Problem 1. Ants

Input file:	input.txt
Output file:	output.txt
Time limit:	1 second
Memory limit:	256 mebibytes

Treasure hunting is a challenge that takes an inventive mind. Behold! ¡¡Ants¿¿, a game that will peak your creativity!

The game is played in a $n \times m$ field, with chairs in some of its cells. There is also an anthill in the field. Beginning from the first second, ants crawl out, one per second, each aiming for its favorite chair. They take the shortest route; it takes an ant one second to move from a cell to any of its neighbor cells. Two cells are considered neighbors if they share a common edge. Having reached its favorite chair, the ant annihilates with joy. All that's left to the player is to observe and count the number of ants coming to their end on each second.

The ant is a tiny creature, and chairs do not block its movement in any way. There can be a chair in any of the game field cells, including the cell with the anthill.

Input

The first line of the input file contains five integers n, m, k, r, c — the field size, number of chairs and anthill coordinates, respectively ($1 \le n, m \le 100$, $1 \le k \le n \cdot m$, $1 \le r \le n, 1 \le c \le m$).

Next comes the description of the field — a rectangular table of n lines and m columns. Each cell of the table contains a non-negative integer i ($i \le k$). If $i = 0$, the cell is empty, otherwise it contains a chair with the number i.

It is guaranteed that all integers from 1 to k occur in this table once. The favorite chair of the ant which crawls out of the anthill on the ith second has the number i.

Output

In the first line of the output file, print the integer e — the number of events describing the self-annihilation of ants. In the following e lines, describe the events — two integers per each — the number of the second and the number of ants that have annihilated during that second. Moments in time must be strictly ascending, and the number of ants in each event must be positive.

Examples

input.txt	output.txt
3 5 4 2 5 0 0 1 0 0 0 0 0 0 3 0 4 0 0 2	3 3 2 4 1 8 1

Problem 2. Blend

Input file:	`input.txt`
Output file:	`output.txt`
Time limit:	3 seconds
Memory limit:	256 mebibytes

A popular architectural CAD program has an operation to create a solid body called "Blend". This operation takes two contours lying in parallel planes and builds a body between these contours. The area bounded by each of the contours in its plane becomes a top or bottom face of the body, and a set of lateral faces is constructed between the two contours.

The lateral surface is built in the following manner (see figure 1 below samples). On each of the contours, a start point is chosen. Each contour has a predefined orientation, i.e. a direction of traversal. The selected start points are joined by an *active segment* (which is a straight line segment). Next, the ends of the active segment are moved along the contours: the end on the bottom contour is moved along the bottom contour, and the end on the top contour is moved along the top contour. As the active segment moves, it sweeps the lateral surface. To obtain the complete lateral surface of the resulting body, each of the endpoints must go one full turn around the corresponding contour. It is forbidden to move an end of the active segment against the chosen direction of contour transversal.

For the purpose of this problem, assume that each contour is a closed polyline consisting of line segments. Note that the rules described above are ambiguous: you can build all sorts of lateral surfaces for the contours by varying how fast you move the ends of the active segment. To resolve the ambiguity, additional rules are introduced. When one end of the active segment is at a vertex of the polyline, the other end must also be at a vertex of the other polyline. Such positions of the active segment define the *lateral edges* of the created body, and the parts of the lateral surface between consecutive lateral edges are called *lateral faces*.

Hence each lateral face of the body must either join a segment of one of the polylines with a vertex of the other polyine (a triangle), or join a segment of one of the polylines with a segment on the other polyline (a quadrangle). The quadrangle can even be curvilinear: in this case, its precise geometry depends on the relative speed of the ends of the active segment. In reality, these speeds are also regulated, but this is irrelevant for this problem. Ultimately, the resulting body is strictly defined if we select which lateral edges must be built, i.e. how the vertices of the given polylines must be matched.

Generally, this is defined by the engineer — the CAD software user. For simplicity, the program provides a default matching. It is chosen in such a way that the total length of all the lateral edges is minimal possible. Write a program that will calculate this default matching based on the provided input contours.

Note that this description does not tell anything about self-intersections of the lateral surface. Naturally, if these problems do occur, the body will not be a solid body, strictly speaking. Such anomalies are possible and allowed with the "Blend" operation, including the case when lateral edges are chosen by default. The discovery and correction of these problems is done with completely different algorithms.

Input

The first line contains three integers: M — number of vertices in the bottom polyline, N — number of vertices in the top polyline, and H — height of the body ($3 \le M, N \le 300$, $1 \le H \le 10^6$).

The following M lines define the coordinates of the vertices of the bottom polyline. Each line contains two integers: x- and y- are the coordinates of the vertex. Vertices are listed according to the selected direction of traversal of the polyline; the last defined vertex is followed by the first defined vertex. The remaining N lines contain the coordinates of the top polyline in the same format.

All coordinates do not exceed 10^6 in absolute value. The bottom polyline lies in the plane $z = 0$, and the top polyline lies in the plane $z = H$. All vertices on every polyline are distinct. Self-intersections are allowed.

Output

In the first line of the output file, print a real number A — the total length of the lateral edges in your solution, and an integer K — the number of lateral edges ($\max(M, N) \le K \le M + N$).

In the remaining K lines, print the lateral edges in the order as they are sweeped by the active segment. You can start from any lateral edge. For each lateral edge, print a separate line with two integers i and j — the indices of the vertices joined by the edge the in bottom and top polylines, respectively ($1 \le i \le M$, $1 \le j \le N$).

The absolute or relative deviation of the value A from the optimal one must not exceed 10^{-9}.

Examples

input.txt	output.txt
3 3 1 0 0 2 0 1 1 3 -1 1 2 -1 -1	4.878315177510850 3 1 3 2 1 3 2
9 3 3 0 2 1 0 2 0 7 0 8 1 8 5 3 7 2 7 0 6 6 2 2 6 2 2	33.210944197060996 9 1 3 2 3 3 3 4 1 5 1 6 1 7 2 8 2 9 2

Illustration

(on the next page)

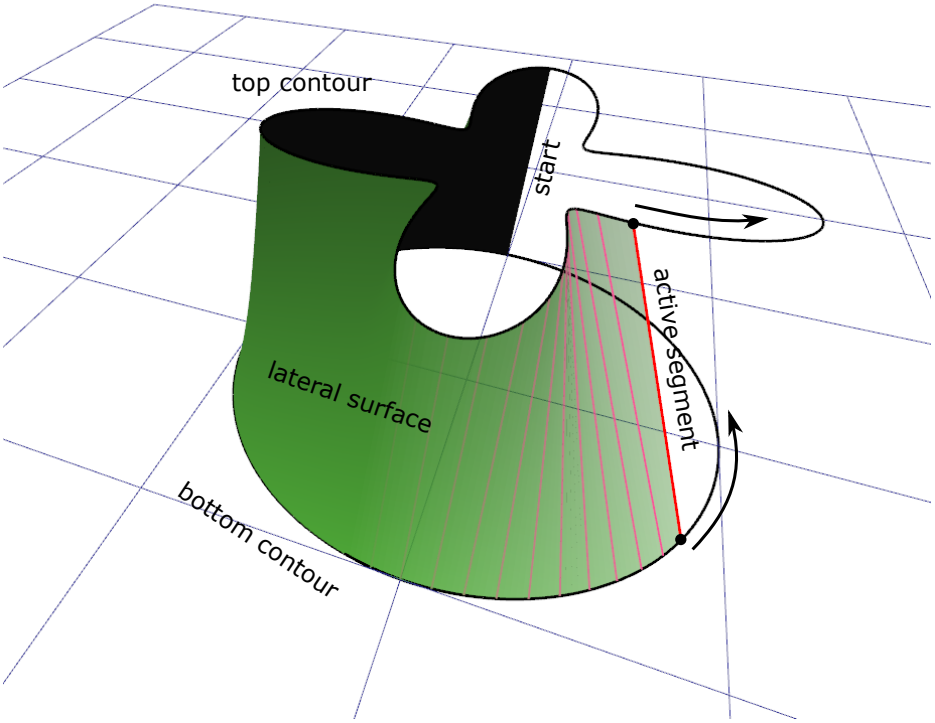


Figure 1: Lateral surface sweeped by the active segment.

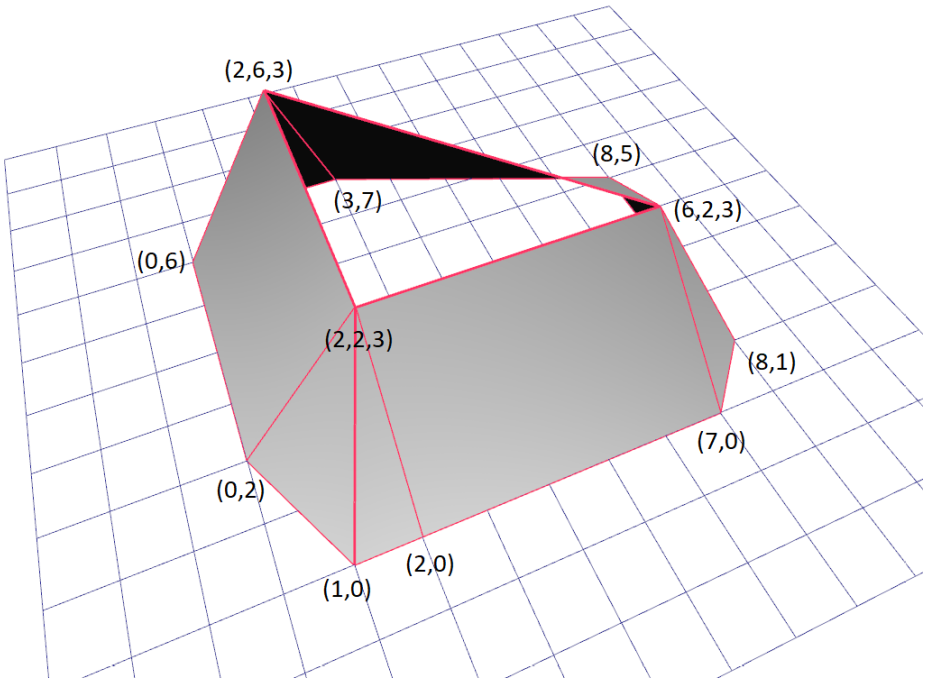


Figure 2: Second sample: lateral edges and faces.

Problem 3. Backup

Input file:	`input.txt`
Output file:	`output.txt`
Time limit:	2 seconds
Memory limit:	256 MB

There are many different backup strategies. The easiest to understand are the differential and incremental backups.

With incremental backup, we create a copy of the data which were changed from the moment of the last backup: for instance, after a full backup on Sunday, on Monday we copy the data that were changed during Monday; the same goes for Tuesday, etc. With such strategy, the volume of data copied is relatively small, but it may require a lot of files to restore the data. For instance, to restore data on Friday, we would need backup copies from Sunday (full backup), Monday, Tuesday, Wednesday and Thursday.

With differential backup, we regularly create a full copy — for instance, every Sunday. Next, on each day of the week we make a copy of all data modified after the last full backup: a Monday copy on Monday, a Monday and Tuesday copy on Tuesday, etc. The total volume of the backed up data is relatively large, but you only need two files to restore data: the last differential and the last full backup.

Dump(1), a popular backup freeware, operates with the concept of backup level, which can perform incremental or differential backup as well as more complex strategies, which cannot be reduced to the two former methods. The idea of backup levels as such is pretty simple: when we make an N level backup, we copy all files modified from the moment of the last backup with the level lower than N. If there are no preceding lower-level backups, we make a complete backup copy of all data.

Write a program to figure out which backup files should be used for recovery based on the backup schedule.

Input

The first line of the input file contains the number M – the total number of tests in the file ($0 < M \le 100\,000$).

Each following line of the input file consists of eight space-separated integers. The first seven numbers are schedule of backups, i.e. the levels of backup N_i, which are performed on the corresponding days of the week i: on Sunday, Monday, Tuesday, etc. ($0 \le N_i \le 9$). It is guaranteed that the zero-level backup is performed on Sunday, i.e. the first number is zero.

The eighth number is the day d when data must be recovered ($0 \le d \le 6$). Note that 0 is Sunday and 6 is Saturday.

Output

For each line of the output file, print a sequence of numbers d_j corresponding to the days when the backup copies necessary for recovery were made ($0 \le d_j \le 6$). Note that the earliest copy is always recovered first, followed by a later copy, hence the following must hold true: $d_j < d_{j+1}$ and $N_{d_j} < N_{d_{j+1}}$. Moreover, the recovery always ends with the latest copy, so the last number in the line must equal d.

Examples

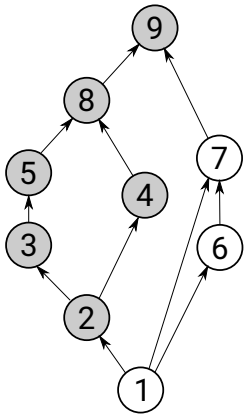
input.txt	output.txt
4 0 0 0 0 0 0 0 4 0 8 8 8 8 8 8 3 0 2 3 4 5 6 7 4 0 7 2 6 3 5 4 4	4 0 3 0 1 2 3 4 0 2 4

Problem 4. bit gisect

Input file:	`standard input stream`
Output file:	`standard output stream`
Time limit:	3 seconds (6 seconds for Java)
Memory limit:	256 mebibytes

Vasya the programmer is using Bit, a version control system for storing his code. The history of changes in Bit can be viewed as an oriented acyclic graph. The nodes of this graph are called revisions. When Vasya wants to make changes in the code, he can take an arbitrary revision A and change the code, resulting in a new revision D. In these case, we call the revision D dependent of the revision A. Each of these dependencies in Bit is expressed as an arc from the revision node A to the revision node D. Mergers are a special revision type in Bit: Vasya can join any two arbitrary revisions A and B and merge them into a new revision D. In this case, the revision D will depend on two revisions: A and B. The repository also contains a single "initial" revision: it is the only revision which does not depend on any other revision. All other revisions in Bit depend on one or more other revisions. Vasya has finished working on his project: there is strictly one revision which does not have any dependent revisions.

But one day, something went wrong, and the code became bug-ridden. Vasya is still trying to find them. Here is what Vasya's bugs do: if a bug appears in a revision A, it affects all revisions accessible from A in the Bit graph: the revision A, all revisions dependent on A, all revisions dependent on revisions dependent on A, etc. To find the bug, Vasya can switch between revisions and check for the bug in the different revisions, thus narrowing down the suspect list. In the end, he will find the initial revision containing the bug.



Vasya is getting bug reports from the users, one after another, and he must find all these bugs, one by one. Help Vasya fix the code as quickly as possible.

The figure shows an example with 9 revisions. The revision `1` is the initial revision. The revisions `7`, `8`, and `9` are mergers, each depending on two other revisions. The revision `2`, which sprouted a bug, and all dependent revisions suffering from that bug are highlighted in gray.

Interaction Protocol

This is an interactive problem. Instead of file input-output, you will be working with a special program "— the interactor. You will be interacting with this program through the standard input-output streams.

Upon the start, your program is fed information about revisions through the standard input stream. The first line contains an integer N — the number of revisions in Bit ($1 \le N \le 1\,000$). The following N lines contain the descriptions of revisions, each beginning with a revision number — a string of 6 hex digits (digits from `0` to `9` and lower case latin letters from '`a`' to '`f`'). It is guaranteed that this description does not contain a revision with the number `000000`. It is followed by an integer k — the number of revisions from which the current revision depends ($0 \le k \le 2$), followed by k space-separated revision numbers. It is guaranteed that by the moment of describing a new revision all descriptions of the revisions on which it depends have been provided.

Vasya must now find his bugs. It is guaranteed that there are no more than 10 000 bugs in the code.

A search for bugs begins with a line sent to your program through the input stream, containing the string "`bug`" and the number of the revision where the new bug has been located. If string "`000000`" is provided as the revision number, it means that all bugs have been found and the program must stop. Otherwise, the search for a bug begins.

Next, your program must send queries to the standard input stream. Each query must consist of a single line containing a command and the number of the revision upon which that command must be executed. The command can be one of the following:

- `check` — check the revision for bugs. The answer is provided as the line "`bad`" if the revision is bugged, and "`good`" if there are no bugs in the revision.
- `fix` — means that you have found a bug-ridden revision. After the command `fix` your program must start looking for the next bug.

For each bug, your program can make no more than 20 queries, otherwise the solution will receive the Wrong Answer verdict.

Make sure to print the line break symbol and clear the output stream buffer (the `flush` command of the language) after every printed query. Otherwise, the solution can get Timeout.

Example

For ease of reading, the commands in the example are separated by blank lines.

standard input stream	standard output stream
9 000001 0 000002 1 000001 000003 1 000002 000004 1 000002 000005 1 000003 000006 1 000001 000007 2 000006 000001 000008 2 000004 000005 000009 2 000007 000008 bug 000008	
bad	check 000004
good	check 000001
bad	check 000002
bug 000009	fix 000002
bad	check 000007
good	check 000006
bug 000000	fix 000007

Problem 5. Slots

Input file:	`input.bin`
Output file:	`output.bin`
Time limit:	1 second (1.5 seconds for Java)
Memory limit:	256 mebibytes

Game programming has its own tips and tricks for solving commonly occuring problems; these tricks are called "patterns". One of the patterns has to do with placing game objects within the array.

There is an array A of fixed size N consisting of *slots*, numbered from 1 to N. Throughput the game, game *objects* can be created and destroyed. From the moment of its creation, each object must be assigned into a slot in the array A. When an object is destroyed, its slot becomes free and can be eventually reused for newly created objects.

Moreover, each object also has an integer *ID*, which remains unique for the object throughout the game. IDs are assigned to objects consecutively starting from the number 1.

Finally, to assure that the creation of new objects does not take longer than $O(1)$, a stack of all currently free slots of the array A is maintained.

A new object is created in several steps:

1. The slot index x is popped from the top of the free slots stack.
2. The new object is assigned to the slot x.
3. The new object is assigned an ID, which is either greater by 1 than the last assigned ID, or equals 1 if it is the very first object in the game.

An object is destroyed in the following steps:

1. The slot x, which contains the object being destroyed, becomes free.
2. The slot index x is pushed onto the top of the free slots stack.

Such a system allows to emplace game objects into the array of slots conveniently. The operations of creation and destruction both take constant time. Moreover, we can even store "weak references" to objects thanks to having IDs! Weak reference as such a reference that you can check whether the object it points to has already been destroyed or is still alive.

In this problem, you must find out whether it is possible to reach the given configuration of the described system of slots; and if so, how to do that with the least possible number of operations. Two types of operations are allowed: create a new object and destroy the object in the given slot.

For each slot it is known whether it must be free or occupied in the end of the game. If a slot must be occupied, then an object with the prescribed ID must be located in it.

Initially all slots are free, as there are no live objects and the game has just begun. The free slots stack contains all the slots of array A in their natural order; the first slot is at the top of the stack.

Attention: in this problem, the input file **input.bin** and the output file **output.bin** contain **binary data**! All integers are provided in the format native to the checking machines, with little-endian byte order.

Input

The input file consists of $(N + 1)$ 32-bit integers. The first integer is N — the number of slots ($1 \le N \le 10^6$). The remaining N numbers define the required configuration.

Each number defines the contents of the corresponding slot:

- 0 — means that the slot must be free,
- $k > 0$ — means that the slot must contain an object with ID = k.

It is guaranteed that all given IDs are different and do not exceed $2 \cdot 10^6$.

Output

If the required configuration cannot be obtained, the output file must contain a single 32-bit integer -1.

Otherwise print $(M + 1)$ 32-bit integers. The first of these integers is M — the minimal number of operations necessary to produce the configuration. The remaining M numbers define the operations in the order of their execution.

Each operation is encoded as a single integer:

- 0 — means that a new object must be created,
- $1 \le x \le N$ — means that the object in the slot x must be destroyed.

All operations must be correct: you cannot destroy an object in a free slot; you cannot create an object when there are no free slots.

Examples

These examples show the contents of the input and output files in hex format. In the testing system, files will contain binary data. Examples in binary format can be downloaded in the ¡¡News¿¿ tab near the problem statements.

input.bin
0A 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00 05 00 00 00 0A 00 00 00 09 00 00 00 08 00 00 00 00 00 00 00 0C 00 00 00
output.bin
0F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 06 00 00 00 07 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 09 00 00 00

input.bin
05 00 00 00 05 00 00 00 04 00 00 00 03 00 00 00 02 00 00 00 01 00 00 00
output.bin
FF FF FF FF

Example explanation

In the first example, we consecutively create 8 objects, which occupy the first 8 slots, with the IDs from 1 to 8. Next, the objects in the slots 6 and 7 are destroyed (with IDs of 6 and 7, respectively), in such order that the slot 7 ends up at the top of the free slots stack. Next, two objects are created: they are assigned IDs 9 and 10, which are next in order, and fall into the slots 7 and 6, respectively. Finally, two more objects with IDs 11 and 12 are created, which are assigned to the last two slots, and the object with ID 11 in the slot 9 is destroyed.

In the second example, it is impossible to achieve the required configuration.

Problem 6. Tea Sort

Input file:	`input.txt`
Output file:	`output.txt`
Time limit:	3 seconds
Memory limit:	256 mebibytes

After a long day of laborious coding, Stepan decided to have five o'clock tea. Programmers love tea of all kinds: kocha, sencha, matcha, mocha and the like. Stepan went into the kitchen and discovered that all packs with tea in the drawer had been mixed. What a mess! Stepan likes order. He wants to have the packs arranged in the usual order.

There are N packs of tea in the drawer, arranged in K rows. In each row, the packs are lined up, from the pack closest to Stepan to the most distant. But the drawer is very low, and you can only pick the closest pack of tea in each row. So, the only possible operation here is to pick the closest pack from a row i and put it into the row j, where it becomes the closest pack. In the programming mumbo-jumbo, each row is a stack, or "LIFO".

Each pack is marked with an integer, which denotes the type of tea. Stepan likes it when:

1. the number of packs in every row is the same,
2. the type number of the pack from the row i is less than or equals the type number of the pack from the row j (for any $i < j$),
3. in every row, packs are arranged by type in the ascending order from the closest to the most distant pack.

Suggest a plan with no more than $13 \cdot N$ operations of moving packs, resulting in everything being in place according to Stepan's taste. You do not need to minimize the number of operations.

Input

The first line of the input file contains two integers N and K — the number of packs of tea and the number of rows ($1 \le N \le 10^5$, $11 \le K \le 111$).

It is followed by K lines, with each r-th line defining the r-th row of tea packs. For each row, an integer T_r is provided, denoting the number of packs in the row, followed by T_r integers — the types of tea in the packs of this row ($0 \le T_r \le N$). Packs in a row are listed from the closest to the most distant.

It is guaranteed that the sum of all T_r equals N, and that N is divisible by K. Integers denoting tea types are not greater than 10^9 in absolute value.

Output

In the first line of the output file, print one integer M — the number of operations in your plan ($0 \le M \le 13 \cdot N$).

In the remaining M lines, print the operations in the order of their execution. For each operation, print two space-separated integers: i — the number of the row, from which Stepan must pick the closest pack, and j — the number of the row where he must put it ($1 \le i, j \le K$).

Examples

input.txt	output.txt
33 11 3 -1 1 1 3 1 1 1 3 1 2 2 3 3 3 3 3 3 4 4 3 5 7 7 3 7 7 7 3 7 8 8 3 8 9 9 3 9 9 9 3 9 9 9	0
33 11 3 -1 1 1 3 1 1 1 3 1 2 2 0 5 3 4 3 3 4 4 5 7 7 3 3 7 7 7 3 7 8 8 3 8 9 9 3 9 9 9 3 9 9 9	13 6 7 6 7 6 7 6 4 7 6 7 6 7 6 5 4 5 11 5 4 5 4 11 5 4 5

Example explanation

In the first example, all packs are already placed in Stepan's favorite order, so there's nothing to do here.

In the second example, the row 4 is empty, and the packs that should be there have been placed in the rows 5 and 6. The first seven operations of the plan pick the packs of the type 3 from the row 6. Note that this pack is the most distant, so to pick it, the preceding three packs must be picked first. These packs are placed to the row 7 in reverse order; upon the removal of the desired pack of the type 3, they are placed back in the right order. The remaining six operations move two packs of the type 3 from the row 5 to the row 4. This shifts a pack of the type 4 further in this row.

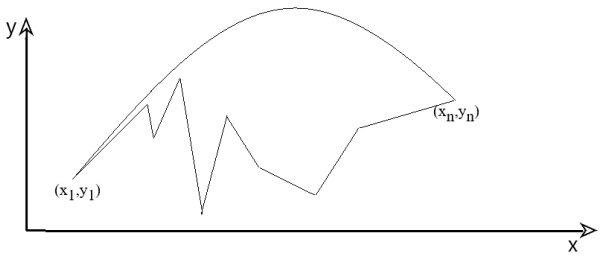
Problem 7. Shooting

Input file:	input.txt
Output file:	output.txt
Time limit:	1 second
Memory limit:	256 mebibytes

Father Theodore, in possession of a piece of stolen sausage, has hidden in the mountains; the concessioners have got nothing else to do but try to strike him in the eye with a rock, just like the elders have always advised.

Ostap and Kisa are at the foot of the mountains; they are throwing a rock, aiming at the thief's eye. Your task is to find the minimal initial speed such that the rock will reach Father Theodore, avoiding all obstacles during its flight.

For simplicity, assume that the landscape, where the three are situated, is a polyline on a plane. The coordinate y for the plane denotes height. The sizes of the people are negligible, so Ostap and Kisa can be reduced to a single point on the terrain, the other point being Father Theodore's eye.



A rock flies in a parabola, subjected to gravity acceleration g, which, for convenience, we will assume equal to 10. The air resistance is negligibly small. The rock must fly above the polyline.

Input

The first line of the input file contains the number n — the number of nodes in the polyline ($2 \le n \le 10^5$). It is followed by n lines. The i-th line contains two integers x_i, y_i — the coordinates of the i-th node of the polyline ($0 \le x_i, y_i \le 10^7$, $1 \le i \le n$). It is guaranteed that the sequence x_i is strictly ascending, i.e. $x_i < x_j$ when $i < j$.

We know that Ostap and Kisa are in the first node of the polyline (x_1, y_1), and Father Theodore is in the last node (x_n, y_n).

Output

The output file must contain a single real number — the minimal initial speed of the rock sufficient to give Father Theodore a black eye.

The relative or absolute error of the answer must not exceed 10^{-6}.

Examples

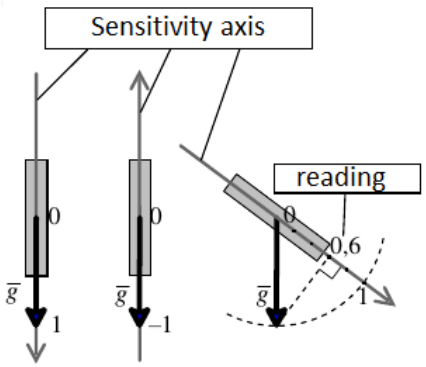
input.txt	output.txt
6 1 2 4 4 5 0 6 3 8 4 11 2	10
3 0 0 1 3 5 2	10.4963363572

Problem 8. Accelerometers calibration

Input file:	`input.txt`
Output file:	`output.txt`
Time limit:	1 second
Memory limit:	256 mebibytes

Modern smart devices can serve as GPS navigators and step counters; they can tell whether the user is walking, running, taking a taxi or a bus; they can automatically re-orient the image on the screen, etc. To do all of these and many other things, they rely on the so called accelerometers. A simplest single-axis accelerometer is connected with a direction, which is called its *sensitivity axis*; readings from a stationary acccelerometer allow to compute the deviation of its axis from a downward vertical, i.e. from the vector of gravity. If there are several single-axis accelerometers mounted on the device, the locations of their axes can be used to define the spatial orientation of the device as a whole.

A perfect accelerometer measures the cosine of the angle between its sensitivity axis and the direction of gravity. If the sensitivity axis is pointed downwards, i.e. along the gravity vector $\overline{g}$, the accelerometer shows the value 1, if the sensitivity axis is pointed upwards, it shows the value -1. If the axis is tilted at an angle to the vertical, the readings of the perfect accelerometer equals the projection of a single gravity vector $\overline{g}$ on the axis:



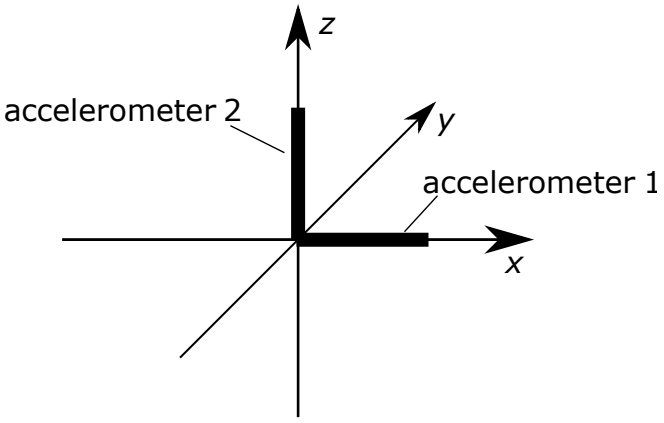
However, the production of the miniature accelerometers is not entirely fault-free. Defects of sensors lead to the following errors in readings:

1. deviation of the sensitivity axis of the accelerometer from the correct direction (bad mounting);
2. proportional change in readings, i.e. increase or decrease in all readings according to a common factor;
3. systemic shift of all readings by some constant;

If there are several accelerometers installed in the device, the errors described above can manifest differently in each of the accelerometers. However, if we examine a series of readings of an accelerometer in different positions, it turns out that the errors affect all of its readings identically.

To figure out the precise nature of the errors in the accelerometer readings in order to correct them digitally, accelerometers are calibrated. One of the ways to calibrate is to collect their readings in several strictly defined positions of the device and to calculate the parameters of the defects described above for each of the accelerometers of the device. These parameters can be then applied to compute the orientation of the device in arbitrary position.

In this problem, we will examine a device with two rigidly mounted accelerometers. When the device is oriented in space in the standard manner, one of the accelerometers must point to the right along the axis X, and the other one must point up along the axis Z (opposite to gravity). The user can rotate the device: assume that rotation is only possible in the plane XZ. In this case, the orientation of the device is defined completely by the angle of counter-clockwise rotation around the axis Y from the standard orientation (counter-clockwise means from the axis X to the axis Z).



To calibrate such an accelerometer, the accelerometer readings are taken when the device is rotated by degrees divisible by 90, which allows to define all parameters of the defects.

Based on the provided calibration datasets, find out the parameters of the defects and find a way to produce the real position of the device based on the accelerometer readings.

Input

The first four lines of the input file contain the readings of accelerometers when device is rotated by 0, 90, 180 and 270 degrees, respectively.

The next line of the input file contains an integer T "— the number of readings, for which the position of the device must be found ($1 \le T \le 1\,000$). Each of the remaining T lines contains accelerometer readings for the device orientation which must be computed.

Each reading contains two space-separated real numbers "— the readings of the first accelerometer (oriented along the axis X) and of the second accelerometer (oriented along the axis Z), respectively. Both numbers are given with 15 digits after the decimal point.

It is guaranteed that the readings in the input file have been acquired using the model of measurements and errors described in the problem statement. Errors in readings cannot be too big: the deviation of the sensitivity axis from the correct direction is never greater than 30°, and the proportional change of readings never changes them more than two-fold, and the shift is never greater than 5 in absolute value.

Output

In the output file, print T numbers, one per line. Each number is the turning angle for the device in degrees for the corresponding accelerometers readings. The turning angle must be within the range of 0° to 360° inclusive. The relative or absolute error of each answer must not exceed 10^{-6}.

Examples

input.txt	output.txt
0 -1 -1 0 0 1 1 0 5 -0.707106781186547 -0.707106781186547 -0.707106781186547 0.707106781186547 0.707106781186547 0.707106781186547 0.707106781186547 -0.707106781186547 -0.866025403784438 -0.5	45 135.00 225 315 60
-0.091012995433623 -0.946575228282571 -0.983288528313429 0.028440168472892 0.291012995433623 0.846575228282571 1.183288528313429 -0.128440168472892 5 -0.801067248717891 -0.628508848717885 -0.530934079986151 0.639439998807080 1.001067248717891 0.528508848717886 0.730934079986151 -0.739439998807080 -0.933661882864499 -0.430356435566630	45.000000000000043 135.000000000000000 225.000000000000000 315.000000000000000 59.999999999999986

Example explanation

In the first example, there are no defects in the accelerometers.

In the second example, all three error components influence each of the two accelerometers, and to a different degree.

Problem 9. Trees

Input file:	input.txt
Output file:	output.txt
Time limit:	1 second
Memory limit:	256 mebibytes

Vasya has gone deep into the graph theory. He's read a chapter about trees, and a problem has been bothering him: he must build a rooted tree with N nodes, with each node, except the leaf nodes, having strictly K children. The answer must be written as a list of edges; of all possible variants, he must find the lexicographically minimal one.

The list of edges is written into a string in the following manner. Each edge is described by a pair of integers — the numbers of nodes, which the edge connects. These two numbers must be written without leading zeroes, and there must be strictly one space character between them. The string consists of the descriptions of all $N - 1$ edges of the graph, written consecutively and separated by a single space character. It is assumed that all nodes are numbered from 1 to N, the root being number 1.

Vasya must find a lexicographically minimal string, which can be obtained in this manner for a rooted tree of the required kind. In lexicographical comparison assume that the space as a character is smaller than all digits.

For instance, let's build a tree with 5 nodes, with all non-leaf nodes having 2 children. A tree with edges $(1, 4), (1, 5), (4, 3), (4, 2)$ fits the requirement. The list of its edges can be written in a string in different ways:

- 4␣2␣4␣3␣1␣4␣1␣5
- 2␣4␣3␣4␣1␣4␣1␣5
- 1␣4␣1␣5␣2␣4␣3␣4

Here, each variant is smaller than the preceding one, but none are optimal. With these values of N and K the lexicographically minimal string 1␣2␣1␣3␣2␣4␣2␣5 is produced by a different tree.

Help Vasya solve this task, he's got a test on graph theory coming up!

Input

The first line of the input file contains two integers N and K, where N — is the number of nodes in the required tree, K — is the number of children of non-leaf nodes ($2 \le N \le 10^5$, $1 \le K \le 10^5$).

Output

If the tree with the specified parameters does not exist, print the word No into the only line of the output file.

Otherwise, in the first line of the output file, print the word Yes; in the second line, print the required lexicographically minimal string.

Examples

input.txt	output.txt
5 2	Yes 1 2 1 3 2 4 2 5
4 10	No

Problem 10. Team order

Input file:	input.txt
Output file:	output.txt
Time limit:	6 seconds
Memory limit:	256 mebibytes

N teams are planning to go to the All-Siberian Programming Contest, but none of them has registered yet. After the registration, teams are sorted lexicographically by their names. The contest jury want to know the order of the teams in the list. They did a little study. Having studied the teams' performance at other contests, the jury have compiled a list of possible names for every team. They found that the team with the number i could use any of its favorite names $S_{i1}, \ldots, S_{iK_i}$ to register.

Here comes the question. Is it true that their little study is absolutely worthless? In other words, is it true that teams can end up in the list in any order? If this is wrong, the jury at least wants to know a single order of the teams which would be **impossible**.

Input

The first line of the input file contains a single integer N — the number of teams ($1 \le N \le 350$). It is followed by N blocks of lines, each describing a team. Teams are numbered from 1 to N in the order as they are described in the input file.

The first line of the description of the i-th team contains a single integer positive number K_i. The description block of the i-th team consists of $K_i + 1$ lines, including the line with the number K_i. The following K_i lines contain the possible names of the i-th team, one per line: $S_{i1}, \ldots, S_{iK_i}$. A team name can only contain lowercase Latin letters. Each name is non-empty and is not longer than 100 characters. All K_i names are distinct.

It is guaranteed that different teams do not have matching names. It is also guaranteed that $\sum_{i=1}^{N} K_i \le 350$.

Output

If the teams names can end up in the list in any order, print the word `YES` in the output file.

Otherwise, print the word `NO` in the first line; in the second line, print any impossible teams order as a list of N space-separated integers. Here k-th integer is the number of the team being k-th in the lexicographically sorted list of names.

Examples

input.txt	output.txt
3 1 teamname 2 vanechka ivan 4 albatross teddybear vitalya pythonists	YES

input.txt	output.txt
2 2 geometrylovers epsiszero 1 speedcoderz	NO 2 1

Problem 11. Ostap's dream

Input file:	input.txt
Output file:	output.txt
Time limit:	3 seconds
Memory limit:	256 mebibytes

Ostap is having a bad dream. In his dream, he is locked inside a convex polygon with N vertices. The boundary of the polygon is split into three continuous parts, with each edge of the polygon belonging strictly to one part. If Ostap finds himself near one part of the polygon boundary, he can fall prey to the Sweet Widow. Near the second part lives Korobeinikov, the record keeper, who is holding a grudge against Ostap; the third part is occupied by his archenemy competitor, Father Theodore.

Ostap wants to stay away from the evil three. All three threats are equally serious, so he wants to be at equal distance from the three parts of the polygon. Find the right spot!

Input

The first line of the input file contains an integer N — the number of vertices ($3 \le N \le 40\,000$).

Each of the following N lines contain two integers X_i and Y_i — the coordinates of ith vertex of the polygon. The coordinates do not exceed 10^6 in absolute value. The vertices are listed in counter-clockwise order. It is guaranteed that the polygon is strictly convex.

The last line contains three distinct integers C_1, C_2, C_3, defining how the polygon is split into parts ($1 \le C_j \le N$). Each of the three vertices with these numbers has one incident side of the polygon belonging to one part and another incident side in another part. The vertices are numbered from 1 to N in the order of their definition.

Output

If such a point exists, print `Yes` in the first line of the output file. In the next line, print two real numbers X_c and Y_c — the coordinates of the point to which Ostap wants to move.

The distances from this point to the three parts of the polygon boundary must differ from each other by no more than 10^{-6} in absolute or relative value.

If there is no such point, print `No` in the only line of the output file.

Examples

input.txt	output.txt
4 8 0 5 3 3 3 0 0 4 3 2	Yes 3.62132025 1.49999996