Problem A
String Factoring
source code: factoring.*

Spotting patterns in seemingly random strings is a problem with many applications. E.g. in our efforts to understand the genome we investigate the structure of DNA strings. In data compression we are interested in finding repetitions, so the data can be represented more efficiently with pointers. Another plausible example arises from the area of artificial intelligence, as how to interprete information given to you in a language you do not know. The natural thing to do in order to decode the information message would be to look for recurrences in it. So if the SETI project (the Search for Extra Terrestrial Intelligence) ever get a signal in the H21-spectra, we need to know how to decompose it.

One way of capturing the redundancy of a string is to find its factoring. If two or more identical substrings A follow each other in a string S, we can represent this part of S as the substring A, embraced by parentheses, raised to the power of the number of its recurrences. E.g. the string $DOODOO$ can be factored as $(DOO)^2$, but also as $(D(O)^2)^2$. Naturally, the latter factoring is considered better since it cannot be factored any further. We say that a factoring is *irreducible* if it does not contain any consecutive repetition of a substring. A string may have several irreducible factorings, as seen by the example string $POPPOP$. It can be factored as $(POP)^2$, as well as $PO(P)^2OP$. The first factoring has a shorter representation and motivates the following definition. The *weight* of a factoring, equals the number of characters in it, *excluding* the parentheses and the exponents. Thus the weight of $(POP)^2$ is 3, whereas $PO(P)^2OP$ has weight 5. A *maximal* factoring is a factoring with the smallest possible weight. It should be clear that a maximal factoring is always an irreducible one, but there may still be several maximal factorings. E.g. the string $ABABA$ has two maximal factorings $(AB)^2A$ and $A(BA)^2$.

Input

The input consists of several rows. The rows each hold one string of at least one, but less than 80 characters from the capital alphabet A-Z. The input is terminated by a row containing the character '*' only. There will be no white space characters in the input.

Output

For each string in the input, output one line containing the weight of a maximal factoring of the string.

Example

Input:	Output:	Maximal factorings:
PRATTATTATTIC	6	$PR(A(T)^2)^3IC$
GGGGGGGGG	1	$(G)^9$, and $((G)^3)^3$
PRIME	5	$PRIME$
BABBABABBABBA	6	$(BAB)^2(A(B)^2)^2A$, and $(BAB)^2A((B)^2A)^2$
ARPARPARPARPAR	5	$(ARP)^4AR$, $A(RPA)^4R$, and $AR(PAR)^4$

*

Problem B
Mountain Village
source code: mountain.*

After a long summer's march through the rough terrain of northern America, the indian tribe had found a place where they hopefully would be left alone. The chief proclaimed that this would be the new place for their village, despite the rocky nature of the landscape. They set a temporary camp for the night, content with the piece of land they had discovered. The very next day however, it stood clear that some effort planning the locations of the Teepee tents had to be made. It was simply too great a difference in altitude between the tents, making the walk along some paths extremely tiresome. Therefore, the chief ordered his witty son, Fast Thought, to find a connected area in their vicinity, large enough to host all tents of the tribe, having as small difference between the highest and lowest point as possible.

The task called for some altitude measurements of their whereabouts, which caused no problem for Fast Thought, since he was wise in the ways of trigonometry. He divided the land into squares big enough to host a tent each and estimated the altitude of each square. Now the problem was reduced to finding a connected region containing at least as many squares as there were tents, having the smallest difference between the highest and lowest altitude. He drew a rectangular matrix A, representing the area, where the entry $a_{i,j}$ at row i and column j, was the altitude of the square with coordinates (i, j). He considered an entry $a_{i,j}$ *adjacent* to the entries $a_{i,j+1}, a_{i+1,j}, a_{i,j-1}$, and $a_{i-1,j}$, and called a set of entries *connected* if for every pair of entries in the set, there was a connecting path of adjacent entries in it. Since the size of the tribe altered with time, Fast Thought decided to solve the problem for a general number of tents. Thus the problem left to solve for him was to find a set of at least k connected entries a_{ij} in the matrix A, such that the difference between the largest and the smallest entry in the set was minimized.

Input

On the first line of input there are two integers, $r, c \leq 40$, giving the dimension of the matrix A. The following r lines, each containing c integers between 0 and 99, are the entries $a_{i,j}$ of the matrix. The next line contains a single integer $n \leq 100$, and is followed by n lines each holding a single positive integer $k_i \leq rc$.

Output

For each k_i, output one line containing the minimum difference between the largest and the smallest entry for any connected set of at least k_i entries.

Example

```
Input:                              Output:
5 10                                0
 0  0  3 46  0 46  0  0 12 12      0
 0  0 13 50 49 46 11 10 10 11      3
 0 51 51 49 99 99 89  0  0 10      4
 0  0 48 82 70 99  0 52 13 14      89
51 50 50 51 70 35 70 10 14 11      99
6
1
5
10
12
47
50
```

Problem C
Pebble Solitaire

source code: pebble.*

I bet you have seen a pebble solitaire game. You know the game where you are given a board with an arrangement of small cavities, initially all but one occupied by a pebble each. The aim of the game is to remove as many pebbles as possible from the board. Pebbles disappear from the board as a result of a move. A move is possible if there is a straight line of three adjacent cavities, let us call them A, B, and C, with B in the middle, where A is vacant, but B and C each contain a pebble. The move constitutes of moving the pebble from C to A, and removing the pebble in B from the board. You may continue to make moves until no more moves are possible.

In this problem, we look at a simple variant of this game, namely a board with twelve cavities located along a line. In the beginning of each game, some of the cavities are occupied by pebbles. Your mission is to find a sequence of moves such that as few pebbles as possible are left on the board.

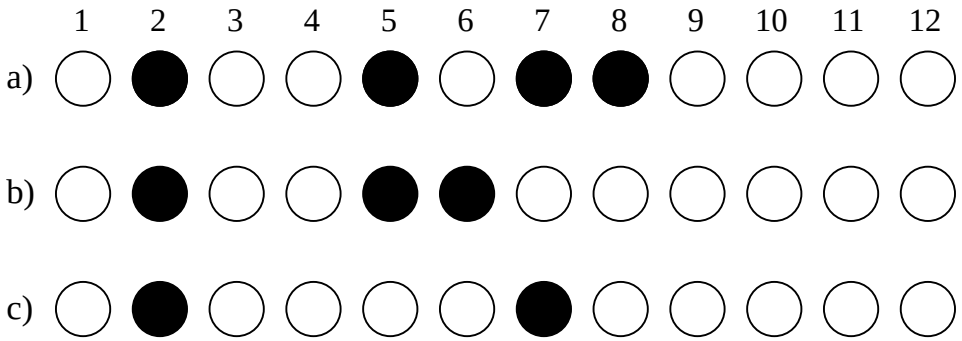


Fig 1. In a) there are two possible moves, namely $8 \to 6$*, or* $7 \to 9$*. In b) the result of the* $8 \to 6$ *move is depicted, and again there are two possible moves,* $5 \to 7$*, or* $6 \to 4$*. Making the first of these results in c), from which there are no further moves.*

Input

The input begins with a positive integer n on a line of its own. Thereafter n different games follow. Each game consists of one line of input with exactly twelve characters, describing the twelve cavities of the board in order. Each character is either '-' or 'o'. A '-' character denotes an empty cavity, whereas an 'o' character denotes a cavity with a pebble in it. There is at least one pebble in all games.

Output

For each of the n games in the input, output the minimum number of pebbles left on the board possible to obtain as a result of moves, on a row of its own.

Example

Input:	Output:
5	1
---oo-------	2
-o--o-oo----	3
-o----ooo---	12
oooooooooooo	1
ooooooooo-o	

Problem D
Manhattan

source code: manhattan.*

You are the mayor of a city with severe traffic problems. To deal with the situation, you have decided to make a new plan for the street grid. As it is impossible to make the streets wider, your approach is to make them one-way (only traffic in one direction is allowed on a street), thus creating a more efficient flow of traffic.

The streets in the city form an orthogonal grid – like on Manhattan *avenues* run in north-south-direction, while *streets* run in east-west-direction. Your mission is to make all the streets and avenues one-way, i.e. fix the direction in which traffic is allowed, while maintaining a short driving distance between some ordered pairs of locations. More specifically, a *route* in the city is defined by two street-avenue crossings, the start and goal location. On a one-way street grid, a route has a *legal* path if it is possible to drive from the start location to the goal location along the path passing streets and avenues in their prescribed direction only.

A route does not define a specific path between the two locations – there may be many possible paths for each route. A legal path in a one-way street grid is considered *simple* if it requires at most one turn, i.e. a maximum of one street and one avenue need to be used for the path.

When travelling by car from one location to another, a simple path will be preferred over a non-simple one, since it is faster. However, as each street in the grid is one-way, there may always be routes for which no simple path exists. On your desk lies a list of important routes which you want to have simple paths after the re-design of the street grid.

Your task is to write a program that determines if it is possible to fix the directions of the one-way streets and avenues in such a way that each route in the list has at least one simple path.

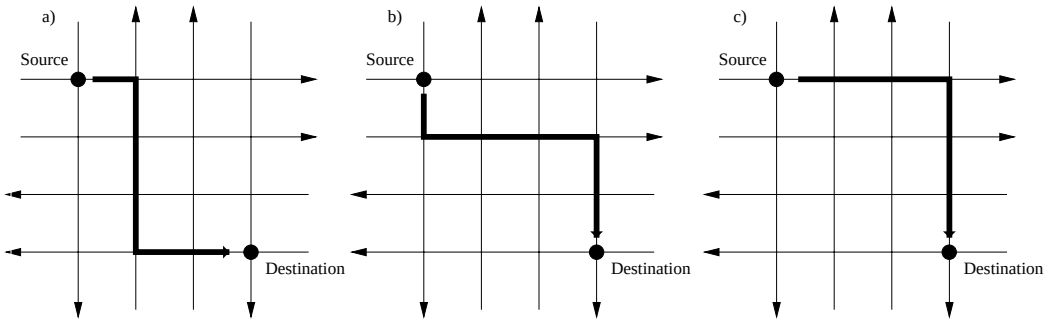


Fig 1. a) An illegal path. b) A legal but non-simple path. c) A simple path.

Input

On the first line of the input, there is a single integer n, telling how many city descriptions that follows. Each city description begins with a line containing three integers: the number of streets $0 < S \leq 30$, and avenues $0 < A \leq 30$ in the street grid, and the number of routes $0 < m \leq 200$ that should have at least one simple path. The next m lines define these routes, one on each line. Each route definition consists of four integers, s_1, a_1, s_2, a_2, where the start location of the route is at the crossing of street s_1 and avenue a_1, and the goal location is at the crossing of street s_2 and avenue a_2. Obviously, $0 < s_1, s_2 \leq S, 0 < a_1, a_2 \leq A$.

Output

For each city, your program should output 'Yes' on a single line if it is possible to make the streets and avenues one-way, so that each route has at least one simple path. Otherwise the text 'No' should be printed on a line of its own.

Example

```
Input:
3
6 6 2
1 1 6 6
6 6 1 1
7 7 4
1 1 1 6
6 1 6 6
6 6 1 1
4 3 5 1
9 8 6
2 2 4 4
4 5 3 2
3 4 2 2
3 2 4 4
4 5 2 2
2 1 3 4
```

```
Output:
Yes
No
No
```

Problem E
Mushroom Misery
source code: mushroom.*

The Institute of Ubiquitousness in Lichtenstein, LIU, conducts a project where the effect of a special type of fungi, *sphera carnelevarium*, are studied. This fungus is very special, since it grows in a circular fashion from its centre, without interference from other objects or other individuals.

Until now, the impact of this forest-living creature has been measured by examining a grid consisting of $n \times n$ squares of size 1 m^2. A grid is considered affected if fungi are present in the square, otherwise it is considered clean. Grids where the fungus merely touches the border of the square are *not* considered affected.

Of course, the task of examining grids is tedious and now unnecessary, since Prof. Müggel discovered that the grow rate is exactly 2.718281828 m/day. Now, counting the number of affected squares should be like a stroll in the park, right? Since there are a lot of old data which will be compared to the new data, the rules of counting the squares must be the same. Do not use data types with unnecessary low precision in your calculations.

Input

On the first line of input there is one integer, $N \leq 50$, giving the number of test cases in the input. After this line, N test cases follows. Each test case starts with a line containing two integers $size, n$ such that $0 < size \leq 1000000, 0 \leq n \leq 1000$, where $size$ is the size of the grid in meters and n is the number of individuals in the test grid. After this line, n lines follows, each line consisting of three decimal number, $0 \leq x \leq size, 0 \leq y \leq size, 0 < r \leq size$, where x and y are the zero-based coordinates of the centre of the individual and r is the (estimated) radius in meters.

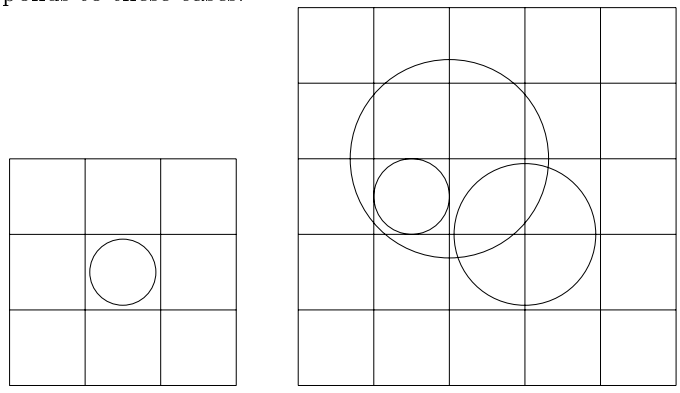
Output

For every test case, output one line containing the number of affected squares (a number between 0 and $size^2$). However, the number of affected squares are always less than 1000000000 (10^9).

Example

Input:	Output:
2	1
3 1	13
1.5 1.5 0.44	
5 3	
2 2 1.31	
1.5 2.5 0.5	
3 3 0.94	

The example corresponds to these cases:



Problem F
Board Wrapping
source code: wrapping.*

The small sawmill in Mission, British Columbia, has developed a brand new way of packaging boards for drying. By fixating the boards in special moulds, the board can dry efficiently in a drying room.

Space is an issue though. The boards cannot be too close, because then the drying will be too slow. On the other hand, one wants to use the drying room efficiently.

Looking at it from a 2-D perspective, your task is to calculate the fraction between the space occupied by the boards to the total space occupied by the mould. Now, the mould is surrounded by an aluminium frame of negligible thickness, following the hull of the boards' corners tightly. The space occupied by the mould would thus be the interior of the frame.

Input

On the first line of input there is one integer, $N \leq 50$, giving the number of test cases (moulds) in the input. After this line, N test cases follow. Each test case starts with a line containing one integer n, $1 < n \leq 1000$, which is the number of boards in the mould. Then n lines follow, each with five floating point numbers x, y, w, h, ϕ where $0 \leq x, y, w, h \leq 10000$ and $-90° < \phi \leq 90°$. The x and y are the coordinates of the center of the board and w and h are the width and height of the board, respectively. ϕ is the angle between the height axis of the board to the y-axis in degrees, positive clockwise. That is, if $\phi = 0$, the projection of the board on the x-axis would be w. Of course, the boards cannot intersect.

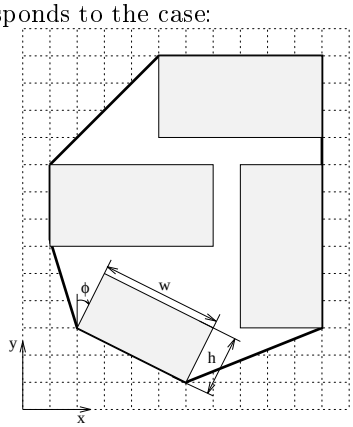
Output

For every test case, output one line containing the fraction of the space occupied by the boards to the total space in percent. Your output should have one decimal digit and be followed by a space and a percent sign ('%').

Example

```
Input:                              Output:
1                                   64.3 %
4
4 7.5 6 3 0
8 11.5 6 3 0
9.5 6 6 3 90
4.5 3 4.4721 2.2361 26.565
```

The example corresponds to the case:



Problem G
Circular Lock

source code: circular.*

For the fourth time this month only, the absent minded mathematics professor Bolkás had lost his keys. Realizing, that a change of personality to one that keep track of trivialities seemed unlikely, he started to invent a lock with a non-material key. Of course, he could just have chosen a combination lock, but as there were far too many numbers flying around in his daily thoughts, this idea was indeed a poor one. What he needed, was a lock which could be deactivated by solving a mathematical problem. For this type of lock, he could always be certain to have the "key" on him. However, for the sake of safety, the problem had to be hard enough for anyone trying to find the solution merely by testing all possibilities. Still, if you knew the mathematics behind the lock, you should be able to deactivate it.

Being a big fan of modular arithmetics, Bolkás designed the lock as a 2×2 array of four modular devices (cf. fig. 1). A modular device is a discrete counter with a display showing the current state of the counter, and a trigger. Each counter can be in any of the states $0, 1, .., p-1$, where $p \geq 1$ is the period of the counter. By trigging a counter, the state of the counter changes from i to $i+1$, except when the state is $p-1$, in which case the new state is 0. For each row and each column of the array, the professor put an increaser button on the panel of the lock. By pushing the increaser button for a row or column, both counters in that row or column are trigged. The lock is deactivated exactly when all four modular devices are in state 0. Bolkás also added a mechanism to activate the lock. Upon locking, each modular device is put in a randomly chosen state, not all equal to 0 (since in that case the lock would be immediately deactivated). Sadly enough, the professor soon found that for some quadruples of periods, it sometimes happened that the lock was put in a combination of states such that it could not be deactivated! However, it did not take him long to find an interesting relation between the locks, which helped him, and hopefully you, to decide if a lock could be deactivated or not:

He observed that a lock whose modular devices had the periods p_{11}, p_{12}, p_{21}, and p_{22}, and were put in the states s_{11}, s_{12}, s_{21}, and s_{22}, could be deactivated *if and only if* a lock with all modular devices having the same period $p = \gcd(p_{11}, p_{12}, p_{21}, p_{22})$, and the states $s'_{ij} = s_{ij} \bmod p$, could. The function $\gcd(a, b, c, d)$ is the greatest common divider of a, b, c, and d.

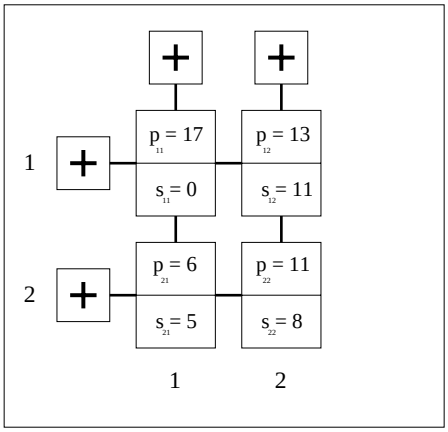


Fig 1. Pressing the increaser buttons of row 2 once, and column 2 twice, deactivates the lock.

Input

The first line of input consists of a single positive integer n. Thereafter n scenarios follow. Each scenario consists of two rows of four integers each. Row i of each scenario, $(1 \leq i \leq 2)$, contains the four integers s_{i1}, s_{i2}, p_{i1}, and p_{i2} in that order. s_{ij} and p_{ij} are the current state and period respectively, of the modular device at row i and column j. No period p_{ij} is larger than 10000, and $0 \leq s_{ij} < p_{ij}$.

Output

For each scenario, output one line containing the text "Yes", if it is possible to deactivate the lock by pressing the four row and column increaser buttons in some order. Otherwise the text "No" should be printed.

Example

Input:	Output:
2	Yes
0 11 17 13	No
5 8 6 11	
1 1 3 3	
1 0 3 3	