Colombian Collegiate Programming League
CCPL 2013

Contest 5 -- May 18

Computational Geometry

Problems

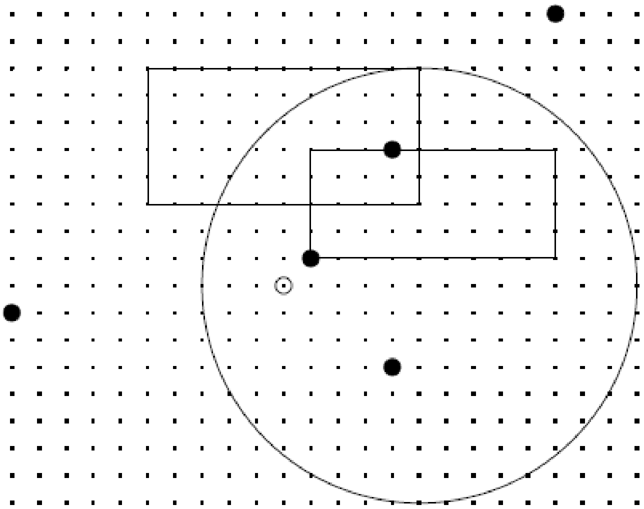
This set contains 10 problems; pages 1 to 18.

(Borrowed from several sources online.)

A - Hitting the Targets

Source file name: `targets.c, targets.cpp` *or* `targets.java`

A fundamental operation in computational geometry is determining whether two objects touch. For example, in a game that involves shooting, we want to determine if a player's shot hits a target. A shot is a two dimensional point, and a target is a two dimensional enclosed area. A shot hits a target if it is inside the target. The boundary of a target is inside the target. Since it is possible for targets to overlap, we want to identify how many targets a shot hits.



The figure above illustrates the targets (large unfilled rectangles and circles) and shots (filled circles) of the sample input. The origin $(0, 0)$ is indicated by a small unfilled circle near the center.

Input

The input contains several test cases. Each test case starts with an integer $1 \leq m \leq 30$ indicating the number of targets. Each of the next m lines begins with the word `rectangle` or `circle` and then a description of the target boundary. A rectangular target's boundary is given as four integers x_1 y_1 x_2 y_2, where $x_1 < x_2$ and $y_1 < y_2$. The points (x_1, y_1) and (x_2, y_2) are the bottom-left and top-right corners of the rectangle, respectively. A circular target's boundary is given as three integers x y r. The center of the circle is at (x, y) and the $0 < r \leq 1000$ is the radius of the circle.
After the target descriptions is an integer $1 \leq n \leq 100$ indicating the number of shots that follow. The next n lines each contain two integers x y, indicating the coordinates of a shot. All x and y coordinates for targets and shots are in the range $[-1000, 1000]$.
The input ends with $m = 0$.

The input must be read from standard input.

Output

For each of the `n` shots, print the total number of targets the shot hits.

The output must be written to standard output.

Sample input

```
3
rectangle 1 1 10 5
circle 5 0 8
rectangle -5 3 5 8
5
1 1
4 5
10 10
-10 -1
4 -3
0
```

Sample output

```
2
3
0
0
1
```

B - Multi-touch gesture classification

Source file name: `multitouch.c`, `multitouch.cpp` *or* `multitouch.java`

With the advent of touch-screen based interfaces, software designers have had to reinvent computer control for finger-based gestures. The most basic task in interpreting a gesture is classifying the touches into one of several possible gestures. This can be difficult because the touches are really blobs of pixels on the screen (not unique points), and gestures involve movement.

We define a *touch* as a connected set of pixels. A pair of pixels is part of the same touch if they are horizontally or vertically adjacent. The *touch point* is the position of a touch. It's defined as the average location of all pixels that are in the set.

A *grip* is the set of all touch points for an image. The *grip point* is the average location of all touch points in the grip. For each touch point we define its *touch vector* as the vector from the grip point to the touch point. A *gesture* is a pair of grips (initial and final), and both grips have the same number of touches. There is a one-to-one correspondence between touches in the initial and final grip. The correspondence is chosen such that the sum of squared distances between corresponding touch points is minimized.

There are three kinds of gestures: pan, zoom, and rotation. There is a distance associated with each kind. The *pan distance* is measured between the two grip points. The *grip spread* is the average distance between each touch point and the grip point. The *zoom distance* is the difference between the two grip spreads. The *touch rotation* is the signed angle (in $(-\pi, \pi)$ radians) between the two touch vectors for corresponding touch points (which is zero if either touch vector has length zero). The *grip rotation* is the average touch rotation. The *rotation distance* is the arc length of the grip rotation along a circle with radius equal the grip spread of the initial grip.

You need to classify the gestures based whether it is a pan, zoom, or rotation. Pick the one which has the greatest associated distance.

Input

Each test case is one pair of images, given side-by-side. Both images are 15×30 pixels, and there will be a single space on each line separating the two images. Each image uses `X` to represent pixels of a touch, and a period (`.`) to represent no touch. Both images have the same number of touches, between 1 and 5. The input ends with an end of file.

The input must be read from standard input.

Output

For each test case, print the number of touches and the type of gesture (pan, zoom, or rotate) as well as the direction of the gesture for zoom or rotate. The direction for a zoom is `in` if the grip spread of the final grip is smaller than that of the initial grip, otherwise it is `out`. The direction of a rotate is either `clockwise` or `counter-clockwise`. All distance comparisons that affect the output will be robust against numerical error of 10^{-5}.

The output must be written to standard output.

Sample input

```
.............................. ..............................
.............................. ..............................
....XXXX...................... ..............................
....XXXX...................... ............XXXX..............
....XXXX...................... .............XXX..............
.............................. ..............................
.............................. ..............................
.............................. ..............................
.............................. ..............................
.............................. ..............................
.............................. ..............................
.............................. ..............................
.............................. ..............................
.............................. ..............................
.............................. ..............................
.............................. ..............................
.............................. ..............................
.............................. .................XX...........
.............................. .......XXX......XXXX..........
.............................. ......XXXX........X...........
........XXX....XXX............ ........X.....................
.........XXX...XX............. ..............................
.............................. ..............................
............XX................ ..............................
...........XXX................ ..............................
.............................. ..............................
.............................. ..............XX..............
.............................. .............XXXX.............
.............................. ...............XX.............
.............................. ..............................
.............................. ..............................
.............................. ..............................
........XX.................... ..............................
........XX.................... .................XXXXX........
.............................. ..................XXX.........
.............................. ..............................
..XXX................XXX...... .....XX.......................
...XX.................XXX..... .....XXX......................
.............................. ..............................
.............................. ..............................
.............................. .....XXX......................
.........XXX.................. ......XX......XXXX............
..........X................... ..............XXXX............
.............................. ..............................
.............................. ..............................
```

Sample output

```
1 pan
3 zoom out
4 rotate counter-clockwise
```

C - Cinoc Mountain

Source file name: `cinoc.c, cinoc.cpp` *or* `cinoc.java`

Cinoc is a mountain region that offers a spectacular natural environment. The area is sometimes called the last wilderness area, but in reality the Cinoc Mountains area is a cultural landscape. The mountains of Cinoc are also an important recreational area for people from around the world and have a particular characteristic: all of them have conic form (a right cone with circular base) and are placed on a completely flat land.

In order to attend any possible emergency, the administration of the Cinoc National Park requires to install two towers of communication, to connect two distant places in the park. The connection between the towers is only possible if there is a *line of sight* between the two towers, that is to say, the highest point of one tower can be seen from the highest point of the other tower. Also, to avoid interferences, the distance from any point in the line of sight to any point on the territory must be greater than zero.

Your task is to write a program that, given a map of the park and the description of the two towers, decides if the connection is possible or not.

Input

The input consists of several test cases. For each test case: the first line contains an integer number k, $0 \leq k \leq 50$, and is followed by $k + 2$ lines. In each case the number k is the number of mountains in Cinoc Mountains region. Each of the next k lines contains four positive, integer numbers xm, ym, hm and rm, $0 \leq xm, ym, hm, rm \leq 10000$, describing a conic mountain ((xm, ym, hm) denotes the coordinates of the top of the mountain and rm denotes the radius of the base). The last two lines contain the information associated with the towers: three numbers per line: xt, yt and ht, $0 \leq xt, yt, ht \leq 10000$, ($(xt, yt, ht)$ denotes the location of the tower's top). The last case is followed by a single line containing -1.

The input must be read from standard input.

Output

For each test case, if there exists a valid line of sight between the corresponding top points of the towers, the following line must be printed:
`Yes`
In other case, the following line must be printed:
`No`
The answers for the different cases must preserve the order of the input.

The output must be written to standard output.

Sample Input

```
2
10 10 2 5
10 2 2 2
10 2 3
2 2 2
1
10 2 2 2
5 2 1
15 2 1
-1
```

Sample Output

```
Yes
No
```

D - Skyline

Source file name: `skyline.c, skyline.cpp` *or* `skyline.java`

Last time I visited Shanghai I admired its beautiful skyline. It also got me thinking, "Hmm, how much of the buildings do I actually see?" since the buildings wholly or partially cover each other when viewed from a distance.
In this problem, we assume that all buildings have a trapezoid shape when viewed from a distance. That is, vertical walls but a roof that may slope. Given the coordinates of the buildings, calculate how large part of each building that is visible to you (i.e., not covered by other buildings).

Input

The input contains several test cases. The first line of each test case contains an integer, N ($2 \leq N \leq 100$), the number of buildings in the city. Then follow N lines each describing a building. Each such line contains 4 integers, x_1, y_1, x_2, and y_2 ($0 \leq x_1 < x_2 \leq 10000$, $0 < y_1, y_2 \leq 10000$). The buildings are given in distance order, the first building being the one closest to you, and so on.
The input ends with $N = 0$.

The input must be read from standard input.

Output

For each building, output a line containing a floating point number between 0 and 1, the relative visible part of the building, each one listed in the order that it was given. The absolute error for each building must be less than 10^{-6}.

The output must be written to standard output.

Sample input

```
4
2 3 7 5
4 6 9 2
11 4 15 4
13 2 20 2
0
```

Sample output

```
1.00000000
0.38083333
1.00000000
0.71428571
```

E - Playground

Source file name: `playground.c, playground.cpp` *or* `playground.java`

George has $K \leq 20$ steel wires shaped in the form of half-circles, with radii $a_1, a_2, \ldots, a_K$. They can be soldered (connected) at the ends, in any angle. Is it possible for George to make a closed shape out of these wires? He does not have to use all the wires.
The wires can be combined at any angle, but may not intersect. Beware of floating point errors.

Input

Each test case consists of a number $0 < K \leq 20$ on a line by itself, followed by a line of K space-separated numbers $a_1, a_2, \ldots, a_K$. Each number is in the range $0 < a_i < 10^7$ and has at most 3 digits after the decimal point.
The input will be terminated by a zero on a line by itself.

The input must be read from standard input.

Output

For each test case, there should be one word on a line by itself; "YES" if it is possible to make a simple connected figure out of the given arcs, and "NO" if it isn't.

The output must be written to standard output.

Sample input	**Sample output**

`1`
`4.000`
`2`
`1.000 1.000`
`3`
`1.455 2.958 4.424`
`7`
`1.230 2.577 3.411 2.968 5.301 4.398 6.777`
`0`
	`NO`
	`YES`
	`NO`
	`YES`

F - Flight Control

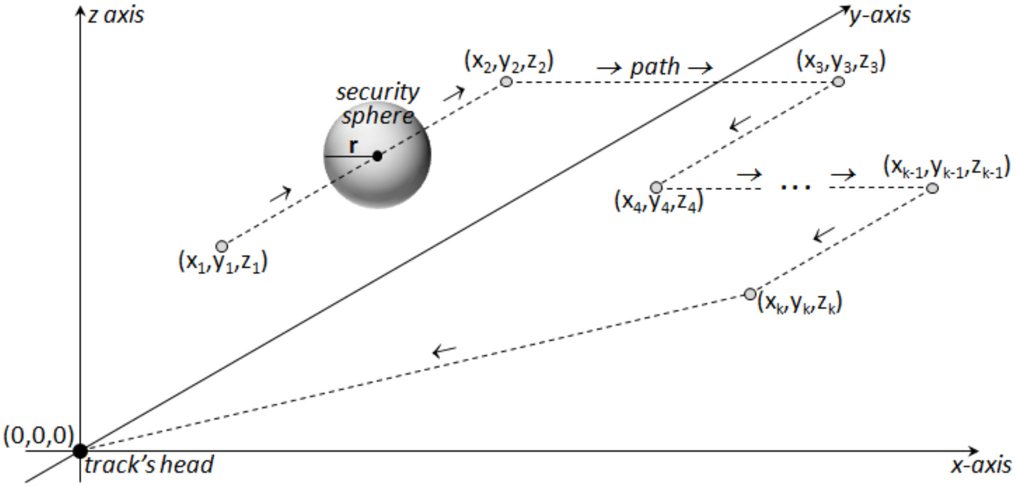
Source file name: `flight.c`, `flight.cpp` *or* `flight.java`

Air traffic controllers are the people who expedite and maintain a safe and orderly flow of air traffic in the global air traffic control system. The position of the air traffic controller is one that requires highly specialized skills. Because controllers have an incredibly large responsibility while on duty, this profession is, according to Wikipedia, "regarded around the world as one of the most difficult jobs today, and can be notoriously stressful depending on many variables (equipment, configurations, weather, traffic volume, human factors, etc.)".

An air traffic controller has access to the following information for each aircraft on the screen:

- *size*: a positive integer number r indicating the radius (measured in meters) of a *security sphere* whose center always is the current position of the aircraft;
- *speed*: a positive integer number s indicating the constant speed (measured in meters per second) of the aircraft along its route;
- *route*: a sequence of points $(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_k, y_k, z_k)$ with integer coordinates (measured in meters) in the three-dimensional Cartesian plane, indicating the path followed by the aircraft before it returns to the head of the *track* which is located at the point $(0, 0, 0)$.

Each aircraft begins its journey at the position (x_1, y_1, z_1) and then, it directly flies in a straight line at constant speed from (x_1, y_1, z_1) to (x_2, y_2, z_2), ..., from $(x_{k-1}, y_{k-1}, z_{k-1})$ to (x_k, y_k, z_k), and finally from (x_k, y_k, z_k) to $(0, 0, 0)$, where each position is relative to the head of the track. After the aircraft arrives at the head of the track, it disappears from the controller's screen.



On a day-to-day basis, air traffic controllers deal with conflict detection and resolution. Therefore, for an air traffic controller is very important to have an alarm system to indicate the specific points where it must take corrective actions to prevent accidents.

It is noteworthy that, geometrically speaking, a *conflict warning* occurs when the security spheres of two aircrafts touch. Formally, a *conflict warning* begins when two aircrafts approach at a distance less than or equal to the sum of the radius of their security spheres, is maintained while this condition is satisfied, and ends when their security spheres stop touching (i.e., when the distance between both is greater than the sum of the radius of their security spheres). Distances are measured with an acceptable error threshold $\varepsilon > 0$.

Despite years of effort and the billions of dollars that have been spent on computer software designed to assist air traffic control, success has been largely limited to improving the tools at the disposal of the controllers. However, today you have the chance to improve the impact of computer software in the air traffic control world.

You have been hired by the *International Center of Planning Control* (ICPC) to determine the quality of the traffic routes defined by the *Aircraft Controller Management* (ACM), through the measurement of the number of dangerous situations that should fix the air traffic controller. Your task is to write a program that, given the information of two aircrafts, determines the number of different conflict warnings that would arise if both aircrafts follow the scheduled route starting at the same time and finishing at the track's head.

Input

The first line of the input contains the number of test cases. Each test case specifies the information of the two studied aircrafts, where each aircraft is described as follows:

- The first line contains three integer numbers r, s and k separated by blanks ($1 \leq r \leq 100$, $1 \leq s \leq 1000$, $1 \leq k \leq 100$), where r is the radius of the security sphere (in meters), s is the speed (in meters per second), and k is the number of points that define the route of the aircraft.
- Each one of the next k lines contains three integer numbers x_i, y_i, and z_i separated by blanks ($-10^4 \leq x_i \leq 10^4$, $-10^4 \leq y_i \leq 10^4$, $0 \leq z_i \leq 10^4$), describing the coordinates (in meters) of the i-th point on the route of the aircraft ($1 \leq i \leq k$).

For each route, you may assume that either $x_i \neq x_{i+1}$, $y_i \neq y_{i+1}$ or $z_i \neq z_{i+1}$ for all $1 \leq i < k$, and that either $x_k \neq 0$, $y_k \neq 0$ or $z_k \neq 0$. The acceptable error threshold is $\varepsilon = 10^{-10}$ meters.

The input must be read from standard input.

Output

For each test, print one line informing the number of different conflict warnings.

The output must be written to standard output.

Sample Input

```
7
20 300 2
10000 1000 5000
1000 100 500
20 100 3
10000 1000 2000
1000 100 500
100 0 0
20 300 2
0 10000 5000
0 -10000 5000
20 300 3
10000 0 5010
-10000 0 5010
-8000 1000 3010
20 300 2
0 10000 5000
0 -10000 5000
20 300 2
10000 0 5010
-10000 0 5010
25 200 2
3000 6000 3000
4000 5000 3000
25 200 2
3000 6000 3005
4000 5000 3005
20 300 2
5000 4000 3000
4000 5000 3000
20 300 2
3000 6000 3005
4000 5000 3005
10 100 1
-1000 0 0
10 100 2
1000 21 0
-1000 21 0
10 100 1
-1000 0 0
10 100 2
1000 20 0
-1000 20 0
```

Sample output

```
0
1
2
1
1
0
1
```

G - Galactic Warlords

Source file name: `galactic.c, galactic.cpp` *or* `galactic.java`

Will the galaxy see peace at last? All the warlords have gathered to divide all of space between themselves. The negotiations have come quite far and the warlords have finally agreed on a peaceful way of deciding who gets what. The 2-dimensional galactic map must first be divided into sectors by splitting it along a set of infinite lines. The warlord with the largest battle fleet will choose one sector, then the warlord with the second largest fleet will choose some other sector and so on, until everyone has gotten a sector. This is then repeated until there are no sectors left.

Different sets of lines have been suggested and it is up to you to present these alternatives to the meeting. To make sure that there will be peace, you are ready to modify the suggestions slightly. You have some experience with warlords and know that no warlord will settle for less space than anyone else, so for there to be peace, all of them must get the exact same area on the map.

Since space is infinite, so is the map. Some sectors will therefore have infinite area, so that is the amount of space everyone will want. How many extra lines will you have to add to make sure each warlord can get at least one sector with infinite area?

Input

The input consists of several test cases. The first line of each test case contains two positive integers W and N, ($1 \leq W, N \leq 100$) denoting the number of warlords and the number of lines in the suggested division of space. This is followed by N lines each containing four integers x_1, y_1, x_2 and y_2, each with an absolute value no higher than 10000. This means that one line is intersecting the two points (x_1, y_1) and (x_2, y_2) on the galactic map. These two points will not be the same.

The input ends with $W = N = 0$.

The input must be read from standard input.

Output

Output the number of lines you will have to add to this suggestion to satisfy all warlords.

The output must be written to standard output.

Sample input	**Sample output**

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

H - Hard Evidence

Source file name: `evidence.c, evidence.cpp` *or* `evidence.java`

The young reporter Janne is planning to take a photo of a secret government installation. He needs to obtain evidence of the many serious crimes against good sense that are being committed there, so as to create a scandal and possibly win a Pulitzer. Unfortunately, the base is surrounded by a high fence with high voltage wires running around. Janne does not want to risk being electrocuted, so he wants to take a photo from outside the fence. He can bring a tripod as high as the fence to take a photo, so if he wants he can stand right beside the fence and take his picture.

The secret installation is a convex polygon. The fence has a form of a circle. Of course Janne wants to make a photo with maximal possible detail level. The detail level of the photo depends on the view angle of the base at the point from which the photo is taken. Therefore he wants to find a point to maximize this angle.

Input

The input contains several test cases. The first line of each test case contains two integer numbers: n and r, the number of vertices of the polygon and the radius of the fence ($3 \leq n \leq 200$, $1 \leq r \leq 1000$). The following n lines contain two real numbers each, respectively, the coordinates of the vertices of the polygon listed in counterclockwise order. It is guaranteed that all vertices of the polygon are strictly inside the fence circle and that the polygon is convex. The center of the fence circle is located at the origin $(0, 0)$.

The input ends with $n = r = 0$.

The input must be read from standard input.

Output

Output the maximal view angle a for the photo ($0 \leq a < 2\pi$). Any answer with either absolute or relative error smaller than 10^{-6} is acceptable.

The output must be written to standard output.

Sample input	Sample output

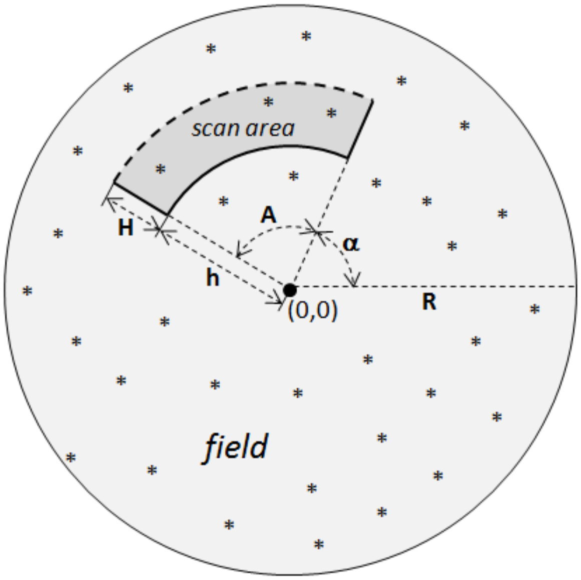
`4 2`
`-1.0 -1.0`
`1.0 -1.0`
`1.0 1.0`
`-1.0 1.0`
`1.000 1.000`
`0 0`
	`1.5707963268`

I - Inspecting Radars

Source file name: `radar.c, radar.cpp` *or* `radar.java`

Radars Inc. is a worldwide renowned radar maker, whose excellent reputation lies on strict quality assurance procedures and a large variety of radar models that fit all budgets. The company hired you to develop a detailed *inspection* that consists of a sequence of E *experiments* on a specific *surveillance model*.

There is a *field* represented with a polar coordinate plane that contains N objects placed at positions with integer polar coordinates. The inspected model is located at the origin $(0, 0)$ of the field and can detect objects at a distance less than its *detection range* R through a *scan area* defined by four adjustment parameters α, A, h, and H, whose meaning is illustrated with the following figure:



Formally, the *scan area* of the model is the region described by the set of polar points

$$\{(r, \theta) |\, h \le r < h+H,\ \alpha \le \theta \le \alpha+A\}$$

α, A, h and H are four integer values where:

- α specifies the *start angle* of the radar's scan area ($0 \le \alpha < 360$);
- A specifies the *opening angle* of the radar's scan area ($0 \le A < 360$);
- h gives the *internal radius* of the radar's scan area ($0 \le h < R$); and
- H gives the *height* of the radar's scan area ($1 \le H \le R$).

An object placed at (r, θ) will be displayed by the model if $h \le r < h+H$ and $\alpha \le \theta \le \alpha+A$, where the last inequality should be understood modulo 360^o (i.e., adding and comparing angles in a circle).

Given N objects placed on the field, you must develop an inspection of the surveillance model through the implementation of E experiments with specific parameterizations. For each experiment you have to find the maximal number of objects on the field that the radar should display if the parameters α ($0 \leq \alpha < 360$) and h ($0 \leq h < R$) are free to set (as integer numbers), and the parameters H ($1 \leq H \leq R$) and A ($0 \leq A < 360$) are given.

Input

The input consists of several test cases. Each test case is described as follows:

- A line with two integer numbers N and R separated by blanks, representing (respectively) the number of objects located on the field and the detection range of the model ($1 \leq N \leq 10^4$, $2 \leq R \leq 10^2$).
- Each one of the following N lines contains two integer numbers r_i and θ_i separated by blanks, specifying the integer polar coordinates (r_i, θ_i) of the i-th object ($1 \leq r_i < R$, $0 \leq \theta_i < 360$, $1 \leq i \leq N$).
- The next line has an integer number E indicating the number of experiments of the inspection ($1 \leq E \leq 10^2$).
- Each one of the following E lines contains two integer numbers H_j and A_j separated by blanks, representing (respectively) the fixed height and the fixed opening angle that parameterize the j-th experiment ($1 \leq H_j \leq R$, $0 \leq A_j < 360$, $1 \leq j \leq E$).

For each test case you can suppose that there are not two different objects placed at the same integer polar coordinate. The last test case is followed by a line containing two zeros.

The input must be read from standard input.

Output

For each test case of the input, print E lines where the j-th line contains the maximal number of objects on the field that the radar should display according to the parameterization given for the j-th experiment ($1 \leq j \leq E$).

The output must be written to standard output.

Sample Input

```
6 100
15 7
15 60
40 15
50 15
45 30
45 90
2
2 1
100 359
9 100
15 7
15 60
40 15
50 15
45 30
45 90
40 45
50 45
78 100
6
100 359
11 30
10 30
11 29
5 30
11 10
0 0
```

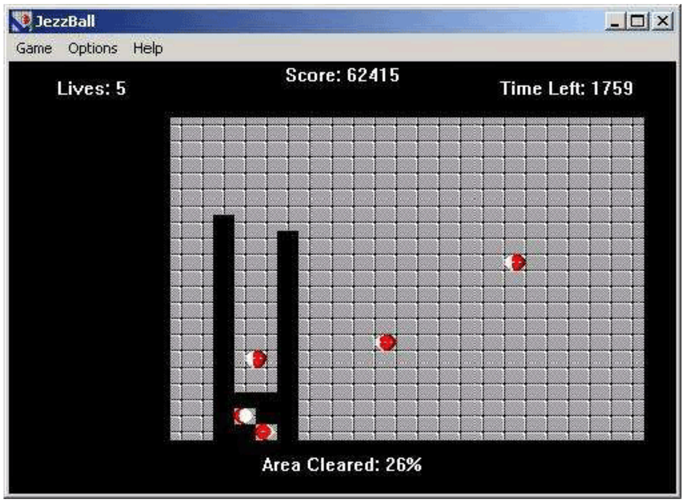
Sample output

```
1
6
9
5
3
3
2
2
```

J - Jezzball

Source file name: `jezzball.c, jezzball.cpp` *or* `jezzball.java`

"JezzBall is a computer game in which red-and-white 'atoms' bounce about a rectangular field of play. The player advances to later levels (with correspondingly higher numbers of atoms and lives) by containing the atoms in progressively smaller spaces, until at least 75% of the area is blocked off." (wikipedia.org)



The picture is a screenshot from the original game, where the player has already covered some space (the black part). In this problem we will consider a slightly different, non-discrete, version of the game. That is, while the length unit is still pixels, you should treat them as non-discrete in the sense that all objects can be at non-integer coordinates and all movements are continuous. The size of the playing field will be 1024 × 768 pixels. The atoms that bounce around will be infinitely thin (and not round balls like in the screenshot). The atoms will move at a constant speed and only change direction when hitting the edge of the playing field (x-coordinate 0 and 1024 or y-coordinate 0 and 768), where they bounce without loss of energy. The atoms do not hit each other.

The player can divide the playing field in two by shooting a horizontal or vertical ray from (in this problem) a fixed point on the playing field. The ray will then extend in both directions simultaneously (up and down for vertical rays, or left and right for horizontal rays) at a uniform speed (in this problem always 200 pixels per second). The rays will also be infinitely thin. If no atom touches any part of the ray while it's still being extended, the field has sucessfully been divided. Otherwise the player loses a life.

If an atom touches the endpoint of an extending edge, this will not be counted as a hit. Also, if an atom hits the ray at the same instant it has finished extending, this will also not count as a hit. Write a program that determines the minimum time the player must wait before he can start extending a ray so that an atom will not hit it before the ray has been completed.

Input

The input consists of several test cases. Each test case starts with a line containing a single integer n, the number of atoms ($1 \leq n \leq 10$). Then follows a line containing two integers, x and y, the position where the two ray ends will start extending from ($0 < x < 1024$, $0 < y < 768$). Then n lines follow, each containing four integers, x, y, v_x and v_y describing the initial position and speed of an atom ($0 < x < 1024$, $0 < y < 768$, $1 \leq |v_x| \leq 200$, $1 \leq |v_y| \leq 200$). The speed of the atom in the x direction is given by v_x and the speed in the y direction is given by v_y. All positions in each input will be distinct. The input is terminated by a case where $n = 0$, which should not be processed. There will be at most 25 test cases.

The input must be read from standard input.

Output

For each test case, output the minimum time (with exactly 5 decimal digits) until the player can extend either a horizontal or vertical ray without an atom colliding with it while it is being drawn. The input will be constructed so that the first time this occurs will be during an open interval at least 10^{-5} seconds long. If no such interval is found during the first 10000 seconds, output "Never" (without quotes).

The output must be written to standard output.

Sample input	**Sample output**

`3`
`700 420`
`360 290 170 44`
`900 150 -53 20`
`890 100 130 -100`
`4`
`10 10`
`1 1 192 144`
`513 385 192 144`
`1023 767 -192 -144`
`511 383 -192 -144`
`0`
	`2.80094`
	`Never`