Problem A. Monster Path

This contest is open for practice. You can try every problem as many times as you like, though we won't keep track of which problems you solve. Read the Quick-Start Guide to get started.

Small input 7 points	Solve A-small
Large input 8 points	Solve A-large

Problem

Codejamon is a mobile game in which monster trainers walk around in the real world to catch monsters. You have an old smartphone with a short battery life, so you need to choose your path carefully to catch as many monsters as possible.

Suppose the Codejamon world is a rectangular grid with **R** rows and **C** columns. Rows are numbered from top to bottom, starting from 0; columns are numbered from left to right, starting from 0. You start in the cell in the $\mathbf{R_S}$th row and the $\mathbf{C_S}$th column. You will take a total of **S** unit steps; each step must be to a cell sharing an edge (not just a corner) with your current cell.

Whenever you take a step into a cell in which you have not already caught a monster, you will catch the monster in that cell with probability **P** if the cell has a monster attractor, or **Q** otherwise. If you do catch the monster in a cell, it goes away, and you cannot catch any more monsters in that cell, even on future visits. If you do not catch the monster in a cell, you may still try to catch the monster on a future visit to that cell. The starting cell is special: you have no chance of catching a monster there before taking your first step.

If you plan your path optimally before making any move, what is the maximum possible expected number of monsters that you will be able to catch?

The battery can only support limited steps, so hurry up!

Input

The first line of the input gives the number of test cases, **T**. **T** test cases follow.

Each test case starts with a line of five integers: **R**, **C**, $\mathbf{R_S}$, $\mathbf{C_S}$ and **S**. **R** and **C** are the numbers of rows and columns in the grid; $\mathbf{R_S}$ and $\mathbf{C_S}$ are the numbers of the row and column of your starting position, and **S** is the number of steps you are allowed to take.

The next line contains two decimals **P** and **Q**, where **P** is the probability of meeting a monster in cells with a monster attractor, and **Q** is the probability of meeting a monster in cells without a monster attractor. **P** and **Q** are each given to exactly four decimal places.

Each of the next **R** lines contains contains **C** space-separated characters; the j-th character of the i-th line represents the cell at row i and column j. Each element is either `.` (meaning there is no attractor in that cell) or `A` (meaning there is an attractor in that cell).

Output

For each test case, output one line containing `Case #x: y`, where x is the test case number (starting from 1) and y is the largest possible expected number of monsters that the player can catch in the given amount of steps.

y will be considered correct if y is within an absolute or relative error of 10^{-6} of the correct answer. See the FAQ for an explanation of what that means, and what formats of real numbers we accept.

Limits

$1 \leq \mathbf{T} \leq 100$.
$0 \leq \mathbf{R_S} < \mathbf{R}$.
$0 \leq \mathbf{C_S} < \mathbf{C}$.
$0 \leq \mathbf{Q} < \mathbf{P} \leq 1$.

Small dataset

$1 \leq \mathbf{R} \leq 10$.
$1 \leq \mathbf{C} \leq 10$.
$0 \leq \mathbf{S} \leq 5$.

Large dataset

$1 \leq \mathbf{R} \leq 20$.
$1 \leq \mathbf{C} \leq 20$.

$0 \leq \mathbf{S} \leq 9$.

Sample

Input	Output
2 4 4 0 0 5 0.8000 0.2000 . . . . A . . A . A 10 10 9 1 4 0.6121 0.1000 . . A A . . . . . . A . . . . A . . . . A . . . . . A A . . . . . . A A A . . . . . A A . A A . . . . A . . A . . . . . A . . . . . . A A . . . . . . A . . . A . . A . . . . A . . A . .	Case #1: 1.6000000 Case #2: 1.0495336

In Case #1, one of the best paths is (0,0)->(0,1)->(0,2)->(1,2)->(2,2)->(2,3). On this path, the expected number of monsters that you will catch is 0.2 + 0.2 + 0.2 + 0.8 + 0.2 = 1.6. Remember that there is no chance of catching a monster before taking your first step, which is why there are five probabilities in the calculation, not six.

In Case #2, one of the best paths is (9,1)->(9,2)->(8,2)->(8,3)->(8,2). On this path, the expected number of monsters that you will catch is 0.1 + 0.6121 + 0.1 + 0.23743359 = 1.04953359. Since we accept results within an absolute or relative error of 10^{-6} of the correct answer (1.04953359), 1.0495336 is accepted.

Problem B. Safe Squares

Small input 6 points	Solve B-small
Large input 13 points	Solve B-large

Problem

Codejamon trainers are actively looking for monsters, but if you are not a trainer, these monsters could be really dangerous for you. You might want to find safe places that do not have any monsters!

Consider our world as a grid, and some of the cells have been occupied by monsters. We define a *safe square* as a grid-aligned **D** × **D** square of grid cells (with **D** ≥ 1) that does not contain any monsters. Your task is to find out how many safe squares (of any size) we have in the entire world.

Input

The first line of the input gives the number of test cases, **T**. **T** test cases follow. Each test case starts with a line with three integers, **R**, **C**, and **K**. The grid has **R** rows and **C** columns, and contains **K** monsters. **K** more lines follow; each contains two integers $\mathbf{R_i}$ and $\mathbf{C_i}$, indicating the row and column that the i-th monster is in. (Rows are numbered from top to bottom, starting from 0; columns are numbered from left to right, starting from 0.)

Output

For each test case, output one line containing `Case #x: y`, where x is the test case number (starting from 1) and y is the the total number of safe zones for this test case.

Limits

1 ≤ **T** ≤ 20.

($\mathbf{R_i}$, $\mathbf{C_i}$) ≠ ($\mathbf{R_j}$, $\mathbf{C_j}$) for i ≠ j. (No two monsters are in the same grid cell.)

0 ≤ $\mathbf{R_i}$ < **R**, i from 1 to **K**

0 ≤ $\mathbf{C_i}$ < **C**, i from 1 to **K**

Small dataset

1 ≤ **R** ≤ 10.

1 ≤ **C** ≤ 10.

0 ≤ **K** ≤ 10.

Large dataset

1 ≤ **R** ≤ 3000.

1 ≤ **C** ≤ 3000.

0 ≤ **K** ≤ 3000.

Sample

Input	Output
2 3 3 1 2 1 4 11 12 0 1 0 3 0 4 0 10 1 0 1 9 2 0 2 4 2 9 2 10 3 4 3 10	Case #1: 10 Case #2: 51

The grid of sample case #1 is:

```
0 0 0
0 0 0
0 1 0
```

Here, 0 represents a cell with no monster, and 1 represents a cell with a monster. It has 10 safe squares: 8 1x1 and 2 2x2.

The grid of sample case #2 is:

```
0 1 0 1 1 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 1 0
1 0 0 0 1 0 0 0 0 1 1
0 0 0 0 1 0 0 0 0 0 1
```

Note that sample case #2 will only appear in the Large dataset. It has 51 safe squares: 32 1x1, 13 2x2, 5 3x3, and 1 4x4.

Problem C. Evaluation

Small input 12 points	Solve C-small
Large input 15 points	Solve C-large

Problem

Given an unordered list of assignment statements, write a program to determine whether the assignment statements can be put in some order in which all variables can be evaluated.

For our problem, an assignment statement will consist of an assignment variable, an assignment operator, and an expression, in that order. Statements will be evaluated one at a time, in the order you choose for them. A variable can be evaluated if and only if it has been the assignment variable of a previous assignment statement.

To simplify the problem, all the expressions are single function calls. Functions can take an arbitrary number of arguments, including zero; a function with zero arguments is always valid, and a function with variable arguments is valid as long as all of the variables are evaluatable.

For example, for the following list of assignment statements:

```
a=f(b,c)
b=g()
c=h()
```

this is one order that makes every statement valid:

```
b=g()
c=h()
a=f(b,c)
```

This is because: (1) b and c can be evaluated because the expressions `g()` and `h()` don't depend on any variables; and (2)a can also be evaluated because expression a depends on b and c, which are evaluatable.

However, the order

```
b=g()
a=f(b,c)
c=h()
```

would not be valid, because `f(b, c)` has variable c as an argument, but variable c has not been an assignment variable yet.

Another example is: `a=f(a)`. This list of statements can't be evaluated because the expression `f(a)` depends on the variable a itself, which makes it impossible to evaluate the statement.

Input

The first line of the input gives the number of test cases, **T**. **T** test cases follow. The first line of each test case contains an integer **N**: the number of assignment statements. Then, **N** lines follow. Each contains one assignment statement.

Each assignment statement consists of three parts: the assignment variable, the assignment operator, and the expression, with no spaces in between. The assignment operator is always =. All expressions consist of a function name, then `(`, then zero or more comma-separated variable names, then `)`. All variables and function names consist of one or more lowercase English alphabet letters. No variable has the same name as a function. No variable will appear more than once as the assignment variable. However, variables may appear more than once in various functions (even within the same function), and functions may appear more than once.

Output

For each test case, output one line containing `Case #x: y`, where x is the test case number (starting from 1) and y is `GOOD` if all variables are evaluatable or `BAD` otherwise.

Limits

1 ≤ **T** ≤ 20.
All functions take between 0 and 10 arguments, inclusive. All variable names consist of between 1 and 20 lowercase English alphabet letters.

Small dataset

1 ≤ **N** ≤ 100.

Large dataset

1 ≤ **N** ≤ 1000.

Sample

Input	Output
4 3 a=f(b,c) b=g() c=h() 2 a=f(b) b=f(a) 2 aaa=foo(x,y) bbb=bar(aaa,bbb) 2 x=f() y=g(x,x)	Case #1: GOOD Case #2: BAD Case #3: BAD Case #4: GOOD

Problem D. Soldiers

Small input 16 points	Solve D-small
Large input 23 points	Solve D-large

Soldiers

General Alice and General Bob are playing a war game. There are **N** soldiers in the game. Each soldier has two stats: attack and defense.

Before the game starts, General Alice and General Bob will take turns selecting soldiers, with General Alice going first. In each turn, a player can select one soldier, as long as that soldier either has an attack stat greater than each of the attack stats of all soldiers selected so far, *or* has a defense stat greater than each of the defense stats of all soldiers selected so far. To be precise: let $\mathbf{A}_i$ and $\mathbf{D}_i$ be the attack and defense values for the i-th soldiers, for i from 1 to **N**, and let S be the set of soldiers that have been selected so far. Then a player can select soldier x if and only if at least one of the following is true:

- $\mathbf{A}_x > \mathbf{A}_s$ for all s in S
- $\mathbf{D}_x > \mathbf{D}_s$ for all s in S

If no selection can be made, then the selection process ends and the players start playing the game.

General Alice wants to select more soldiers than General Bob, and General Bob wants to avoid that. If both players play optimally to accomplish their goals, can General Alice succeed?

Input

The first line of each case contains a positive integer **N**, the number of soldiers. **N** more lines follow; the i-th of these line contains two integers $\mathbf{A}_i$ and $\mathbf{D}_i$, indicating the attack and defense stats of the i-th soldier.

Output

For each test case, output one line containing `Case #x: y`, where x is the test case number (starting from 1) and y is YES or NO, indicating whether General Alice can guarantee that she selects more soldiers than General Bob, even if General Bob plays optimally to prevent this.

Limits

$1 \le \mathbf{T} \le 10$;
$1 \le \mathbf{A}_k, \mathbf{D}_k \le 10000$.

Small dataset

$1 \le \mathbf{N} \le 200$.

Large dataset

$1 \le \mathbf{N} \le 4000$.

Sample

Input	Output
3 3 10 2 1 10 10 3 3 10 1 10 10 1 10 3 10 2 1 10 4 9	Case #1: NO Case #2: YES Case #3: YES