Guadalajara 2010 hosted by ITESO
acm International Collegiate Programming Contest
IBM event sponsor
ITESO

Problem A (Blue)
The Lazy Lumberjacks
Input: Lumber.in

Once upon a time in a far, far away forest, there was a team of lazy lumberjacks. Since they were too lazy to cut trees, they were always figuring out ways to sneak out of work. Their foreman, on the other side, was always trying to put them all to work.

After a lot of discussions the foreman and the lumberjacks came to an agreement: they will work, but only if the area of the forest assigned to each one was a triangle. If it was any other shape they will be free not to work that week. The idea was to give each lumberjack three numbers representing the length of each of the triangles side. If the numbers were correct and form a triangle, the lumberjacks had to work, else, they were free to leave and not work.

Since our lumberjacks are as cunning as they are lazy, they convince the foreman to let them determine the surface and the site in the forest were they will work. As a result, the lumberjacks keep passing the foreman sets of numbers that could not form the sides of a triangle. After a while, the foreman began to suspect and decide to write a program that validates the input of each lumberjack. Now when the lumberjacks decide to pass wrong numbers they get a fine of $1000.00 (more than a day’s salary).

Your job is to write the program that the foreman has to use to determine if the numbers (all integers) passed by the lumberjacks can be the sides of a triangle. If they can, you have to print “OK” else you have to print “Wrong!!”

Input

The input consist in a data set describing the numbers of that each lumberjack has passed to the foreman for the day: The data is formatted as follows:

The first line is an integer N (2 <= N <= 20). Then follows N lines, each one containing three integers separated by a space.

Output

For each line in the input you have to find if the integers can represent the sides of a triangle. If they can you have to print “OK” for each line in the input, else you have to print “Wrong!!”

Sample Input	Output for the Sample Input
6	Wrong!!
1 2 3	Wrong!!
3 2 5	OK
3 4 5	OK
6 6 1	OK
3 3 3	Wrong!!
7 3 10	

Problem B (Red)
Balanced ScoreCard
Input: bsc.in

The Balanced ScoreCard (BSC), first introduced by Robert Kaplan and David Norton, is a strategic planning and management system that is used extensively in organizations worldwide (industry, business, government and nonprofit) primarily to monitor individual and organizational performance against strategic goals. The BSC provides a tree-shaped hierarchical view of the status of every objective required to fulfill a specific organization goal. The root of the tree denotes the organization goal itself whereas the leaves refer to the most specific objectives (performance of an employee, sales of a specific product); inner nodes denote objectives per organizational area (business unit, department, product line, customers from a specific region).

The performance of a given organization goal is in terms of what was expected to be done and what was actually done. This relationship is known as the *result* of and has to be calculated from the bottom-up. Every objective in the lowest level of the tree has a predefined expected value and a post-defined actual value. The percentile relationship between the actual and expected values gives the result of the objective. The result of inner objectives (including the root) is calculated according to a rule given to every objective. Typical rules include the following: 1) the result of an objective is the weighted average of the results of the children objectives, 2) the result of an objective is the worst result of the children objectives. To implement the first rule, objectives have to be accompanied with a *weight* value given as a percentage. To illustrate better this, consider the following example:

General rule = 2
1. Weight = 50%, rule = 1
    - 1.1 Weight = 60%, value = 95.0
    - 1.2 Weight = 30%, value = 85.0
    - 1.3 Weight = 10%, value = 63.0
2. Weight = 50%, rule = 2
    - 1.1 Weight = 50%, value = 72.0
    - 1.2 Weight = 50%, value = 92.0

The result of the organization goal is then calculated as follows:

Result = Min { ((95.0 x 0.60) + (85.0 x 0.30) + (63.0 x 0.10), Min { 72.0, 92.0 } }
= Min { 88.8, 72.0 } = 72.0

Notice that *weight* values are ignored when parent's rule is 2.

The problem consists in figuring out the result of an organization goal, given a BSC containing:

1) The weight and the result, for every lowest-level objective

2) The weight, the rule and the children objectives, for every inner objective.

And assuming the BSC is well-formed (no syntactic errors are expected).
Nevertheless, the BSC may be unbalanced: the sum of the weight of a given group of sibling objectives is not equals to 100.0% ± 0.01, no matter the parent's rule.

Input

The first line contains an integer number, $N > 0$, denoting the number of test cases.
The next N lines contain each the definition of an organization goal, having the following syntax:
<rule><blank><objective>{<blank><objective>}*, where:

- <rule> is either 1 or 2, and follows the definition explained above.
- <blank> is a blank space
- <objective> can be either <leaf-objective> or <inner-objective>
- {ω}* denotes 0 or more occurrences of ω
- <leaf-objective> is written as: <(><weight><;><value><)>
  - <weight> is a number greater than 0.0 but not greater than 100.0
  - <value> is a number between 0.0 and 100.0
  - <(> is the symbol (
  - <)> is the symbol)
  - <;> is the symbol ;
- <inner-objective> is written as:
  <(><weight><;><rule><;><{><objective>{<,><objective>}*<}><)>
  - <{> is the symbol {
  - <}> is the symbol }

Output

The output consists of N lines, where every line $k \leq N$ contains a number in the range [0.0 – 100.0] which denotes the result of organization goal k. Keep only the two most significant decimals. If the BSC k is not balanced, line k contains the word *Unbalanced* instead.

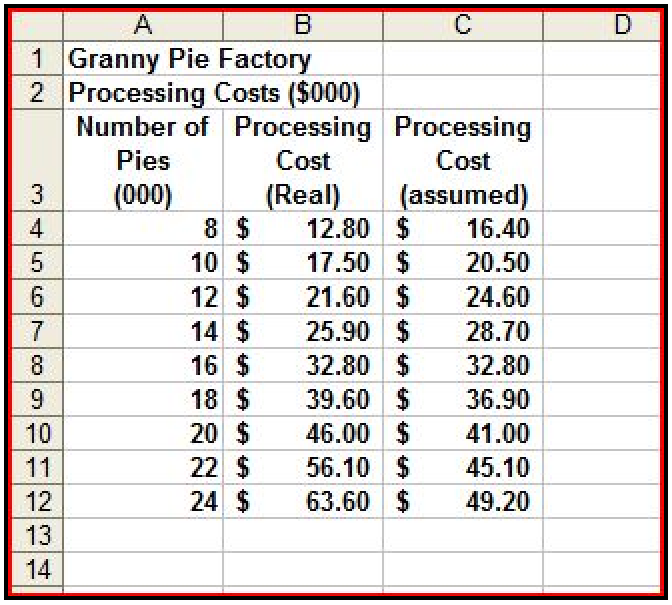
Sample Input	**Output for the Sample Input**
3 1 (35;95) (65;78) 1 (72;83) (28;1;{(60;100),(30;97),(15;85)}) 2 (50;1;{(60;95),(30;85),(10;63)}) (50;2;{(50;72),(50;92)})	83.95 Unbalanced 72.00

Problem C (Green)
Granny Pie Factory
Input: granny.in

Granny has always loved to bake pies. She has a very special recipe to prepare an apple pie. His grandson, Simon, has decided to turn Granny's hobbie into a family business. To do so, Simon has to combine two ingredients: fruit and dough. He buys these two ingredients and combines them to prepare apple pies through the appropriate process: cooking, packaging, delivering, etc.). He sells the pies to local grocery stores.
As a good businessman, Simon wants to have the highest profit so he prepared a financial model for Granny Pie Factory where he defines input and output variables. Since the main idea is to get immediate profits, the only decision variable is the selling price for each pie.
As we all know, the equation to obtain the profit is as follows: Profit = Income – Costs

In order to get the cost right, Simon has to determine the process costs which he has assumed is a constant multiplied by the number of pies processed (Simon assumed that Processing Cost = 2.05 * Number of pies). After several months of collecting data, Simon has realized that his assumption is false and needs to find a way to accurately predict processing costs.

The next figure shows the data that Simon has collected



	A	B	C	D
1	Granny Pie Factory			
2	Processing Costs ($000)			
3	Number of Pies (000)	Processing Cost (Real)	Processing Cost (assumed)	
4	8	$ 12.80	$ 16.40	
5	10	$ 17.50	$ 20.50	
6	12	$ 21.60	$ 24.60	
7	14	$ 25.90	$ 28.70	
8	16	$ 32.80	$ 32.80	
9	18	$ 39.60	$ 36.90	
10	20	$ 46.00	$ 41.00	
11	22	$ 56.10	$ 45.10	
12	24	$ 63.60	$ 49.20	
13				
14				

Figure 1: Granny Pie Factory Processing Cost Data

In short, the idea is to find the relationship between the number of pies processed and the processing cost.

Input

The input consists in datasets describing the number of pies processed (in thousands), the real processing cost and the assumed processing cost. The dataset is formatted as follows.

First an integer denoting the number of pies processed followed by two reals denoting (in that order) the real processing cost and the assumed processed cost. Each number is separated by an space and there could be N lines of numbers in a dataset.

A line consisting in 0 0 0 denotes the end of the dataset.

Output

For each dataset you will need to find the relationship between the number of pies processed and the real processing cost.

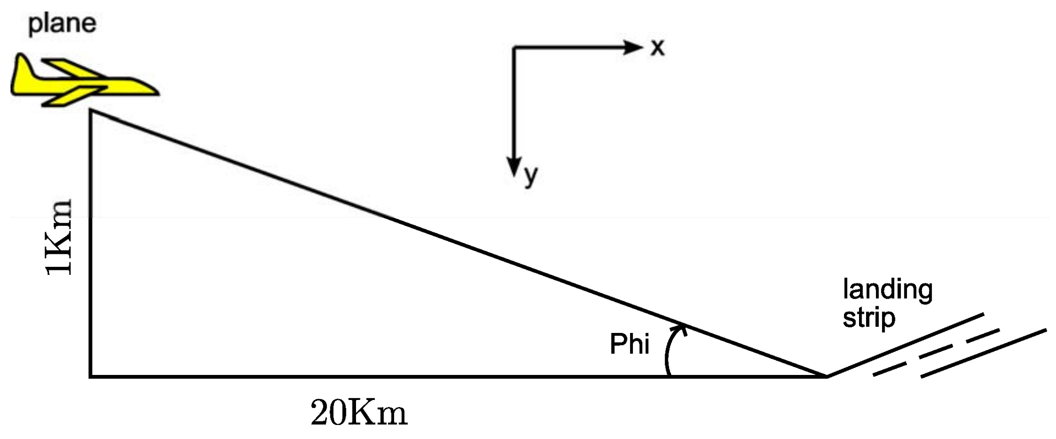
Sample Input	Output for the Sample Input

8 12.80 16.40
10 17.50 20.50
12 21.60 24.60
14 25.90 28.70
16 32.80 32.80
18 39.60 36.90
20 46.00 41.00
22 56.10 45.10
24 63.60 49.20
0 0 0
	Linear coefficient: 0.9331
	Quadratic coefficient: 0.0712
	
	Equation: y = 0.0712 x2 + 0.9331 x

Problem D (Black)
Landing Navigation
Input: navigation.in

Landing a plane is a difficult operation. Modern planes include landing aids, in which a computer indicates, based on analysis of a number of variables, whether it is safe to attempt landing and, if so, how best to proceed.

In this problem we will program a simplified landing aid. We assume the pilot engages this aid when the plane is in a specific position relative to the landing strip:



That is, when the plane is exactly at a height of one kilometer, and 20 kilometers away from the landing strip, the pilot starts the landing procedure and engages our landing aid. As shown in the picture, the plane is descending at an angle *Phi*, measured counterclockwise from ground; furthermore, we consider the horizontal *x* direction to be positive in the direction towards the landing strip, and the vertical *y* direction to be positive downwards towards the ground. We also assume the plane velocity to be always 200Km/h in the direction from the plane straight towards the landing strip. The vertical and horizontal (scalar) speeds may vary, but the velocity will always be 200Km/h.

When our landing aid is first engaged, it will inform the pilot the following: a) the current time into the descent is zero; b) condition is GO (meaning landing may proceed); c) what descent angle *Phi* should the plane be set to; d) the remaining flight time; and e) the current horizontal and vertical speeds.

Under static conditions this would be all that is needed to safely land the plane. However, wind bursts may affect the plane's position and velocity during the descent. We assume bursts only occur in the *x* or *y* directions, but they may have positive or negative speed. The burst velocity is added to the plane's. The changes introduced by the burst will affect the plane velocity and the angle *Phi* needed to reach the landing strip. However, to have a safe landing we need to meet the following two conditions:

- the descent angle *Phi* must be equal to or greater than 1 degree, and equal to or less than 4 degrees
- the plane speed in the *y* direction must be less than 4 meters per second

The plane has a burst detector, which indicates to our landing aid the direction and speed of the burst.
We assume it is too risky for the pilot to make course changes during a burst, so our landing aid will behave as follows:

- It will indicate to the pilot that a burst has started, as well as current time into the descent
- Every five seconds, it will calculate the new angle *Phi* and plane velocity required to land, and will determine if a safe landing would still be possible if the burst ended at that exact moment. If safe landing is still possible, it will give the pilot a GO signal. Otherwise, it will give an ABORT signal.
- At the end of the burst, it will indicate the pilot that the burst has ended, a GO or ABORT signal, and in case of a GO signal, the new flight conditions.

Input and Sample Input

Your program must read test cases from a file called “navigation.in”. Each test case has one or more lines. Each line has four numbers separated by spaces. The first number is the time into the flight (in seconds) when a burst starts. The second is the time when the burst stops. The third number is equal to 0 if the wind moves in the *x* direction and different than 0 if the wind moves in the *y* direction. The fourth and final number is the burst speed in Km/h; it may be positive or negative.

Test cases are separated by lines where the first number is 0. There may or may not be any number of such lines at the start, middle or end of the file. It may be assumed that bursts always occur in isolation and never coincide in time. Bursts have duration of at least one second. When ABORT conditions are set, the rest of the test case must be ignored, since the plane has interrupted the landing. There will always be at least one burst per test case, and one test case per input file.
As an example, consider the following input file:

```
50 52 0 10
0 0 0 0
100 117 0 50
150 170 1 -60
```

This file defines two test cases. In the first, there is a wind burst along the *x* direction that starts 50 seconds into the descent and lasts two seconds. Wind speed is 10km/h; the burst is pushing the plane towards the landing strip. The second test case has two bursts. The first burst starts 100 seconds into the flight and lasts 7 seconds, and moves in the *x* direction at 10km/h. The second burst occurs 150 seconds into the flight, moves along the *y* direction, and has negative speed (that is, it pushes the plane upwards). Times and speeds are not restricted to be integers and could be fractional.

Output and Sample Output for Sample Input

```
---Start of test case---
TIME = 0.00, GO
  RTIME = 360.45
  ANGLE = 2.86
  VX = 55.49
  VY = 2.77
BURST START AT TIME = 50.00
BURST END AT TIME = 52.00
TIME = 52.00, GO
  RTIME = 308.35
  ANGLE = 2.86
  VX = 55.49
  VY = 2.78
---End of test case---
---Start of test case---
TIME = 0.00, GO
  RTIME = 360.45
  ANGLE = 2.86
  VX = 55.49
  VY = 2.77
BURST START AT TIME = 100.00
TIME = 105.00, GO
TIME = 110.00, GO
TIME = 115.00, GO
BURST END AT TIME = 117.00
TIME = 117.00, GO
  RTIME = 239.21
  ANGLE = 2.91
  VX = 55.49
  VY = 2.82
BURST START AT TIME = 150.00
TIME = 155.00, GO
TIME = 160.00, GO
TIME = 165.00, ABORT
---End of test case---
```

Let us analyze the program's output.

Test cases are separated by lines ---Start of test case--- and ---End of test case---.

At the start of each test case, the landing aid outputs the following information:

```
TIME = 0.00, GO       The current time (0.00) and a GO signal
  RTIME = 360.45      Remaining flight time if conditions don't change
  ANGLE = 2.86        Angle of descent that pilot must set, in degrees
  VX = 55.49          Speed along x axis in m/s
  VY = 2.77           Speed along y axis in m/s
```

(Note that the last four lines are indented by two spaces, and all numbers are printed with two decimal figures). When a burst is detected, the pilot is informed of this, along with the current flight time in seconds, with a line as follows:

```
BURST START AT TIME = 100.00
```

Then, every five seconds a GO or ABORT signal is given, along with the current flight time:

```
TIME = 105.00, GO
TIME = 110.00, GO
TIME = 115.00, GO
```

The pilot is informed of the end of the burst, along with the time, and if conditions are good, a GO signal and new flight information:

```
BURST END AT TIME = 117.00    Burst end is indicated, with time
TIME = 117.00, GO             GO signal is given (if appropriate)
  RTIME = 239.21              Updated flight information is provided
  ANGLE = 2.91
  VX = 55.49
  VY = 2.82
```

In case the landing cannot be completed safely, an ABORT signal is given and the test case ends immediately:

```
BURST START AT TIME = 150.00
TIME = 155.00, GO
TIME = 160.00, GO
TIME = 165.00, ABORT           When ABORT is given, test case ends
---End of test case---
```

Problem E (Brown)
Face the Maze
Input: maze.in

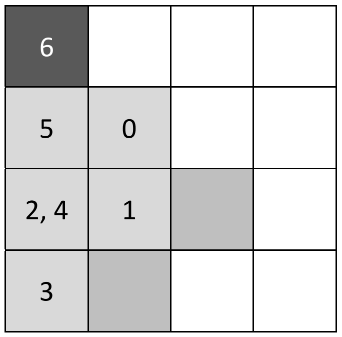
This problem consists of traversing deterministically a discrete surface represented by a matrix of *n* x *n* cells (*n* > 2), starting at a *source* cell, ending at a *target* cell, and considering the existence of static obstacles in the form of *unavailable* cells. The key idea is that from the *source* cell, the position of the *target* cell is unknown; thus, a path starting from *source* cell must be created on the run until *target* cell is found.

Only vertical and horizontal movements are allowed. The choice of the next cell visited is given by the following list of priorities.

1. Go to the *target* cell, if it is adjacent vertical or horizontally
2. Go down, if the bottom cell is available and not visited yet
3. Go to the right, if the right cell is available and not visited yet
4. Go to the left, if the left cell is available and not visited yet
5. Go up, if the top cell is available and not visited yet
6. Go back to the previous cell.

If either *source* cell or *target* cell is trapped in a dead end, eventually the path will return to the start point. In such a case, the travel must end.

For a better understanding of this problem, consider the first test case from Sample Input, where *source* cell is at (2, 2), *target* is at (1, 1), and cells (3, 3) and (2, 4) are unavailable. As can be seen in the following figure, the path from *source* to *target* considering the priorities enlisted above is as follows: 1) go down, 2) go to the left, 3) go down, 4) go back, 5) go up, 6) go to the *target*.



Input

The first line contains an integer *N* > 0 denoting the number of test cases.
The next *N* lines contain a space-separated list starting with an integer *n* > 1 denoting the number of rows (or columns) in the surface, and followed by *m* ≥ 2 pairs of integers (i,j), such that:

a) i is the column index

b) j is the row index
c) $1 \leq i, j \leq n$
d) The first pair (i_1, j_1) denotes the position of the *source* cell
e) The second pair (i_2, j_2), such that (i_2, j_2) ≠ (i_1, j_1), denotes the position of the *target* cell
f) Every pair (i_k, j_k), such that $k \geq 3$, (i_k, j_k), ≠ (i_1, j_1), and (i_k, j_k) ≠ (i_2, j_2), denotes an *unavailable* cell (obstacle)

Output
The output consists of *N* lines containing the path produced at each test case. The path is specified by means of a space-separated list of *m* pairs of integers (i,j), such that:
a) $1 \leq i, j \leq n$
b) The first pair (i_1, j_1) denotes the position of the *source* cell
c) Every pair (i_k, j_k), where $1 < k <$ m, denotes the position of a cell employed in the path, such that (i_k, j_k) is not an *unavailable* cell and is horizontally or vertically adjacent to (i_{k-1}, j_{k-1})
d) The last pair (i_m, j_m) denotes either
    a. The position of the *target* cell, if it was successfully reached
    b. The position of the *source* cell, otherwise

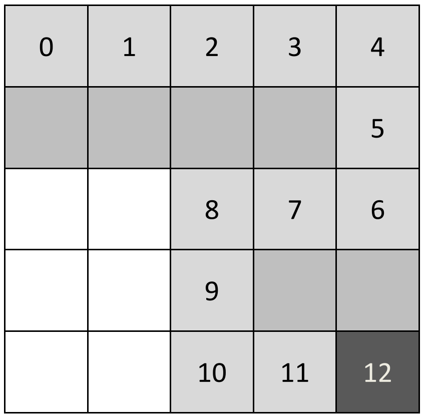
Sample Input

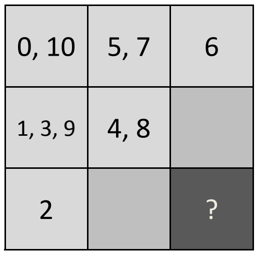
```
3
4 (2,2) (1,1) (3,3) (2,4)
5 (1,1) (5,5) (1,2) (2,2) (3,2) (4,2) (4,4) (5,4)
3 (1,1) (3,3) (3,2) (2,3)
```

Output for the Sample Input

```
(2,2) (2,3) (1,3) (1,4) (1,3) (1,2) (1,1)
(1,1) (2,1) (3,1) (4,1) (5,1) (5,2) (5,3) (4,3) (3,3) (3,4) (3,5) (4,5) (5,5)
(1,1) (1,2) (1,3) (1,2) (2,2) (2,1) (3,1) (2,1) (2,2) (1,2) (1,1)
```

Images from test cases 2 and 3





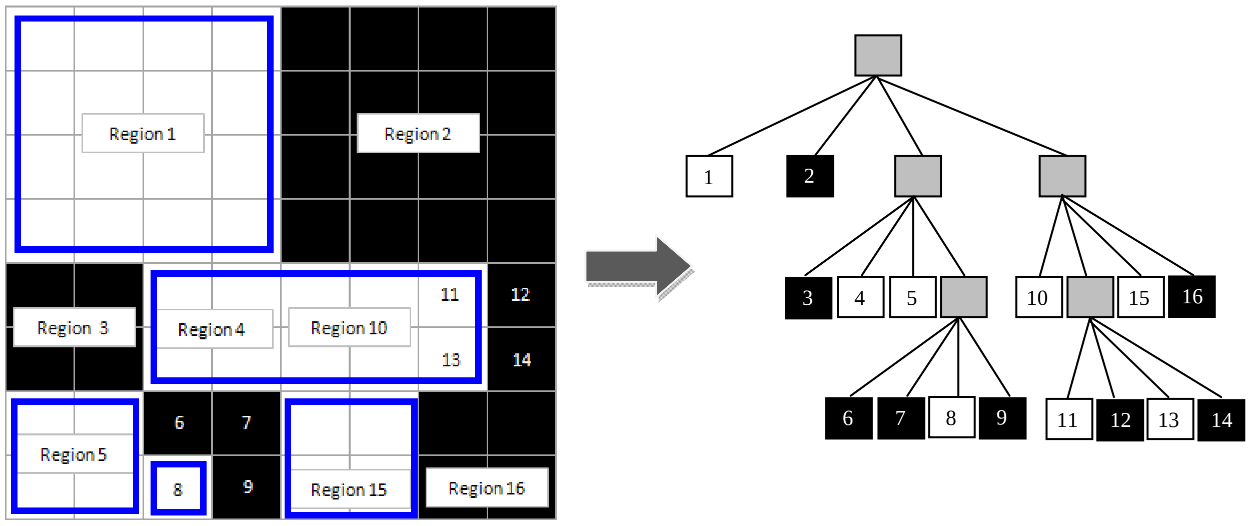
Problem F (Purple)

Creating a Quadtree

Input: quadtree.in

A *quadtree*, first introduced by Finkel and Bentley, is a tree data-structure in which each internal node has exactly four children. Quadtrees are often used for problems that can be mapped into a two-dimensional space which is then recursively subdivided it into four equally-sized regions while a certain condition holds. Such problems include the following: collision detections, spatial data indexing, image compression and Conway's Game of Life. The problem consists in compressing a binary image represented by an *n* x *n* matrix, where *n* > 1 is a power of 2. The binary image is specified by means of rectangular areas containing only white pixels. Each area is given by the location of its upper-left and lower-right corners. The compressed image is given by a quadtree, where each node denotes a region in the image. A region is subdivided if it is not a pixel yet and it has both white and black pixels. The order in visiting regions is left-right, top-down.

For a better understanding of this problem, consider the third test case from Sample Input where the binary image is a 8 x 8 matrix composed by five white areas. In the following figure, such areas are highlighted with blue rectangles. Its corresponding quadtree is made up of five internal nodes and sixteen leaves (regions).



Input

The first line contains an integer $N > 0$ denoting the number of test cases.
The next N lines contain a space-separated list starting with an integer $n > 1$, power of two, which denotes the matrix size, and followed by $m \geq 2$ pairs of integers (i_k,j_k), such that:

a) i_k is the column index
b) j_k is the row index

c) $1 \le i_k, j_k \le n$
d) Two consecutive pairs (i_{k-1}, j_{k-1}), (i_k, j_k), where k is even, denotes respectively the position of the upper-left corner and lower-right corner of the ½kth area
e) If *m* is odd, the pair (i_m, j_m) is ignored

Output

The output consists of *N* lines containing either the quadtree produced at each test case, or the text *Size is invalid* if the matrix size is not a power of 2. The quadtree is specified by means of a sequence of 0's and 1's. For each black node add a 0 to the output; for each white node add a 1; for each gray node add a * and repeat this process for each child node. Do not consider the root in the output. Traverse the quadtree in pre-order.

Sample Input

```
3
4 (1,1) (1,1) (4,1) (4,1) (1,3) (2,4)
15
8 (1,1) (4,4) (3,5) (7,6) (1,7) (2,8) (3,8) (3,8) (5,7) (6,8)
```

Output for the Sample Input

```
*1000*010010
Size is invalid
10*011*00101*101010
```