Rujia Liu's Present 4

A Contest Dedicated to

Geometry and CG Lovers

(A computer-rendered image by my company)

October 1, 2011
UVa Online Judge

Problems

= Warm-up Problems =

A. Smallest Regular Polygon
B. An Angular Puzzle
C. Nine-Point Circle

= Geometry Computations =

D. Composite Transformations
E. 2D Geometry 110 in 1!
F. Polishing a Extruded Polygon
G. My SketchUp

= Geometry Algorithms =

H. Smallest Enclosing Rectangle
I. Smallest Enclosing Box
J. A Strange Opera House II
K. Point Location
L. All-Pair Farthest Points

= Computer Graphics and Motion Planning =

M. Bounding Volume Hierarchy
N. A Tiny Raytracer
O. The Cleaning Robot

I'm trying my best to reduce potential floating-point issues by carefully designing the test cases and allowing some difference between your output and the standard one, and the margins are always quite generous. If your programs still suffer from these kinds of errors, it usually means that they are numerically instable. Please redesign your algorithm or refactor your code.

I know that geometry problems can be very difficult to debug, even if you're given the judge data! Thus, I decided to provide some additional data and debugging tools that make your life (a little bit) easier. You can download them on the contest website.

I'd like to thank Yiming Li for validating problem A, B, C and M, Yi Chen and Di Tang for validating problem E, Yeji Shen and Dun Liang for discussing and validating other problems.

Hello, everyone! My name is Rujia Liu. I used to do a lot of problem solving and problemsetting, but after graduated from Tsinghua University, I'm spending more and more time on my company **L**

(You may realized that the paragraph above is copied from the texts of my 3rd contest, but that's me, lazy me.)

This time, my contest is all about geometry and computer graphics (CG). This is a bit strange, since I'm a bit weak in geometry during my "programming contest career". However, my company, www.qeyj.com "forced" me to write a lot of codes for 3dsmax, AutoCAD, developed some real-time 3D applications, and even our own off-line render (that's why I wrote problem M and N!). As a result, I'm getting more and more familiar with geometry and CG and finally decided to arrange this contest.

This contest is partially educative. I'm trying to cover a lot of important areas of geometry and CG, so the problems themselves are not very imaginative. Nevertheless, some of the problems (like problem N) are still attractive (at least to me), so try your best to enjoy the contest!

However, I believe this contest is hard, not only for most (if not all!) contestants, but also for myself. I mean, geometry codes can be very error-prone, so I'm NOT 100% sure that there's no mistake in the judge data (though I've spent a looooot of time to prevent such things). Don't hesitate to write emails to me (rujia.liu@gmail.com) during the contest if you suspect the judge data might be wrong (or suffered from floating-point issues). The best way is to email me your solution, and then I'll check ASAP.

This is an Internet contest, so don't forget that you can always use google. I will NOT be angry if you use codes from someone else. That's also a way to learn. However, please try to google formulae, ideas and algorithms only, and write your own code. That's best for you.

Best regards,
Rujia Liu

A. Smallest Regular Polygon

Given two different points A and B, your task is to find a *regular* polygon of *n* sides, passing through these two points, so that the polygon area is minimized.

Input

There will be at most 100 test cases. Each case contains 5 integers x_A, y_A, x_B, y_B, n ($0 <= x_A, y_A, x_B, y_B <= 100$, $3 <= n <= 10000$), the coordinates of A and B, and the number of sides of the regular polygon. The two points A and B are always different. The last test case is followed by a line with five zeros, which should not be processed.

Output

For each test case, print the smallest area of the regular polygon to six decimal places.

Sample Input

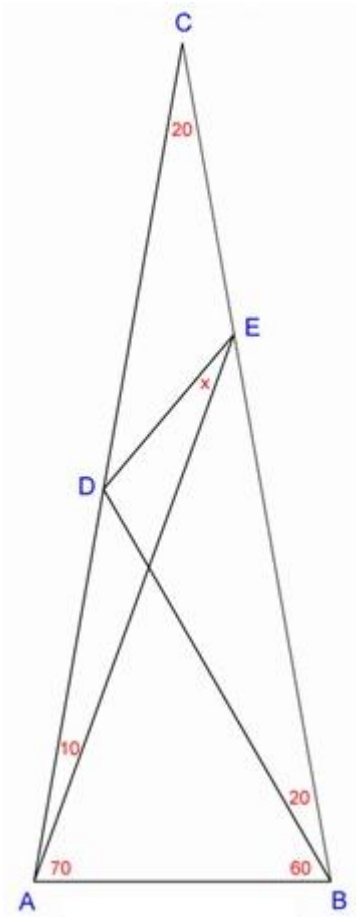
```
0 0 1 1 4
1 2 3 4 5
2 3 4 5 6
0 0 0 0 0
```

Output for Sample Input

```
1.000000
5.257311
5.196152
```

B. An Angular Puzzle

Here is an old interesting puzzle: In the picture below, what is the angle of DEA, in degrees? Note that 5 angles are already given. The picture is drawn to scale.



You're to solve a generalized problem: let *a*, *b*, *c*, *d*, *e* be the angle of ACB, CAE, EAB, CBD, DBA (in degrees), what is the angle of DEA, in degrees?

Note that E must be strictly on segment BC (cannot coincide with B or C), and D must be strictly on segment AC (cannot coincide with A or C). The triangle ABC must be non-degenerated (i.e. ABC cannot be collinear). Not all combination of parameters *a*, *b*, *c*, *d* and *e* corresponds to a valid figure described above. Your program should be able to detect this.

Input

There will be at most 100 test cases. Each case contains 5 integers *a*, *b*, *c*, *d*, *e* ($0 < a, b, c, d, e < 90$). The last test case is followed by five zeros, which should not be processed.

Output

For each test case, print the answer to two decimal places. If there is more than one solution, print "`Multiple solutions`". If the input is incorrect (i.e. there is no valid picture for these parameters), print "`Impossible`" (without quotes).

Sample Input

```
20 10 70 20 60
30 5 70 15 60
60 30 30 30 30
30 40 40 40 40
0 0 0 0 0
```

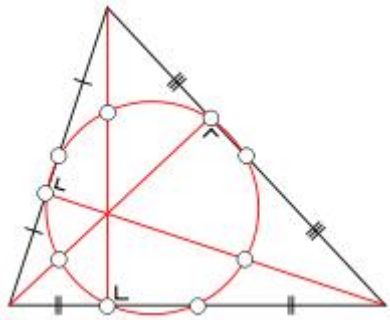
Output for Sample Input

```
20.00
12.96
30.00
Impossible
```

C. Nine-Point Circle

In geometry, the nine-point circle is a circle that can be constructed for any given triangle. It is so named because it passes through nine significant points defined from the triangle. These nine points are:

- The midpoint of each side of the triangle
- The foot of each altitude
- The midpoint of the line segment from each vertex of the triangle to the orthocenter (where the three altitudes meet; these line segments lie on their respective altitudes).



The nine-point circle is also known as Feuerbach's circle, Euler's circle, Terquem's circle, the six-point circle, the twelve-point circle, the n-point circle, the medioscribed circle, the mid circle or the circum-midcircle.

Given three non-collinear points A, B and C, you're to calculate the center position and radius of triangle ABC's nine-point circle.

Input

There will be at most 100 test cases. Each case contains 6 integers x1, y1, x2, y2, x3, y3 (0 <= x1, y1, x2, y2, x3, y3 <= 1000), the coordinates of A, B and C. The last test case is followed by a line with six -1, which should not be processed.

Output

For each test case, print three real numbers x, y, r, indicating that the nine point circle is centered at (x,y), with radius r. The numbers should be rounded to six decimal places.

Sample Input

```
0 0 10 0 3 4
-1 -1 -1 -1 -1 -1
```

Output for Sample Input

```
4.000000 2.312500 2.519456
```

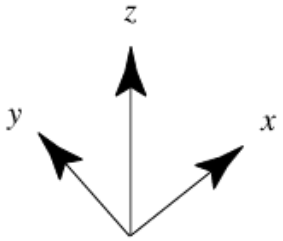
D. Composite Transformations

Given *n* points and *m* planes in 3D place, you will need to perform *t* transformations on them, and then calculate their final states. By "transforming a plane", we mean transforming all the points on that plane.

There are three kinds of transformations (in the text below, P means the point being transformed):

TRANSLATE a b c	If P's position was (x, y, z), it becomes (x+a, y+b, z+c) after the transformation.
ROTATE a b c theta	P is rotated. The rotation axis is vector (a,b,c), the angle of rotation is theta degrees. The rotation follows the right-hand rule, so if the vector (a,b,c) points toward you, the rotation will be counterclockwise from your point of view. The rotation axis always passes through (0,0,0)
SCALE a b c	If P's position was (x, y, z), it becomes (ax, by, cz) after the transformation.

This problem uses right-hand coordinate system:



Tips: An example test case can be downloaded from the contest website.

Input

There will be only one test case, beginning with three integers *n*, *m*, *t* (1<=*n*,*m*<=50,000, 1<=*t*<=1,000). Next *n* lines contain the coordinates of each point. Next *m* lines contains four integers a, b, c, d to describe a plane ax+by+cz+d=0 (at least one of a, b, c will be non-zero). Next *t* lines contain the operations. All the input coordinates and parameters a, b, c, d are real numbers with absolute values not larger than 10. These input real numbers will have at most two digits after the decimal point. Parameter theta is an integer between 0 and 359 (inclusive).

Output

For each point, print three real numbers x, y, z on a single line. For each plane, print four real numbers a, b, c, d on a single line. To avoid floating-point issues, $a^2+b^2+c^2$ must be 1, but if there is more than one choice of (a, b, c, d) to represent the answer, anyone is acceptable. Output each real number to two decimal places. To reduce the impact of floating-point errors, each number you print could differ from the standard output by up to 0.05.

Sample Input

```
1 1 3
1 2 3
0 0 1 -2
TRANSLATE 2 3 4
ROTATE 1 0 0 90
SCALE 3 2 1
```

Output for Sample Input

```
9.00 -14.00 5.00
0.00 1.00 0.00 12.00
```

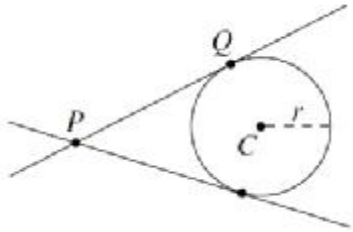
E. 2D Geometry 110 in 1!

This is a collection of 110 (in binary) 2D geometry problems.

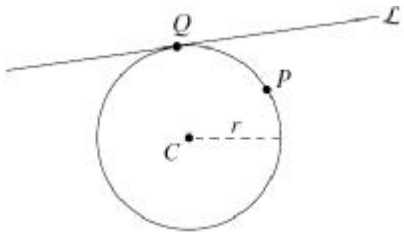
CircumscribedCircle x1 y1 x2 y2 x3 y3
Find out the circumscribed circle of triangle (x1,y1)-(x2,y2)-(x3,y3). These three points are guaranteed to be non-collinear. The circle is formatted as (x,y,r) where (x,y) is the center of circle, r is the radius.

InscribedCircle x1 y1 x2 y2 x3 y3
Find out the inscribed circle of triangle (x1,y1)-(x2,y2)-(x3,y3). These three points are guaranteed to be non-collinear. The circle is formatted as (x,y,r) where (x,y) is the center of circle, r is the radius.

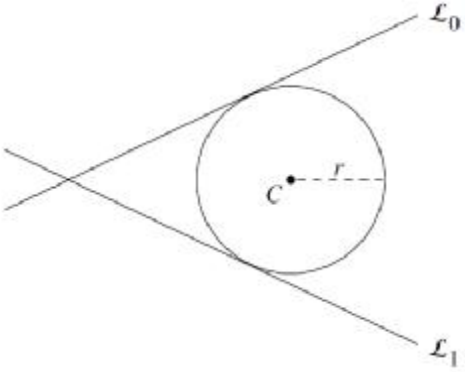
TangentLineThroughPoint xc yc r xp yp
Find out the list of tangent lines of circle centered (xc,yc) with radius r that pass through point (xp,yp). Each tangent line is formatted as a single real number "angle" (in degrees), the angle of the line (0<=angle<180). Note that the answer should be formatted as a list (see below for details).



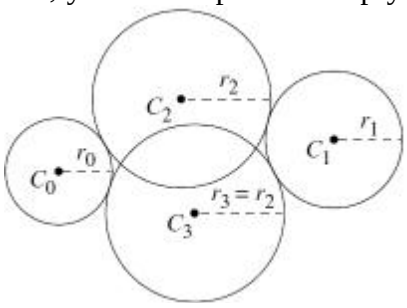
CircleThroughAPointAndTangentToALineWithRadius xp yp x1 y1 x2 y2 r
Find out the list of circles passing through point (xp, yp) that is tangent to a line (x1,y1)-(x2,y2) with radius r. Each circle is formatted as (x,y), since the radius is already given. Note that the answer should be formatted as a list. If there is no answer, you should print an empty list.



CircleTangentToTwoLinesWithRadius x1 y1 x2 y2 x3 y3 x4 y4 r
Find out the list of circles tangent to two *non-parallel* lines (x1,y1)-(x2,y2) and (x3,y3)-(x4,y4), having radius r. Each circle is formatted as (x,y), since the radius is already given. Note that the answer should be formatted as a list. If there is no answer, you should print an empty list.



CircleTangentToTwoDisjointCirclesWithRadius x1 y1 r1 x2 y2 r2 r
Find out the list of circles tangent to two disjoint circles (x1,y1,r1) and (x2,y2,r2), having radius r. Each circle is formatted as (x,y), since the radius is already given. Note that the answer should be formatted as a list. If there is no answer, you should print an empty list.



For each line described above, the two endpoints will not be equal. When formatting a list of real numbers, the numbers should be sorted in increasing order; when formatting a list of (x,y) pairs, the pairs should be sorted in increasing order of x. In case of tie, smaller y comes first.

Input
There will be at most 1000 sub-problems, one in each line, formatted as above. The coordinates will be integers with absolute value not greater than 1000. The input is terminated by end of file (EOF).

Output
For each input line, print out your answer formatted as stated in the problem description. Each number in the output should be rounded to six digits after the decimal point. Note that the list should be enclosed by square brackets, and tuples should be enclosed by brackets. There should be no space characters in each line of your output.

Sample Input

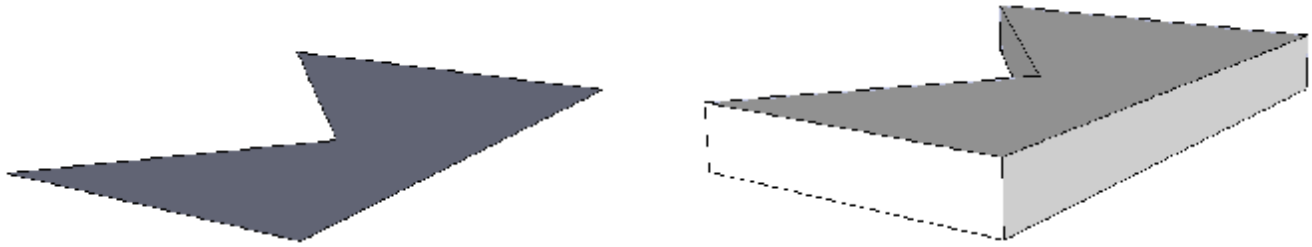
```
CircumscribedCircle 0 0 20 1 8 17
InscribedCircle 0 0 20 1 8 17
TangentLineThroughPoint 200 200 100 40 150
TangentLineThroughPoint 200 200 100 200 100
TangentLineThroughPoint 200 200 100 270 210
CircleThroughAPointAndTangentToALineWithRadius 100 200 75 190
185 65 100
CircleThroughAPointAndTangentToALineWithRadius 75 190 75 190
185 65 100
CircleThroughAPointAndTangentToALineWithRadius 100 300 100
100 200 100 100
CircleThroughAPointAndTangentToALineWithRadius 100 300 100
100 200 100 99
CircleTangentToTwoLinesWithRadius 50 80 320 190 85 190 125 40
30
CircleTangentToTwoDisjointCirclesWithRadius 120 200 50 210
150 30 25
CircleTangentToTwoDisjointCirclesWithRadius 100 100 80 300 250 70 50
```

Output for Sample Input

```
(9.734940,5.801205,11.332389)
(9.113006,6.107686,5.644984)
[53.977231,160.730818]
[0.000000]
[]
[(112.047575,299.271627),(199.997744,199.328253)]
[(-0.071352,123.937211),(150.071352,256.062789)]
[(100.000000,200.000000)]
[]
[(72.231286,121.451368),(87.815122,63.011983),(128.242785,144.270867
),(143.826621,85.831483)]
[(157.131525,134.836744),(194.943947,202.899105)]
[(204.000000,178.000000)]
```

F. Polishing a Extruded Polygon

You have a simple polygon on z=0 plane, then you make an extrusion so that its height becomes h unit (i.e. the top surface is now on z=h). We call the extruded polygon *the model*. Note that the model is a solid.



After that, you apply m "cutting" operations on the model. Each operation is described by four integers a, b, c, d, that is to remove the set of points (x_p, y_p, z_p) satisfying $ax_p+by_p+cz_p+d>0$. Finally, you need to calculate the volume and surface area of the polished model.

Input

There will be at most 10 test cases. Each case begins with three integers n, h, m ($1<=n<=1000$, $1<=h<=10$, $1<=m<=100$), the number of vertices of the polygon, the height after the extrusion, and the number of operations. The next n lines contain the vertices of the polygon in counter-clockwise order (note that they're in z=0 plane, so the z coordinates are not given. Coordinates are real numbers whose absolute values are not greater than 1000). Each of the next m lines contains four real numbers a, b, c, d, describing one operation. At least one of a, b, c is non-zero. The last test case is followed by a line with n=h=m=0, which should not be processed.

Output

For each test case, print the volume and surface area of the original model and polished model *after every operation* (i.e. output m+1 lines in total). Each number should be rounded to three digits after the decimal point. It is guaranteed that the volume and surface area is always strictly positive. To reduce the impact of floating-point errors, each number you print could differ from the standard output by up to 0.005.

Sample Input

```
4 5 2
0 0
10 0
10 10
0 10
1 0 0 -9
0 0 1 -3
5 5 1
0 0
4 0
4 4
2 2
0 4
0 -1 0 3
5 5 1
0 0
4 0
4 4
2 2
0 4
0 1 0 -3
0 0 0
```

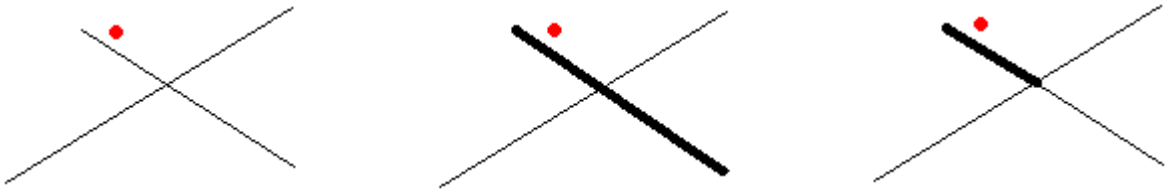
Output for Sample Input

```
500.000 400.000
450.000 370.000
270.000 294.000
60.000 112.284
5.000 36.142
60.000 112.284
55.000 96.142
```

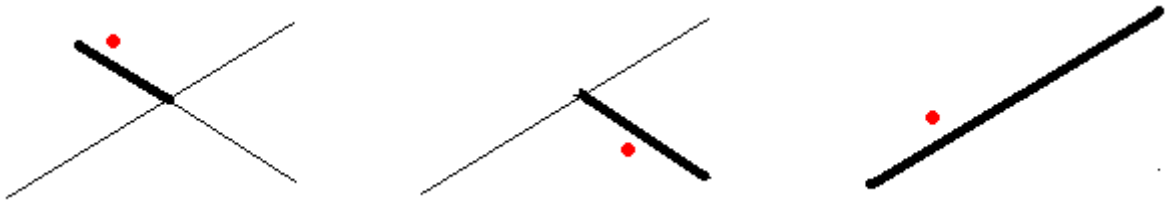
G. My SketchUp

Google SketchUp is an easy-to-use program that lets you create, modify and share 3D models. In this problem, you're to write a simplified version of SketchUp called My SketchUp. Since 3D is complex, My SketchUp will be in 2D only.

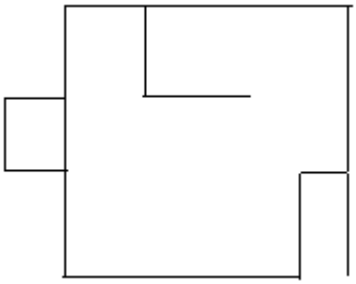
SketchUp is intuitive. To understand this, suppose you've drawn two line segments that intersect other, like this:



Then, you click your mouse at the red dot. If you're using AutoCAD, you'll select a whole segment (shown in the middle picture), but in My SketchUp, you'll only select a small segment (shown in the right picture), because the two segment you've drawn cut each other! What's more, if you remove two small segments as shown below, the other two small segments will automatically join up to become one segment again! Note that in the middle picture, you cannot select the "long" segment as a whole, because it's still cut into two pieces.



Here is the general rule: segments that intersect each other actually cut each other; and there will be no "redundant" points that can be removed without affecting the appearance of the picture. As a result, the in-memory data structure of the segments can be deduced merely from the appearance. *If two pictures look the same, they are the same internally.* For example, if you draw (0,0)-(1,0), then (1,0)-(2,0), you'll have only one segment: (0,0)-(2,0), if you draw (0,0)-(1,0) twice, you'll only get one. In the picture below, there are 14 vertices and 15 segments (no matter how you draw this picture!!).



Your task is to execute a sequence of commands (described in the input format section) and print the description of the resulting picture. Vertices are sorted in ascending order of x, then ascending order of y; Segments are represented by a pair of integers a and b (a<b), that means the segment is connecting vertex a and vertex b (vertices are numbered from 1).

Input

There will be at most 25 test cases. Each case begins with one integer ***n*** (1<=***n***<=100), the number of operations. Each of the following ***n*** lines is formatted as one of:

```
DRAW x1 y1 x2 y2 x3 y3 …
```

or

```
REMOVE x y d
```

In the *draw* operation, you're drawing a poly-line (x1,y1)-(x2,y2)-(x3,y3)-…, note that if the last point equals to the first point, you're actually drawing closed poly-line (but not necessarily a polygon, since the poly-line could be self-intersecting. There will be at least 2 and at most 20 points in a draw operation.

In the *remove* operation, all the line segments whose distance from (x,y) is at most ***d***, are removed ***simultaneously*** (be careful about this!). If no segments satisfy this condition, this operation takes no effect. -1000<=x1,y1,x2,y2<=1000, 0<=d<=10.

There is a special command "END" following the last draw/remove operation. The last test case is followed by a line with n=0, which should not be processed.

Output

For each test case, print the number of vertices, followed by the coordinates of the vertices (one vertex per line), sorted as stated in the problem statement. The next line contains the number of segments, followed by the descriptions of the segments.

Tips: In this problem, your output must match the standard output perfectly. In order to prevent you from printing "-0.00" instead of "0.00", you're encouraged to add to small number (e.g. 1e-6) to each number you print.

Sample Input

```
7
DRAW 10 0 50 0 50 30 10 30 10 0
DRAW 0 10 10 10 10 20 0 20 0 10
DRAW 40 0 40 10 50 10 50 0
REMOVE 45 1 1
DRAW 50 20 20 20 20 30
DRAW 40 20 40 30
REMOVE 42 23 3
END
7
DRAW 0 0 10 0
DRAW 1 0 11 0
DRAW 12 0 15 0
DRAW 5 0 5 1
DRAW 8 0 8 1
REMOVE 5 2 1
DRAW 11 0 12 0
END
0
```

Output for Sample Input

```
14
0.00 10.00
0.00 20.00
10.00 0.00
10.00 10.00
10.00 20.00
10.00 30.00
20.00 20.00
20.00 30.00
40.00 0.00
40.00 10.00
40.00 20.00
50.00 0.00
50.00 10.00
50.00 30.00
15
1 2
1 4
2 5
3 4
3 9
4 5
5 6
6 8
7 8
7 11
8 14
9 10
10 13
12 13
13 14
4
0.00 0.00
8.00 0.00
8.00 1.00
15.00 0.00
3
1 2
2 3
2 4
```

H. Smallest Enclosing Rectangle

There are n points in 2D space. You're to find a smallest enclosing rectangle of these points. By "smallest" we mean either area or perimeter (yes, you have to solve both problems. The optimal rectangle for these two problems might be different). Note that the sides of the rectangle might not be parallel to the coordinate axes.

Input

There will be at most 10 test cases in the input. Each test case begins with a single integer n ($3 <= n <= 100{,}000$), the number of points. Each of the following n lines contains two real numbers x, y ($-10^5 <= x,y <= 10^5$), the coordinates of the points. The points will not be collinear. The last test case is followed by a line with $n=0$, which should not be processed.

Output

For each line, print the area of the minimum-area enclosing rectangle, and the perimeter of the minimum-perimeter enclosing rectangle, both rounded to two decimal places.

Sample Input

```
5
0 0
2 0
2 2
0 2
1 1
5
1 1
9 0
7 10
0 5
2 11
3
5 3
7 2
6 6
4
6 3
9 1
9 6
8 10
0
```

Output for Sample Input

```
4.00 8.00
95.38 39.19
7.00 11.38
27.00 23.63
```

I. Smallest Enclosing Box

There are *n* points in 3D space. You're to find a smallest enclosing box of these points. By "smallest" we mean volume. Note that the sides of the box might not be parallel to the coordinate axes.

Input

There will be at most 10 test cases in the input. Each test case begins with a single integer *n* (4 <= *n* <= 10), the number of points. Each of the following n lines contains three integers *x, y, z* (-100 <= *x,y,z* <= 100), the coordinates of the points. The points will not be coplanar. The last test case is followed by a line with *n*=0, which should not be processed.

Output

For each line, print the volume of the smallest enclosing box, rounded to two decimal places.

Sample Input

```
9
0 0 0
0 2 0
2 0 0
2 2 0
0 0 2
0 2 2
2 0 2
2 2 2
1 1 1
4
0 0 0
1 1 0
1 0 1
0 1 1
5
0 0 0
3 0 1
2 4 3
0 5 7
3 4 9
5
3 2 0
8 9 0
0 9 7
1 9 0
8 6 6
0
```

Output for Sample Input

```
8.00
1.00
71.09
385.48
```

Note

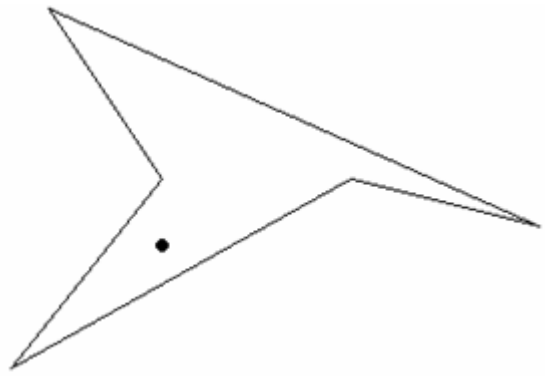
In the fourth example, the vertices of the minimal bounding box are:

```
(9.33269, 4.89595, 7.61936),   (2.62752, 2.26606, 7.37561)
(9.70517, 4.62989, 0.243756),  (3, 2, 0)
(6.70509, 11.6301, 7.24374),   (0, 9, 7)
(7.07757, 11.3641, -0.131862), (0.372395, 8.73416, -0.375618)
```

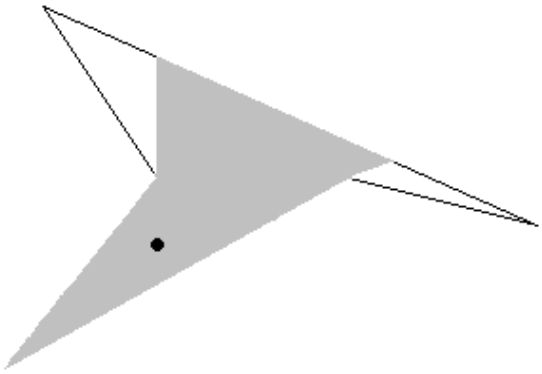
J. A Strange Opera House II

Yesterday evening, I have dreamed of a strange opera house which is in the form of a simple polygon. I was standing on the stage at (x, y) singing "That's All I Ask of You" with my girlfriend - that's our favorite song.

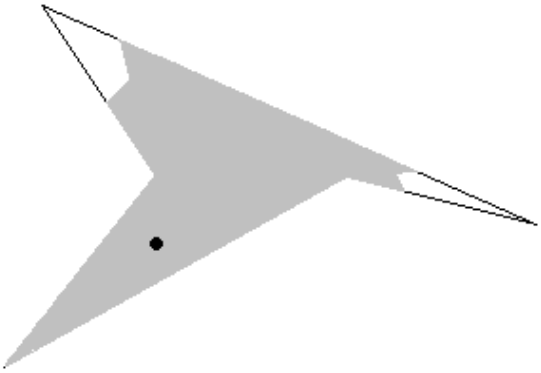
The walls can reflect our voice at most k times. The following 4 figures show how our voice is reflected.



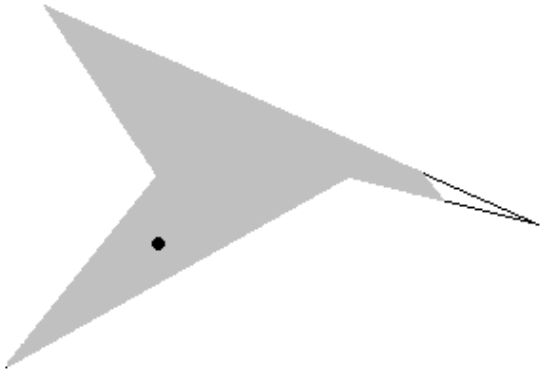
(a) the opera house



(b) our original voice



(c) reflecting the voice once



(d) reflecting the voice twice

I wonder how much area in opera house could hear our song, either directly or indirectly. Can you tell me?

Input

The input consists of at most 10 test cases. Each case contains four integers n, k, x and y ($3 \le n \le 50$, $0 \le k \le 5$), the number of vertices of the opera house, the maximal number of reflections of our voice, and the location of the stage. The stage will never be on a wall. The following n lines each contain two integers x_i and y_i, the coordinates of the vertices. The vertices are arranged either clockwise or counterclockwise. The last case is followed by a single zero, which should not be processed. All the coordinates are integers with absolute values not greater than 1000.

Output

For each test case, print the total area of the places that could hear our song, to two decimal places.

Sample Input

```
5 0 100 135
20 200
200 100
300 125
40 10
100 100
8 1 25 15
0 0
0 20
30 20
30 0
20 0
20 10
10 10
10 0
0
```

Output for Sample Input

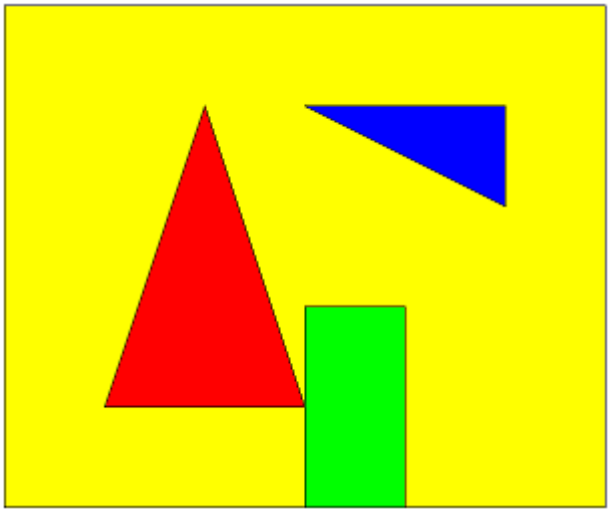
```
9368.00
466.67
```

K. Point Location

Given a partition of the plane into disjoint regions (i.e. a planar subdivision), your task is to determine the region where each query point lies (We use labels to distinguish the regions, see below).

The planar subdivision will be given as a planar straight-line graph (PSLG) with every vertex having at least two adjacent vertices. The two sides of each segment will always belong to different regions.

Here is a sample PSG with 5 regions in the picture below (don't forget there is an exterior infinite region outside the PSGL):



Input

There will be at most 10 test cases. Each test case begins with four integers n, m, p, q ($1 \le n \le 10{,}000$, $1 \le m \le 30{,}000$, $1 \le p \le 20{,}000$, $1 \le q \le 100{,}000$), where n and m are the number of vertices and edges in PSLG, p is the number of labels, q is number of queries. The next n lines contain the coordinates of the vertices (coordinates are integers whose absolute values do not exceed 10^6). The next m lines contain the edges of the PSGL (vertices are numbered 1 to n). There will be no self-loops or parallel-edges. No two edges will be crossing each other at non-endpoints, and each vertex will be connected to at least two edges, and the two sides of each edge will always belong to different regions. The next p lines contain the coordinates of the labels (numbered 1 to p). There will be at most one label strictly inside each region (including the infinite region) of the PSGL. The vertices of the PSGL will be connected. The next q lines contain the coordinates of the query points (coordinates are real numbers whose absolute values do not exceed 10^6). The input terminates with $n=m=p=q=0$, which should not be processed.

Output

For each query, print the label of the region in which the query point lies. If the region does not have a label, print 0. It is guaranteed that for each label and query point, the distance to the boundary of its region will be at least 10^{-4}.

Sample Input

```
14 16 5 5
0 0
30 0
40 0
60 0
60 50
0 50
20 40
10 10
30 10
30 20
40 20
50 30
50 40
30 40
1 2
2 9
9 8
8 7
7 9
9 10
10 11
11 3
3 4
4 5
5 6
6 1
12 13
13 14
14 12
2 3
20 20
10 20
35 10
45 39
1 60
28 11
29 14
34 7
40 38
70 1
0 0 0 0
```

Output for Sample Input

```
1
2
3
4
5
```

L. All-Pair Farthest Points

Given a convex polygon in 2D space, you're to find out the farthest vertex for *each* vertex.

Input

There will be at most 10 test cases in the input. Each test case begins with a single integer n ($3 <= n <= 30{,}000$), the number of points. Each of the following n lines contains two integers x, y ($0 <= x,y <= 10^8$), the coordinates of the vertices, in counter-clockwise order. The last test case is followed by a line with $n=0$, which should not be processed.

Output

For each test case, print n lines, the farthest vertices for each vertex. The vertices in the input are numbered 1 to n. If there are multiple farthest vertex, output the smallest index.

Sample Input

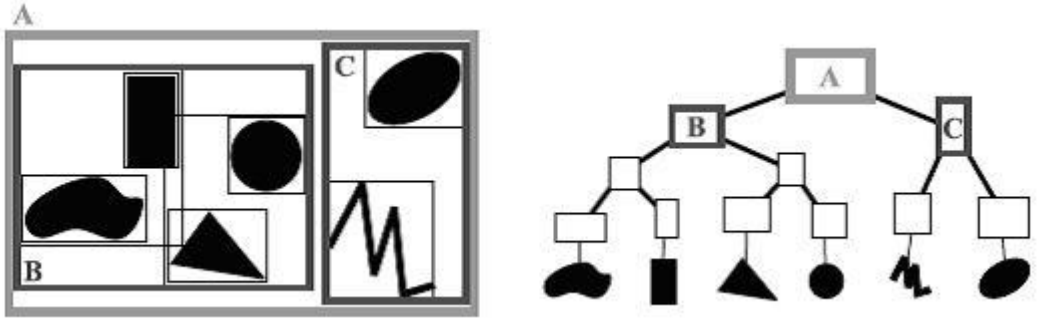
```
3
0 0
1 0
0 10
0
```

Output for Sample Input

```
3
3
2
```

M. Bounding Volume Hierarchy

A **bounding volume hierarchy (BVH)** is a tree structure on a set of geometric objects. All geometric objects are wrapped in bounding volumes that form the leaf nodes of the tree. These nodes are then grouped as small sets and enclosed within larger bounding volumes. These, in turn, are also grouped and enclosed within other larger bounding volumes in a recursive fashion, eventually resulting in a tree structure with a single bounding volume at the top of the tree.



(An example of a bounding volume hierarchy using rectangles as bounding volumes)

BVH can be used as an accelerating structure in ray tracing algorithms. In this problem, you're to implement a static BVH to answer ray-triangle intersection queries efficiently (by static we mean the scene does not change). Note that if the ray hits more than one triangle, only the closest one is output.

For people who have no idea how the BVHs are constructed, here is a simple way: compute the *axis-aligned bounding box* (AABB) of all the objects, and then sort the objects by the coordinates of the "widest" dimension. After that, build the left sub-tree with the first half of objects and the right sub-tree with the remaining, recursively. Note that the AABBs of the objects in the left sub-tree and the right sub-tree might be overlapping, so when traversing the BVH, you need to visit both sub-trees if the ray intersects with the AABBs of both halves of objects. When the number objects is small, we do not split them, so the node becomes a leaf of BVH.

The detail above is just one way of implementing BVH. Another way is to split a node with an axis-aligned *plane*. Sorting is avoided (for each object, it takes constant time to determine which sub-tree it should go to), but you may need to place a single object in both sub-trees. Feel free to employ your own ideas in your program if you like. As long as your program is both correct and fast, it will be accepted.

Tips: An example test case can be downloaded from the contest website.

Input

There will be at most 10 test cases. Each case begins with an integers p ($3 <= p <= 40{,}000$), the number of vertices, Each of the next p lines contains 3 integers, the coordinates of each vertex. The next line is an integer t ($1 <= t <= 80{,}000$), the number of triangles. Each of the next t lines contains 3 integers, the vertex indices of each triangle. Vertices and triangles are both numbered sequentially from 0.

The next line contains an integer q ($1 <= q <= 10{,}000$), the number of queries. Each of the next q lines contains 6 integers, the positions of origin and target of a ray. The origin is guaranteed to be strictly outside any triangle. Note that we're talking about rays, not segments, so "target" is actually just a point that the ray passed through. The coordinates are real numbers with small absolute values. The last test case is followed by a line with $p=0$, which should not be processed.

The triangle meshes are extracted from real-world models.

Output

For each ray, print the ID of the triangle that the ray is hit, and the position of the hit point. If nothing is hit, print a single -1. We've carefully designed the judge I/O, but to further reduce the impact of floating-point errors, the *percentage* of incorrect output lines can be up to 5%, but please do not omit or add any line, since the comparison is done sequentially.

Sample Input

```
4
0 0 0
9 0 0
3 4 0
4 2 7
4
0 1 2
0 1 3
0 2 3
1 2 3
5
-3 0 0 4 2 3
3 -1 -2 4 2 3
6 -3 5 4 2 3
12 7 5 4 2 3
4 2 8 5 3 9
0
```

Output for Sample Input

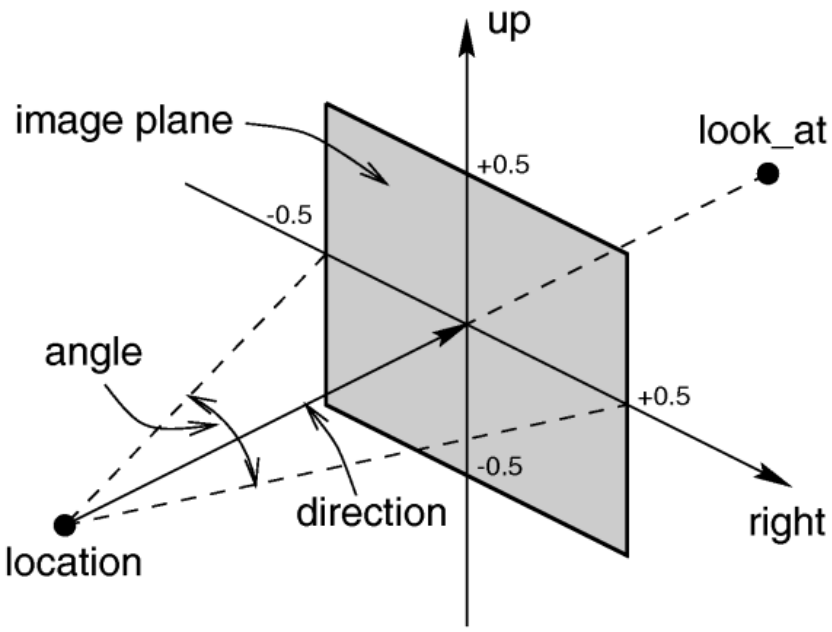
```
2 1.741935 1.354839 2.032258
0 3.400000 0.200000 0.000000
1 4.410256 0.974359 3.410256
3 4.568889 2.355556 3.142222
-1
```

N. A Tiny Raytracer

In this problem, you're to implement a tiny *ray tracer* that renders a scene consisting of a single *point-light* and some triangle meshes.

The camera model

The camera in this problem is a perspective camera located at **camera_pos** (3D point) and looking at **camera_target** (3D vector) with an up vector **camera_up** (3D vector). Its horizontal field of view is **f** degrees, and can produce pictures of **W** x **H** (in pixels).



Don't worry if you could not understand the last paragraph. Just look at the picture above. Imagine a so-called image plane (it's actually a rectangle) in front of the camera. The final image (i.e. the output of this problem!) is the discrete version of the image plane.

Then **camera_up** is the y-axis of the picture, and the ray passing through **camera_pos** and **camera_target** is passing through the center of the image, note that we do not care about how far the target is; only the origin (i.e. **camera_pos**) and direction of the ray is relevant. The horizontal field of view is the angle between the left and right boundaries of the image.

To calculate the projected coordinate (i.e. the corresponding location of a 3D point in the image plane) of an arbitrary point **P** in 3D, simply make a ray from **camera_pos** to **P**, then the intersection of this ray with the image plane is what you want. It's not difficult to show that the position of the image plane does NOT change the projected coordinate of any point (given that the projected coordinate is normalized, as in the picture), so you can choose any plane to be the image plane, as long as its y-axis is **camera_up**, and the ray from **camera_pos** to **camera_target** passes through the center of the image plane. Note that every pixel is a square, so the width/height ratio of the image plane is always W:H.

Raytracing overview

Here is how ray tracing works: for each pixel in the final image, we define a ray that extends from the camera to the **center** of the pixel (please look at the picture above again). We follow this ray out into

the scene and as it bounces off of different objects. The final color of the ray (and therefore of the corresponding pixel) is given by the colors of the objects hit by the ray as it travels through the scene.

In the simplest case, all the objects are neither reflective nor transparent, so every time a ray hits an object, we follow a single new ray from the point of intersection *directly towards* the light source, to calculate the color of the ray (see below). If this new ray is blocked by an object, it is shadowed.

In realistic scenes with like glass and water, we need to consider multiple bounces from objects to handle reflections and refractions. If an object is reflective we simply trace a new *reflected ray* from the point of intersection towards the direction of reflection. The reflected ray is the mirror image of the original ray, pointing away from the surface. If the object is to some extent transparent, then we also trace a *refracted ray* into the surface. If the materials on either side of the surface have different indices of refraction, such as air on one side and water on the other, then the refracted ray will be bent, like the image of a straw in a glass of water. If the same medium exists on both sides of the surface then the refracted ray travels in the same direction as the original ray and is not bent. The exact behavior of refraction is captured by *Snell's law*: the ratio of sines of the angles of incidence and refraction is equivalent to the opposite ratio of the indices of refraction:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1}$$

Naturally, reflected rays and refracted rays themselves can spawn other ways, so we're actually defining a **tree of rays**. In order to prevent the tiny render from being too slow, we need to restrict the height of the tree, so each leaf of the tree is either a ray hitting a non-reflective and non-transparent object (or missing everything!), or the tree depth reached a maximum. If the maximum depth is set to zero, we're disabling reflections and refractions (but direct lighting is still working). In this problem, the maximum depth is always 4. Here is a pseudo-code for this recursive procedure:

```
Color trace_ray(int depth, Ray ray) {
  Color point_color = BLACK, reflect_color = BLACK, refract_color = BLACK;
  Intersection i = get_first_intersection(ray);

  if(i.objID >= 0) { // intersection exists
    double refl = scene.obj[i.objID].refl;
    double refr = scene.obj[i.objID].refr;
    point_color = get_point_color(i) * (1 - refl - refr);
    if(depth < maxdepth && refl > 0)
      reflect_color = trace_ray(depth+1, get_reflected_ray(ray, i)) * refl;
    if(depth < maxdepth && refr > 0)
      refract_color = trace_ray(depth+1, get_refracted_ray(ray, i)) * refr;
  }
  return point_color + reflect_color + refract_color;
}
```

When the depth does not reach the maximum, always trace the reflected ray as long as the object's *reflectiveness* if positive, no matter how small it is. The same is true for the *refractiveness*. The addition of two colors will be defined shortly.

Shading

There are many different ways to determine color at a point of intersection (i.e. the "get_point_color" function above). In this problem, we use *Lambertian shading* (also known as cosine shading), which determines the brightness of a point based on the normal vector at the point and the vector from the point to the light source (recall that there is only one point light). If the two coincide (the surface directly faces the light source) then the point is at full intensity. As the angle between the two vectors increases, as when the surface is tilted away from the light, then the brightness diminishes. This model

is known as cosine shading because that mathematical function easily implements the above effect: it returns the value 1 when given an angle of zero, and returns zero when given a ninety degree angle (when the surface and light source are perpendicular).

In case the result of the dot product is zero, we still may not want that part of the object to be pitch-black. After all, even when an object is completely blocked from a light source, there is still light bouncing around that illuminates it to some extent. For this reason we make use of **ambient_coefficient** and **diffuse_coefficient**, which is just (1-ambient_coefficient), in the following pseudo-code. Note that the triangles are two-sides, so we take the absolute value of the dot product. The variable *object_color* is an attribute of the object (just like **reflectiveness**, **refractiveness** or **index**), which is the color of the object when fully illuminated.

```
double shade;
if(is_shadowed(i)) // check whether the intersection point i is shadowed
  shade = 0;
else
  shade = fabs(Dot(light_vector, normal_vector)); // both vectors are normalized
return object_color * light_color * (ambient_coeff + diffuse_coeff*shade);
```

Recall that the intersection point is shadowed if and only if the segment connecting that point and the light source is blocked by an object (even if the object is reflective or refractive!). This can lead to incorrect shadows, but this is a limitation of this simplified problem, so just ignore this "bug".

In this problem, we use a triple (r, g, b) to represent an RGB color, where 0<=r,g,b<=1. To add two colors, multiply two colors or multiply a color with a constant, just treat the color as a 3D vector. It's not difficult to see that if you strictly follow the rules above, the result of adding two colors will always be a valid color (i.e. each component is between 0 and 1).

Input

There will be at most 20 test cases. Each case begins with one integer n (n<=20), the number of objects, followed by the descriptions of each object.

Each object begins with an integers p (3<=p<=25), the number of vertices. Each of the next p lines contains 3 real numbers, the coordinates of each vertex. The next line is an integer t (1<=t<=50), the number of triangles. Each of the next t lines contains 3 integers, the vertex indices of each triangle. Vertices and triangles are both numbered sequentially from 0. The last line of the object description contains 6 real numbers r, g, b, refl, refr and idx, where r, g, b (0<=r,g,b<=1) describe the object's color (for example, 1.0 1.0 1.0 means white, 0.0 0.0 0.0 mean black), refl and refr are the reflectiveness, refractiveness and media index of this object (refl,refr>=0, refl+refr<=1). Note that the index of vacuum is 1. Each triangle is two-sided, so the vertices are arbitrarily ordered. The next line contains real numbers x, y, z, a real number amb and a color c, that means the point light is located at (x,y,z), with ambient coefficient amb and color c. All the objects are sane (they have a reasonable structure, not just random triangles) and they do not overlap. As a result, when refraction occurs, the ray is either shooting from the vacuum or to the vacuum, but never from one object directly into another object.

The next line contains an integer q, the number of images to render. Each of the next q lines contains 10 real numbers and 2 integers. The first nine numbers are *camera_pos, camera_target* and *camera_up*, the last three numbers are f, W and H(10.286 <= f <= 100.389, 10 <= W <= 200, 10<=H<=200). The last test case is followed by a line with n=0, which should not be processed.

Output

For each test case, output two integer W and H (same as the input), and then H lines, with W colors formatted as RRGGBB (R, G, B values in hexadecimal. Letters can be either lowercase or uppercase) in each line. You can convert the output into a JPG file using the script available if the gift package,

downloadable on the contest website. To reduce the impact of floating-point errors, the percentage of incorrect pixels in each image can be up to 1%. By “incorrect” we mean either r, g, b value differ from the standard output by at least 2.

Sample Input

```
7
4
552.8 0.0   0.0
0.0 0.0   0.0
0.0 0.0 559.2
549.6 0.0 559.2
2
0 1 2
2 3 0
1.0 1.0 1.0 0 0 0
4
556.0 548.8 0.0
556.0 548.8 559.2
0.0 548.8 559.2
0.0 548.8   0.0
2
0 1 2
2 3 0
1.0 1.0 1.0 0 0 0
4
549.6   0.0 559.2
0.0   0.0 559.2
0.0 548.8 559.2
556.0 548.8 559.2
2
0 1 2
2 3 0
1.0 1.0 1.0 0 0 0
4
0.0   0.0 559.2
0.0   0.0   0.0
0.0 548.8   0.0
0.0 548.8 559.2
2
0 1 2
2 3 0
0.0 1.0 0.0 0 0 0
4
552.8   0.0   0.0
549.6   0.0 559.2
556.0 548.8 559.2
556.0 548.8   0.0
2
0 1 2
2 3 0
1.0 0.0 0.0 0 0 0
8
130.0 165.0  65.0
82.0 165.0 225.0
240.0 165.0 272.0
```

```
290.0 165.0 114.0
290.0   0.0 114.0
240.0   0.0 272.0
130.0   0.0  65.0
82.0   0.0 225.0
12
0 6 3
6 4 3
2 0 3
0 2 1
7 6 0
7 0 1
4 5 3
5 2 3
6 5 4
7 5 6
2 5 1
5 7 1
1.0 1.0 1.0 0 0 0
8
423.0 330.0 247.0
265.0 330.0 296.0
314.0 330.0 456.0
472.0 330.0 406.0
423.0   0.0 247.0
472.0   0.0 406.0
314.0   0.0 456.0
265.0   0.0 296.0
12
2 0 3
0 2 1
0 5 3
5 0 4
6 2 3
5 6 3
7 0 1
0 7 4
2 7 1
6 7 2
7 5 4
7 6 5
1.0 1.0 1.0 0 0 0
278.0 548.0 79.5   0.1   1 1 1
1
278 273 -800    278 273 0   0 1 0   54.432   80 60
0
```

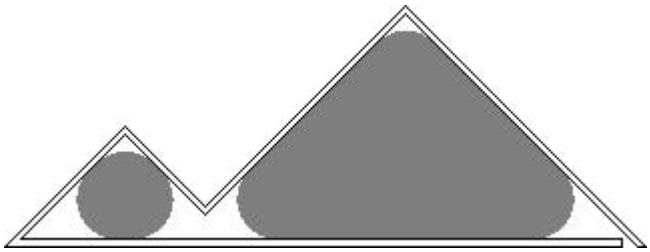
Output for Sample Input

0. The Cleaning Robot

You have a cleaning robot which is a perfect disc (circle with interior) of radius *r*. Given a polygonal obstacle, you need to place the robot in an empty space so that the area that can be cleaned is maximized. During the cleaning process, the robot cannot cross the obstacle, but touching is ok. Note that the robot must be "trapped" by the obstacle (i.e. it cannot move arbitrarily far from the obstacle), otherwise the robot may get lost forever.

The picture below shows a simple scenario. The area that can be cleaned is colored gray.

The picture below shows a more complex scenario. There are two "rooms" for you to choose. Obviously the right one is better.



Warning: this problem is harder than it seems. Please double-check your solution before submitting.

Input

There will be at most 25 test cases. Each case begins with two integers *n*, *r* (1<=*n*<=100, 1<=*r*<=20), the number of vertices of the polygon and the radius of the robot. The next *n* lines contain the coordinates of the polygon vertices (integers whose absolute values do not exceed 1000), given in counter-clockwise or clockwise order. The last test case is followed by a line with *n*=*r*=0, which should not be processed.

Output

For each test case, print the maximal total area that can be cleaned, to two decimal places. If you cannot place the robot, print "`Impossible`" (without quotes).

Sample Input

```
4 4
0 0
1 0
1 1
0 1
10 5
0 0
1 0
1 21
11 21
11 1
2 1
2 0
12 0
12 22
0 22
0 0 0
```

Output for Sample Input

```
Impossible
178.54
```