Problem A. A Knight's Journey

Input file: journey.in
Output file: journey.out

The knight is getting bored of seeing the same black and white squares again and again and has decided to make a journey around the world. Whenever a knight moves, it is two squares in one direction and one square perpendicular to this.

The world of a knight is the chessboard he is living on. Our knight lives on a chessboard that has a smaller area than a regular 8×8 board, but it is still rectangular. Can you help this adventurous knight to make travel plans?

Find a path such that the knight visits every square once. The knight can start and end on any square of the board.

Input

The input begins with a positive integer n in the first line. The following lines contain n test cases. Each test case consists of a single line with two positive integers p and q, such that $1 \leq pq \leq 26$.

This represents a $p \times q$ chessboard, where p describes how many different square numbers $1, \ldots, p$ exist, q describes how many different square letters exist. These are the first q letters of the Latin alphabet: $A, \ldots$.

Output

The output for every scenario begins with a line containing "`Scenario #i:`", where i is the number of the scenario starting at 1. Then print a single line containing the lexicographically first path that visits all squares of the chessboard with knight moves followed by an empty line. The path should be given on a single line by concatenating the names of the visited squares. Each square name consists of a capital letter followed by a number.

If no such path exist, you should output `impossible` on a single line.

Example

journey.in	journey.out
3 1 1 2 3 4 3	Scenario #1: A1 Scenario #2: impossible Scenario #3: A1B3C1A2B4C2A3B1C3A4B2C4

Problem B. Line Segments

Input file: line.in
Output file: line.out

Line segments are a very common element in computational geometry. A line segment is the set of points forming the shortest path between two points (including those points). Although they are a very basic concept it can be hard to work with them if they appear in huge numbers unless you have an efficient algorithm.

Given a set of line segments, count how many distinct pairs of line segments are overlapping. Two line segments are said to be overlapping if they intersect in an infinite number of points.

Input

The first line contains the number of scenarios.

Each scenario starts with the number n of line segments ($1 \leq n \leq 100000$). Then follow n lines consisting of four integers x_1, y_1, x_2, y_2 in the range $[0, 1000000]$ each, representing a line segment that connects the points (x_1, y_1) and (x_2, y_2). It is guaranteed that a line segment does not degenerate to a single point.

Output

The output for every scenario begins with a line containing "`Scenario #i:`", where i is the number of the scenario starting at 1. Then print a single line containing the number of distinct pairs of overlapping line segments followed by an empty line.

Example

line.in	line.out
2 8 1 1 2 2 2 2 3 3 1 3 3 1 10 0 20 0 20 0 30 0 15 0 25 0 50 0 100 0 70 0 80 0 1 0 0 1 1	Scenario #1: 3 Scenario #2: 0

Problem C. Pimp My Ride

Input file: pimp.in
Output file: pimp.out

Today, there are quite a few cars, motorcycles, trucks and other vehicles out there on the streets that would seriously need some refurbishment. You have taken on this job, ripping off a few dollars from a major TV station along the way.

Of course, there's a lot of work to do, and you have decided that it's getting too much. Therefore you want to have the various jobs like painting, interior decoration and so on done by garages. Unfortunately, those garages are very specialized, so you need different garages for different jobs. More so, they tend to charge you the more the better the overall appearance of the car is. That is, a painter might charge more for a car whose interior is all leather. As those "surcharges"depend on what job is done and which jobs have been done before, you are currently trying to save money by finding an optimal order for those jobs.

Individual jobs are numbered 1 through n. Given the base price p for each job and a surcharge s (in US$) for every pair of jobs (i, j) with $i \neq j$, meaning that you have to pay additional $\$s$ for job i, if and only if job j was completed before, you are to compute the minimum total costs needed to finish all jobs.

Input

The first line contains the number of scenarios. For each scenario, an integer number of jobs n, $1 \leq n \leq 14$, is given. Then follow n lines, each containing exactly n integers. The i-th line contains the surcharges that have to be paid in garage number i for the i-th job and the base price for job i. More precisely, on the i-th line, the i-th integer is the base price for job i and the j-th integer ($j \neq i$) is the surcharge for job i that applies if job j has been done before. The prices will be non-negative integers smaller than or equal to 100000.

Output

The output for every scenario begins with a line containing "`Scenario #i:`", where i is the number of the scenario starting at 1. Then print a single line:

"`You have officially been pimped for only $p`"

with p being the minimum total price. Terminate the output for the scenario with a blank line.

Example

pimp.in
2 2 10 10 9000 10 3 14 23 0 0 14 0 1000 9500 14

pimp.out
Scenario #1: You have officially been pimped for only $30 Scenario #2: You have officially been pimped for only $42

Problem D. A Bug's Life

Input file: bugs.in
Output file: bugs.out

Professor Hopper is researching the sexual behavior of a rare species of bugs. He assumes that they feature two different genders and that they only interact with bugs of the opposite gender. In his experiment, individual bugs and their interactions were easy to identify, because numbers were printed on their backs.

Given a list of bug interactions, decide whether the experiment supports his assumption of two genders with no homosexual bugs or if it contains some bug interactions that falsify it.

Input

The first line of the input contains the number of scenarios. Each scenario starts with one line giving the number of bugs (at least one, and up to 2000) and the number of interactions (up to 1000000) separated by a single space. In the following lines, each interaction is given in the form of two distinct bug numbers separated by a single space. Bugs are numbered consecutively starting from one.

Output

The output for every scenario is a line containing "`Scenario #i:`", where i is the number of the scenario starting at 1, followed by one line saying either "`No suspicious bugs found!`" if the experiment is consistent with his assumption about the bugs' sexual behavior, or "`Suspicious bugs found!`" if Professor Hopper's assumption is definitely wrong.

Example

bugs.in	bugs.out
2 3 3 1 2 2 3 1 3 4 2 1 2 3 4	Scenario #1: Suspicious bugs found! Scenario #2: No suspicious bugs found!

Problem E. Rdeaalbe

Input file: rdeaalbe.in
Output file: rdeaalbe.out

As you probably know, the human information processor is a wonderful text recognizer that can handle even sentences that are garbled like the following:

`The ACM Itrenntaoial Clloegaite Porgarmmnig Cnotset (IPCC) porvdies clolgee stuetnds wtih ooppriuntetiis to itnrecat wtih sutednts form ohetr uinevsrtieis.`

People have claimed that understanding these sentences works in general when using the following rule: The first and last letters of each word remain unmodified and all the characters in the middle can be reordered freely.

Since you are a skeptical ACM programmer, you immediately set on to write the following program: Given a sentence and a dictionary of words, how many different sentences can you find that could potentially be mapped to the same encoding?

Input

The first line contains the number of scenarios. Each scenario begins with a line containing the number n of words in the dictionary ($0 \leq n \leq 10000$), which are printed on the following n lines. After this there is a line containing the number m of sentences that should be tested with the preceding dictionary ($0 \leq m \leq 10000$) and then m lines containing those sentences. The sentences consist of letters from a..z, A..Z and spaces only and have a maximal length of 10000 characters. For each word in the dictionary a limitation of 100 characters can be assumed.

Output

The output for every scenario begins with a line containing "`Scenario #i:`", where i is the number of the scenario starting at 1. For each sentence output the number of sentences that can be formed on an individual line. It is guaranteed that this number can be expressed using a signed 32-bit datatype. Terminate the output for the scenario with a blank line.

Example

rdeaalbe.in
2 3 ababa aabba abcaa 2 ababa abbaa 14 bakers brakes breaks binary brainy baggers beggars and in the blowed bowled barn bran 1 brainy bakers and beggars bowled in the barn

rdeaalbe.out
Scenario #1: 2 2 Scenario #2: 48

Problem F. Acid Text

Input file: acid.in
Output file: acid.out

A couple of months ago the web standards project (WaSP) has come up with a test for modern browsers and their CSS implementation called acid2. This test ensures that all the browsers have similar results when it comes to parsing and displaying cascaded style sheet files (CSS) for HTML. Since you want to beat all the other text-based browsers on standard compliance you directly start implementing the CSS capabilities into your favorite text-browser Lynks.

Your text-browser will be given a set of graphic files and a simplified css-style-sheet. A graphic is defined by a name, height, width and a 2-dimensional array of characters. All characters are to be printed

except for the character '.' which denotes a transparent pixel. Here is an example picture:

```
owl.png 5 7
.-----.
|O...O|
|..v..|
|.<_>.|
.-----.
```

Given the style-sheet your task is it to produce the graphical result that the browser is supposed to display. A CSS-file is made up from a number of entries where each entry looks like this:

```
#<id> {
 pos-x : <x> px ;
 pos-y : <y> px ;
 position : <relative = <id of graphic>|absolute> ;
 file : <filename> ;
 layer : <layer-number> ;
}
```

The following rules hold for the CSS-entries:

Lines Each CSS-entry will be given on exactly 7 lines as in the input above.

Ordering Each CSS-entry will contain exactly the 5 attributes pos-x, pos-y, position, file and layer, in this order, each attribute on a separate line.

Whitespace There may be zero or more white-spaces (spaces and tabs) at the beginning of lines, at the end of lines or everywhere where the sample above has a space.

Here are the rules for composing the picture:

Background The background is assumed to be black (i.e. just spaces).

Positioning The top left corner of the viewing device is assumed to be x : 0, y : 0. Absolute positioning always is based on this top-left corner. Relative positioning information is always based on the topleft pixel of another graphic. There will not be any circular references between CSS elements. All resulting positions will be zero or greater in x and y.

Layering Graphics with a higher layer number are to be printed after graphics with a lower layer number. Graphics with the same layer number are to be printed in the order they appear in the CSS.

Input

The first line of the input is the number of scenarios that will follow. For each scenario the following information is given: The first line contains the number of files to follow (at least one, at most 100), each of which is given by a space separated triple of a filename f, a height h, a width w ($1 \leq w, h \leq 100$) and then h lines, each with exactly w characters. Following the file definition is a single line with a number m (at least one, at most 500), which is followed by a CSS file of m entries.

You can assume the resulting picture to be at most 1000×1000 characters large. All coordinates in CSS entries will be given as integers with an absolute value less than 1000000. All filenames and identifiers are made up from alphanumeric characters and dots only. No two files have the same name and no two identifiers are equal. The layer attribute will be at least 0 and at most 1000000.

Output

The output for every scenario begins with a line containing "`Scenario #i:`", where i is the number of the scenario starting at 1. For each scenario print the resulting picture from overlaying all the given graphics following the instructions in the CSS file. Your result for each scenario should be rectangular as small as possible. However, transparent pixels always belong to the resulting picture, even if they are located directly at the border. The top-left corner of the result should always contain position (0, 0). All empty areas should be padded with spaces. Terminate the output for every scenario with a blank line.

Example

acid.in

```
1
4
bg.png 5 7
.-----.
|.....|
|.....|
|.....|
.-----.
eye.jpg 1 1
O
nose.bmp 1 1
v
mouth.png 1 3
<_>
5
#bg {
  pos-x: 1 px;
  pos-y: 1 px;
  position: absolute;
  file: bg.png;
  layer: 0;
}
#leftEye {
  pos-x: 1 px;
  pos-y: 1 px;
  position: relative=bg;
  file: eye.jpg;
  layer: 1;
}
#rightEye {
  pos-x: 4 px;
  pos-y: 0 px;
  position: relative=leftEye;
  file: eye.jpg;
  layer: 1;
}
#nose {
  pos-x: 2 px;
  pos-y: 1 px;
  position: relative=leftEye;
  file: nose.bmp;
  layer: 1;
}
#mouth {
  pos-x: -1 px;
  pos-y: 1 px;
  position: relative = nose;
  file: mouth.png;
  layer: 1;
}
```

acid.out

```
Scenario #1:

 -----
|O   O|
|  v  |
| <_> |
 -----
```

Problem G. Diophantus of Alexandria

Input file:	`diophantus.in`
Output file:	`diophantus.out`

Diophantus of Alexandria was an Egypt mathematician living in Alexandria. He was one of the first mathematicians to study equations where variables were restricted to integral values. In honor of him, these equations are commonly called diophantine equations. One of the most famous diophantine equation is $x^n + y^n = z^n$. Fermat suggested that for $n > 2$, there are no solutions with positive integral values for x, y and z. A proof of this theorem (called Fermat's last theorem) was found only recently by Andrew Wiles.

Consider the following diophantine equation:

$$\frac{1}{x}+\frac{1}{y}=\frac{1}{n}$$

where $x, y, n \in \mathbb{N}^+$.

Diophantus is interested in the following question: for a given n, how many distinct solutions (i. e., solutions satisfying $x \le y$) does this equation have? For example, for $n = 4$, there are exactly three distinct solutions:

$$\frac{1}{5}+\frac{1}{20}=\frac{1}{4}$$
$$\frac{1}{6}+\frac{1}{12}=\frac{1}{4}$$
$$\frac{1}{8}+\frac{1}{8}=\frac{1}{4}$$

Clearly, enumerating these solutions can become tedious for bigger values of n. Can you help Diophantus compute the number of distinct solutions for big values of n quickly?

Input

The first line contains the number of scenarios. Each scenario consists of one line containing a single number n ($1 \le n \le 10^9$).

Output

The output for every scenario begins with a line containing "`Scenario #i:`", where i is the number of the scenario starting at 1. Next, print a single line with the number of distinct solutions of the equation for the given value of n. Terminate each scenario with a blank line.

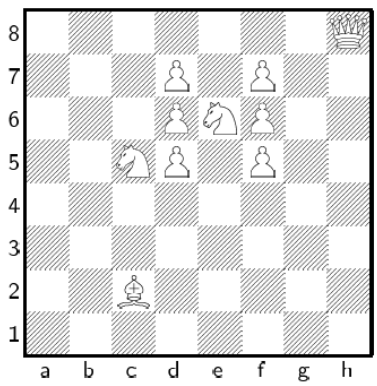
Example

diophantus.in	diophantus.out
2 4 1260	Scenario #1: 3 Scenario #2: 113

Problem H. Traveling Queen Problem

Input file: `queen.in`
Output file: `queen.out`

Black has been defeated and the white army has won, but unfortunately the white king has been killed in the fight, and so the white queen is looking for a new mate. She is unsure whom of the knights to marry and has decided to visit them all. Afterwards she plans to see the bishop to arrange for the marriage.



Given a chessboard with the current situation, find the shortest number of moves such that the queen visits every knight and, finally, visits the bishop.

The queen visits by standing on one of the (at most) eight neighbouring squares and she does not necessarily have to move between two visits. For each move the queen can go an arbitrary number of squares in one of the eight directions (horizontal, vertical or diagonal). No move may pass through or stop at a non-empty square.

Input

The first line contains the number of scenarios. Each scenario consists of a chessboard description. The rows $8, \ldots, 1$ are given in this order, one line per row. Each line contains 8 characters to represent the squares at columns $a, \ldots, h$ of this row. Each description is followed by an empty line. There is one character Q to denote the starting position of the queen, and one B to denote the square on which the bishop stands. There is an arbitrary number of pawns, given as P, who simply block movement, as well as $2 - 14$ knights denoted as N. All other squares are given as '.' and are empty.

Output

The output for every scenario begins with a line containing "`Scenario #i:`", where i is the number of the scenario starting at 1.

Then print the lexicographical first path that contains the minimal number of moves, ends adjacent to the bishop and visits each knight at least once. The path shall be given on a single line by concatenating in order the names of the squares on which the queen stands. Each square name consists of a small letter followed by a decimal digit. If no such path exists, output "`impossible`" on a single line. Terminate the output for the scenario with a blank line.

Example

queen.in	queen.out
2 ....Q ...P.P.. ...PNP.. ..NP.P.. ....B..... .... B.P..... ..P..... PPP..N.. .... .N...Q..	Scenario #1: h8h2e5d4b2 Scenario #2: impossible

Problem I. Mine Map

Input file: `minemap.in`
Output file: `minemap.out`

After the recent theft of the problemset for the ACM ICPC World finals (by notorious british super spy James B.-we reported on this some weeks ago), the ACM has decided to store all future problemsets in a high security building. The security board endowed with the job of creating this new vault had the brilliant idea to build it in the form of a giant maze. Essentially, this maze consists of a bunch of square rooms, arranged in the form of a square matrix, with all the rooms connected to each other by a series of doors. Going through them is the only way to get to the center where the problemsets are stored. Obviously, it is not that hard to get through a maze in which all room are connected to each other. So, to make things more dangerous for would-be intruders, some of the rooms are booby trapped with mines. If somebody enters the central room of the vault containing the problemsets, these mines are activated. Afterwards, opening a door leading to a room with a mine in it will trigger an alarm, and all security doors close immediately, trapping the intruder. This way, the ACM can find out who sent the spy and disqualify all teams of that nation.

But recently the security board became aware of a new scanning device able to detect the mines, once they are activated. This detector could be used from within any of the rooms of the vault, and would be able to tell the user whether any of the up to eight adjacent rooms contains a mine, or not. Unfortunately, the board already ordered a batch of these

mines, and now doesn't want to have to admit that this might have been a mistake. Instead, they simply want to spread the mines in such a way that it is difficult to leave the center by just using the device. Your are assigned to the team building the vault in order to help them evaluate their designs.

A vault has the form of a quadrangle, with sides that have odd length. Each room in the vault can be described by a pair of coordinates, indicating that horizontal and vertical offset relative to a fixed corner of the building. In each room, there are doors leading to the neighboring rooms; more importantly, the mine detector can detect mines in all of the up to eight adjacent rooms. The device can only tell you whether there are any mines nearby, but not how many there are. Your job is to create a special map from each of the vault design drafts. On the map, mark all rooms somebody starting from the center room (which is guaranteed to not contain a mine) could safely reach with the help of the new detector and the following simple strategy: When you are in a room where the detector reports no adjacent mines, search all surrounding rooms. Otherwise, do not risk triggering a mine and do not advance farther from this room (you might reach one of the surrounding rooms via another "safe" route later on, though).

Thus, if the intruder is in a room not next to any mines, he will be able to go to all surrounding rooms- mark such a room with a ".". If the intruder enters a room which is next to one or more mines, he will retreat-mark these rooms with a "#". To be able to verify your work, the security board also wants you to mark the position of each mine with a "*". Finally, all remaining rooms should be marked with a "?".

Input

The first line contains the number of scenarios. Each scenario starts with a line containing the odd integer n ($1 < n < 300$) of the vault, indicating the length of one of its outer walls. This is followed by the number m of mines (which is positive and only limited by the number of rooms in the vault).

Next comes m lines, each containing two integers r and c ($1 \le r, c \le n$), which give the row and the column of a mine.

Output

The output for every scenario begins with a line containing "`Scenario #i:`", where i is the number of the scenario starting at 1. Then print an ASCII representation of the map of the vault as described above. Terminate the output for the scenario with a blank line.

Example

minemap.in	minemap.out
3 3 1 1 2 5 4 3 1 1 3 3 5 5 3 5 2 1 1 5 3	Scenario #1: ?*? ?#? ??? Scenario #2: ??*?? ?###? *#.#* ?###? ??*?? Scenario #3: *#... ##... ..... .###. .#*#.

Problem J. Knights of the Round Table

Input file: `knights.in`
Output file: `knights.out`

It is the year 573 AD. King Arthur rules all of Britain. He is a just ruler, loved by the people. But governing such a big country is a daunting task, and doing it requires a lot of effort. Luckily, in this HE is aided by the most noble Knights of the Round Table. Initially, this body was composed only out of the most valiant heroes to be found in the whole kingdom, like Sir Gawain, Sir Lancelot, or Sir Galahad. Recently, though, King Arthur found it to be a political necessity to admit additional members to this most holy round. The main reason for this is that due to the lasting peace under his reign, the guilds demanded more influence on the ruling of the country. "After all, in times of peace you need bread more than swords, don't you? said the head of the guild of bakers. Being an exceptionally just and noble ruler, Arthur agreed.

This has lead to a certain swelling of the ranks of the "knights"of the round table. Arthur found that he had to order a much bigger table from the guild of carpenters (who in turn immediately demanded some seats on it), and subsequently had to add a new wing to Castle Camelot to contain it (you may guess what the stone masons requested afterwards).

As a result, the weekly meetings of the Round Table are rather crowded now. In fact, there are so many people that it can be quite hard to understand each other when sitting at opposites sides of the table. To find out how big the problem really is, Arthur wants you to compute how far away the two farthest members of the Round Table sit from each other. Things are complicated by the fact that there is only a finite number of people sitting at the table, and they are not spread equally around the table-some of them are sitting closely together (to discuss important matters), while others prefer to distance themselves from their neighbors (e. g., the representative of the guild of healers and the head of the guild of assassins).

Input

The first line contains the number of scenarios. Each scenario starts with a line containing the number n of chairs ($3 \le n \le 10\,000$). After that follow n lines, each consisting of two integers p and q ($0 \le p < q < 10^9$), denoting the angle $2\pi p/q$.

Output

The output for every scenario begins with a line containing "`Scenario #i:`", where i is the number of the scenario starting at 1. Then print the maximal occurring distance, rounded to two digits after the decimal point. Terminate the output for the scenario with a blank line.

Example

knights.in	knights.out
1 4 0 1 1 4 2 3 4 5	Scenario #1: 7.90

Problem K. Honeymoon Hike

Input file: `honeymoon.in`
Output file: `honeymoon.out`

Emma is on a hiking trip with Eric, her freshly-married husband, for their honeymoon. They are hiking from one cabin to the next every day. Unfortunately, Eric is not as fit as Emma and is slowly getting tired. Since Emma does not want to start their newly-formed marriage with a serious conflict (and needs somebody to keep her warm in the nights), she decides to plan the next day trips so that they are not so strenuous for Eric.

In the past days, Emma has discovered a surprising fact about her husband. He is not so much tired by the length of their daily trip or the total amount of meters they had to climb. Instead, Eric is tired the more, the bigger the difference between the highest and the lowest point of today's route becomes. Emma assumes this is due to psychological factors. It just sounds a lot more difficult to climb once from 500 meter to 1 500 meters than to climb from 200 to 400 meters ten times, although you climbed twice as much in the latter case.

Given an altitude map of the terrain, you should help Emma in finding a path that minimizes the difference between its highest and its lowest elevation, so that Eric does not feel as tired. The cabin they start at is located at the top-left corner and their destination is the bottom-right corner of the map. They can move along any of the four major directions but not on a diagonal.

Input

The first line contains the number of scenarios. Each scenario starts with a number n ($2 \leq n \leq 100$), the size of the area. The elevations of the terrain are given as a $n \times n$ integer matrix $(h_{i,j})$ ($0 \leq h_{i,j} \leq 200$) on n lines, where each line contains n space-separated elevations.

Output

The output for every scenario begins with a line containing "`Scenario #i:`", where i is the number of the scenario starting at 1. Then, print a single line containing the difference between the highest and the lowest elevation on the optimal path. Terminate the output for the scenario with a blank line.

Example

honeymoon.in	honeymoon.out
1 5 1 1 3 6 8 1 2 2 5 5 4 4 0 3 3 8 0 2 2 4 4 3 0 3 1	Scenario #1: 3

Problem L. Relocation

Input file: `relocation.in`
Output file: `relocation.out`

Emma and Eric are moving to their new house they bought after returning from their honeymoon. Fortunately, they have a few friends helping them relocate. To move the furniture, they only have two compact cars, which complicates everything a bit. Since the furniture does not fit into the cars, Eric wants to put them on top of the cars. However, both cars only support a certain weight on their roof, so they will have to do several trips to transport everything. The schedule for the move is planed like this:

1. At their old place, they will put furniture on both cars.
2. Then, they will drive to their new place with the two cars and carry the furniture upstairs.
3. Finally, everybody will return to their old place and the process continues until everything is moved to the new place.

Note, that the group is always staying together so that they can have more fun and nobody feels lonely. Since the distance between the houses is quite large, Eric wants to make as few trips as possible.

Given the weights w_i of each individual piece of furniture and the capacities C_1 and C_2 of the two cars, how many trips to the new house does the party have to make to move all the furniture? If a car has capacity C, the sum of the weights of all the furniture it loads for one trip can be at most C.

Input

The first line contains the number of scenarios. Each scenario consists of one line containing three numbers n, C_1 and C_2. C_1 and C_2 are the capacities of the cars ($1 \leq C_i \leq 100$) and n is the number of pieces of furniture ($1 \leq n \leq 10$). The following line will contain n integers $w_1, \ldots, w_n$, the weights of the furniture ($1 \leq w_i \leq 100$). It is guaranteed that each piece of furniture can be loaded by at least one of the two cars.

Output

The output for every scenario begins with a line containing "`Scenario #i:`", where i is the number of the scenario starting at 1. Then print a single line with the number of trips to the new house they have to make to move all the furniture. Terminate each scenario with a blank line.

Example

relocation.in	relocation.out
2 6 12 13 3 9 13 3 10 11 7 1 100 1 2 33 50 50 67 98	Scenario #1: 2 Scenario #2: 3