Problem A. Bubbly Troubly

Source file name:	bubbly.c, bubbly.cpp, bubbly.java, bubbly.py
Input:	`Standard`
Output:	`Standard`

You may have seen a champagne tower at a wedding or an exclusive Hollywood A-list party. In a typical three-level tower, the first (lowest) level contains 9 glasses touching in a square pattern. The second level contains 4 glasses touching in a square pattern centered above the first level, and the third level contains 1 glass centered above the other levels. Figure A.1 shows a a top-down view of this tower.

Figure A.1

Champagne is always poured directly into the top glass. In this example, once the top glass fills and starts to overflow, it *immediately* begins filling the 4 glasses below (i.e., assume overflowing champagne travels instantaneously to any glasses below). Once the 4 glasses on the second level fill, they begin overflowing to the 9 on the bottom level. Note that in this example the 4 on the second level finish filling at the same time, but the 9 on the lowest level finish filling at different times. This means that there will be some amount of spilled champagne before the tower has finished filling – this is an acceptable price to pay for such a beautiful sight.

The new fad is to make interesting patterns or imagery out of champagne glasses. These new-fangled "towers" needn't appear structurally sound; they can be held in place with complex support systems designed so as not to interfere with the overflowing champagne. Each of these new towers will always have a single highest glass into which all champagne is directly poured.

If two glass rims coincide vertically (i.e., have the same center and radius), then no accumulation occurs into the lower glass from the upper glass (though the overflow champagne from the upper glass may still be collected by other lower glasses). Additionally, a single point of champagne overflow causes no measurable accumulation. In other words, measurable accumulation only occurs when a non-point arc of champagne overflows to the interior of a glass.

Your task is to determine whether a proposed champagne tower will fill to completion, and if so, how long it will take.

Input

The input begins with a single integer n representing the number of champagne glasses in the tower ($1 \leq n \leq 20$). The next n lines each describe a champagne glass. Each glass description consists of 5 values x y z r v with (x, y, z) representing the center of the glass's rim ($0 \leq x, y \leq 1000$; $1 \leq z \leq 1000$), r representing its radius ($1 \leq r \leq 1000$), and v representing its volume measured in milliliters ($1 \leq v \leq 1000$). All input values are integers, and the top glass is filled at a constant 100 milliliters per second.

Output

Display the number of seconds after which the tower will be completely filled, or `Invalid` if the proposed champagne tower will never fill completely. Round answers to the hundredths place. Always print answers to two decimal places and include the leading 0 on answers between 0 and 1. Output values will always be $\leq 10^6$ seconds (or 11 days, 13 hours, 46 minutes and 40 seconds, whichever you prefer).

Example

Input	Output
14 0 0 1 1 400 0 2 1 1 400 0 4 1 1 400 2 0 1 1 400 2 2 1 1 400 2 4 1 1 400 4 0 1 1 400 4 2 1 1 400 4 4 1 1 400 1 1 2 1 400 1 3 2 1 400 3 1 2 1 400 3 3 2 1 400 2 2 3 1 400	84.00
2 2 1 2 2 10 0 0 1 1 10	0.78
2 0 0 1 1 100 10 10 2 1 100	Invalid

Explanation

Glasses with the same z coordinate do not intersect.

Problem B. Foosball Dynasty

Source file name:	foosball.c, foosball.cpp, foosball.java, foosball.py
Input:	`Standard`
Output:	`Standard`

Balaji and his coworkers like to play Foosball on their lunch break. Foosball is a game typically played by 2 players (individual matchup) or 4 players (team play). However, due to the increasing interest and availability of coworkers, Balaji has created a new variation that supports 5 or more players. Each individual point is played by four players: two on the White team and two on the Black team.

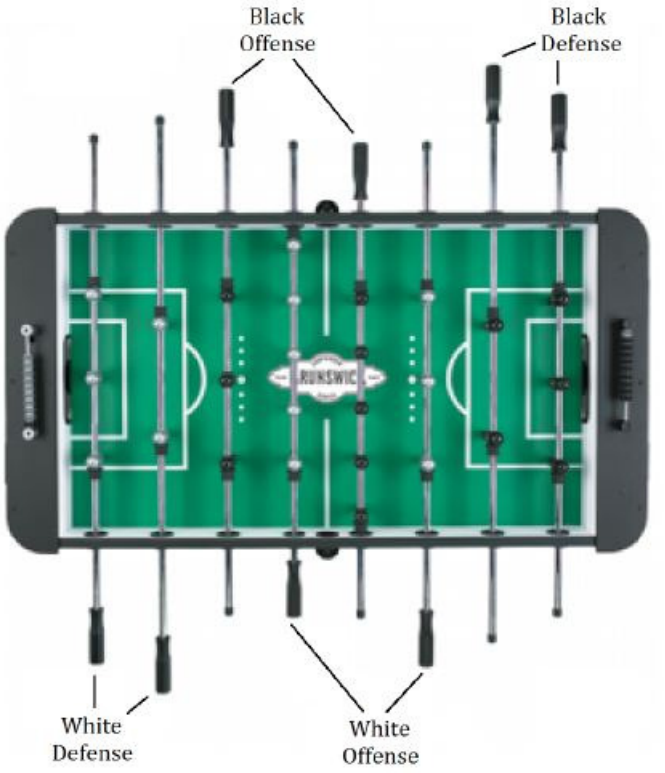

Figure B.1

The remaining players wait in line for their turn to play. On each point, the players on the scoring team switch positions (e.g., if White scores the point, then the White Offense player becomes the White Defense player, and vice versa). The Offense player of the team that lost the point becomes the Defense player of the same team, while the Defense player of the team that lost the point goes to the back of the line and waits for their next chance to play. The person at the front of the line becomes the new Offense player of the team that lost the point.

Based on these rules, a team that continues scoring consecutive points gets to keep playing together (swapping positions with each other after each point) until the other team breaks the streak. Each such streak of points creates a dynasty for the winning team. Dynasties can be short-lived (a single point) or long-lived, but they are always broken when the opposing team scores a point. The "winners" of this variation of foosball are the players that can create the longest dynasty.

Input

Input begins with a line containing an integer n representing the number of players ($5 \leq n \leq 10$). The next line contains the n names of all participating players. The first four are the names of the four players

who initially arrive at the table (in that order): the first person to arrive starts at White Offense, the second starts at Black Offense, the third starts at White Defense, and the fourth starts at Black Defense. The remaining players get in line to wait for their turn. The final input line contains a non-empty string indicating which side scored each point, with a White team point represented by 'W' and a Black team point represented by 'B'. The maximum length of this string will be 1000.

Output

Display the team that has achieved the longest dynasty. If more than one team ties for the record, then display each of these teams in chronological order, one team per line. When displaying a team, names should be displayed in the order in which the players arrived at the table for that team. Note that it is possible for the same team to appear more than once in the output, potentially with the player names in a different order.

Example

Input	Output
6 Balaji David Alex Scott Andrew Ravi WWBWBBWBW	Balaji Alex Andrew David
6 Amy Jinu Kasey Sarah Sheetal Julia BBBBB	Jinu Sarah

Problem C. The Key to Cryptography

Source file name:	cryptography.c, cryptography.cpp, cryptography.java, cryptography.py
Input:	`Standard`
Output:	`Standard`

Suppose you need to encrypt a top secret message like "SEND MORE MONKEYS". You could use a simple substitution cipher, where each letter in the alphabet is replaced with a different letter. However, these ciphers are easily broken by using the fact that certain letters of the alphabet (like 'E', 'S', and 'A') appear more frequently than others (like 'Q', 'Z', and 'X'). A better encryption scheme would vary the substitutions used for each letter. One such system is the *autokey* cipher.

To encrypt a message, you first select a secret word – say "ACM" – and prepend it to the front of the message. This longer string is truncated to the length of the message and called the key, and the n^{th} letter of the key is used to encrypt the n^{th} letter of the original message. This encryption is done by treating each letter in the key as a cyclic shift value for the corresponding letter in the message, where 'A' indicates a shift of 0, 'B' a shift of 1, and so on. Using "ACM" as the secret word, we would encrypt our message as follows:

```
SENDMOREMONKEYS   (message)
ACMSENDMOREMONK   (key)
SGZVQBUQAFRWSLC   (ciphertext)
```

Note that the letter 'E' in the message was encrypted as 'G' the first time it was encountered (since the corresponding letter in the key was 'C' indicating a shift of 2), but then as 'Q' and 'S' the next two times.

Your task is simple: given a ciphertext and the secret word, you must determine the original message.

Input

Input consists of two lines. The first contains the ciphertext and the second contains the secret word. Both lines contain only uppercase alphabetic letters. **Assume both lines are at most 1000 characters long.**

Output

Display the original message that generated the given ciphertext using the given secret word.

Example

Input	Output
SGZVQBUQAFRWSLC ACM	SENDMOREMONKEYS

Problem D. Lost in Translation

Source file name:	lost.c, lost.cpp, lost.java, lost.py
Input:	`Standard`
Output:	`Standard`

The word is out that you've just finished writing a book entitled *How to Ensure Victory at a Programming Contest* and requests are flying in. Not surprisingly, many of these requests are from foreign countries, and while you are versed in many programming languages, most spoken languages are Greek to you. You've done some investigating and have found several people who can translate between languages, but at various costs. In some cases multiple translations might be needed. For example, if you can't find a person who can translate your book from English to Swedish, but have one person who can translate from English to French and another from French to Swedish, then you're set. While minimizing the total cost of all these translations is important to you, the most important condition is to minimize each target language's distance (in translations) from English, since this cuts down on the errors that typically crop up during any translation. Fortunately, the method to solve this problem is in Chapter 7 of your new book, so you should have no problem in solving this, right?

Input

Input starts with a line containing two integers n, m indicating the number of target languages and the number of translators at your disposal ($1 \leq n \leq 100$, $1 \leq m \leq 4500$). The second line will contain n strings specifying the n target languages. After this line are m lines of the form l_1, l_2, c where l_1 and l_2 are two different languages and c is a positive integer specifying the cost to translate between them (in either direction). The languages l_1 and l_2 are always either English or one of the target languages, and any pair of languages will appear at most once in the input. The initial book is always written in English.

Output

Display the minimum cost to translate your book to all of the target languages, subject to the constraints described above, or `Impossible` if it is not possible.

Example

Input	Output
4 6 Pashto French Amheric Swedish English Pashto 1 English French 1 English Amheric 5 Pashto Amheric 1 Amheric Swedish 5 French Swedish 1	8
2 1 A B English B 1	Impossible

Problem E. Red Rover

Source file name:	redrover.c, redrover.cpp, redrover.java, redrover.py
Input:	`Standard`
Output:	`Standard`

One of our older Mars Rovers has nearly completed its tour of duty and is awaiting instructions for one last mission to explore the Martian surface. The survey team has picked a route and has entrusted you with the job of transmitting the final set of instructions to the rover. This route is simply a sequence of moves in the cardinal directions: North, South, East, and West. These instructions can be sent using a string of corresponding characters: `N, S, E`, and `W`. However, receiving the signal drains the rover's power supply, which is already dangerously low. Fortunately, the rover's creators built in the ability for you to optionally define a single "macro" that can be used if the route has a lot of repetition. More concretely, to send a message with a macro, two strings are sent. The first is over the characters {`N,S,E,W,M`} and the second is over {`N,S,E,W`}. The first string represents a sequence of moves and calls to a macro (`M`), while the second string determines what the macro expands out to. For example:

```
WNMWMME
EEN
```

is an encoding of

```
WNEENWEENEENE
```

Notice that the version with macros requires only 10 characters, whereas the original requires 13.

Given a route, determine the minimum number of characters needed to transmit it to the rover.

Input

Input consists of a single line containing a string made up of the letters `N, S, E`, and `W` representing the route to transmit to the rover. The maximum length of the string is 100.

Output

Display the minimum number of characters needed to encode the route.

Example

Input	Output
WNEENWEENEENE	10
NSEW	4
EEEEEEEEE	6

Problem F. Removal Game

Source file name:	removal.c, removal.cpp, removal.java, removal.py
Input:	`Standard`
Output:	`Standard`

Bobby Roberts is totally bored in his algorithms class, so he's developed a little solitaire game. He writes down a sequence of positive integers and then begins removing them one at a time. The cost of each removal is equal to the greatest common divisor (gcd) of the two surrounding numbers (wrapping around either end if necessary). For example, if the sequence of numbers was 2, 3, 4, 5 he could remove the 3 at a cost of 2 (= gcd(2,4)) or he could remove the 4 at a cost of 1 (= gcd(3,5)). The cost of removing 2 would be 1 and the removal of 5 would cost 2. Note that if the 4 is removed first, the removal of the 3 afterwards now has a cost of only 1.

Bobby keeps a running total of each removal cost. When he ends up with just two numbers remaining he takes their gcd, adds that cost to the running total, and ends the game by removing them both. The object of the game is to remove all of the numbers at the minimum total cost. Unfortunately, he spent so much time in class on this game, he didn't pay attention to several important lectures which would lead him to an algorithm to solve this problem. Since none of you have ever wasted time in your algorithm classes, I'm sure you'll have no problem finding the minimum cost given any sequence of numbers.

Input

Input contains multiple test cases. Each test case consists of a single line starting with an integer n which indicates the number of values in the sequence ($2 \leq n \leq 100$). This is followed by n positive integers which make up the sequence of values in the game. All of these integers will be ≤ 1000. Input terminates with a line containing a single 0. There are at most 100 test cases.

Output

For each test case, display the minimum cost of removing all of the numbers.

Example

Input	Output
4 2 3 4 5 5 14 2 4 6 8 0	3 8

Problem G. That's One Hanoi-ed Teacher

Source file name:	hanoi.c, hanoi.cpp, hanoi.java, hanoi.py
Input:	`Standard`
Output:	`Standard`

Roberta Roberts (the older sister of Bobby in Problem F) teaches math at a small college, and has just introduced the Tower of Hanoi to the students in her Discrete Math class. In case you've been in a Tibetan monastery for the past several years and have never heard of the Tower of Hanoi puzzle (doubtful for several reasons), here's a brief description. The puzzle consists of three pegs, and n disks with radii of $1, 2, \cdots, n$. The initial set up has all the disks on a start peg in increasing order of their size from top to bottom, forming a pyramid. The object of the puzzle is to move all of these disks to a *destination* peg using the following rules:

1. You can move only one disk at a time.
2. At no point may a larger disk lie on top of a smaller disk.

It's well known that the optimal (i.e., shortest) solution for a Tower of Hanoi puzzle with n disks involves $2^n - 1$ moves. The optimal solution for $n = 3$ is shown below (where the start peg is the leftmost peg and the destination peg is the rightmost peg):

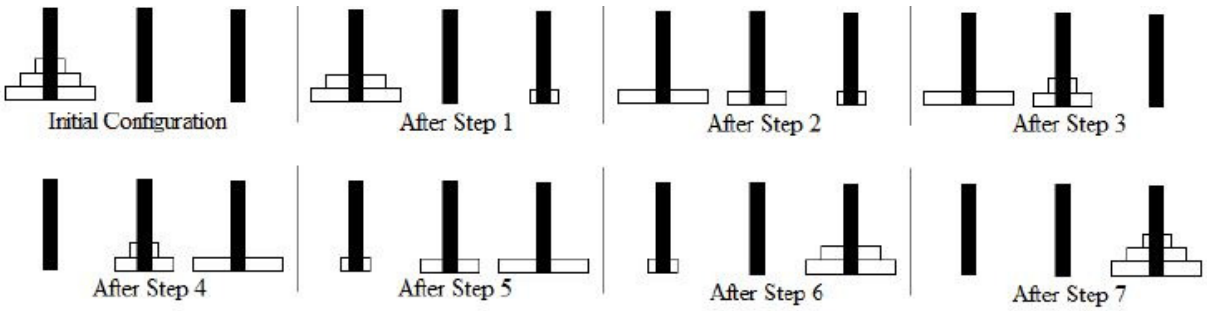

Figure G.1

As part of an in-class lab, Roberta will hand out Tower of Hanoi sets to her students and let them try to solve the problem on their own. As she goes around the room, she realizes that for the larger size sets, she has trouble looking at a current layout of the disks and determining whether the student is on the right track or not. In other words, she wishes to know whether or not a given configuration of the puzzle is one of the 2^n configurations in the optimal solution sequence. She would also like to be able to tell her students how close they are to the final configuration (i.e., all the disks in increasing sizes, top to bottom, on the destination peg). Needless to say, this has caused her a bit of consternation, so she has come to you for help.

Input

Input consists of three lines, each line representing one peg of a Tower of Hanoi configuration. Each of these lines starts with a non-negative integer m indicating the number of disks on that peg, followed by m integers indicating the disks, starting with the disk on the bottom of the peg. The first line refers to the start peg and the last line refers to the destination peg. Disks are numbered consecutively starting at 1 with each number indicating the disk's radius. All disk numbers used form a consecutive sequence. The maximum number of disks in any test case is 50.

Output

Display `No` if the given configuration is not in the optimal solution sequence; otherwise display the minimum number of remaining moves required to get to the final configuration.

Example

Input	Output
1 3 2 2 1 0	4
1 3 0 2 2 1	No

Problem H. Vin Diagrams

Source file name:	vindiagrams.c, vindiagrams.cpp, vindiagrams.java, vindiagrams.py
Input:	`Standard`
Output:	`Standard`

Venn diagrams were invented by the great logician John Venn as a way of categorizing elements belonging to different sets. Given two sets A and B, two overlapping circles are drawn – a circle representing the elements of A, and another representing the elements of B. The overlapping region of the circles represents element that belong to both A and B, i.e., the intersection of the two sets $A \cap B$. A classic Venn diagram might look like this:

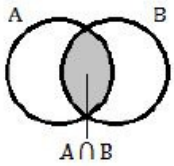

Figure H.1

One of John's biggest fans was his grandson, Vin Vaughn Venn. Vin was inspired by his grandfather's diagrams, but Vin was a very creative individual. Simple overlapping circles struck Vin as too boring of a way to visualize the sometimes messy intersections of categories, so he set out to make his grandfather's diagrams more interesting. Just like Venn diagrams, Vin diagrams are used as a way of categorizing elements belonging to different sets A and B, but the representation of each set is not required to be a circle. In fact, each set can have any shape as long as there is single overlapping section for elements in the intersection of A and B.

In this problem, Vin diagrams will be laid out on a grid. Each set representation is a loop of '`X`' characters, with one '`X`' in each loop replaced by an '`A`' or '`B`' to identify the loop. All empty positions (both inside and outside of the loops) are represented by period ('.') characters, and the set of positions inside a loop is contiguous. Each loop character touches exactly two other loop characters either vertically or horizontally. Loops do not self-intersect, and other than the allowed horizontal/vertical paths and right angle connections, different parts of the loop do not touch (see Figures H.2 and H.3 below).

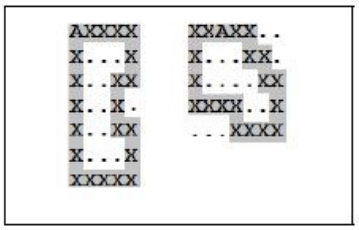

Figure H.2: Two legal loops

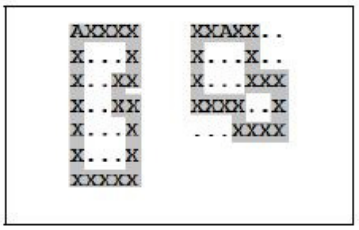

Figure H.3: Two illegal loops

Loops A and B intersect at exactly two points. Loop intersection points always follow the pattern shown in Figure H.4 (including the four '.' positions around the intersection). No loop makes a right angle turn at an intersection point but always flows straight through the intersection, either vertically or horizontally. An example of legally intersecting loops is shown in Figure H.5.

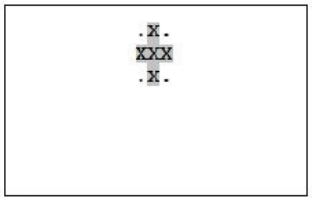

Figure H.4: Intersection point and surrounding positions

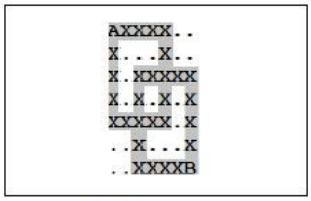

Figure H.5: Legally intersecting loops

Input

The input starts with two integers r c describing the number of rows and columns in the Vin diagram ($7 \leq r,\ c \leq 100$). The following r rows each contain a string of c characters. All positions that are not part of loop A or loop B are marked with a period ('.') character. The loop labels 'A' and 'B' are placed somewhere around the loops' perimeters at non-intersection positions and are never on the same loop.

Output

Display, in order, the area of the Vin diagram exclusive to set A, the area exclusive to set B, and the area of the intersection. Given the representation of Vin diagrams, the area of a section is defined as the number of periods ('.') it encloses.

Example

Input	Output
7 7 AXXXX.. X...X.. X.XXXXX X.X.X.X XXXXX.X ..X...X ..XXXXB	5 5 1
11 13 XXXXXXA...... X.....X...... X..XXXXXXXXX. X..X..X....X. X..X..XXX..XX X..B....X...X X..X.XXXX...X X..X.X......X XX.XXXXXX...X .X...X..X.XXX .XXXXX..XXX..	21 22 10

Problem I. Waif Until Dark

Source file name:	waifuntil.c, waifuntil.cpp, waifuntil.java, waifuntil.py
Input:	`Standard`
Output:	`Standard`

"Waif Until Dark" is a daycare center specializing in children of households where both parents work during the day. In order to keep the little monsters ... that is, darlings ... occupied, the center has a set of toys that the kiddies can play with. Some of these toys belong to one of several categories – sports toys, musical toys, dolls, etc. In order to save wear and tear on these types of toys, the teachers allow only certain numbers of toys of each category to be used during playtime. Of course, kids being kids, not all of the toys are liked by everyone, so sometimes it's difficult to make sure every kid has a toy they like. Given all of these constraints, the CEO of Waif has come to you to write a little program to determine the maximum number of these monsters (let's be honest here) who can be satisfied.

Input

Input starts with a line containing three integers n m p indicating the number of children, the number of toys and the number of toy categories ($1 \leq n, m \leq 100, \quad 0 \leq p \leq m$). Both children and toys are numbered starting at 1. After this line are n lines of the form k i_1 i_2 ... i_k ($1 \leq k, i_1, i_2, \ldots i_k \leq m$); the j^{th} of these lines indicates that child j is willing to play with toys i_1 through i_k. Next are p lines of the form l t_1 t_2 ... t_l r ($1 \leq r \leq l \leq m, \quad 1 \leq t_1, t_2, \ldots t_l \leq m$); the j^{th} of these lines indicates that toys t_1 through t_l are in category j and that at most r of these toys can be used. Toys can be in at most one category and any toy not listed in these p lines is not in any toy category and all of them can be used. No toy number appears more than once on any line.

Output

Display the maximum number of children that can be satisfied with a toy that they like.

Example

Input	Output
4 3 1 2 1 2 2 1 2 1 3 1 3 2 1 2 1	2

Problem J. Yes, Yes, It's Nonograms

Source file name:	nonograms.c, nonograms.cpp, nonograms.java, nonograms.py
Input:	`Standard`
Output:	`Standard`

Nonograms (also known as Paint by Numbers or Hanjie) is a logic puzzle which encodes a black- and-white picture using sequences of numbers. The object of the puzzle is to recreate the picture from the numbers. The puzzle initially consists of a blank $n \times m$ grid, with a sequence of numbers associated with each row and each column. These numbers indicate the lengths of runs of black squares in a row (from left to right) or column (from top to bottom). For example, if the numbers for a row are 4 5 1 it indicates that somewhere in the row there is a run of 4 consecutive black squares followed later by a run of 5 consecutive black squares which is in turn followed by a run of a single black square. There must be 1 or more white spaces between each black run, and there can be 0 or more white squares before the first or after the last black run. In our example, if the length of a row is 13, then there are four possible layouts of black and white squares:

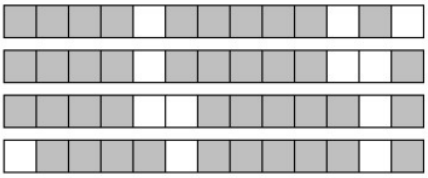

Figure J.1

Note that in all four of the possible layouts certain squares are always black, as shown in Figure J.2, while others can be either white or black (indicated by '?')

Figure J.2

In fact, this is a major technique in solving Nonograms, since they not only help in filling black squares in a particular row (as above), but the black squares then constrain the possible layouts in the intersecting columns. This helps in filling in black squares in the columns, which in turn lead to more constraints on black squares in the rows, and so on. Similarly, we may sometimes deduce that certain tiles must be white, based solely on the sequence of runs for a given row or column and the colors of tiles in that row or column that were determined previously. For many puzzle instances, applying this approach repeatedly suffices to find a solution. In more complicated Nonograms, other methods need to be used as well, but for the purposes of this problem we will not only ignore these methods but insist that you use no technique other than the one described above.

Input

Input starts with a line containing two integers n m, where n is the number of rows in the grid and m is the number of columns ($1 \leq n,\ m \leq 100$). Following this are n lines each giving the sequence of numbers for a row, starting with the uppermost row. Each of these lines has the form $p\ v_1\ v_2\ \cdots\ v_p$, where p is the number of values in the sequence, and $v_1\ \cdots\ v_p$ is the sequence. After these n lines are m lines which describe the columns in a similar way, starting with the leftmost column. The Nonogram puzzle is guaranteed to have at least one arrangement of black and white squares consistent with all the sequences, which may or may not be fully discoverable with the technique described above.

Output

Display the most complete picture that can be constructed using only the technique described above. Your output should consist of n lines containing m characters each. Use an '`X`' for a black square, a '.' for a white square, and a '?' for a square that cannot be determined.

Example

Input	Output
3 7 3 2 1 1 1 3 2 2 2 1 1 2 1 1 1 1 1 2 1 1 1 2 2 1 1	XX.X..X ...XXX. .XX..XX
5 5 1 3 1 3 1 3 1 1 1 1 1 3 1 3 1 3 1 1 1 1	??X?? ??X?? XXX.. ??.?? ??.??