Problem A:
Save the water

Source file: `water{.c| .cpp| .java}`
Input file: `water.in`

GCC countries face an ever growing challenge on water management and preservation. Water is becoming scarce with every passing year due to global warming. Officials believe that careful management of the water quantities that rains yield, can help reduce the severity of the problem.

However, the main issue is rain water is susceptible to evaporation as it has to be stored in open reservoirs. This evaporation can be reduced if the reservoirs have a smaller surface area that can be heated by the sun. And here is where your role begins: a number of rectangular reservoirs are available, each with its own depth, width and length. Your task is to calculate which reservoirs to use to minimize the water evaporation given a specific amount of water that has to be kept in them.

A reservoir can be filled to the top or used partially. However, the total surface of the water that is exposed to the sun has to be minimized.

For example, let's assume that we have the following three reservoirs:

Then 18000 m^3 of water can be best stored by using reservoirs #2 and #3, exposing only 1000m^2 to the sun. If reservoir #1 were to be used, then 5000m^2 would be exposed instead.

Input

The input starts with a line with a single integer $N<100$, which is the number of cases to be examined. Each of the N cases, starts with a line containing the number of reservoirs $M<16$ and the total water capacity $W<100000$ m^3, separated by a single space. M lines follow, each containing 3 integers *depth*, *width* and *length* (all in meters) separated by a single space, that describe the dimensions of one reservoir. Dimensions are positive integers smaller than 1000. W is always smaller than the total reservoir capacity.

Output

For each test case, you should output the best total surface of the reservoirs that need to be used, in a separate line.

Sample Input

```
1
15 55000
10 5 100
20 2 30
15 1 25
10 50 10
2 25 30
11 10 25
11 50 10
22 25 30
12 10 25
11 50 10
21 25 30
15 10 25
10 70 10
20 50 30
150 1 25
```

Output for Sample Input

```
2585
```

Problem B:
Trouble in Sales Land

Source file: `sales{.c| .cpp| .java}`
Input file: `sales.in`

The travelling sales person is a well-known problem in computer science: a set of destinations/cities that someone has to visit exactly once, returning to the originating city.

The travelling costs between the cities are typically given in terms of time, distance or some monetary expense.

In a realistic scenario though, the cost of travelling between the cities is time dependent: it varies based on the time-of-day. During peak traffic hours one would expect to get a much more costly "journey". This is the setting of the problem you have to solve: given a set of cities and three different time costs for each road connecting them (one for 8:00-15:59, one for 16:00-23:59 and one for 00:00 – 7:59), you have to find the smallest possible route time-wise, that goes through all the cities.

You can assume that there is no cost for visiting a city (business is bad now with the recession) and that the cost of using a particular "road" connecting two cities A and B, is determined when the journey starts from A. The sales person starts off his/her journey at 8:00am.

Input

The first line contains an integer $N<100$, representing the number of cases. Each input case starts with a line containing three integers separated by a single space: $V<20$, E and S, where V is the number of cities (numbered from zero), E is the number of streets connecting the cities and S is the starting city. E lines follow, one for each street, containing 5 non-negative numbers separated by single space: V1, V2, T1, T2 and T3. V1 and V2 are the indices of the connected cities; T1 is the journey time in minutes when departure is between 0:00 and 7:59. T2 is the journey time in minutes when departure is between 8:00 and 15:59. Finally, T3 is the journey time in minutes when departure is between 16:00 and 23:59. All times are less than 1000 minutes.

Output

For each test case, the output should consist of a line containing the sequence of cities visited, including the starting city. The line should end with the duration of the journey in minutes, separated from the city sequence with the " : " characters. If a journey is not possible, then "`No solution exists`" should be output instead.

Sample Input

```
2
5 8 0
0 1 1 2 1
0 3 2 2 1
0 2 3 2 1
2 1 4 3 1
2 4 5 4 1
2 3 6 5 1
3 4 7 1 1
1 4 8 2 5
14 20 1
0 1 60 180 60
1 2 60 180 60
1 3 60 180 60
2 3 60 180 60
3 4 60 180 60
3 5 60 180 60
4 5 60 180 60
5 6 60 180 60
5 7 60 180 60
6 7 60 180 60
7 8 60 180 60
7 9 60 180 60
8 9 60 180 60
9 11 60 180 60
9 10 60 180 60
10 11 60 180 60
11 12 60 180 60
11 13 60 180 60
12 13 60 180 60
0 12 60 180 60
```

Output for Sample Input

```
0 2 1 4 3 0 : 10
1 0 12 13 11 10 9 8 7 6 5 4 3 2 1 : 1200
```

Problem C:
Black or White

Source file: `bw.{c | cpp | java}`
Input file: `bw.in`

The neighborhood of Grand Prairie City (GPC) is expected to grow fast. All the old villas in the neighborhood are either black or white. To maintain this pattern, the city council has imposed some strange rules in painting new villas in the neighborhood:

1) All the new villas must be painted either black or white
2) When a new villa is built, it should be painted with the same color of its nearest old villa.

The neighborhood of GPC has *N* old villas. Your job is to determine the color of any new villa using rules 1 and 2. In case of a tie (i.e., if more than one nearest), choose the old villa that comes first in the input. Each villa is considered as a point (*x*, *y*) in a two dimensional coordinate system (*X*, *Y*). The distance between two villas *a*, *b* is computed using the following formula:

$$\boldsymbol{d_{ab} = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}}$$

Where d_{ab} is the distance between villa *a* and villa *b*; and the coordinates of villas *a*, and *b* are (x_a, y_a), and (x_b, y_b), respectively.

Input

The input starts with a line containing the integer *N* (0 < *N* <= 1,000), the number of old villas in the neighborhood. This follows *N* lines, each line containing the description of an old villa. An old villa is described by three variables *x*, *y* and *c*, separated by spaces. (*x*, *y*) is the coordinate of the villa, where *x* and *y* are both nonnegative real numbers (0 <= x, y <= 10,000), and *c* is the color of the villa, which is either "W" or "B", denoting white or black, respectively. After the old villa descriptions, there will be one or more (at most 100) lines of input, each line containing the *x*, *y* coordinate of a new villa. The input will be terminated by a line containing -1.

Output

For each new villa of the input, you are to output the color of the villa, printing "W" for white and "B" for black.

Sample Input

```
4
1 2 W
2 0 B
2 4 W
3 2 B
2 2
3 4
2.5 2.5
-1
```

Output for Sample Input

```
W
W
B
```

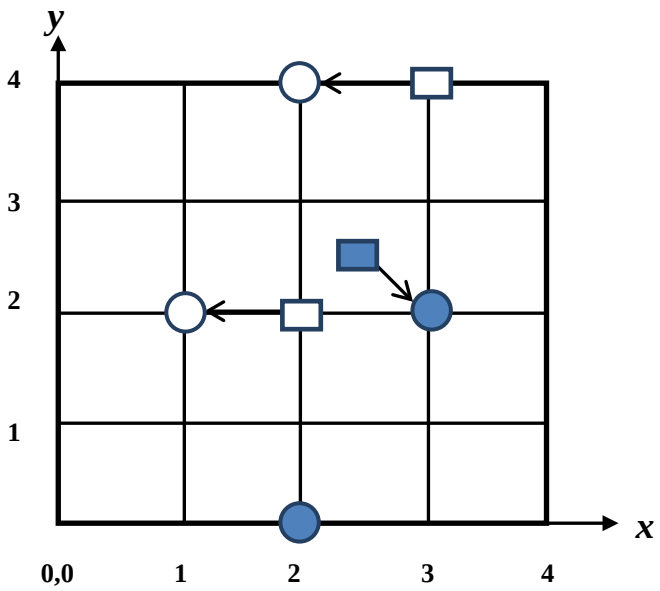


Figure C.1 shows the villas in the sample input. The circles are the old villas and the rectangles are the new villas. Also, the white circles and rectangles are “White” and the shaded circles and rectangles are “Black”. The arrow from each new villa points to its nearest old villa. In case of the new villa at (2,2), there is a tie (two old villas are nearest). So, the old villa that comes first in the input (i.e., 1,2) is chosen as the nearest neighbor, and hence its color is white.

Problem D:
Alien Invasion

Source file: `alien.{c | cpp | java}`
Input file: `alien.in`

Aliens have landed on planet Xi. And this time they came with the intention of invasion, not excursion! They are going to wipe out the intelligent life-forms of planet Xi (i.e., the Xians). However, the Xians may have a last chance for survival. The reason is to be explained shortly. The surface of Xi is a perfect square, divided into N x N estates. Each estate has a unit square area. An estate may have up to eight adjacent estates (shown in the illustration). However, each estate has different *height*, which reveals the vital weakness of the aliens. The aliens can move from one estate *x* to an adjacent state *y* only if the height of *y* is less than or equal to the height of *x*. The aliens have landed on an estate that they call Apparently Unoccupied Den (AUD). Their goal is to invade the whole planet in minimum amount of time by moving from one estate to all adjacent ones, starting from the AUD estate. However, since they cannot move to a higher adjacent estate, they may not succeed in occupying all the estates – which gives the Xians the only hope for survival.

Input

The first line of input shall contain an integer *K*, the number of planets to consider. Each planet's description will start with an integer *N* ($0 < N <= 1000$) on a line by itself, denoting the dimension of the square world. The planet will be then given as a matrix *C* of *N* x *N* non-negative integers. The number at row *i*, col *j* (i.e., $C_{i,j}$) of the matrix ($0 <= i, j < N$) denotes to the height the corresponding estate ($0 <= C_{i,j} <= 100$). Each row of the matrix (i.e., $C_{i,0}$, …, $C_{i,N-1}$) will be in one line of the input. Two adjacent integers in each input line will be separated by a space. After this matrix, there will be a single line containing two integers *r* and *c* ($0 <= r, c <$ N), denoting the row# and col# of the AUD.

Output

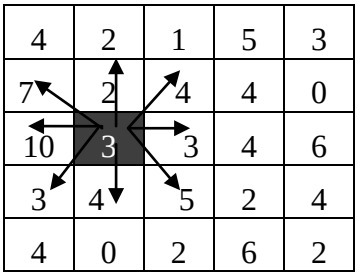
For each planet, you are to output two numbers, namely, the maximum number of estates that the aliens can invade and the minimum amount of time required to invade those estates. Assume that the aliens need one unit of time to move from one estate to all the adjacent estates (off course maintaining the constraint mentioned above). Also assume that there will always be enough aliens to invade all the adjacent estates from a given estate.

Sample Input

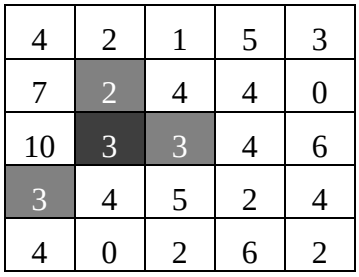
```
2
5
4 2 1 5 3
7 2 4 4 0
10 3 34 6
3 4 5 2 4
4 0 2 6 2
2 1
2
10 2
5 9
0 1
```

Output for Sample Input

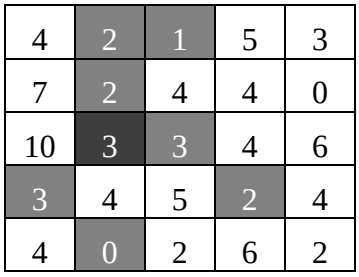
```
10 3
1 0
```



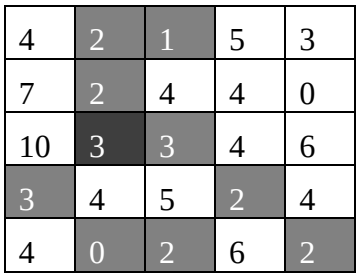
Time = 0.
Total estates invaded = 1.
The shaded estate (2, 1) is the AUD.
Arrows point to the adjacent estates.



Time = 1.
The shaded estates are invaded.
Total estates invaded = 4.



Time = 2.
Total estates invaded = 8.



Time = 3.
Total estates invaded = 10.
That's it! Aliens can't invade any other estates.

Figure D.1 Illustration of alien invasion in the first planet of the sample input.

Problem E:
Sum 'em up

Source file: `sum.{c | cpp | java}`
Input file: `sum.in`

Two integers *x* and *y* are relatively prime if *x* and *y* do not have any common divisor other than 1. For example, 9 and 4 are relatively prime, whereas 9 and 6 are not. Let *S*(*n*) be the set of all positive integers less than *n* that are relatively prime with *n*.

For example, *S*(8) = {1, 3, 5, 7}, and *S*(9) = {1, 2, 4, 5, 7, 8}

Let *Q*(*n*) = *U*(*n*) – *S*(*n*), where *U*(*n*) is the set of all positive integers less than *n*. Therefore, *Q*(*n*) is the set of all positive integers that are *not* relatively prime with *n*. Again, continuing from our previous examples,

Q(8) = *U*(8) – *S*(8) = {1,2,3,4,5,6,7} – {1,3,5,7} = {2, 4, 6} and

Q(9) = *U*(9) – *S*(9) = {1,2,3,4,5,6,7,8} – {1,2,4,5,8,7} = {3, 6}

Also, *Q*(2147483646) = {2, 3, 6, …more than 1 billion other numbers…}

In this problem, we are interested in finding the sum of the integers in Q(*n*), i.e., computing the function:

$$F(n) = \sum_{i=1}^{K} Q_i(n)$$

Where $K = |Q(n)|$, i.e., the number of elements in *Q*(*n*), and $Q_i(n)$ is the *i*-th element of *Q*(*n*).

For example, *F*(8) = 2 + 4 + 6 = 12, and *F*(9) = 3 + 6 = 9

Input

Each line of the input will contain a positive integer *n* ($1 < n < 2^{31}$). The input will be terminated by a line containing -1. There will be at most 25 lines of input.

Output

For each number *n* in the input, you are to output the value of *F*(*n*). Be assured that the value of *F*(*n*) shall always be less than 2^{63}.

Sample Input

```
2
4
8
9
10000
999999999
2147483646
-1
```

Output for Sample Input

```
0
2
12
9
29995000
1756766468243233 53
1731820625269184835
```

Problem F:
Bacterial growth

Source file: `bacteria.{c | cpp | java}`
Input file: `bacteria.in`

A curious biology student left a bacteria culture, with extremely low concentration, for few weeks. Then, the student measured the concentration and found that it grew to almost saturation level. The reason for the huge growth is that bacteria not only grow every second, but also new bacteria may enter the culture from the outside environment.

Let c be the initial concentration of bacteria; r be the growth rate per second; a be the amount of bacteria per second coming into the colony from the environment; and t be the time elapsed in seconds. The future concentration of bacteria after 1 second can be computed by the following formula: $f = c\ r + a$

Your task is to compute f for a number of colonies and select the maximum f, where t is a very large number.

Input

The input starts with the number of test cases nCases, on a separate line. The first line of each test case specifies the number of colonies *nColonies*, also on a separate line. *nColonies* lines follow, where each line consists of four numbers: $c \leq 0.001$, $r \leq 0.01$, $a \leq 0.001$, $t \leq 110{,}000$. The first number is the original concentration of bacteria; the second is the rate of growth per second; the third is the additional bacteria per second; and the fourth is the number of seconds.

Output

For each test case, you should output the value of f for the colony with largest f after t seconds, as well as the index of that colony (first colony has index 1). The largest value for f for the sample input is the fourth colony (fourth input line). You should format the values of f as fixed scientific notation 0.0e+000, as in sample output. Make sure that the decimal digit is always shown. For example, 3.0e+124 should be output instead of 3e+124.

Sample Input

```
1
7
0.0003 0.004 0.0007 52232
0.0005 0.002 0.0008 59152
0.0007 0.001 0.0003 37439
0.0005 0.01 0.0006 101835
0.0002 0.004 0.0002 84086
0.001 0.001 0 28623
0.001 0.006 0.0003 47311
```

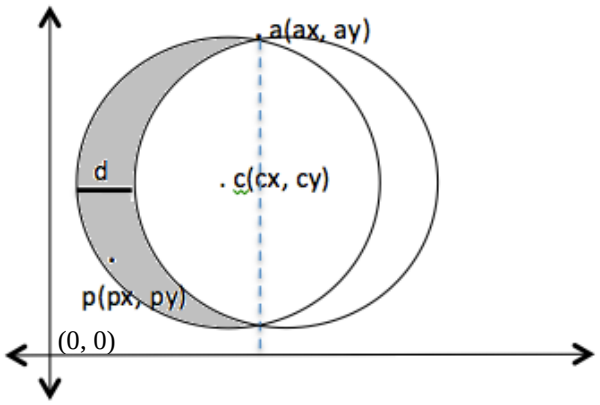
Output for Sample Input

```
7.1e+438 from Colony 4
```

Problem G:
Inside a crescent?

Source file: `crescent.{c | cpp | java}`
Input file: `crescent.in`

The gray crescent in the figure below is formed by a circle (point c is the center), which is moved by *d* units along the positive direction of the x-axis. The point *a* is the upper tip point of a crescent. Your task is to check whether a given point *p* is inside the crescent, on one of the crescent lines, or outside the crescent. On crescent refers to points on the outer-crescent line or inner-crescent line. If the distance between a point and one of the crescent lines is less than 1 unit, the point should be considered as on crescent. Note that the entire crescent is always in the first quarter of coordinate system and faces the right direction. Further, the axis that connects the upper and the lower tip of the crescent is always perpendicular on the x-axis.



Input

The input begins with an integer *n* on a separate line, which represents the number of test cases. Each case specifies a crescent and a set of points to consider whether they are in, on or outside that crescent. Each case consists of several lines. The first line consists of the following integers: *ax ay cx cy d*, where *ax ay* are the coordinates of the tip point, *cx cy* are the coordinates of the center *c* and *d* is the widest-distance. The second line has an integer *nP*, which corresponds to the number of candidate points *P*. Each of the following *nP* lines has two integers *px* and *py*, the coordinates of a point *p*. Note that all numbers are separated by spaces.

Output

For each input, the program should output "inside crescent", "on crescent" or "outside crescent". In the above figure, *p* is inside the crescent.

Sample Input

```
1
970 1955 1047 1071 461
3
565 195
565 326
565 335
```

Output for Sample Input

```
outside crescent
on crescent
inside crescent
```

Problem H: Offside!

Source file: `offside.{c | cpp | java}`
Input file: `offside.in`

The Fédération Internationale de Football Association (FIFA) is the international governing body of football and other sports. FIFA's law 11 sets the rules for offside positions and offside offences as follows:

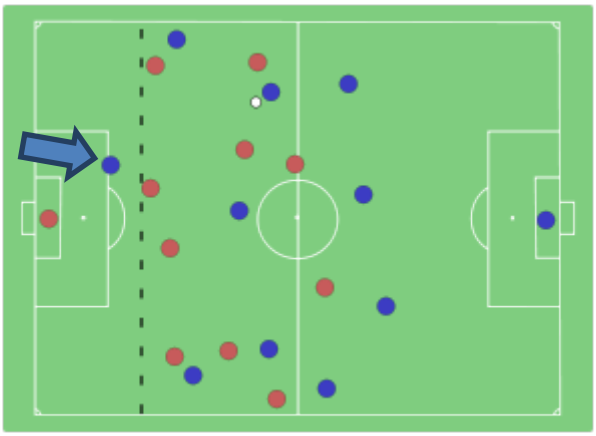
A player is in an offside position if the following two conditions are met:

- He is nearer to his opponents' goal line than both the ball and the second last opponent.
- He is in his opponents' half of the field of play.

A player commits and offside offence if the following two conditions are met:

- He is in an offside position at the moment the ball is played by a teammate.
- He becomes actively involved in the play.

In the following diagram, the player indicated by the arrow on the left of the diagram is in an offside position, as he is in front of both the second-to-last defender (marked by the dotted line) and the ball (white circle).



Currently, offside offences are called for by assistant line referees who need to be highly observant and concentrated during the entire period of the match. They still, however, make mistakes by missing offside offences or calling ones that did not exist. Those mistakes sometimes change the entire course of a football match, and even a whole football season. To avoid such mistakes, we would like you to write a program that processes locations of both teams' players at the moment the ball is played, the last player who played the ball, and the actively involved players in the play and discover if an offside offence has been committed.

Input

The first line of the input file contains an integer T indicating the number of test cases. Each test case starts with two integers L and W ($0 < L,W \leq 1{,}000$) describing the length and width of the football field. 22 lines follow with two integers on each line describing the X and Y positions of the players. The first 11 positions are the left team players' positions. The following 11 positions are the

right team players' positions. The lower left corner of the field is the reference origin, i.e. position (0,0). All players will be positioned within the field area. The players are numbered from 1 to 22 in order of appearance in the input file. The following line contains an integer *P* describing the number of the player who played the ball last. The position of the ball is assumed to be the same as that of the last player who played it. The next line contains an integer *N* describing the number of active players in the play. *N* lines follow containing the numbers of the active players in the play (i.e. each line contains a number of an active player in the play).

Output

The output for each test case is in this form:

k. R

where ***k*** represents the test case number (starting at 1) followed by a single space, and ***R*** is "Offside!" without the quotes if an offside offence has been committed, or "`No Offside`!" without the quotes if no offside offence has been committed.

Sample Input

```
2
90 50
1 25
20 27
22 49
24 23
26 5
30 6
35 30
40 48
43 1
45 27
50 20
89 25
10 29
25 49
30 5
35 26
40 35
41 10
50 3
55 40
60 27
65 30
17
```

Output for Sample Input

```
1. Offside!
2. No Offside!
```

```
2
13
14
90 50
1 25
20 27
22 49
24 23
26 5
30 6
35 30
40 48
43 1
45 27
50 20
89 25
22 29
25 49
30 5
35 26
40 35
41 10
50 3
55 40
60 27
65 30
17
2
13
14
```

Problem I:
The Champions League

Source file: `champions.{c | cpp | java}`
Input file: `champions.in`

The Champions League is an annual continental club football competition organized by the Union of European Football Associations (UEFA) since 1992. It replaced the European Champion Clubs' Cup, or simply European Cup, which had run since 1955, adding a group stage to the competition and allowing multiple entrants from certain countries. It is one of the most prestigious tournaments in the world and the most prestigious club competition in European football. The final of the 2012–13 tournament was the most watched UEFA Champions League final to date, as well as the most watched annual sporting event worldwide in 2013, drawing 360 million television viewers.[1]

In this problem, we are interested in the first knock-out stage of this competition. This stage consists of 16 teams, of which only 8 qualify to the next round. To determine the teams that qualify, the 16 teams are paired by means of a draw. Each pair plays two matches on a home-and-away basis. This means that each team will play a match on their home field (home match) and another match on the opponent's home field (away match). The winner of each pair is the team that scores the greater aggregate of goals in the two matches. If the aggregate scores are equal, then the winner is the team that scores more away goals. That is, the team that scores more goals on the opponent's field wins. If the two teams score the same number of away goals, then extra time is given and if the tie continues, kicks from the penalty mark are played under different rules.

Given the scores of the two matches (the home and away matches), you are to determine the winner team or call for extra time.

Input

The first line of the input file contains an integer *N* indicating the number of test cases. Each test case consists of two lines with two space separated integers on each line. The first line describes the score of the match played in the first team's home field, while the second line describes the score of the match played in the second team's home field. The score always starts with the number of goals scored by the home team followed by the number of goals scored by the away team. All numbers in the input file are a maximum of 100.

Output

The output for each test case is in this form:

k. R

where ***k*** represents the test case number (starting at 1) followed by a single space, and ***R*** is "The first team wins!" without the quotes if the first team wins, "The second team wins!" without the quotes if the second team wins, or "Extra time is needed!" without the quotes if the two teams are still tied.

[1] Wikipedia

Sample Input

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

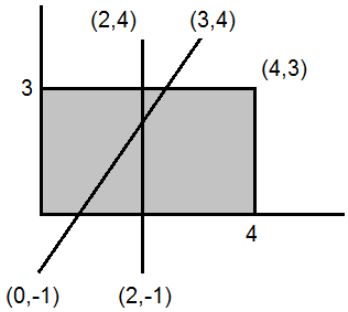
Output for Sample Input

```
1. The first team wins!
2. The second team wins!
3. Extra time is needed!
```

Problem J:
Cake Divider

Source file: `cake.{c | cpp | java}`
Input file: `cake.in`

Given a piece of cake, and a knife to cut through it a number of times, state the number of pieces the cake is divided to. Knife movement always starts at one point *outside* the cake, and stops at another point *outside* the cake moving in a *straight* line only. Cake is represented with a *rectangle*: a set of four points with edges that connects them in the 2 D Cartesian coordinates. Each knife movement is represented with an edge: two points with straight line between them. The cake is always placed in the Cartesian coordinates such that its lower left corner is at (0, 0).



Input

The input file will consist of a sequence of problem instances, each starts with two numbers (x, y) that represents the upper right corner of the cake, the value of x=0 and y=0 indicates the end of instances. Each instance starts with *n* that represents the number of knife cuts, followed by *n* edges, each is represented by two points (x1, y1), (x2, y2). All given numbers are integers. Instances are terminated when point (0, 0) is encountered for cake edge. Ranges: $1 \le n \le 5$
Cake upper corner: $1 \le x \le 9$, $1 \le y \le 9$,
Knife cut coordinates: $-99 \le x_i \le 99$, $-99 \le y_i \le 99$

Output

For each problem instance, your program should output a single line consisting of one value that represents the number of pieces the cake is divided into.

Sample Input

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

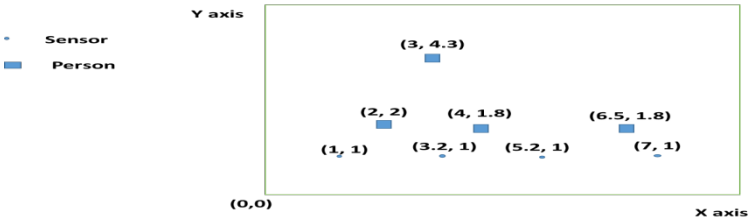
Output for Sample Input

```
2
4
```

Problem K:
Pressure Sensors

Source file: `pressure.{c | cpp | java}`
Input file: `pressure.in`

A mesh of sensors is used to detect the pressure of the people on a stage to be used for various purposes. Write a program that accept the number of sensors, the number of people, the average weight of each person, the range of the sensor, the dimensions of the stage, the locations of the sensors, and the locations of the people. The sensor could detect pressure within a range provided. The program should display the weights detected by each sensor.
The following diagram shows a sample of the stage with four sensors and four people distributed on it. This will show how the coordinates of the system are defined.



Input

The input file contains a number of test cases separated by a single empty line. Each test case data are described as 5 consecutive lines. These are:
The first line contains:

- The x-axis length of the stage X, where 1<= X <=10
- The y-axis length of the stage Y, where 1<= Y <=10
- The average weight of a person W, where 0.1<= W <=100.0
- The range of the sensor R, where 0<= R<=10.0
- The number of the people in the stage NP, where 0<= NP <=100
- The number of the available sensors NS, where 1<= NS <=50

The second line contains the x-axis coordinates of the sensors separated by single space. The third line contains the corresponding y-axis coordinates of the sensors separated by single space. The fourth line contains the x-axis coordinates of the people separated by single space. Finally, the fifth line contains the correspondent y-axis coordinates of the people separated by single space. Following is an example of input with clarifications:

```
10(X) 10(Y) 80(W) 10(R) 3(NP) 2(NS)
7 8      The x-coordinates of the sensors
8 3      The y-coordinates of the sensors
4 4 5    The x-coordinates of the people
5 8 4    The y-coordinates of the people
```

Output

For each problem instance, your program should output the pressures measured by the sensors in one line separated by a single space. The order of the sensors is the same order of their coordinates in the input file.

Sample Input

```
10 10 60 5 0 2
7 8
8 3

10 10 80 10 3 2
7 8
8 3
4 4 5
5 8 4

10 10 77 1 3 6
7 8 5 1 2 9
8 3 8 2 9 4
4 4 5
5 8 4

10 10 55 10 8 5
7 8 4.5 2.3 9.9
8 3 7.1 1.1 4.3
4 4 5 5.1 4.4 7.7 8.8 2.2
5 8 4 2 1 9.9 5.5 8.9

10 10 90 5 1 2
7 8
8 3
4
5
```

Output for Sample Input

```
0 0

240 240

0 0 77 0 0 0

440 440 440 385 440

90 90
```