Problem A. Enjoy Arithmetic Progressions

Input file:	`arithmetic.in`
Output file:	`arithmetic.out`
Time limit:	5 seconds
Memory limit:	256 megabytes

Young Andrew loves arithmetic progressions. His even younger sister Anya has written a sequence of numbers on a blackboard. Now Andrew wants to split this sequence into several arithmetic progressions. For example, a sequence `1 2 5 7 9 11 3` can be split into three arithmetic progressions: `1 2`, `5 7 9 11` and `3`. There is another way to split it into three arithmetic progressions (`1 2`, `5 7 9` and `11 3`), but there is no way to split it into two or less progressions.

Obviously, Andrew would love to split the sequence into as little progressions as possible. He's even willing to change some of the numbers so that the resulting number of arithmetic progressions is smaller. But it would be boring to just change all numbers to `1 2 3 ...`.

So Andrew has decided to assign a score of c to each change operation, and a score of p to each resulting arithmetic progression. If he changes n_c numbers and splits the result into n_p progressions, his total score is $cn_c + pn_p$. He wants his total score to be as small as possible.

Input

The first line of the input file contains three integers n, $1 \leqslant n \leqslant 3\,000$ (the length of Anya's sequence), c and p, $1 \leqslant c, p \leqslant 10\,000$. The second line of the input file contains n integers between -1000 and 1000, inclusive — Anya's sequence.

Output

In the first line of the output file print minimal total score. In the second line print the number of arithmetic progressions that Andrew will form. Then output the progressions themselves, one per line. Each line should start with the number of elements in the progression, followed by the numbers themselves. Every number should be written either as integer between -10^9 and 10^9, inclusive, or as a rational number with numerator between -10^9 and 10^9, inclusive, and denominator between 2 and 10^9, inclusive, with the greatest common divisor of numerator and denominator equal to 1. Andrew never uses numbers that can't be written under above restrictions.

In case there are several possible solutions with the same total score, output any.

Examples

arithmetic.in	arithmetic.out
11 2 5 -100 -100 -100 1 1 2 2 3 100 100 100	19 3 3 -100 -100 -100 5 1 3/2 2 5/2 3 3 100 100 100

Note

Let us remind you that an arithmetic progression is a sequence of numbers such that all differences between adjacent numbers are the same.

Problem B. Cheater Detection

Input file:	`cheaters.in`
Output file:	`cheaters.out`
Time limit:	5 seconds
Memory limit:	256 megabytes

Let's consider a simplified model of programming contests. Let there be n *contestants* participating in m *contests*. The measure of success of contestant i in contest j is t_{ij} — the time it took this contestant to solve all problems in this contest.

Contest j has *difficulty* d_j, and contestant i has *strength* s_i. The times t_{ij} are then randomly and independently chosen from Poisson distribution with $\lambda = \frac{d_j}{s_i}$.

Let us remind you that Poisson distribution with a given λ is a discrete probability distribution that only allows non-negative values, and value k occurs with probability $e^{-\lambda}\frac{\lambda^k}{k!}$.

However, as if things aren't complicated enough, some of the contestants are cheaters. They don't really know how to solve anything, however, they somehow get all solutions in advance before each contest, and they can thus solve each contest in zero time. However, that would be stupid since it would be immediately noticeable. A cheater takes a more complicated approach: he/she has a *perceived strength* s_i (we denote it with the same letter as normal strength since exactly one of those two values is defined for each contestant, cheater or not). For each contest, he/she randomly and independently chooses its *expected difficulty* (they don't know the real difficulty since they can't solve anything by themselves) d'_{ij}. He/she chooses the time to take to solve contest j randomly and independently from Poisson distribution with $\lambda = \frac{d'_{ij}}{s_i}$. That way, the cheater will look like a normal contestant with strength s_i. Or at least they think they will.

Given the times it took each contestant to solve each contest, find out which contestants are cheaters.

Input

The first line of the input file contains an integer n, denoting the number of contestants, $400 \leqslant n \leqslant 500$. The second line of the input file contains an integer m, denoting the number of contests, $400 \leqslant m \leqslant 500$.

The next n lines contain m integers each. The j-th integer in the i-th line is the time t_{ij} that it took contestant i to solve contest j.

In each testcase n and m will be chosen randomly, independently and uniformly between 400 and 500, inclusive. Each contestant will be a cheater with probability 0.5, those events being also independent. The strength or perceived strength for each contestant will be a floating-point number chosen randomly, independently and uniformly between 1.0 and 10.0. The difficulty of each contest will be a floating-point number chosen randomly, independently and uniformly between 10.0 and 100.0. Each cheater will choose the perceived difficulty of each contest randomly, independently and uniformly between 10.0 and 100.0.

Output

Output n lines. The i-th line should contain "Cheater" (without quotes) if contestant i is a cheater, or "Honest" otherwise.

Examples

cheaters.in	cheaters.out

6
6
12 10 21 19 14 14
2 11 5 8 3 4
4 4 8 11 11 16
5 4 7 1 5 8
14 3 22 20 7 9
9 18 22 6 9 11
	Honest
	Honest
	Honest
	Cheater
	Cheater
	Cheater

Note

Note that the example has too small n and m — the real testcases won't have that. Apart from that, it is generated according to the above description.

Problem C. Completely Non-zero Determinant

Input file:	`determinant.in`
Output file:	`determinant.out`
Time limit:	2 seconds
Memory limit:	256 megabytes

Given an integer n, you need to construct a $n \times n$ matrix M of zeroes and ones such that for every m, $1 \leqslant m \leqslant n$, a, $1 \leqslant a \leqslant n - m + 1$ the sub-matrix formed by rows 1 through m and columns a through $a + m - 1$ of M is non-singular over $\mathbb{F}_2$.

It is guaranteed that such matrix exists for any n.

Input

The first and only line of the input file contains an integer n, $1 \leqslant n \leqslant 100$.

Output

Output the required matrix in n lines of n integers (zeroes or ones) each, separated with single spaces inside a line. You can output any $n \times n$ matrix that satisfies the above condition.

Examples

determinant.in	determinant.out

3	1 1 1
	1 0 1
	1 0 0

Note

Let us remind you that a $m \times m$ matrix P over $\mathbb{F}_2$ is non-singular when there's an odd number of permutations p of $1, 2, \ldots, m$ such that elements P_{1,p_1}, P_{2,p_2}, $\ldots$, P_{m,p_m} are all equal to one.

Problem D. Disconnected Graph

Input file:	`disconnected.in`
Output file:	`disconnected.out`
Time limit:	3 seconds
Memory limit:	256 megabytes

You are given a connected undirected graph and several *small* sets of its edges. For each set, you need to determine whether the graph stays connected with edges from the set removed.

Remember that a graph is *connected* when for every two distinct vertices there's a path connecting them.

Input

The first line of the input file contains two integers n and m ($1 \leqslant n \leqslant 10\,000$, $1 \leqslant m \leqslant 100\,000$), denoting the number of vertices and edges in the graph, respectively. Vertices are numbered from 1 to n.

The next m lines contain the description of the edges. Each line contains two integers a and b — the numbers of the vertices connected by this edge. Each pair of vertices is connected by at most one edge. No edge connects a vertex to itself. Edges are numbered from 1 to m in the order they are given in the input.

The next line contains an integer k ($1 \leqslant k \leqslant 100\,000$), denoting the number of small sets to test. The next k lines contain the descriptions of the small sets. Each line starts with an integer c ($1 \leqslant c \leqslant 4$), denoting the number of edges in the set, followed by c numbers of the edges from the set. The numbers of the edges inside one small set will be distinct.

Output

Output k lines, one per each given small set. The i-th line should contain "Connected" (without quotes), if removal of the corresponding small set leaves the graph connected, or "Disconnected" otherwise.

Examples

disconnected.in	disconnected.out

4 5
1 2
2 3
3 4
4 1
2 4
3
1 5
2 2 3
2 1 2
	Connected
	Disconnected
	Connected

Problem E. Expanding Lake

Input file:	`lake.in`
Output file:	`lake.out`
Time limit:	2 seconds
Memory limit:	256 megabytes

You've been walking around your garden when rain started. At first, you were feeling relieved because it was the first rain in two months. But suddenly you've realized that this is not just a normal rain, but the all-drowning rain the news were talking about. So now you need to get home *fast*.

Imagine your garden as an infinite plane. You are at point (x_0, y_0) on this plane, while your home is at point $(0, 0)$. You can move with speed v_0. Easy: $\frac{\sqrt{x_0^2+y_0^2}}{v_0}$, you might think.

But there's also a dreadful lake in your garden. It is a circle centered at (x_1, y_1) and radius r_1. Moveover, because of the rain, it is expanding with speed v_1, so its radius after time t is equal to $r_1 + v_1 t$ (its center doesn't move).

You can't go into the lake as you're suddenly very afraid of water. It's OK for you to stand on the border of the lake.

Is it possible for you to get home? If yes, what is the minimum time required to do that?

Input

The first line of the input file contains three integers x_0, y_0 and v_0, $x_0^2 + y_0^2 > 0$, $-1000 \leqslant x_0, y_0 \leqslant 1000$, $1 \leqslant v_0 \leqslant 1000$.

The second line of the input file contains four integers x_1, y_1, r_1 and v_1, $x_1^2 + y_1^2 > r_1^2$, $-1000 \leqslant x_1, y_1 \leqslant 1000$, $1 \leqslant r_1, v_1 \leqslant 1000$, $(x_0 - x_1)^2 + (y_0 - y_1)^2 > r_1^2$.

It is guaranteed that in case it's possible to get home, it would still be possible when the initial radius of the lake is $r_1 + 10^{-3}$; in case it's impossible to get home, it would still be impossible when the initial radius of the lake is $r_1 - 10^{-3}$.

Output

Output one floating-point number denoting the minimum time required to get home. Your answer will be considered correct if it's within 10^{-9} relative or absolute error of the correct one.

In case you can't get home, output -1.

Examples

lake.in	lake.out
10 0 3 7 0 1 1	3.753280475952816
10 0 2 7 0 1 1	-1

Problem F. Discrete Logarithm Graph

Input file:	`modulo.in`
Output file:	`modulo.out`
Time limit:	3 seconds
Memory limit:	256 megabytes

The *discrete logarithm graph* for a given prime p is a directed graph with $p-1$ vertices, numbered from 1 to $p-1$. There is an arc from vertex a to vertex b in this graph if and only if a is a discrete logarithm of b, meaning that there exists such x between 1 and $p-1$ that $x^a = b$ modulo p.

You need to find a hamiltonian cycle in the discrete logarithm graph for the given p or report that no such cycle exists. Let us remind you that a hamiltonian cycle in a graph is a cycle that passes through each vertex exactly once.

Input

The first and only line of the input file contains a prime number p, $3 \leqslant p \leqslant 100\,000$.

Output

When there is no hamiltonian cycle, output one integer -1. When the hamiltonian cycle exists, output $p-1$ pairs of numbers denoting the arcs that form the cycle, in the order they go along the cycle. Each arc should be described by the vertex a where this arc starts and the number x such that this arc ends at x^a modulo p.

Examples

modulo.in	modulo.out

`5`	`1 3`
	`3 3`
	`2 2`
	`4 2`

Problem G. Questionable Genetic Detection

Input file:	`questionable.in`
Output file:	`questionable.out`
Time limit:	2 seconds
Memory limit:	256 megabytes

Detecting possible future illnesses via DNA analysis is the new hot trend, and you certainly want to follow it. You've obtained a very fuzzy description of a DNA signature that signals a very dangerous illness, and you want to create a quick test that will tell people that they don't have it — since people agree to pay much more money when you tell them good news than when the news are not so good.

The *DNA signature* is a string of letters `A`, `C`, `G`, `T`. However, you don't know the signature itself — you just know the set of possible letters for each position in the string. For example, your knowledge can be: the first letter is `A`, the second letter is either `A` or `C`, the third and fourth letters are arbitrary, the fifth letter is `C`, `G` or `T`.

Your future test will be just a substring test. So your goal now is to find the shortest string of letters `A`, `C`, `G`, `T` that is for sure NOT a substring of the DNA signature, no matter which possibility we choose for each letter.

Input

The first line of the input file contains an integer n, $1 \leqslant n \leqslant 25$ — the length of the DNA signature. The next n lines contain at least 1 and at most 4 characters each, with i-th line denoting the possible letters for i-th position of the signature. All characters in one line are different.

Output

Output the shortest string that is for sure not a substring of the given signature. In case there are several possible answers, output any.

Examples

questionable.in	questionable.out
5 A AC GACT ATCG GCT	TAA

Problem H. Strictly Off Permutations

Input file:	`strictly-off.in`
Output file:	`strictly-off.out`
Time limit:	2 seconds
Memory limit:	256 megabytes

A group of n friends meet each Saturday. They sit along one side of a long straight (*not* round) table. Moreover, they have always sat in exactly the same sequence along this table.

We will number all n available seats from 1 to n as they go along the table. We will also number all n friends from 1 to n as they usually sit along the table, so that friend i has always sat in seat i.

Now they have decided to change their usual order significantly. They have decided that each friend should choose a seat at least a and at most b seats away from his/her usual seat. Formally, we need to find a permutation p_i of numbers between 1 and n such that $a \leqslant |p_i - i| \leqslant b$ for all i.

You need to find a new seating plan for the friends.

Input

The first and only line of the input file contains three integers n, a and b, $2 \leqslant n \leqslant 100\ 000$, $1 \leqslant a \leqslant b \leqslant n-1$.

Output

When the required plan exists, output n integer numbers, the i-th number should denote the friend that will sit in seat i.

In case there are several possible answers, output any. In case there's no such seating plan, output single number -1.

Examples

`strictly-off.in`	`strictly-off.out`
`9 3 8`	`4 5 6 7 8 9 3 2 1`

Problem I. 2D to 3D

Input file:	`threed.in`
Output file:	`threed.out`
Time limit:	10 seconds
Memory limit:	256 megabytes

Given three non-zero area convex polygons, one on each coordinate plane in space (Oxy, Oyz, Ozx), you need to check if there exists a set of points in space which has these three polygons (including their interiors) as its orthogonal projections to the coordinate planes.

Input

The first line of the input file contains an integer n_{xy}, the number of points of the Oxy polygon. The next n_{xy} lines contain two integers each, denoting x and y coordinates of the vertices of the polygon, in either clockwise or counter-clockwise order.

The next block of lines describes the Oyz polygon in the same format (for coordinates, y goes before z). The next block of lines describes the Ozx polygon in the same format (for coordinates, z goes before x). All coordinates don't exceed 10^6 by absolute value, $3 \leqslant n_{xy}, n_{yz}, n_{zx} \leqslant 6000$.

Output

Output "Yes" (without quotes) if such set of points in space exists, and "No" (without quotes) otherwise.

Examples

threed.in	threed.out

4
0 0
0 1
1 1
1 0
4
0 0
1 0
1 2
0 2
4
0 0
2 0
2 1
0 1
	Yes

4
0 0
0 1
1 1
1 0
4
0 0
0 1
1 1
1 0
4
0 0
1 0
1 1
0 2
	No

Problem J. Automatic Input Verifier

Input file:	`verifier.in`
Output file:	`verifier.out`
Time limit:	3 seconds
Memory limit:	256 megabytes

Tons and tons of programming contest problems contain integers separated with spaces and newlines in the input. In many cases, the amount of numbers found inside the file and in each particular line is determined by the numbers themselves.

More formally, we define an *input description* as a sequence of statements of the following types:

1. "`The first/next line contains` *number* `non-negative integer(s),` *variable*$_1$`,` *variable*$_2$`, ...,` *variable*$_{number}$`.`". Here, *number* is a positive integer between 1 and 3, and *variable*s are arbitrary distinct variable names that have never appeared in the input description before this statement. When *number* is equal to 1, "`integer(s)`" should be replaced by "`integer`", otherwise it should be replaced by "`integers`".

2. "`The first/next line contains` *variable* `integers.`". Here, *variable* is a variable name that has appeared in this description before this statement.

3. "`The first/next line contains a non-negative integer` *variable*`, followed by` *variable* `integers.`". Here, *variable* is a variable name that has never appeared in the input description before this statement, and will never appear in the input description after this statement. Please note that the same variable name is repeated twice in this statement.

4. "`The first/next line contains` *number* `integer(s).`". Here, *number* is a positive integer. When *number* is equal to 1, "`integer(s)`" should be replaced by "`integer`", otherwise it should be replaced by "`integers`".

5. "`The first/next` *variable* `lines contain` *number* `integer(s) each.`". Here, *variable* is a variable name that has appeared in this description before this statement, and *number* is a positive integer. When *number* is equal to 1, "`integer(s)`" should be replaced by "`integer`", otherwise it should be replaced by "`integers`".

6. "`The first/next` *variable*$_1$ `lines contain` *variable*$_2$ `integers each.`". Here, *variable*$_1$ and *variable*$_2$ are (possibly the same) variable names that have appeared in this description before this statement.

7. "`The first/next` *variable*$_1$ `lines contain a non-negative integer` *variable*$_2$`, followed by` *variable*$_2$ `integers each.`". Here, *variable*$_1$ is a variable name that has appeared in this description before this statement. *variable*$_2$ is a variable name that has never appeared in this description before this statement, and will never appear in this description after this statement, and is different from *variable*$_1$. Please note that, unlike all other variables, *variable*$_2$ may have different values for each line described by this statement. This is useful, for example, for listing several heaps with varied amount of items in each heap.

Each occurrence of "`first/next`" in the above statements should be subsituted by "`first`" in the first statement of the input description, and by "`next`" in all other statements.

Given several input files, you need to verify whether there exists an input description containing not more than 4 statements that fits most of the given input files, and highlight the input files that don't fit this description.

Input

The first line of the input file contains an integer n, $1 \leqslant n \leqslant 10$, the number of input files given. Please note that the meaning of words "input file" below corresponds to the second meaning of that term in this paragraph, not the first one.

The description of each given input file starts with a line containing a dash ("`-`"). The actual given input file follows, with each line appended an exclamation mark ("`!`") to avoid difficulties in processing empty lines.

After excluding the exclamation mark at the end, each line of each given input file will contain between 0 and 1000 integers, each between -1000 and 1000, separated with single spaces. Each given input file will contain at most 10000 lines and at most 100000 integers.

Output

In case there exists an input description with not more than 4 statements describing all given input files, output "`All good!`" (without quotes) to the first line of the output file, followed by the input description itself, one statement per line. Use non-empty strings of at most 10 lowercase English letters to denote variables. In case there are several possible descriptions, output any.

In case there exists an input description with not more than 4 statements describing at least one given input file, find any such description that describes the most given input files. Output "`Bad format:` *numbers*`.`" (without quotes) to the first line of the output file, where *numbers* is a single-space separated list of input file numbers that don't fit the given input description. The input files are numbered from 1 to n in the order they are given. Then output the input description itself.

In case no input description with not more than 4 statements fits any given input file, output "`FAIL.`" to the only line of the output file.

Examples

verifier.in
3 - 2 3! 1 2! 3 5! -100 24! - 5 0! - 2 2! 2 2! 2 2!
verifier.out
All good! The first line contains 2 non-negative integers, v, e. The next e lines contain 2 integers each.

verifier.in

```
4
-
0!
1 0!
!
-
2 1 2!
2 0!
!
!
-
4 -100 -100 0 100!
3 0!
!
!
!
!
-
3 3 3 3!
3 1!
!
!
!
```

verifier.out

```
Bad format: 4 3.
The first line contains a non-negative integer c, followed by c integers.
The next line contains 2 non-negative integers, number, line.
The next number lines contain line integers each.
```

verifier.in

```
1
-
-1000!
-1000 -1000!
-1000 -1000 -1000!
-1000 -1000 -1000 -1000!
-1000 -1000 -1000 -1000 -1000!
```

verifier.out

```
FAIL.
```