Problem 2. Matrix

Input file:	*standard input*
Output file:	*standard output*
Time limit:	4 seconds
Memory limit:	512 mebibytes

You are given an $n \times m$ matrix filled with zeroes and ones. We consider all the $m!$ possible reorderings of columns of the matrix. In particular, we are interested in those reorderings for which the ones in every row are placed in adjacent columns (i.e., they form a contiguous part of the row).

Your task is to compute the number of such reorderings of columns, modulo a given number x.

Input

The first line of the input contains three positive integers n, m, and x ($1 \leqslant n, m \leqslant 2\,000$, $2 \leqslant x \leqslant 1\,000\,007$); The following n lines describe the subsequent rows of the matrix. Each of these lines contains exactly m space-separated digits 0 or 1.

Output

Your program should output a single integer: the number of interesting reorderings of columns of the matrix, modulo x.

Examples

standard input	*standard output*
3 4 42 1 1 0 1 1 0 1 1 1 1 0 0	2

Problem 3. To Kill the Dragon

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

A small group of brave magicians gathered together to kill the bad dragon. Since they do it every week (sometimes in several attempts), they know the dragon's behaviour by heart and can predict all his moves with one second precision.

There are two kinds of magicians: attackers and healers.

The purpose of attackers is to damage the dragon, and the purpose of healers is to heal injured magicians. Each attacker can surround the dragon with an anti-magic aura that continuously damages the dragon. Similarly, each healer can surround all magicians on the battlefield with an aura that continuously restores their health.

This damage and healing in a unit of time are directly proportional to the power of the respective aura. While a magician maintains his spell, the power of the aura increases linearly and proportionally to the time of maintaining the spell. But the power of the aura of each magician can not exceed some fixed value associated with the magician. While a magician does not maintain the spell, the power of the aura decreases linearly and proportionally to time. The power of an aura can never become negative, that is, it does not decrease after falling to zero.

Auras of different magicians are independent and don't influence each other. The total damage the dragon gets equals to the sum of damages from every attacker's aura. Similarly, the total healing each magician gets equals to the sum of amounts of healing from every healer's aura. Hit points of a magician can not exceed their initial value and are the same for each magician.

During the battle, the dragon uses different spells in some fixed order. The effect of each of his spells lasts until he starts to cast the next one. The same type of spell can be used multiple times in a row.

The first spell is Fire Storm. It creates a wall of fire all over the field except for a small plot of land in the center of the storm. While a magician is located in the fire, he gets constant damage and feels too uncomfortable to cast a spell. There is enough space in the center of the storm to locate all magicians without getting damaged, and there they can cast spells. The center of the storm can appear in different places each time this spell is used.

The second spell is Dragon's Breath. It also covers the whole field with fire, but this time, there is no place to hide from it. Fortunately, there are some relatively safe locations, where at least magicians' ears are not blocked by howling wind, so they can cast their spells. However, such locations are too small, and at each of them, there is room for only one magician. The number of such locations always equals to the number of magicians. Similar to the Fire Storm, relatively safe locations can appear in different places each time. Since Dragon's Breath takes much magic power, the dragon never uses it more than two times in a row.

After a while, the dragon gets tired of the battle and strikes all magicians with his tail, instantly killing them.

Magicians don't like to be fried, so each of them runs to the center of the fire storm along a straight line right after it starts. In case of Dragon's Breath, each magician chooses one of the relatively safe locations for himself and runs straight to that location. The size of a magician is negligibly small compared to the size of the field, so one can consider that several magicians can run through the same point at the same time. However, it doesn't deny the fact that each relatively safe point can be considered only for exactly one magician. Dragon's spells can cause different damage but it is always strictly greater than the maximal total healing from all the healers.

The dragon does not like to hurry, so the duration of his spells is long enough for all magicians to reach every safe location of the current spell from every safe location of the previous spell regardless of which

magicians go to which safe locations. (Note however that there is *no* guarantee that they can get to their destinations alive.) Moreover, if the current spell is Fire Storm, then its duration is long enough not only to reach its center but also to increase the power of all attacking and healing spells to maximum value and to totally restore all magicians' health regardless of how much health they had.

If at some moment hit points of any magician fall to zero, all magicians die and their souls go to the nearest cemetery. However, it is not so easy to kill magicians' souls, they can come back to their bodies and, having restored their powers, try to start a new battle. In this problem, however, this scenario does not interest us at all.

Killing the dragon is slightly more complicated. For this problem, it is necessary not only to make his hit points equal to zero but also to cast a final spell. Casting the final spell requires all magicians to gather at some safe point where they don't get damage, that is, in the center of the storm. Obviously, magicians must be alive to cast the final spell. The final spell can be cast instantly. At the very moment when the dragon switches two attacks, the first of them is considered to have effect.

The first attack starts at time 0. Each subsequent attack starts at the time the previous attack is finished. At time 0, all magicians are located at the origin and every healer have had enough time to restore his healing aura to the maximal level. However, the attackers can not establish any positive damage aura before the battle starts.

Input

In the first line of input, you are given three nonnegative integers n, m and k: the number of attackers, the number of healers and the total number of dragon's attacks ($1 \leqslant n + m \leqslant 10$; $1 \leqslant k \leqslant 10$). In the second line, there are three integers hp_m, v and hp_d: hit points and velocity of each magician and hit points of the dragon ($1 \leqslant hp_m, hp_d \leqslant 10^9$; $1 \leqslant v \leqslant 10^6$).

The next n lines describe attackers. On i-th of them, there are three integers dps_i, id_i and dd_i: the maximal damage per second for i-th attacker, increment of power of aura per second while i-th attacker maintains the spell and weakening of power of aura per second while this attacker runs to the next safe location ($1 \leqslant dps_i, id_i, dd_i \leqslant 10^6$).

The next m lines describe healers. On i-th of them, there are three integers hps_i, ih_i and dh_i: the maximal amount of hit points that can be restored per second by aura of i-th healer, increment of power of aura per second while i-th healer maintains the spell and weakening of power of aura per second while this healer doesn't maintain the spell ($1 \leqslant hps_i, ih_i, dh_i \leqslant 10^6$).

After that, there are k blocks which describe dragon's attacks in chronological order.

If an attack is a Fire Storm, the first line of its description contains a single word "`Storm`". On the second line, you are given space-separated integers d and t: the amount of damage per second caused by this attack and its duration in seconds ($1 \leqslant d, t \leqslant 10^6$). On the third line, there are two integers x and y: coordinates of the center of this storm ($-1000 \leqslant x, y \leqslant 1000$).

If an attack is a Dragon's Breath, the first line contains a single word "`Breath`". On the second line, you are given space-separated integers d and t: the amount of damage per second caused by this attack and its duration in seconds ($1 \leqslant d, t \leqslant 10^6$). The next $n + m$ lines contains two integers each x_i and y_i: coordinates of relatively safe locations. All relatively safe points are different.

If the attack is a Tail Strike, its description contains a single word "`Tail`".

It is guaranteed that among all attacks, there is exactly one strike with dragon's tail and this attack is the last one. It is also guaranteed that there can not be more than two Dragon's Breaths in a row.

All times are given in seconds, coordinates in meters, velocities in meters per second. Powers of auras are given in hit points per second, speeds of their strengthening and weakening in hit points per square second.

Note that, although the given values are all integers, all movement, damage, healing and changes in aura powers happen continuously.

Output

If the magicians are not able to kill the dragon, output a single line "`You are not prepared`".

Otherwise, on the first line, output "`No useful loot again`". On the second line, output one integer: the minimal number of attack during which the dragon can die. Attacks are numbered from 1.

Examples

input.txt	output.txt
1 1 2 1 5 10 1 3 2 1 1 1 Storm 7 11 0 0 Tail	No useful loot again 1
1 1 2 1 5 10 1 3 2 1 1 1 Storm 7 10 0 0 Tail	You are not prepared

Problem 5. Asynchronous Exceptions

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	256 mebibytes

Your job is to simulate the operation on a machine called ACM (Asynchrounous Calculating Machine) to determine how long each program takes to finish their job. This helps detecting unexpected thread behaviors, like race and deadlock.

ACM has many processing units and can run many threads simultaneously. It runs a global scheduler, which maintains all the threads, assigns a thread on each processing unit and manages all semaphores. Each thread has its own thread id and every thread is one of three states: `RUNNING`, `READY`, and `WAITING`. Global scheduler has one thread queue which contains all the threads in `READY` state.

A program consists of one or more code blocks. Each code block has a list of ACM operations, including the actual computations and thread creation. Each thread runs operations in only one of the given code blocks, and it runs operations from the top of the block to the bottom, unless it receives killThread-signal from other threads.

ACM simulation program simulates 9 operations. Here I describe them all. Please note that word enclosed by '<>' should have some actual name or value.

- `compute <clock>` — spend *clock* clocks for computation. When this operation is interrupted after spending T clocks by other threads or the scheduler and is resumed later, it consumes $clock - T$ clocks.
- `<threadVar> <- forkR <code block>` — generate a native thread running on *code block* and store its id in a thread variable $threadVar$. It can run simultaneously with any threads which are generated in other processing unit. If $threadVar$ is used before in the thread, its previous value is overwritten and lost (this means that we will not be able to refer to the thread although it might run forever).
- `<threadVar> <- forkI <code block>` — generate a virtual thread running on *code block* and store its id in a thread variable $threadVar$. A virtual thread shares some OS resources with its parent thread, so they can't run simultaneously even there are idle processing units. This restriction is applied to any two threads which are directly or indirectly connected by the parent-child relatinship of $forkI$. If $threadVar$ is used before in the thread, its previous value is overwritten and lost.
- `yield` — make the current thread from running state to ready state. The thread goes to the end of the thread queue.
- `killThread <threadVar>` — send a kill signal to the $threadVar$ thread. $threadVar$ guaranteed to be used in either $forkR$ or $forkI$ operations in the same code block before this operation. When a thread receives a kill signal, it immediately cancels its resource requests by lock, changes into ready state if it's in block state, and ends its operation when it's in running state. This operation does not change the semaphore values. If the target thread is already ended, this operation does nothing.
- `lock <semaphore name> <amount>` — request *amount* of *semaphore name*. If the value of *semaphore name* is greater or equal to the *amount*, it just subtracts the *amount* from the semaphore variable. Otherwise, the operation blocks the current thread until the variable is greater or equal to the *amount*. Once the variable gets greater or equal to the *amount*, the thread gets into `READY` state and goes into the thread queue after subtracting *amount* from the variable. If there are more than one thread requesting for the same semaphore, the thread which locked the semaphore first always gets `READY` first. So, the other threads won't get `READY` even if the semaphore variable meets their demands. If there are more than one thread getting `READY`, the thread which locked the semaphore earlier goes into the queue first.

- `unlock <semaphore name> <amount>` — add *amount* to *semaphore name*. Values of semaphores can be larger than initial values. The thread doesn't get blocked at all by this operation.
- `loop <loop count>` — run the code snippet between loop and corresponding `next` command for *loop count* times.
- `next` — this corresponds a loop command in the same code block. The code snippet between `loop` and `next` should be run.
- `end` — end the operation. This command only appears in the end of the code block, and each code block has this operation at the end.

Each parameter should either be a name or a digit. A name is an alphabetic string (case-sensitive) and its length is at most 200. Here are some explanations and constraints:

- *threadVar* is a name. It only refers to the id in the same thread. Each thread has its own value even it has the same name;
- *clock* is a non-negative integer which is at most 1000;
- *semaphore name* is a name;
- *amount* is a non-negative integer which is at most 1000;
- *loop count* is a non-negative integer which is at most 1000;
- total number of lines that all threads evaluate throughout simulation never exceeds 10^5;
- size of input file never exceeds 100K.
- threre is 1 thread queue;
- the id of the n-th created thread is n;
- CPU ids are assigned from 1 to number of CPUs.

Threads are assigned to CPUs when:

- a simulation starts (A thread is assigned to CPU 1; its code block is the first code block of the input);
- there are more than one threads that are in ready state which can be assigned to a CPU (threads are asssinged in the thread queue order; if there are more than one CPUs that can be assinged to the thread, the smallest CPU is selected);
- time step is a multiple of time slice (if the CPU has a thread that is running state, the thread goes to ready state in asscending order of the CPU id; after this, threads are assigned to each CPU in asscending order of the CPU id).

In each step, the simulator does round-robin preemption (when time step is a multiple of time slice), executes operations of each running threads and increments time step. When a compute operation is executed, the thread gets computing time. A thread that has computing time greater than 0 cannot do any operations. After each step, the simulator decrements positive computing time of each running thread.

Operations executed in the same time step is executed in ascending order of the CPU id. In the same time step, a CPU id of an operation is greater or equal to CPU ids of operations which have already been executed. Time step starts from 0.

Input

Input file begins with a line that contains two integers N and T ($1 \leqslant N, T \leqslant 1000$), separated by a space. They mean the number of time steps of the simulation and the maximum number of threads that ACM is capable, respectively. The next line contains a single integer C ($1 \leqslant C \leqslant 100$), which indicates the number of CPUs available.

The following line has an integer q ($1 \leqslant q \leqslant 1000$) that means time slice of the scheduler.

Description of semaphores follows.

It begins with the number of semaphores S ($0 \leqslant S \leqslant 1000$) which is followed by information of each semaphore, one per line. The information of a semaphore consists of an alphabet string (case-sensitive) and an integer, which specify the name of the semaphore and its initial value, respectively.

Finally code blocks are given. The number of code blocks B ($1 \leqslant B \leqslant 1000$) comes first, and then each code block is described. The first line of the description of each code block is the name of the block (case-sensitive alphabet string), followed by a colon (‘:’). Following lines describe content of the code block. A code block is an array of operations described above, and one operation is written in one line. You may assume that every code block always ends with an ‘`end`’ operation.

Output

If the number of living threads exceeds ACM’s capacity, put information of all threads which terminated before exceeding the capacity, and then put “`<<oops>>`”. If not, and if there are one or more threads not finished at the end of the simulation, put information of all threads terminated, and then put “`<<loop>>`”. Otherwise (i. e. when all threads terminates within the time), just put information of all threads. All quotes are for clarity.

Information of threads must be written in increasing order of thread ID, one per line. Information of a single thread is denoted by two integers, the thread ID and the time it terminated, separated by a single space character.

Examples

standard input	*standard output*
50 50 1 10 1 semaphore 1 1 codeBlockA: loop 2 compute 10 next end	1 20
50 50 1 10 1 semaphore 1 2 codeBlockA: hoge <- forkR codeBlockB yield compute 1 killThread hoge lock semaphore 1 compute 1 end codeBlockB: compute 1 lock semaphore 2 end	1 3 2 3
5 5 1 3 1 semaphore 1 1 codeBlockA: hoge <- forkI codeBlockA compute 1 end	1 1 2 2 3 4 4 4 5 5 <<loop>>
5 5 1 2 1 semaphore 1 1 codeBlockA: compute 1 hoge <- forkI codeBlockA hoge <- forkI codeBlockA end	1 1 2 3 3 3 <<oops>>

Problem 7. El Clasico

Input file:	`standard input`
Output file:	`standard output`
Time limit:	2 seconds
Memory limit:	256 mebibytes

From year to year, the Byteland football championship is a face-off between the two top teams: «Integer» and «Bytelona».

Given this, the Byteland Football League decided to perform a draft before each season in order to distribute strong new players between these teams.

The procedure works like this: players line up on the stage in a row from left to right in random order. Also for each player is given his transfer value.

Team managers take turns picking players. For each turn, they can choose either the player in the far right position, the player in the far left position, or both at once. Selected players leave the stage and do not participate in the rest of the selection process.

Given that most new players are immediately leased to other European clubs, the management of each team tries to assure that the total transfer price of all the players selected by his team is as maximal as possible.

This year, «Bytelona» gets to pick first.

Using the data for the transfer value of the players listed in the order in which they are standing on the stage, calculate what the total transfer price of the players selected by «Bytelona» will be, and what the total transfer price of the players selected by «Integer» will be.

Input

The first line of the input file contains one integer N — the number of players participating in the draft ($1 \leqslant N \leqslant 10^6$). The next line lists N integers p_i ($1 \leqslant p_i \leqslant 10^9$) — the transfer price of each player. Players are listed from left to right in the order in which they are arranged on the stage.

Output

Output two integers — the total transfer price of the players selected by «Bytelona» and the total transfer price of the players selected by «Integer».

Examples

standard input	standard output
4 3 1 5 2	8 3

Problem 11. Kickout

Input file:	`standard input`
Output file:	`standard output`
Time limit:	2 seconds
Memory limit:	256 mebibytes

The board game «Kickout» goes like this. An n number of identical playing tokens are arranged on n squares, which are numbered by sequential integers from 1 to n. Players take turns. A player's turn consists of moving a token from a square with the number i to a square with the number $2^k \cdot i$ (if such a square exists), where k is a positive integer (for example, a player can choose a token that is on square 3 and move it to one of the squares 6, 12, 24, 48, and so on). If the corresponding square already has a token on it, the two tokens kick each other out (self-destruct) and the square is left empty. The the player who cannot make a move will lose.

Depending on the value of n, in an optimal game, either the first or the second player can win. We'll define the sequence S like this: the j-th element is equal to the j-th value of n in ascending order in which the second player wins. Your task is to use the defined j to calculate S_j.

Input

The first line of the input file contains a single integer j ($1 \leqslant j \leqslant 10^9$) — the index of the required element in the S sequence.

Output

Output a single integer - the j-th element in the S.

Example

standard input	standard output
3	11

Problem 13. Edges Are Too Sharp!

Input file:	*standard input*
Output file:	*standard output*
Time limit:	1 second
Memory limit:	256 mebibytes

The company "Nanozavod, Inc." is specialized on design and assembly of nanobots. Now they meet another problem on their long way. Artificial joints require pieces of nanoribbons of exact length. Developers can create nanoribbon segments of the known lengths, but they are too long. Now they are trying to cut a segment into two parts of equal length.

A nanocutter looks like a hollow polygonal prism with another prism inside it. The inner prism has very sharp edges and is used to cut nanoribbon: when a piece of nanoribbon touches any edge of the inner prism, it is immediately cut in the contact point. Unfortunately, if a ribbon touches the inner prism in more than one point but by some longer segment (for example, it contacts the entire edge of the prism at the same moment), the whole piece of nanoribbon is completely destroyed. The outer prism is used to hold a piece of nanoribbon: the endpoints of the ribbon must be put onto the side surface of the outer prism. The tension of the ribbon is exactly uniform.

So, the developers now have to find two points on the outer prism's side surface such that the segment connecting these points has exactly one common point with the inner prism, and this common point is the middle point of the segment. All edges of both prisms which connect their base faces are parallel, so the problem can reduced to the same two-dimensional problem for the base faces of the prisms. Coordinates of both prisms' vertices are integers, bottom vertices form two polygons that are strictly convex (that is, there are no 180-degree corners of polygons). The inner polygon is strictly inside the outer one (their borders have no common points). Your task is to find coordinates of ends of any segment that satisfies the conditions of the problem. It is guaranteed that at least one such segment exists.

Input

The first line of input contains two integers: M is number of outer polygon vertices and N is number of inner polygon vertices ($3 \leqslant M, N \leqslant 3 \cdot 10^5$).

The following $M + N$ lines contain two integers x and y each ($|x|, |y| \leqslant 10^7$) that are coordinates of the polygon vertices (first M lines for the outer polygon, next N lines for the inner polygon). Vertices of each polygon are given in counterclockwise order.

It is guaranteed that the inner polygon is strictly inside the outer one.

Output

Print four real numbers x_1, y_1, x_2, y_2 specifying the coordinates of the nanoribbon's endpoints. The coordinates must be given with absolute error no more than $\varepsilon = 10^{-4}$ (that is, there must exist an exact solution to the problem (X_1, Y_1, X_2, Y_2) such that $|x_1 - X_1| < \varepsilon$ etc.). If there are multiple answers, print any one of them.

Examples

standard input	*standard output*
3 3 0 0 4 0 0 4 1 1 2 1 1 2	2.000000 0.000000 2.000000 2.000000

Problem 17. Card Duel

Input file:	*standard input*
Output file:	*standard output*
Time limit:	1 second
Memory limit:	256 mebibytes

Two players play a game. Each of players has initial amount of cards (n_1 and n_2 correspondently). On every turn players choose one card each and open it. Weaker card goes to retreat, the stronger one is taken back by the player who opened it. If the players have shown same cards, both go to retreat. The game is continued, until at least one of the players is over of cards. If one of the players still has at least a card when that happens, than he gets 1 point, and his rival 0. If both of the players are over of cards, than each gets 0.5 points. There are N kinds of cards totally. Strength relation of cards is nontransitive and defined with matrix A. If card i is stronger than j, A_{ij} is 1, and 0 otherwise. Define price of that game for the first player, supposing that second player plays optimally.

Input

The first line contains number N. Following N lines contain N numbers each, which define matrix A. Next line contains number n_1 and n_1 more numbers, each descripting a kind of corresponding card of the first player. The last line contains similar description of the second player's cards.

$1 \leqslant n_1, n_2, N \leqslant 8$.

$A_{ij} + A_{ji} = 1$ with $i \neq j$, $A_{ii} = 0$.

Output

Output price of the game for the first player with accuracy no less than 10^{-8}.

Example

standard input	*standard output*
3 0 1 1 0 0 1 0 0 0 2 3 2 1 1	0.00000000
3 0 1 0 0 0 1 1 0 0 3 1 2 3 3 1 2 3	0.50000000
3 0 1 0 0 0 1 1 0 0 3 1 2 3 3 2 2 3	0.66666667

Problem 19. Multisets

Input file:	*standard input*
Output file:	*standard output*
Time limit:	30 seconds
Memory limit:	512 mebibytes

We say a multiset of positive integers $\mathcal{A}$ is *s-smaller* than a multiset $\mathcal{B}$ if the smallest integer x that has a different number of occurrences in the two multisets appears more times in the multiset $\mathcal{A}$.

For example, $\{1, 2, 3\}$ is *s-smaller* than $\{1, 3, 3, 5\}$ and $\{1, 1, 4, 4\}$ is *s-smaller* than $\{1, 1, 4\}$.

We are looking at an integer sequence S. Note that every contiguous subsequence of S induces a multiset (the set of its elements with each element occurring as many times as it does in the subsequence). Your task is to find the k-th *s-smallest* multiset induced by a nonempty contiguous subsequence of S.

Input

The first line of the standard input contains two integers n and k ($1 \leqslant n \leqslant 150\,000$, $1 \leqslant k \leqslant \frac{n(n+1)}{2}$), the length of S, and the index of the sought multiset. The second line consists of n positive integers, none of which exceeds 10^6 — the sequence S.

Output

Output a single line containing the multiset induced by the k-th *s-smallest* contiguous subsequence of S. To avoid ambiguity, output the elements of the multiset in ascending order.

Example

standard input	*standard output*
6 5 1 2 1 3 5 1	1 1 2

The multisets induced by the 21 nonempty contiguous subsequences of $1, 2, 1, 3, 5, 1$ are, in sorted order: $\{1, 1, 1, 2, 3, 5\}$, $\{1, 1, 2, 3, 5\}$, $\{1, 1, 2, 3, 5\}$, $\{1, 1, 2, 3\}$, $\{1, 1, 2\}$, $\{1, 1, 3, 5\}$, $\{1, 2, 3, 5\}$, $\{1, 2, 3\}$, $\{1, 2\}$, $\{1, 2\}$, $\{1, 3, 5\}$, $\{1, 3, 5\}$, $\{1, 3\}$, $\{1, 5\}$, $\{1\}$, $\{1\}$, $\{1\}$, $\{2\}$, $\{3, 5\}$, $\{3\}$, $\{5\}$.

Problem 23. Far Eastern Federal Clock

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

Once upon a time there was a large country with many provinces and a great government. The government noticed that citizens of the furthest province are unhappy, and decided to do something for them.

After a serious sociological research, the government decided that the main problem of the province is the fact that every citizen has to buy his own watch to measure time. Thus the government decided to build a large tower with a giant analog clock on it, so that citizens could look at the common clock and save money on watches.

Many efforts and resources were spent, and finally the clock has been built and officially started in a grand ceremony. As the ceremony finished, people noticed that the clock has a small problem — it measures time incorrectly.

Since all the money allocated to this project was already spent, it was impossible to fix the clock. Instead, the government introduced a new position of Senior Clock Manager, whose responsibility was to adjust the clock by manually moving its hands.

It was decreed that:

- The clock will always be adjusted exactly at midnight.
- During the rest of the day, the clock will be adjusted periodically, every p minutes.
- The difference between the time displayed by the clock and the actual time must never exceed m minutes. Note that the smallest of all possible differences for a given hand positions is picked, for example, if the clock calculated time as 23:50 while the actual time is 00:05, the difference is 15 minutes.

Since the clock hands are very heavy, the Clock Manager's job is not an easy one. To help him, find the period of adjustment minimizing the total distance by which he must move the clock hands throughout the day.

The clock has two hands — for minutes and hours. Every minute, the minute hand jumps clockwise by t degrees, and the hour hand jumps by $t/12$ degrees. (A correct clock should have $t = 6$).

An adjustment is made immediately after the jump, and the effort is equal to the sum of angles between the current and the correct positions of both minute and hour hands.

Input

Input file contains floating point number t followed by integer m ($0 \leqslant t \leqslant 10^4$, $1 \leqslant m \leqslant 10^4$, t has no more than 3 digits after decimal point).

Output

Output file must contain minimum total effort s in degrees, with absolute error less than 10^{-2}, and the corresponding adjustment period p, $1 \leqslant p \leqslant 1440$. If there are several answers with the same total effort, output the one with maximum p.

Examples

input.txt	output.txt
12 1	9360.0000 2
18.0 90	1440.0000 30
6 1	0.0000 1440

Note

In the first sample, the clock moves twice as fast as it should, so every minute the error is increased by one minute. This requires an adjustment every two minutes.

In the second sample, the clock moves three times as fast as it should, but the acceptable error is much higher. It turns out that every 30 minutes the position of the minute hand coincides with the correct one, so if we choose the interval of 30 minutes, only the hour hand must be moved.

Problem 29. History Course

Input file:	*standard input*
Output file:	*standard output*
Time limit:	16 seconds
Memory limit:	256 mebibytes

You are to give a series of lectures on important historical events, one event per lecture, in some order. Each event lasted for some time interval $[a_i, b_i]$. We say that two events are related if their intervals have a common point. It would be convenient to schedule lectures on related events close to each other. Moreover, lectures on unrelated events should be given in the order in which the events have taken place (if an event A preceded an unrelated event B, then the lecture on A should precede the lecture on B).

Find the minimum integer $k \geqslant 0$ and an order of the lectures such that any two related events are scheduled at most k lectures apart from each other (lectures number i and j are considered to be $|i - j|$ lectures apart).

Input

The first line of input contains the number of test cases T. The descriptions of the test cases follow:

The first line of each test case contains the number n, $1 \leqslant n \leqslant 5 \cdot 10^4$. Each of the next n lines contains two integers a_i and b_i , $-10^9 \leqslant a_i, b_i \leqslant 10^9$ — the ends of the i-th interval. The intervals are pairwise different.

Output

Print the answers to the test cases in the order in which they appear in the input. The first line of the answer to each test case should contain the minimum value of k. The next n lines should list the intervals (in the same format as in the input) in an order such that any two related events are scheduled at most k lectures apart. Remember to put any unrelated intervals in the proper order!

Examples

standard input	*standard output*
1 3 1 6 2 3 4 5	1 2 3 1 6 4 5

Problem 31. Espionage For Dummies

Input file:	`espionage.in`
Output file:	`espionage.out`
Time limit:	2 seconds
Memory limit:	256 mebibytes

The government in a faraway land consists of n ministers. The king has discovered that exactly 2 (two) of the ministers are spies and tell everything to the enemies, but he doesn't know which ones.

He decided to find out by conducting a series of experiments. In each experiment, he tells a secret to some subset of his ministers. After some time, he checks if this secret is known to the enemies (through his own spies, of course). If it is, then he knows that at least one of the two spies was among the ministers that know the secret. If it isn't, then he knows that all ministers that know the secret are honest.

Of course, the king wants to find the spies in using as few experiments as possible. Each experiment is allowed to depend on the results of previous experiments. How many experiments is he going to need in the worst case? You also need to find the strategy to determine the spies using this many experiments.

Input

The input file contains one integer n, $3 \leqslant n \leqslant 64$.

Output

On the first line on the output file, print the number r of experiments needed in the worst case, and the number s of lines in the optimal strategy. On the next s lines describe the optimal strategy itself. t-th ($1 \leqslant t \leqslant s$) of those lines should be either "`CHECK` k a_1 a_2 ... a_k p q" or "`ANSWER` a b". The first one means "run an experiment by telling the secret to ministers a_1, a_2, ..., a_k ($1 \leqslant a_i \leqslant n$, $a_i \neq a_j$, $0 \leqslant k \leqslant n$), and if there are spies among them, go to line p, if not, go to line q ($t + 1 \leqslant p, q \leqslant s$, so we're only allowed to jump forward in the strategy)". The second one means "declare ministers a and b ($1 \leqslant a, b \leqslant n$, $a \neq b$) the spies". The execution of the strategy starts from the first line. The ministers in each command may be printed in arbitrary order.

The number of lines s must not exceed 10^5. In case there are several strategies that need at most r experiments, output any.

Examples

espionage.in	espionage.out
4	3 11 CHECK 1 1 2 3 CHECK 1 2 4 5 CHECK 1 2 6 7 ANSWER 1 2 CHECK 1 3 8 9 CHECK 1 3 10 11 ANSWER 3 4 ANSWER 1 3 ANSWER 1 4 ANSWER 2 3 ANSWER 2 4

Note

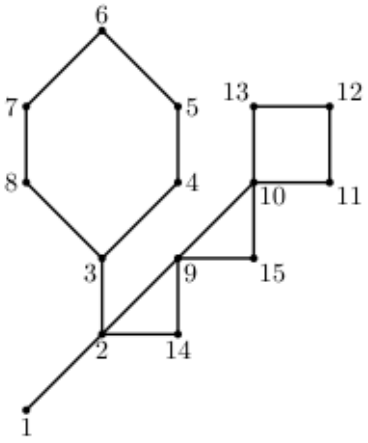
The source code size limit is 262144 bytes.

Problem 37. Cactus Automorphisms

Input file:	`cactus.in`
Output file:	`cactus.out`
Time limit:	3 seconds
Memory limit:	256 mebibytes

NEERC had featured a number of problems in previous years about cactuses — connected undirected graphs in which every edge belongs to at most one simple cycle. Intuitively, cactus is a generalization of a tree where some cycles are allowed.

In 2005, the first year where problems about cactuses had appeared, the problem was called simply "Cactus". In 2007 it was "Cactus Reloaded" and in 2010 it was called "Cactus Revolution". An example of cactus from NEERC 2007 problem is given on the picture below.



The challenge that judges face when preparing test cases for those problems is that some wrong solutions may depend on the numbering of vertices in the input file. So, for the most interesting test cases judges typically include several inputs with the same graph, but having a different numbering of vertices. However, some graphs are so regular that the graph remains the same even if you renumber its vertices. Judges need some metric about the graph that tells how regular the given graph is in order to make an objective decision about the number of test cases that need to be created for this graph.

The metric you have to compute is the number of graph automorphisms. Given an undirected graph (V, E), where V is a set of vertices and E is a set of edges, where each edge is a set of two distinct vertices $\{v_1, v_2\}$ $(v_1, v_2 \in V)$, graph automorphism is a bijection m from V onto V, such that for each pair of vertices v_1 and v_2 that are connected by an edge (so $\{v1, v2\} \in E$) the following condition holds:

$$\{m(v1), m(v2)\} \in E.$$

Each graph has at least one automorphism (one where m is an identity function) and may have up to $n!$ automorphisms for a graph with n vertices. Because the number of automorphisms may be a very big number, the answer must be presented as a prime factorization $\prod_{i=1}^{k} p_i^{q_i}$, where p_i are prime numbers in ascending order ($p_i \geqslant 2,\ p_i < p_i + 1$) and q_i are their corresponding powers ($q_i > 0$).

Input

The first line of the input file contains two integers n and m ($1 \leqslant n \leqslant 5 \cdot 10^4$, $0 \leqslant m \leqslant 5 \cdot 10^4$). Here n is the number of vertices in the graph. Vertices are numbered from 1 to n. Edges of the graph are represented by a set of edge-distinct paths, where m is the number of such paths. Each of the following m lines contains a path in the graph. A path starts with an integer k_i ($2 \leqslant ki \leqslant 1000$) followed by k_i integers from 1 to n. These k_i integers represent vertices of a path.

Adjacent vertices in a path are distinct. Path can go to the same vertex multiple times, but every edge is traversed exactly once in the whole input file. There are no multiedges in the graph (there is at most one edge between any two vertices). The graph in the input file is a cactus.

Output

On the first line of the output file write number k — the number of prime factors in the factorization of the number of graph automorphisms. Write 0 if the number of graph automorphisms is 1. On the following k lines write prime numbers p_i and their powers q_i separated by a space. Prime numbers must be given in ascending order.

Examples

cactus.in	cactus.out
15 3 9 1 2 3 4 5 6 7 8 3 7 2 9 10 11 12 13 10 5 2 14 9 15 10	1 2 2
2 1 2 1 2	1 2 1
15 7 3 1 2 3 3 4 2 5 3 6 2 7 3 8 2 9 3 10 2 11 3 12 2 13 3 14 2 15	6 2 11 3 5 5 2 7 2 11 1 13 1

Note

The first sample input corresponds to the picture from the problem statement. This graphs has $4 = 2^2$ automorphisms.

The second sample input is a simple graph with two vertices and one edge between them that has $2 = 2^1$ automorphisms.

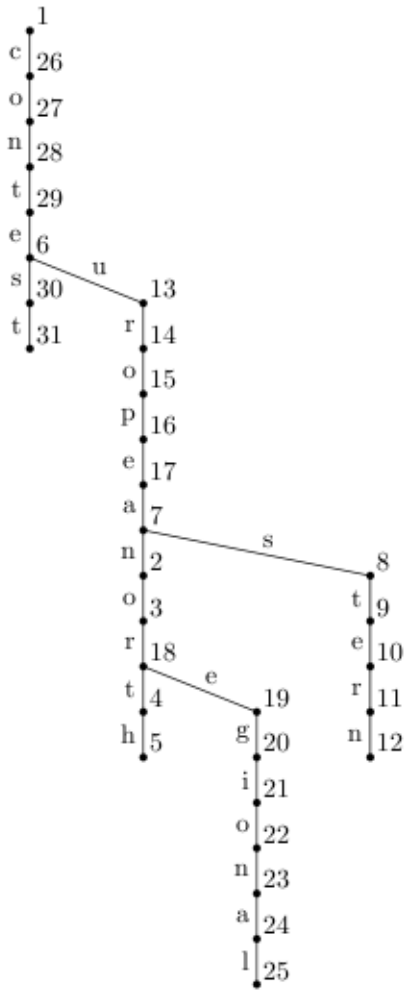
The third sample input is a "star" graph with a center vertex and 14 rays that has $14! = 87178291200 = 2^1 1 \times 3^5 \times 5^2 \times 7^2 \times 11^1 \times 13^1$ automorphisms.

Problem 41. Dictionary

Input file:	`dictionary.in`
Output file:	`dictionary.out`
Time limit:	2 seconds
Memory limit:	256 mebibytes

Petr and Dmitry are working on a novel data compression scheme. Their task is to compress a given set of words. To compress a given set of words they have to build a rooted tree. Each edge of the tree is marked with exactly one letter.

Let us define a dictionary that is produced by this kind of tree as a set of words that can be constructed by concatenating letters on edges on any path from any vertex in the tree (not necessarily root) and going away from root down to the leaves (but not necessarily finishing on a leaf). Boys have to construct such a tree with a dictionary that is a superset of the set of words that they are given to compress. This tree should have the smallest number of vertices between trees that satisfy the above condition. Any tree with the same number of vertices will do. Your task is to help them.



For example, in a tree on the picture above with the root marked as 1, a path from 7 to 5 reads “north”, a path from 16 to 12 reads “eastern”, a path from 29 to 2 reads “european”, a path from 3 to 25 reads “regional”, and a path from 1 to 31 reads “contest”.

Input

The first line of the input file contains the number of words in a given set n ($1 \leqslant n \leqslant 50$). The following n lines contain different non-empty words, one word per line, consisting of lowercase English letters. The

length of each word is at most 10 characters.

Output

On the first line output the number of vertices in the tree m. The following m lines shall contain descriptions of tree vertices, one description per line. Vertices are indexed from 1 to n in the order of their corresponding description lines. If the corresponding vertex is a tree root, then its description line shall contain a single integer number 0, otherwise its description line shall contain an index of its parent node and a letter on the edge to its parent node, separated by a space.

Examples

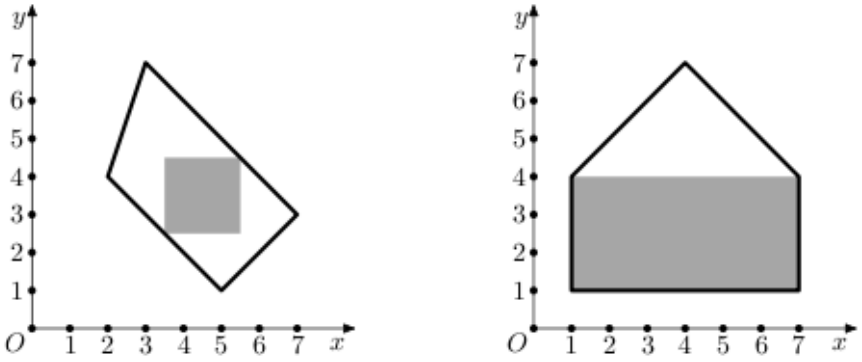
dictionary.in	dictionary.out
5 north eastern european regional contest	31 0 7 n 2 o 18 t 4 h 29 e 17 a 7 s 8 t 9 e 10 r 11 n 6 u 13 r 14 o 15 p 16 e 3 r 18 e 19 g 20 i 21 o 22 n 23 a 24 l 1 c 26 o 27 n 28 t 6 s 30 t

This sample output corresponds to the picture from the problem statement.

Problem 43. Easy Geometry

Input file:	`easy.in`
Output file:	`easy.out`
Time limit:	2 seconds
Memory limit:	256 mebibytes

Eva studies geometry. The current topic is about convex polygons, but Eva prefers rectangles. Eva's workbook contains drawings of several convex polygons and she is curious what is the area of the maximum rectangle that fits inside each of them.



Help Eva! Given the convex polygon, find the rectangle of the maximum possible area that fits inside this polygon. Sides of the rectangle must be parallel to the coordinate axes.

Input

The first line contains a single integer n — the number of sides of the polygon ($3 \leqslant n \leqslant 10^5$). The following n lines contain Cartesian coordinates of the polygon's vertices — two integers x_i and y_i ($-10^9 \leqslant x_i, y_i \leqslant 10^9$) per line. Vertices are given in the clockwise order. The polygon is convex.

Output

Output four real numbers x_{min}, y_{min}, x_{max} and y_{max} — the coordinates of two rectangle's corners ($x_{min} < x_{max}$, $y_{min} < y_{max}$). The rectangle must fit into the polygon and have the maximum possible area.The absolute precision of the coordinates should be at least 10^{-5}.

The absolute or relative precision of the rectangle area should be at least 10^{-5}. That is, if A' is the actual maximum possible area, the following must hold: $min(|A - A'|, \frac{A-A'}{A'}) \leqslant 10^{-5}$.

Examples

easy.in	easy.out
4 5 1 2 4 3 7 7 3	3.5 2.5 5.5 4.5
5 1 1 1 4 4 7 7 4 7 1	1 1 7 4