosses a boundary only if we enter the interior of *p*, instead of just clipping off the vertex. We cross a boundary if and only if the vertices neighboring *p* lie on different sides of line *l*. Crawling along an edge of the

polygon does not change the boundary count, although it raises the application-specific question of whether such a point on the boundary is considered inside or outside *p*.

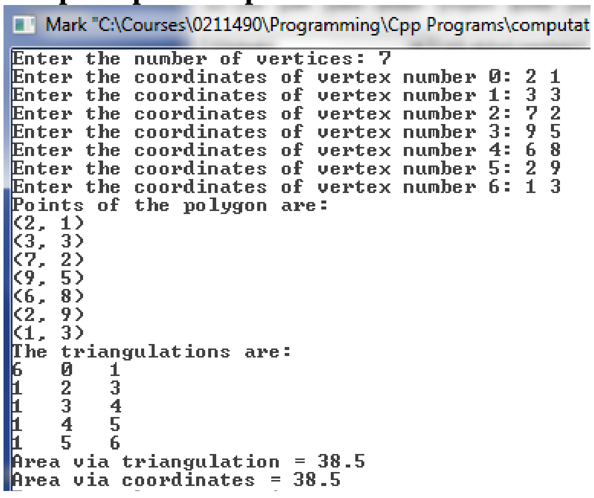
The following program includes functions to find the area of a polygon.

```
// triangulation.cpp

#include "geometry.cpp"

void add_triangle(triangulation *t, int i, int j, int k, polygon *p)
{
        int n;

        n = t->n;

        t->t[n][0] = i;
        t->t[n][1] = j;
        t->t[n][2] = k;

        t->n = n + 1;
}

bool point_in_triangle(point p, triangle t)
{
        int i;

        for (i=0; i<3; i++)
                if (cw(t[i],t[(i+1)%3],p)) return(false);

        return(true);
}

bool ear_Q(int i, int j, int k, polygon *p)
{
        triangle t;
        int m;

        copy_point(p->p[i],t[0]);
        copy_point(p->p[j],t[1]);
        copy_point(p->p[k],t[2]);

        if (cw(t[0],t[1],t[2])) return(false);

        for (m=0; m<p->n; m++)
        {
                if ((m!=i) && (m!=j) && (m!=k))
                        if (point_in_triangle(p->p[m],t)) return(false);
        }
        return(true);
}

void triangulate(polygon *p, triangulation *t)
{
        int l[MAXPOLY], r[MAXPOLY];
        int i;

        for (i=0; i<p->n; i++)
        {
                l[i] = ((i-1) + p->n) % p->n;
                r[i] = ((i+1) + p->n) % p->n;
        }

        t->n = 0;
        i = p->n-1;
        while (t->n < (p->n-2))
        {
                i = r[i];
                if (ear_Q(l[i],i,r[i],p))
                {
                        add_triangle(t,l[i],i,r[i],p);
                        l[ r[i] ] = l[i];
                        r[ l[i] ] = r[i];
                }
        }
}

double area_triangulation(polygon *p)
{
        triangulation t;
        double total = 0.0;
   int i;

        triangulate(p,&t);
        for (i=0; i<t.n; i++)
                total += triangle_area(p->p[t.t[i][0]],p->p[t.t[i][1]], p->p[t.t[i][2]]);

        return(total);
}

double area_via_coordinates(polygon *p)
{
        double total = 0.0;
        int i, j;

        for (i=0; i<p->n; i++) {
                j = (i+1) % p->n;
                total += (p->p[i][X]*p->p[j][Y]) - (p->p[j][X]*p->p[i][Y]);
        }

        return(total / 2.0);
}
```

The following program is the driver.

```
#include "triangulation.cpp"

void main()
{
        polygon p;
        triangulation t;
        int i;

        cout << "Enter the number of vertices: ";
        cin >> p.n;
        for (i=0; i<p.n; i++)
        {
                cout << "Enter the coordinates of vertex number " << i << ": ";
                cin >> p.p[i][X] >> p.p[i][Y];
        }

        print_polygon(&p);
        triangulate(&p, &t);
        print_triangulation(&t);

        cout << "Area via triangulation = " << area_triangulation(&p) << endl;
        cout << "Area via coordinates = " << area_via_coordinates(&p) << endl;

}
```

Sample input / output:



```
Enter the number of vertices: 7
Enter the coordinates of vertex number 0: 2 1
Enter the coordinates of vertex number 1: 3 3
Enter the coordinates of vertex number 2: 7 2
Enter the coordinates of vertex number 3: 9 5
Enter the coordinates of vertex number 4: 6 8
Enter the coordinates of vertex number 5: 2 9
Enter the coordinates of vertex number 6: 1 3
Points of the polygon are:
(2, 1)
(3, 3)
(7, 2)
(9, 5)
(6, 8)
(2, 9)
(1, 3)
The triangulations are:
6    0    1
1    2    3
1    3    4
1    4    5
1    5    6
Area via triangulation = 38.5
Area via coordinates = 38.5
```

Algorithms on Grids:

That polygons drawn on rectilinear and hexagonal grid points can be naturally decomposed into individual cells makes it useful to be able to solve certain computational problems on these cells:

- *Area* — The formula *length* × *width* computes the area of a rectangle. For triangles, it is ½ × *altitude* × *base*. An equilateral triangle where each side has length r has area $\sqrt{3}r^2/4$; so a regular hexagon with radius r has area $3\sqrt{3}r^2/2$.
- *Perimeter* — The formula 2 × (*length* + *width*) computes the perimeter of a rectangle. For triangles, we sum the side lengths, $a + b + c$, which reduces to $3r$ for equilateral triangles. Regular hexagons of radius r have perimeter $6r$; observe how they approach the circumference of a circle $2\pi r \approx 6.28r$.
- *Convex Hulls* — Squares, equilateral triangles, and regular hexagons are all inherently convex, so they are all their own convex hulls.
- *Triangulation* — Inserting either one of the two diagonals in a square or all three diagonals radiating from any point in a regular hexagon triangulates it. This works only because these figures are convex; notches and bumps make the process harder.
- *Point location* — As we have seen, a point lies in an axis-oriented rectangle if and only if $x_{\max} > x > x_{\min}$ and $y_{\max} > y > y_{\min}$. Such tests are slightly more difficult for triangles and hexagons, but surrounding these shapes by a bounding box usually reduces the need for the complicated case.

We conclude this section with two interesting algorithms for geometric computing on grids. They are primarily of interest for rectilinear grids but can be adapted to other lattices if the need arises.

Range Queries:

Orthogonal range queries are a common operation in working with $n \times m$ rectilinear grids. We seek a data structure which quickly and easily answers questions of the form: “What is the sum of the values in a given sub-rectangle of the matrix?”

Any axis-oriented rectangle can be specified by two points, the upper-left-hand corner (x_l, y_l) and the lower-right-hand corner (x_r, y_r). The simplest algorithm is to run nested loops adding up all values $m[i][j]$ for $x_l \le i \le x_r$ and $y_r \le j \le y_l$. But this is inefficient, particularly if you must do it repeatedly in seeking the rectangle of largest or smallest such sum.

Instead, we can construct an alternate rectangular matrix such that element $m_1[x][y]$ represents the sum of all elements $m[i][j]$ where $i \le x$ and $j \le y$. This dominance matrix m_1 makes it easy to find the sum of the elements in any rectangle, because the sum $S(x_l, y_l, x_r, y_r)$ of elements in such a box is

$$S(x_l, y_l, x_r, y_r) = m_1[x_r, y_l] - m_1[x_l - 1, y_l] - m_1[x_r, y_r - 1] + m_1[x_l - 1, y_r - 1]$$

This is certainly fast, reducing the computation to just four array element lookups. Why it is correct? The term $m_1[x_r, y_l]$ contains the sum of all the elements in the desired rectangle, plus all other dominated items. The next two terms subtract this away, but remove the lower-left-hand corner twice so it must be added back again. The argument is that of standard inclusion-exclusion formulas in combinatorics. The array m_1 can be built in $O(mn)$ time by filling in the cells using row-major ordering and similar ideas.

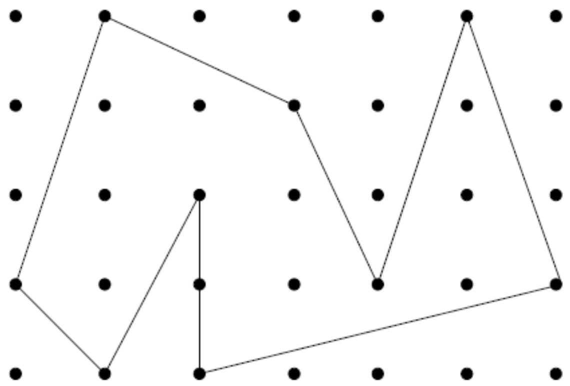
Lattice Polygons and Pick’s Theorem:

Rectangular grids of unit-spaced points (also called lattice points) are at the heart of any grid-based coordinate system. In general, there will be about one grid point per unit-area in the grid, because each grid point can be assigned to be the upper-right-hand corner of a different 1×1 empty

rectangle. Thus the number of grid points within a given figure should give a pretty good approximation to the area of the figure.
Pick's theorem gives an exact relation between the area of a lattice polygon P (a non-intersecting figure whose vertices all lie on lattice points) and the number of lattice points on/in the polygon. Suppose there are $I(P)$ lattice points inside of P and $B(P)$ lattice points on the boundary of P. Then the area $A(P)$ of P is given by

$$A(P) = I(P) + B(P)/2 - 1$$

as illustrated in the figure below.



A lattice polygon with ten boundary points and nine internal points, and hence area 13 by Pick's theorem.

For example, consider a triangle defined by coordinates $(x, 1)$, $(y, 2)$, and $(y+k, 2)$. No matter what x, y, and k are there can be no interior points, because the three points lie on consecutive rows of the lattice. Lattice point $(x, 1)$ serves as the apex of the triangle, and there are $k+1$ lattice points on the boundary of the base. Thus $I(P) = 0$, $B(P) = k + 2$, and so the area is $k/2$, precisely what you get from the triangle area formula.
As another example, consider a rectangle defined by corners (x_1, y_1) and (x_2, y_2). The number of boundary points is

$$B(P) = 2\,|y_2 - y_1 + 1| + 2\,|x_2 - x_1 + 1| - 4 = 2\,(\Delta_y - \Delta_x)$$

with the 4-term to avoid double-counting the corners. The interior is the total number of points in or on the rectangle minus the boundary, giving

$$I(P) = (\Delta_x + 1)\,(\Delta_y + 1) - 2\,(\Delta_y - \Delta_x)$$

Pick's theorem correctly computes the area of the rectangle as $\Delta_x\,\Delta_y$. Applying Pick's theorem requires counting lattice points accurately. This can in principle be done by exhaustive testing for small area polygons using functions that (1) test whether a point lies on a line segment and (2) test whether a point is inside or outside a polygon. Cleverer sweep-line algorithms would eliminate the need to check all but the boundary points for efficiency.

Problems

The Closest Pair Problem

PC/UVa IDs:	111402/10245	**Popularity:**	A	**Success rate:**	low	**Level:**	2

A particularly inefficient telephone company seeks to claim they provide high-speed broadband access to customers. It will suffice for marketing purposes if they can create just one such link directly connecting two locations. As the cost for installing such a connection is proportional to the distance between the sites, they need to know which pair of locations are the shortest distance apart so as to provide the cheapest possible implementation of this marketing strategy.
More precisely, given a set of points in the plane, find the distance between the closest pair of points provided this distance is less than some limit. If the closest pair is too far apart, marketing will have to opt for some less expensive strategy.

Input

The input file contains several sets of input. Each set of input starts with an integer N ($0 \leq N \leq 10{,}000$), which denotes the number of points in this set. The next N lines contain the coordinates of N two-dimensional points. The two numbers denote the x- and y-coordinates, respectively. The input is terminated by a set whose $N = 0$, which should not be processed. All coordinates will have values less than 40,000 and be non-negative.

Output

For each input set, produce a single line of output containing a floating point number (with four digits after the decimal point) which denotes the distance between the closest two points. If there do not exist two points whose distance is less than 10,000, print the line “INFINITY”.

Sample Input

```
3
0 0
10000 10000
20000 20000
5
0 2
6 67
43 71
39 107
189 140
0
```

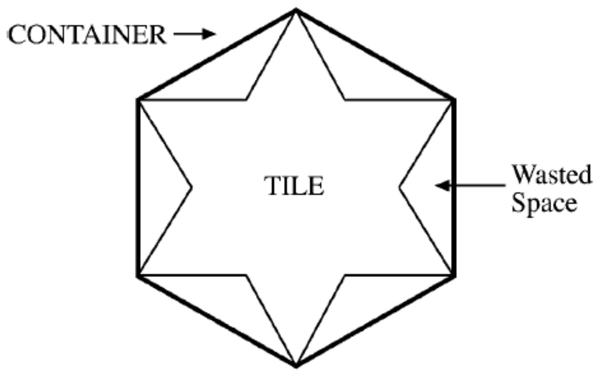
Sample Output

```
INFINITY
36.2215
```

Useless Tile Packers

PC/UVa IDs:	111405/10065	**Popularity:**	C	**Success rate:**	average	**Level:**	3

Useless Tile Packer, Inc., prides itself on efficiency. As their name suggests, they aim to use less space than other companies. Their marketing department has tried to convince management to change the name, believing that “useless” has other connotations, but has thus far been unsuccessful.
Tiles to be packed are of uniform thickness and have a simple polygonal shape. For each tile, a container is custom-built. The floor of the container is a convex polygon that has the minimum possible space inside to hold the tile it is built for.



This strategy leads to wasted space inside the container. Your job is to compute the percentage of wasted space for a given tile.

Input

The input file consists of several data blocks. Each data block describes one tile. The first line of a data block contains an integer N ($3 \le N \le 100$) indicating the number of corner points of the tile. Each of the next N lines contains two integers giving the (x, y) coordinates of a corner point (determined using a suitable origin and orientation of the axes) where $0 \le x, y \le 1{,}000$. The corner points occur in the same order on the boundary of the tile as they appear in the input. No three consecutive points are collinear.
The input file terminates with a value of 0 for N.

Output

For each tile in the input, print the percentage of wasted space rounded to two digits after the decimal point. Each output must be on a separate line.
Print a blank line after each output block.

Sample Input

```
5
0 0
2 0
2 2
```

```
1 1
0 2
5
0 0
0 2
1 3
2 2
2 0
0
```

Sample Output

```
Tile #1
Wasted Space = 25.00 %

Tile #2
Wasted Space = 0.00 %
```

Radar Tracking

PC/UVa IDs:	111406/895	**Popularity:**	C	**Success rate:**	low	**Level:**	2

A ground-to-air radar system uses an antenna that rotates clockwise in a horizontal plane with a period of two seconds. Whenever the antenna faces an object, its distance from that antenna is measured and displayed on a circular screen as a white dot. The distance from the dot to the center of the screen is proportional to the horizontal distance from the antenna to the object, and the angle of the line passing through the center and the dot represents the direction of the object from the antenna. A dot directly above the center represents an object that is north of the antenna; an object to the right of the center represents an object to the east; and so on.
There are a number of objects in the sky. Each is moving at a constant velocity, and so the dot on the screen appears in a different position every time the antenna observes it. Your task is to determine where the dot will appear on the screen the next time the antenna observes it, given the previous two observations. If there are several possibilities, you are to find them all.

Input

The input consists of a number of lines, each with four real numbers: a_1, d_1, a_2, d_2. The first pair a_1, d_1 are the angle (in degrees) and distance (in arbitrary distance units) for the first observation while the second pair a_2, d_2 are the angle and distance for the second observation.
Note that the antenna rotates clockwise; that is, if it points north at time $t = 0.0$, it points east at $t = 0.5$, south at $t = 1.0$, west at $t = 1.5$, north at $t = 2$, and so on. If the object is directly on top of the radar antenna, it cannot be observed. Angles are specified as on a compass, where north is 0^o or 360^o, east is 90^o, south is 180^o, and west is 270^o.

Output

The output consists of one line per input case containing all possible solutions. Each solution consists of two real numbers (with two digits after the decimal place) indicating the angle a_3 and distance d_3 for the next observation.

Sample Input

```
90.0 100.0 90.0 110.0
90.0 100.0 270.0 10.0
90.0 100.0 180.0 50.0
```

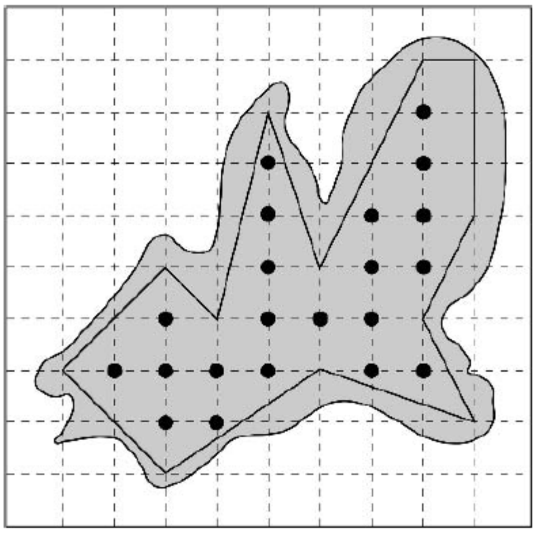
Sample Output

```
90.00 120.00
270.00 230.00
199.93 64.96 223.39 130.49
```

Trees on My Island

PC/UVa IDs:	111407/10088	**Popularity:**	C	**Success rate:**	average	**Level:**	3

I have bought an island where I want to plant trees in rows and columns. The trees will be planted to form a rectangular grid, so each can be thought of as having integer coordinates by taking a suitable grid point as the origin.


A sample of my island

However, my island is not rectangular. I have identified a simple polygonal area inside the island with vertices on the grid points and have decided to plant trees on grid points lying strictly inside the polygon.
I seek your help in calculating the number of trees that can be planted.

Input

The input file may contain multiple test cases. Each test case begins with a line containing an integer *N* ($3 \le N \le 1,000$) identifying the number of vertices of the polygon. The next *N* lines contain the vertices of the polygon in either the clockwise or counterclockwise direction. Each of these *N* lines contains two integers identifying the *x*- and *y*-coordinates of a vertex. You may assume that the absolute value of all coordinates will be no larger than 1,000,000.
A test case containing a zero for *N* in the first line terminates the input.

Output

For each test case, print a line containing the number of trees that can be planted inside the polygon.

Sample Input

```
12
3 1
6 3
```

```
9 2
8 4
9 6
9 9
8 9
6 5
5 8
4 4
3 5
1 3
12
1000 1000
2000 1000
4000 2000
6000 1000
8000 3000
8000 8000
7000 8000
5000 4000
4000 5000
3000 4000
3000 5000
1000 3000
0
```

Sample Output

```
21
25990001
```