Problem A. Chip Installation

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

The new chip is going to be installed to the new aircraft produced by the Airtram company. The chip has the form of a disc. There are n wires that must be plugged to the chip.

For easiness of installation each wire can be plugged to one of the two sockets available for this wire. Sockets are located around the disc at its edge. Each wire has its color, for safety reasons two wires of the same color are not allowed to be plugged into adjacent sockets.

Given the configuration of the sockets on the chip, find the way to connect all wires to it, so that the restrictions above were fulfilled.

Input

The first line of the input file contains n — the number of wires ($1 \le n \le 50\,000$). The second line contains n integer numbers ranging from 1 to 10^9 — colors of the wires. The third line contains $2n$ integer numbers and describes sockets. Each socket is described by one integer number — the number of the wire that can be plugged into it. Each number from 1 to n appears exactly twice in this line. Sockets are described in order they appear along the chip edge. Note that the last socket is also adjacent to the first socket.

Output

If there is no way to plug wires into their sockets so that no two wires of the same color were plugged into adjacent sockets, print "`NO`" at the first line of the output file.

In the other case print "`YES`". The second line must contain n integer numbers. For each wire output the number of socket it must be plugged into. Sockets are numbered from 1 to $2n$ in order they are described in the input file.

Examples

chip.in	chip.out
2 1 1 1 1 2 2	YES 1 3
2 1 1 1 2 1 2	NO
2 1 2 1 2 1 2	YES 1 2

Problem B. Divisible Substrings

Input file:	`divisible.in`
Output file:	`divisible.out`
Time limit:	4 seconds
Memory limit:	256 megabytes

Laboratory of Strings and Divisibility is investigating divisibility of extremely long strings. Now they are interested in the following question. Given a string consisting of decimal digits, find out how many of its different substrings are decimal representations of a number divisible by n. Unnecessary leading zeroes are not allowed. This number can be quite large, so it must be found modulo r.

The problem is that the string given is very long. So it is specified using so called *linear grammar*. Linear grammar is a set of rules of a form $A \to \alpha$ where A is an uppercase character of English alphabet (called nonterminal) and α is a string that consists of digits and uppercase characters of English alphabet, greater than A. There is at most one rule for each character, and exactly one rule for '`A`' and each character that occurs in α for some rule.

The string specified by the grammar is constructed as follows. Initially the string consists of one character '`A`'. After that while there is a nonterminal character in the string, it is replaced with the right side of its rule. For example, let the grammar be $A \to BB$, $B \to CC0$, $C \to 123$. Then the string represented by this grammar is 12312301231230.

Input

The first line of the input file contains n and r ($1 \le n \le 30$, $1 \le r \le 10^9$). The second line contains k — the number of rules. The following k lines contain linear grammar for some string. Right side of each rule has length of at most 100. Rules are listed in order of increasing their left side nonterminal.

Output

Output one number — the number of substrings of the given string divisible by n modulo r.

Examples

divisible.in	divisible.out
2 1000000000 3 A->BB B->CC0 C->123	46

Problem C. Preparing Documents

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

Macrohard company is preparing a set of documents to apply for a new grant provided by Flatland government. The application for the grant must contain n documents. The head of the company recruited two secretaries to prepare the documents. Each document requires 1 hour to be prepared by one secretary.

Preparation of the document can be interrupted at any moment and then continued by the same or by another secretary. But the same document cannot be simultaneously prepared by both secretaries.

The documents depend on each other, so they cannot be prepared in any order. For each document there is a set of requirements for the documents that must be completed before starting to prepare it.

Find out how fast the documents for the grant can be prepared.

Input

The first line of the input file contains n — the number of documents and m — the number of requirements ($1 \le n \le 100\,000$, $0 \le m \le 200\,000$).

The following m lines describe requirements, each requirement is described by two integer numbers: a_i and b_i which mean that document a_i must be completed before starting to prepare document b_i. It is guaranteed that the requirements are acyclic, so it is possible to prepare all documents.

Output

Output one real number — the number of hours required to prepare all documents.

Examples

documents.in	documents.out
6 5 1 4 2 4 3 4 4 5 4 6	3.5

The following process allows to prepare all documents in 3.5 hours.

Time	Secretary 1	Secretary 2
0h00m	Start preparing document 1	Start preparing document 2
0h30m	Stop preparing document 1, switch to document 3	
1h00m		Finish document 2, start to continue preparing document 1
1h30m	Finish document 3, start preparing document 4	Finish document 1
2h30m	Finish document 4, start preparing document 5	Start preparing document 6
3h30m	Finish document 5	Finish document 6

Problem D. GridBagLayout

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

Java user interface libraries AWT and SWING are widely used in various graphical applications. To represent the way various visual components are located in the container they use the concept of *layout*. One of the most powerful layouts is `GridBagLayout`. In this problem you will have to emulate the behavior of simplified `GridBagLayout`.

Consider a container that you consequently position components at. The container has a grid that components are arranged to row after row. Rows and columns of the grid are numbered from 0. The way the new component is positioned on the grid is defined by the contents of special `GridBagConstraints` structure. We will consider the following fields of `GridBagConstraints`:

Field	**Definition**
`gridwidth`	Number of columns that the component occupies
`gridheight`	Number of rows that the component occupies
`gridx`	Number of leftmost column that the component occupies
`gridy`	Number of topmost row that the component occupies

The component occupies grid cells in a rectangle with leftmost column `gridx`, rightmost column `gridx + gridwidth - 1`, topmost row `gridy` and bottommost row `gridy + gridheight - 1`, with the following exceptions: `gridx` and `gridy` can be equal to a special value `GridBagConstraints.RELATIVE` and `gridwidth` and `gridheight` can be equal to a special value `GridBagConstraints.REMAINDER`.

The ways the components are placed if `gridx` and/or `gridy` are equal to `GridBagConstraints.RELATIVE` are summarized in the following table. `REL` is used to represent value `GridBagConstraints.RELATIVE`.

`gridx`	`gridy`	**first in its row**	**Algorithm**
`REL`	`REL`	No	Find the first free cell in the topmost row of the previous component after the rightmost column of the previous component. Use this cell as the top-left cell of the component being added.
`REL`	`REL`	Yes	Find the first row after the topmost row of the previous component that has at least one free cell (if the component is the first component being added, use row 0). Find the first free cell in this row. Use this cell as the top-left cell of the component being added.
`REL`	numeric	Always	Find the first free cell in `gridy` row. Use this cell as the top-left cell of the component being added.
numeric	`REL`	No	Use the topmost row of the previous component as the topmost row of the component being added.
numeric	`REL`	Yes	Use the first row after that topmost row of the previous component that has free cell in `gridx` column as the topmost row of the component being added.

The value `gridwidth` is used to determine the number of columns, occupied by the component. If the value is equal to a special value `GridBagConstraints.REMAINDER`, the component occupies all columns to the end of each row it occupies, and the next component becomes the first component in its row.

The value `gridheight` is used to determine the number of rows, occupied by the component. If the value is equal to a special value `GridBagConstraints.REMAINDER`, the component occupies all rows to the bottom of the container in each column it occupies.

If two components intersect, or there is no required free cell in the algorithms described above, the layout is said to be invalid, and the behavior of the container is undefined.

The number of rows and columns in the grid is calculated after all components are placed, the minimal number of rows and columns required to accommodate all components is used.

You are given a sequence of components being placed to `GridBagLayout` container. For each component find where it will be placed.

Input

The first two lines of the input file contain initialization, they are always the same. The rest of the input file contains blocks of adding components. Each block consists of several lines of initialization of `gbc` object followed by adding a component to the container.

Initial values of `gbc` fields are: `gridwidth`= 1, `gridheight`= 1, `gridx`=`GridBagConstraints.RELATIVE`, `gridy`=`GridBagConstraints.RELATIVE`.

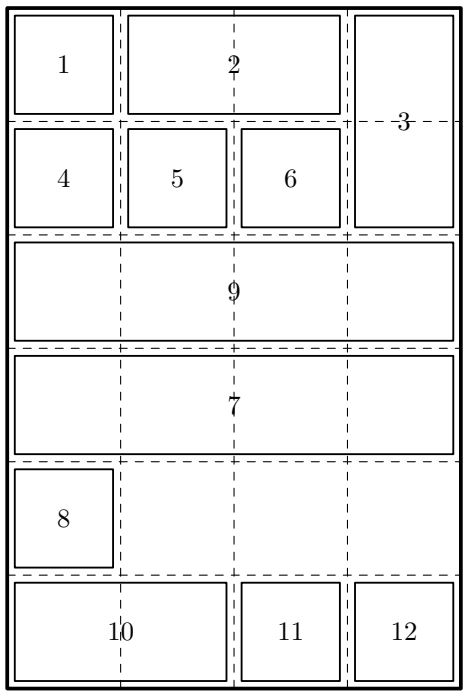
It is guaranteed that the layout is not invalid. All values assigned to `gbc` fields are non-negative and do not exceed 50 (`gridwidth` and `gridheight` are positive). The total number of added components doesn't exceed 50. At least one component is added.

Output

The first line of the output file must contain two integer numbers: the number of rows and the number of columns. Let k components be added in the input file. The following k lines must contain four integer numbers each — the top row, the left column, the bottom row and the right column of each component. Components must be described in order they are added.

Example

The picture below shows the layout produced by `GridBagLayout` for sample input.



gridbaglayout.in

```
Panel container = new Panel(new GridBagLayout());
GridBagConstraints gbc = new GridBagConstraints();
container.add(new Component(), gbc);
gbc.gridwidth = 2;
container.add(new Component(), gbc);
gbc.gridheight = 2;
gbc.gridwidth = GridBagConstraints.REMAINDER;
container.add(new Component(), gbc);
gbc.gridwidth = 1;
gbc.gridheight = 1;
container.add(new Component(), gbc);
container.add(new Component(), gbc);
container.add(new Component(), gbc);
gbc.gridy = 3;
gbc.gridwidth = GridBagConstraints.REMAINDER;
container.add(new Component(), gbc);
gbc.gridy = 4;
gbc.gridwidth = 1;
container.add(new Component(), gbc);
gbc.gridy = 2;
gbc.gridwidth = GridBagConstraints.REMAINDER;
container.add(new Component(), gbc);
gbc.gridx = 0;
gbc.gridy = GridBagConstraints.RELATIVE;
gbc.gridwidth = 2;
container.add(new Component(), gbc);
gbc.gridx = GridBagConstraints.RELATIVE;
gbc.gridwidth = 1;
container.add(new Component(), gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
container.add(new Component(), gbc);
```

gridbaglayout.out

```
6 4
0 0 0 0
0 1 0 2
0 3 1 3
1 0 1 0
1 1 1 1
1 2 1 2
3 0 3 3
4 0 4 0
2 0 2 3
5 0 5 1
5 2 5 2
5 3 5 3
```

Problem E. Hot Potato Routing

Input file:	hot.in
Output file:	hot.out
Time limit:	8 seconds
Memory limit:	256 megabytes

Hot potato routing is a method of routing packets to its destinations in a network that doesn't involve routing tables nor queuing or analyzing packets in the internal nodes.

Network consists of n nodes connected by links. Each node has its *schedule*. Schedule of a node i is a sequence $a_{i,0}, a_{i,1}, \ldots, a_{i,l_i-1}$ of node numbers. The network operates in discrete time units as follows.

Each time unit t there can be several packets in the network. Each packet has its source and destination nodes, and appearance time. At its appearance time unit the packet appears at its source node. If the packet at node i has this node as its destination, it is accepted and considered to be delivered at time t, it disappears from the network. If the packet at node i has some other node as its destination, it is forwarded to node $a_{i,t \bmod l_i}$ where it arrives at the next time unit. If at some time unit two or more packets are at the same node, they are all discarded.

Main disadvantage of hot potato routing comes from its main features — lack of routing tables and packet analyzing. Schedules of nodes and packet appearance times must be chosen very carefully to ensure delivery. Even so it is often impossible to deliver all packets.

You are given a network description and a set of packets. You can choose to deliver some of them. Find the maximum subset of packets that can be delivered to their respective destinations.

Input

The first line of the input file contains n — the number of nodes in the network ($2 \le n \le 100$). The following n lines describe nodes, the i-th node is described by the length of its schedule l_i followed by l_i numbers: $a_{i,0}, a_{i,1}, \ldots, a_{i,l_i-1}$ ($1 \le l_i \le 8$, $a_{i,j} \ne i$).

The next line of the input file contains p — the number of packets ($1 \le p \le 100$) followed by p lines that describe packets. Each packet is described by its source node s_i, its destination node d_i and its appearance time t_i ($s_i \ne d_i$, $0 \le t_i \le 1000$).

Output

The first line of the output file must contain one integer number k — the maximal number of packets that can be delivered simultaneously. The second line must contain k integer numbers — the numbers of these packets. Packets are numbered from 1 to p in order they are given in the input file.

Examples

hot.in	hot.out
4 2 2 3 3 1 1 3 3 4 4 2 1 3 3 1 3 0 3 1 2 4 2 3	2 2 3

Problem F. Knapsack Problem

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

Knapsack problem is stated as follows: given n items, the i-th one with weight w_i and value v_i, find the set of items with total weight not exceeding c (knapsack capacity) and maximal possible total value. It is well known that knapsack problem is NP-hard, dynamic programming solutions are available, but they are linear in sum of all weights (which is not polynomial in problem instance size)

However, in some cases it is possible to solve the problem in greedy manner. One notable case is when the set of all possible knapsack solutions forms a matroid.

Matroid is a pair $\langle X, \mathcal{I} \rangle$ where X is a finite set and $\mathcal{I}$ is a set of subsets of X called independent subsets. $\mathcal{I}$ must satisfy the following three axioms:

1. $\mathcal{I} \neq \varnothing$;
2. If $A \in \mathcal{I}$ and $B \subset A$ then $B \in \mathcal{I}$;
3. If $A, B \in \mathcal{I}$ and $|A| > |B|$ then there exists $x \in A \setminus B$ such that $B \cup \{x\} \in \mathcal{I}$.

For example, edges of an undirected graph where subset is called independent if it is acyclic form a matroid.

Consider a set of items with weights $w_1, w_2, \ldots, w_n$. Let X be integer numbers from 1 to n. Call a subset $\{i_1, i_2, \ldots, i_k\}$ independent if $w_{i_1} + w_{i_2} + \ldots + w_{i_k} \leq c$. Find out whether the described object is a matroid.

Input

The first line of the input file contains n ($1 \leq n \leq 50$). The second line of the input file contains n integer numbers $w_1, w_2, \ldots, w_n$ ($1 \leq w_i \leq 100$). The third line of the input file contains c ($0 \leq c \leq \sum w_i$).

Output

Output "YES" if the set of solutions to knapsack problem is a matroid. In the other case output "NO". In this case the second line must contain one integer number — the number of the axiom that is violated. The following two lines must contain counterexample to the axiom. The counterexample must describe two sets: A and B (clearly, axiom 1 cannot be violated). Each set is described by the number of items in it followed by the numbers of these items.

Examples

knapsack.in	knapsack.out
3 1 2 3 4	YES
3 3 4 5 7	NO 3 2 1 2 1 3

Problem G. Magic Potions

Input file:	magic.in
Output file:	magic.out
Time limit:	4 seconds
Memory limit:	256 megabytes

L'Ashyto is a famous magician. He is working in his laboratory on a new set of magic potions to sell on the upcoming magic fair. He has bought several bottles of each of n different magic substances that can be used to create magic potions. Each magic potion must be composed of two different magic substances. To create a potion the magician mixes the contains of two bottles and pronounces the magic spell.

Of course, L'Ashyto would like to create as many potions as possible. But he would like the quality of their ingredients be as good as possible. L'Ashyto has numbered all substances from 1 to n, substance number 1 being the best, and substance number n being the worst. If there are several ways to create the maximal possible number of potions, he would like to create as many potions from substances 1 and 2, as possible. If there are still several variants, he would like to maximize the number of $1+3$ potions, then $1+4$ potions, etc, $1+n$ potions, $2+3$ potions, etc. Help him to find out how many potions of which ingredients to create.

Input

The first line of the input file contains n — the number of substances L'Ashyto has ($2 \le n \le 100\,000$). The second line contains n integer numbers: $a_1, a_2, \ldots, a_n$ — the number of bottles of the first substance, the number of bottles of the second substance, etc ($1 \le a_i \le 10^9$).

Output

The first line of the output file must contain m — the number of different types of potions L'Ashyto must create. The following m lines must contain three integer numbers each: i, j, c_{ij} — the numbers of substances to create a potion of, and how many such potions to create. For each description there must be $i < j$. Output potion description in lexicographical order.

Examples

magic.in	magic.out
4 1 1 1 1	2 1 2 1 3 4 1

Problem H. Restoring Permutation

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

A permutation $\langle a_1, a_2, \ldots, a_n \rangle$ is said to have a *descent* at position i if $a_i > a_{i+1}$. A permutation $\langle a_1, a_2, \ldots, a_n \rangle$ is said to have a *fixed point* at position i if $a_i = i$.

Let us call a permutation $A = \langle a_1, a_2, \ldots, a_{2n} \rangle$ of $2n$ integer numbers from 1 to $2n$ *alternatively fixed* if the following conditions are satisfied:

- A has exactly n descents, all of its descents are at odd positions (i.e. $a_{2i-1} > a_{2i} < a_{2i+1}$ for all i from 1 to $n-1$ and $a_{2n-1} > a_{2n}$);
- A has exactly n fixed points.

For example, the permutation $\langle 3, 2, 6, 4, 5, 1 \rangle$ is alternatively fixed.

Consider the following transformation of alternatively fixed permutation: remove all fixed points and turn the remaining vector to permutation by replacing each remaining number with the count of remaining numbers not exceeding it. For example, transforming the above permutation yields $\langle 3, 2, 6, 4, 5, 1 \rangle \to \langle 3, 6, 1 \rangle \to \langle 2, 3, 1 \rangle$.

You are given the result of the transformation. Restore the original alternatively fixed permutation.

Input

The first line of the input file contains n — the number of elements in the transformed permutation ($1 \le n \le 100\,000$). The second line contains n integer numbers — the permutation itself.

Output

If there is no solution, output -1 at the first line of the output file. In the other case output the original alternatively fixed permutation of $2n$ numbers. If there are several solutions, output any one.

Examples

restore.in	restore.out
3 2 3 1	3 2 6 4 5 1
1 1	-1

Problem I. Roads

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

After inventing the wheel, the citizens of Flatland decided to connect their n cities by roads. Since building roads turned out to be quite expensive, they decided to build as few roads as possible so that one could get from any town to any other town by roads. Since traffic rules were not invented then yet, it was allowed to travel along each road in any direction. Flatland is flat, so there is no need to go around mountains or valleys, therefore each road is a straight line segment connecting two cities.

Of course, citizens immediately started to argue which cities would be connected by the roads. Citizens of each city wanted to have more roads going from their city. To stop negotiations the king of Flatland issued an order: for each city he claimed the number of roads that would go from it. Since the king was very wise, all these numbers were from 1 to $n-1$ and their sum was $2n-2$.

But the minister of road building was not as wise as king was. Even given these numbers, he couldn't find the way to build roads. The problem was being complicated by the additional fact that bridges, tunnels or traffic lights were not known that time, so roads weren't allowed to intersect.

You are his last hope to be saved from terrible execution.

Input

The first line of the input file contains n — the number of cities in Flatland ($2 \le n \le 1000$). The following n lines contain three integer numbers each: x_i, y_i and d_i. Here x_i and y_i are the coordinates of the corresponding city, and d_i is the number of roads that must go from it. The coordinates do not exceed 10^4 by their absolute values. No three cities are on the same straight line.

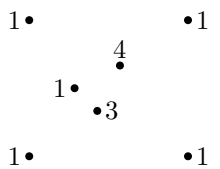
Output

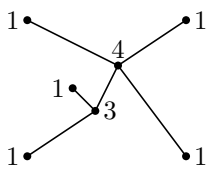
If it is impossible to build the roads, output -1 at the first line of the output file.

In the other case, output $n-1$ lines. Each line must contain two integer numbers — the numbers of the cities to be connected by the corresponding road. The cities are numbered from 1 to n in order they are given in the input file.

Examples

roads.in	roads.out
7 0 6 1 0 0 1 2 3 1 3 2 3 4 4 4 7 0 1 7 6 1	1 5 2 4 3 4 4 5 5 6 5 7





Problem J. Squary Set

Input file:	`squary.in`
Output file:	`squary.out`
Time limit:	4 seconds
Memory limit:	256 megabytes

A set X of k positive integers is called *squary* if any subset of X containing $k-1$ numbers sums up to a square of integer. For example, a set $\{1, 22, 41, 58\}$ is squary because $1+22+41 = 64 = 8^2$, $1+22+58 = 81 = 9^2$, $1+41+58 = 100 = 10^2$, $22+41+58 = 121 = 11^2$.

Given n and k, find the partition of n to k distinct integers that form a squary set, or find out that it is impossible.

Input

Input file contains two integer numbers: n and k ($2 \le k \le 30$, $2 \le n \le 200\,000$).

Output

If there exists a squary partition of n to k distinct positive integers, output "`YES`" at the first line of the output file. The second line must contain k integers — the partition itself.

If there is no squary partition of n to k distinct integers, output "`NO`".

Examples

squary.in	squary.out
122 4	YES 1 22 41 58
2 2	NO