Problem A. Brackets Subsequences

Input file:	`brackets.in`
Output file:	`brackets.out`
Time limit:	1 second
Memory limit:	64 megabytes

Consider bracket sequences with one type of brackets. Given a sequence of brackets, your task is to find the number of its different subsequences that are regular brackets sequences.

For example, the sequence "`(((())())(`" has 8 such subsequences: "`(((())())`", "`((())()`", "`(((()))`", "`(()())`", "`(())`", "`()()`", "`()`", and "".

Input

The input file contains a sequence of brackets. The sequence is not empty, its length does not exceed 300.

Output

Output the number of its different subsequences that are regular brackets sequences.

Examples

brackets.in	brackets.out
(((())())(	8

Problem B. Deadlock

Input file:	`deadlock.in`
Output file:	`deadlock.out`
Time limit:	1 second
Memory limit:	64 megabytes

Roman has recently developed the new deadlock detection algorithm. The algorithm uses non-deterministic computers and oracles to detect deadlocks in multithreaded environments.

The algorithm detects deadlocks in multithreaded programs without cycles that use only one synchronization primitive — *critical sections*. Critical section can be viewed as the flag that only one thread can own at a time. If some thread owns the flag, and the other thread wants to obtain it, it has to wait until the first thread releases it.

Let us consider an example. Let there be three threads, and they all need to obtain one critical section at some point. Let the first thread obtain critical section at some moment. The two other threads, when reaching the point where they need to obtain the critical section, are suspended and wait for it to be released. When the first thread releases the critical section, one of the suspended threads (not necessarily the first one that asked for the critical section) is resumed and obtains the critical section. When it releases the critical section, the third thread is resumed and can proceed.

When there is only one critical section, if no thread locks up, the process always terminates — all threads are sooner or later given the critical section. The situation becomes worse when there are several critical sections. Let us consider another example. Let there be two threads and two critical sections: A and B. Let the first thread obtain critical section A and continue execution. Meanwhile the second thread obtains critical section B. Let at some point the first thread need critical section B. It is suspended at that point waiting for B to be released. Note that it still owns A. The situation becomes tragic if the second thread now needs A before releasing B. Then it is suspended and waits for A to be released while still holding B. Now none of the threads can ever proceed. Such situtation is called a *deadlock*.

Generally a deadlock is the situation when two or more threads own the set S of critical sections, need the set R of critical sections, and $R \subset S$. In a situation of a deadlock none of the unfinished threads can advance.

We will use the following model to describe threads. Each thread is described with the sequence of its synchronization operations on critical sections. Let critical sections be numbered from 1 to m. If some thread needs critical section i we will denote it as "`+i`". If the thread releases critical section i, we will denote it as "`-i`". We will consider all threads being *regular* — that is, if some thread obtains critical sections in some order $a_1, a_2, \ldots a_k$, it releases them in the reverse order $a_k, a_{k-1}, \ldots a_1$. In other words, if we introduce one bracket type for each critical section, and denote the obtaining of the critical section as the opening bracket, and the releasing of the critical section as the closing bracket, the corresponding sequence is the regular brackets sequence. No thread attempts to obtain the critical section that it already owns.

We will denote the k-th synchronization operation of the i-th thread as $[i : k]$. If the i-th thread has completed $k - 1$ synchronization operations and is going to execute the k-th one, we say that it is in the *execution position* before $[i : k]$.

We will consider all threads running simultaneously and independently except that they use the common set of critical sections. The order of operations between different threads that is not defined by critical sections may be arbitrary.

Roman's algorithm allows to find the location at which each thread is suspended in a deadlock. Unfortunately, it does not allow to find the sequence of the operations that may cause the deadlock. Note that the deadlock may not occur in some cases. For example, let there be two threads with the following sequences of operations:

Thread	Operations
1	$+1, +2, -2, -1$
2	$+2, +1, -1, -2$

There is a possible deadlock before $[1:2]$ and $[2:2]$. However, it occurs only if the operation $[2:1]$ is executed before $[1:2]$ and $[1:1]$ is executed before $[2:2]$. If, for example, $[1:1]$ and $[1:2]$ are both executed before $[2:1]$, the deadlock does not occur.

Roman has found the deadlock in his program and now wants to find the sequence of operations that leads to it. Unfortunately, it is not always possible to find such sequence — some deadlocks are unreachable. So Roman would like to find any sequence of operations that leads to a deadlock — not necessarily the one that he has detected with his algorithm. Help him to do it.

Input

The first line of the input file contains two integer numbers n and m — the number of threads and critical sections used, respectively ($2 \le n \le 20$, $2 \le m \le 1000$). The following n lines contain sequences of synchronization operations that threads perform. The i-th of these lines starts with l_i — the number of operations the i-th thread performs ($1 \le l_i \le 1000$) followed by l_i integer numbers — the operations themselves. The number $+k$ denotes obtaining of the critical section k, the number $-k$ — releasing it.

The last line of the input file describes the deadlock. It contains n integer numbers, i-th of which ranges from 1 to $l_i - 1$ and represents the execution position that the i-th thread is before in the deadlock. It is guaranteed that the situation described is indeed a deadlock and all threads take part in it — that is, no thread can advance and perform its next synchronization operation.

Output

Output the sequence of operations that leads to some deadlock. The operation that the i-th thread has performed its k-th operation must be described as "`[i:k]`". Separate operations with spaces and/or line breaks.

The deadlock reached must not necessarily be the one described in the input file. It is not necessary that all threads take part in the deadlock, it is sufficient that at least two threads cannot advance.

Examples

deadlock.in	deadlock.out
2 2 4 +1 +2 -2 -1 4 +2 +1 -1 -2 2 2	[1:1] [2:1]
2 3 6 +1 +2 +3 -3 -2 -1 6 +1 -1 +3 +2 -2 -3 3 4	[2:1] [2:2] [2:3] [1:1] [1:2]
3 4 6 +1 +2 +3 -3 -2 -1 4 +2 +3 -3 -2 4 +3 +1 -1 -3 2 2 2	[2:1] [2:2] [2:3] [2:4] [1:1] [3:1] [1:2]

Please note that in the last example threads 1 and 3 can enter the alternative deadlock as sample output shows. In this case all threads that do not take part in a deadlock may finish their execution. After the sequence of operations in the output file no thread must be able to advance and at least two threads must not have finished execution.

Problem C. Forbidden Subwords

Input file:	`forbidden.in`
Output file:	`forbidden.out`
Time limit:	1 second
Memory limit:	64 megabytes

Consider some finite alphabet Σ. Let us call a mapping $\alpha : \mathbb{Z} \to \Sigma$ a *two-side infinite word*. We will denote the i-th character $\alpha(i)$ as α_i.

Two infinite words α and β are considered equal if there is such $k \in \mathbb{Z}$ that $\alpha_i = \beta_{i+k}$ for all $i \in \mathbb{Z}$.

We call a finite word $x = x_1 x_2 \dots x_l$ a *subword* of a two-side infinite word α if there is $k \in \mathbb{Z}$ such that $x_i = \alpha_{i+k-1}$ for all i from 1 to l. It is clear that if x is a subword of α, it is a subword of any word equal to α.

Let us consider a set $\mathcal{F}$ of finite words that we will call a *set of forbidden words*. We say that a two-side infinite word α *avoids* $\mathcal{F}$ if no word from $\mathcal{F}$ is a subword of α.

For example, there are three two-side infinite words over $\Sigma = \{$'`a`', '`b`'$\}$ that avoid the set $\mathcal{F} = \{$"`ba`"$\}$ — "...`aaaaa`...", "...`aaaabbbbb`...", and "...`bbbbb`...".

Given $\mathcal{F}$ you have to find the number of different two-side inifinite words that avoid $\mathcal{F}$.

Input

The first line of the input file contains two integer numbers n and m — the number of letters in Σ and the number of forbidden words ($1 \le n \le 6$, $0 \le m \le 1000$). Let us use first n letters of the English alphabet as Σ.

The following m lines contain forbidden words, all forbidden words are different and non-empty. Length of any forbidden word doesn't exceed 10.

Output

Output one integer number — the number of two-side infinite words that avoid $\mathcal{F}$. If there are infinitely many such words, output -1.

Examples

forbidden.in	forbidden.out
2 1 ba	3
2 2 aaa bbb	-1
2 1 a	1
2 2 a b	0
3 6 ab ac ba bc ca cb	3

Problem D. Puzzle Championship

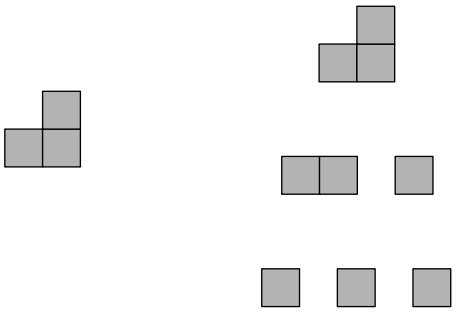
Input file:	`puzzle.in`
Output file:	`puzzle.out`
Time limit:	1 second
Memory limit:	64 megabytes

Eric has just returned from Flatland Puzzle Championship. One of the most challenging puzzles suggested was the following.

Recall that a polyomino is a 4-connected set of squares on a grid. Two polyominoes are considered the same if one can be tranformed to another by rotation and/or flipping.

You are given a polyomino and have to split it to a set of one or more polyominoes. The task is to find the number of ways to do it.

For example, there are three ways to split polyomino on the left picture to smaller polyominoes. These ways are shown on the right picture.



Input

The size of the source polyomino doesn't exceed 2×3. It is described with two lines of three characters each. Each character is either '`*`' or '`.`'.

Output

Output one integer number — the number of ways to split the source polyomino.

Examples

puzzle.in	puzzle.out
.*. **.	3

Problem E. Compact Relations Encoding

Input file:	`relations.in`
Output file:	`relations.out`
Time limit:	1 second
Memory limit:	64 megabytes

Recall that a *binary relation* R on X is a set of ordered pairs of elements from X: $R \subset X \times X$. If X is finite and contains n elements, a relation can be represented as a boolean matrix with n rows and n columns.

In some cases it is important to efficiently encode relations. One compact relation encoding is the following. Consider two functions f and g from X to the set $\mathbb{Z}$ of integer numbers. We say that these functions encode relation R if for any x and y from X the following is true: $(x, y) \in R$ if and only if $f(x) \le g(y)$.

Given the relation R, find out if this relation can be encoded in such a way, and if it can, find the required functions f and g.

Input

The first line of the input file contains n — the number of elements in $X = \{x_1, x_2, \ldots x_n\}$ ($1 \le n \le 1\,000$). The following n lines describe R. Each line contains n characters, the j-th character of the i-th of these lines is '`1`', if $(x_i, x_j) \in R$, and '`0`' in the other case.

Output

If the relation can be encoded in the described way, print "`YES`" at the first line of the output file. In this case, the second line must contain n integer numbers ranging from -10^9 to 10^9 — the values of f for x_1, x_2, …, x_n respectively, and the third line must describe g in a similar way.

If the relation cannot be encoded in such a way, print "`NO`" at the first line of the output file.

Examples

relations.in	relations.out
3 111 110 100	YES 0 1 2 2 1 0
3 110 101 011	NO

Problem F. Peaceful Rooks

Input file:	`rooks.in`
Output file:	`rooks.out`
Time limit:	1 second
Memory limit:	64 megabytes

A rook is a chess piece that moves along ranks and files. Let us consider k white rooks and l black rooks on an $m \times n$ chessboard. We call the arrangement of the rooks on the chessboard *peaceful* if no white rook attacks a black one (and vice-versa).

Given m, n, k and l, find the number of peaceful arrangements of the rooks on the chessboard.

Input

The input file contains four integer numbers: m, n, k and l ($2 \le m, n \le 10$, $1 \le k$, $1 \le l$, $k + l \le mn$).

Output

Output one integer number — the number of peaceful arrangements of k white and l black rooks on the $m \times n$ chessboard.

Examples

rooks.in	rooks.out
3 3 2 2	18

Problem G. Secret Photo

Input file:	`secret.in`
Output file:	`secret.out`
Time limit:	1 second
Memory limit:	64 megabytes

James Pond is planning to take a photo of a new secret military base in Flatland. Unfortunately, the base is surrounded by a high fence with high voltage wires running around. James does not want to risk electrical shock, so he wants to take a photo from outside the fence. He can bring a high tripod to take a photo, so we can consider that if needed he can take it right from some point of the fence.

The secret base is a convex polygon. The fence has a form of a circle. Of course James wants to make a photo with maximal possible detail level. The detail level of the photo depends on the view angle of the base from the point the photo is taken at. Therefore he wants to find the point to maximize this angle.

Help him to find such point.

Input

The first line of the input file contains two integer numbers: n and r — the number of vertices of the polygon and the radius of the fence ($3 \le n \le 200$, $1 \le r \le 10^3$).

Let us introduce the coordinates in such a way, that the center of the fence circle is in the origin. The following n lines contain two real numbers each — the coordinates of the vertices of the polygon listed in counterclockwise order. It is guaranteed that all vertices of the polygon are strictly inside the fence circle, and that the polygon is convex.

Output

Output two real numbers — the coordinates of the point from which the photo must be taken. The point must be outside the fence, although it may be exactly on it. Print at least 8 digits after the decimal point.

Examples

secret.in	secret.out
4 2 -1.0 -1.0 1.0 -1.0 1.0 1.0 -1.0 1.0	2.0000000000000 0.0000000000000

Problem H. TEX Assistant

Input file:	`tex.in`
Output file:	`tex.out`
Time limit:	1 second
Memory limit:	64 megabytes

It is known that people often make small mistakes when typesetting formulas in TEX. The most common mistake is the forgotten curly braces around integer power exponent or index when they are negative or consist of more than one digit.

Let us consider an example. To typeset "10^{-3}" you have to type "`$10^{-3}$`". If you forget the curly braces and type "`$10^-3$`" instead, you would get "10^-3" which is generally not what you want. Similarly, to typeset "k_{13}" you have to type "`$k_{13}$`". If you type "`$k_13$`" you would get "k_13".

Let us describe the part of TEX we need for this problem. Formulas are typed in so called *math mode*. There are two types of formulas: *inline* and *displayed*, inline formulas are typed between two dollar signs '`$`', displayed formulas are typed between two double dollar signs '`$$`'. Note that there is no space between the dollar signs in the double dollar delimiter.

To typeset a dollar sign itself somewhere inside the text, one can *escape* it by putting a backslash before: '`\$`'. The backslash in turn can also be escaped, two backslashes '`\\`' have a special meaning of creating a new line.

Upper index or power exponent is created by placing '`^`' character before it, lower index is created by placing '`_`' character. By default each of this modifiers is only applied to the following character. To put a longer sequence to the exponent or index, you have to place it in curly braces: '`{`' and '`}`'.

Each of '`^`', '`_`', '`{`' and '`}`' as well loses its special meaning when escaped. To escape it, you have to place the non-escaped backslash '`\`' character before it. Two or more non-escaped '`^`' or '`_`' characters in a row are forbidden in TEXand will not occur in the input to this problem.

An integer number is the arbitrary sequence of digits '`0`' to '`9`' that can be preceded by minus sign '`-`'.

You have to find all non-escaped '`^`' and '`_`' characters in the math mode and issue warning for each case when they are immediately followed by an integer number that consists of more than one character.

Input

Input file contains some valid TEX document. It's length doesn't exceed 10 kilobytes.

It is guaranteed that the source is correct in the following sense:

- blocks of text that are in the math mode are correctly delimitered with dollar signs;
- math mode blocks do not reside inside each other;
- the document doesn't end in math mode;
- there are no more that two consecutive dollar signs anywhere;
- no '`^`' or '`_`' character is followed by a space, a dollar sign, another '`^`' or '`_`' character, or the end of file.

Output

For each non-escaped '`^`' and '`_`' character in the math mode that is immediately followed by an integer number that consists of more than one character, you have to print one line containing the warning message. Adhere to the format of sample output.

The position displayed must be the position of the '`^`' or '`_`' character that is followed by a long number.

If there are no warnings, print "`No warnings`" to the output file.

Examples

tex.in

```
It is well known that the kinetic energy equals $$E=mv^2 \atop 2.$$

The tenth power may be mistakenly typeset as $10^10$ when
the proper way is $10^{10}$.

The dollar sign can be typeset as \$. In this case
it doesn't turn the math mode on and 10^10 should issue no warning.

However, backslash can also be escaped and the following
dollar is not escaped any more\\$k_-3$

What can we do with \TeX? Essentialy everything. We can display
physics formulas, such as $p^-+n^+=0$.
```

tex.out

```
Warning at line 3, position 49
Warning at line 10, position 35
```

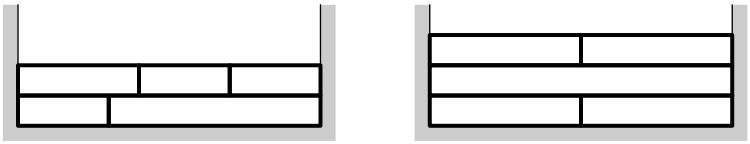
Problem I. Crazy Wall

Input file:	`wall.in`
Output file:	`wall.out`
Time limit:	1 second
Memory limit:	64 megabytes

The authors of the game "*Crazy Tetris*" decided to create the new game, they called "*Crazy Wall*". The goal of the game is to build a wall to lock up the pass. The wall must be as high as possible.

The width of the pass is w. The wall must be built of several bricks. The bricks fall from above one after another. For each brick the positions of its left and right borders are known. The player is not allowed to move the bricks, but she is allowed to choose the order in which the bricks fall. She is also allowed to stop the bricks and confirm that the wall is completed.

The wall consists of several rows, the width of each row is w. Additionally, the wall must be *solid*. The wall is called solid, if no two bricks have the same position of the left border, except at the left side of the pass, nor any two bricks have the same position of the right border, except at the right side of the pass. For example, the wall on the left on the picture below is solid, but the wall on the right is not.



Given the description of the bricks, find out what is the maximal possible height of the solid wall one can build, and what bricks must be used to build it.

Input

The first line of the input file contains n and w — the number of bricks and the width of the pass, respectively ($1 \le n \le 2000$, $1 \le w \le 2 \cdot 10^9$). The following n lines describe bricks. Each brick is described with two integer numbers l_i and r_i — the position of its left and right border ($0 \le l_i < r_i \le w$).

Output

At the first line of the output file print h — the maximal possible height of the wall. At the following h lines describe the rows. Each row description must consist of numbers of bricks used to create the row. List bricks that the row consists of from left to right.

Examples

wall.in	wall.out
6 10 0 3 3 7 7 10 3 10 0 4 4 7	2 1 4 5 6 3
5 10 0 5 0 5 0 10 5 10 5 10	2 1 4 3

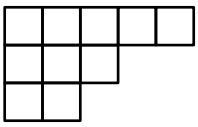
Problem J. Die Young

Input file:	young.in
Output file:	young.out
Time limit:	1 second
Memory limit:	64 megabytes

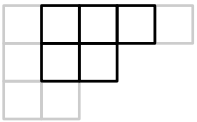
Young diagram is a well known way to describe a partition of a positive integer number. A partition of a number n is a representation as a sum of one or several integer numbers $n = m_1 + m_2 + \ldots + m_k$ where $m_1 \geq m_2 \geq \ldots \geq m_k$.

A diagram consists of n boxes arranged in k rows, where k is the number of terms in the partition. A row representing the number m_i contains m_i boxes. All rows are left-aligned, and sorted from longest to shortest.

The diagram on the picture below corresponds to the partition $10 = 5 + 3 + 2$.



Sometimes it is possible to *inscribe* one Young diagram into the other. Diagram X can be inscribed into the diagram Y if it is possible to delete some boxes from diagram Y so that it turns to diagram X. Note that it is only allowed to remove some boxes, it is not allowed to rotate or flip the diagram. For example, the picture below shows that the diagram for $5 = 3 + 2$ can be inscribed into the diagram for $10 = 5 + 3 + 2$.



On the other hand, for example, it is impossible to inscribe the diagram for $8 = 4 + 4$ into the diagram for $10 = 5 + 3 + 2$.

Given n, your task to find such partition of n that the corresponding Young diagram has the greatest possible number of diagrams that can be inscribed into it.

For example, there are 36 Young diagrams that can be inscribed into the diagram for $10 = 5 + 3 + 2$. However, it is not the maximal possible value. The diagram for $10 = 4 + 2 + 2 + 1 + 1$ has the better value, there are 41 diagrams that can be inscribed into it.

Input

Input file contains n ($1 \leq n \leq 100$).

Output

At the first line of the output file print the maximal number of Young diagrams that can be inscibed into some Young diagram for the partition of n.

At the second line print one or more integer numbers — the number of boxes in each row of the optimal diagram.

Examples

young.in	young.out
10	41 4 2 2 1 1