Problem A. Alter Board

Input file:	`alter.in`
Output file:	`alter.out`

Little Dima gave his little brother Petya interactive chess board of size $n \times m$ as a present. There are many awesome features of this board but one is Petya's favorite. He can choose any rectangle formed by board squares and perform an inversion. Every white cell in the inverted rectangle becomes black and every black one becomes white.

In the initial state the board is colored in chess style, namely every cell is either black or white and every two cells that share a side have different colors. Little Petya wants to perform several inversions described above to turn all cells into the same color. He is impatient, so he asks you to provide him with instructions to do it with the minimal number of inversions.

Input

The input file contains two integers, n and m ($1 \le n, m \le 50$) — the number of rows and columns on the board, respectively.

Output

The first line of the output file must contain k — the number of inversions required to transform the board.

The following k lines must describe inversions, one per line. Each line must contains 4 integers — row and column of one of the corners of the corresponding rectangle and row and column of the opposite corner. Any two opposite corners can be used to specify a rectangle.

Rows of the board are numbered from 1 to n. Columns of the board are numbered from 1 to m.

Sample input and output

alter.in	alter.out

2 2	2
	1 1 1 1
	2 2 2 2

Problem B. Burrito King

Input file:	`burrito.in`
Output file:	`burrito.out`

Two friends Albert and Barney came to the newly opened restaurant “Burrito King”. The restaurant had opened just yesterday, and Albert has got a special gift card, which allows the friends to get a free burrito. However, there is a constraint on the amount of ingredients — the burrito can contain at most g_i grams of ingredient i for all i from 1 to n.

There are two satisfaction parameters a_i and b_i for each ingredient — the amount of Albert's joy per gram of the corresponding ingredient, and the amount of Barney's unhappiness per gram, correspondingly.

Therefore, the total Albert's joy from the burrito is equal to:

$$\sum_{i=1}^{n} s_i \cdot a_i$$

The total Barney's unhappiness from the burrito is equal to:

$$\sum_{i=1}^{n} s_i \cdot b_i$$

Here s_i is the number of grams of the i-th ingredient in the burrito. Note, that s_i is not necessarily an integer, and $0 \le s_i \le g_i$.

Albert wants to make his total joy from the burrito to be at least A. Barney is his best friend, so Albert wants Barney's total unhappiness to be no more than B. Among all possible burritos that satisfy the above constrains, Albert wants to choose one that maximises his total joy.

Your task is to help Albert to choose s_i to satisfy these conditions or to find out that there is no solution.

Input

The first line contains three integers n, A, and B ($1 \le n \le 100\,000$, $0 \le A, B \le 10^9$), the number of ingredients, the least amount of Albert's joy and the maximal amount of Barney's unhappiness. Each of the following n lines contains a description of an ingredient: three integers g_i, a_i, b_i ($0 \le g_i, a_i, b_i \le 100$) — the maximal number of grams allowed, the amount of Albert's joy per gram and the amount of Barney's unhappiness per gram.

Output

The first line of the output must contain two real numbers — the maximal amount of his joy and the amount of Barney's unhappiness that Albert can obtain, satisfying the conditions in the problem statement, or “-1 -1”, if Albert cannot satisfy the conditions.

If the conditions are satisfiable the second line must contain n real numbers — the amount of each ingredient in grams.

Your output must have an absolute or relative error of at most 10^{-8}.

Any way to reach maximal Albert's joy that satisfies the given conditions can be printed.

Sample input and output

burrito.in	burrito.out

`2 5 5`
`2 2 1`
`2 2 4`
	`5.5 5`
	`2 0.75`

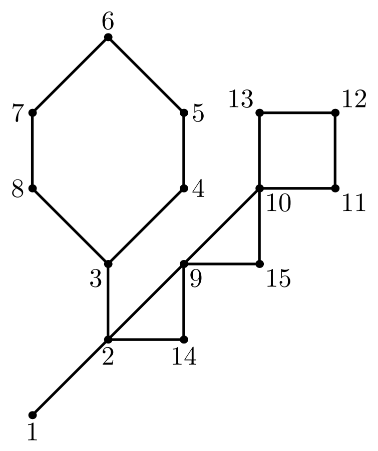
`2 5 5`
`2 2 2`
`2 2 4`
	`-1 -1`

Problem C. Cactus Generator

Input file:	`cactus.in`
Output file:	`cactus.out`

NEERC featured a number of problems 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 a problem about cactuses appeared, the problem was called simply "Cactus". In 2007 it was "Cactus Reloaded", in 2010 it was called "Cactus Revolution", and in 2013 it was called "Cactus Automorphisms". Here is an example of cactus that was used in those problems:



For four years judges had to generate test files for cactuses with thousands of vertices. Of course, a number of test generators of ever-increasing complexity were built, culminating with a domain-specific language called *CGL* — Cactus Generator Language. CGL can compactly define a big cactus for purposes of a test. In this problem you have to parse a simplified version of this language, which we call *SCGL* — Simple Cactus Generator Language, and output a resulting cactus.

A cactus has to be output by listing the minimal set of edge-distinct paths that cover the whole graph.

The syntax of SCGL cactus definition is represented by the *graph* non-terminal in the grammar that is given below using the Extended Backus-Naur Form:

graph	=	"`c`"
	|	"`c(`" *list* "`)`"
	|	"`loop(`" *list* "`)`"
	|	"`t(`" *list* "`)`"
list	=	*graph* { "`,`" *graph* }
	|	(*number* | *range* | *variable*) ["`,`" *graph*]
number	=	*nzdig* { "`0`" | *nzdig* }
nzdig	=	"`1`" | "`2`" | ... | "`8`" | "`9`"
range	=	"`range(`" *variable* "`,`" *numvar* "`,`" *numvar* "`)`"
variable	=	"`A`" | "`B`" | ... | "`Y`" | "`Z`"
numvar	=	*number* | *variable*

A *graph* production rule denotes a graph with two labeled vertices — the *first* and the *last*. Graphs definition rules have the following semantics:

- The basic building block c denotes a graph with just two vertices (one is the first and the other one is the last) and one edge.
- The $c(\sigma)$ rule connects a specified list of graphs σ from left to right into a chain, merging the last vertex of the first graph in the list with the first vertex of the second graph in the list, the last vertex of the second graph with the first of the third one, and so on. The resulting graph's first vertex is the first vertex of the first graph in the list, and the resulting graph's last vertex is the last vertex of the last graph in the list.
- The $loop(\sigma)$ rule connects a specified list of graphs σ from left to right, merging the last vertex of the first graph in the list with the first vertex of the second graph in the list, and so on like in $c(\sigma)$, while the last vertex of the last graph in the list is merged with the first vertex of the first graph in the list to form a loop. The resulting graph's first and last vertices are the first and the last vertices of the first graph in the list. Loop can be applied only to lists with more than one graph.
- The $t(\sigma)$ rule connects a specified list of graphs σ, merging their first vertices. The resulting graph's first and last vertices are the first and the last vertices of the first graph in the list.

The *list* of graphs is either specified explicitly, by a comma-separated list, or using a *list repetition* with a number, a range, or a variable, optionally followed by a comma and a graph. When a graph is not explicitly specified in a list repetition, then the given graph is assumed to be c.

The simplest list repetition is defined using a *number* non-terminal. It denotes a list of graphs with the specified integer number of copies of the given graph.

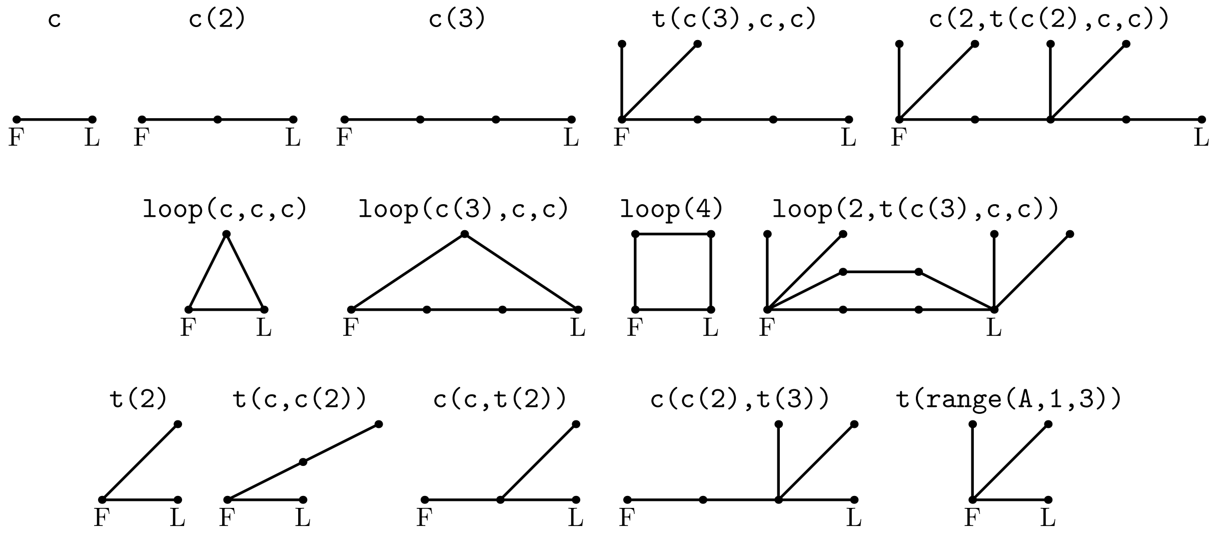
A *range* list repetition is defined by $range(\nu, \alpha, \beta)$ rule which has three components — a variable ν, and numbers α and β. If ξ character sequence is a *graph*, then $c|loop|t(range(\nu, \alpha, \beta), \xi)$ are called *range-enabled* rules and the variable ν is called a *bound variable* in ξ. In the context of a range-enabled rule, ξ is repeated $|\beta - \alpha| + 1$ times to form a list. Every occurrence of variable ν in ξ is replaced by consecutive integer numbers between α and β inclusive in ascending order. That produces a list of $|\beta - \alpha| + 1$ graphs, which are then connected according the specification of the corresponding range-enabled rule. The α and β themselves might refer to variables that are bound in the outer range-enabled rule.

In a *well-formed* graph:

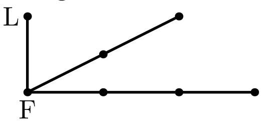
- each *variable* non-terminal (a letter from `A` to `Z`) occurs at most once as ν in $range(\nu, \alpha, \beta)$ rules;
- all other occurrences of *variable* non-terminal that are allowed by the grammar are bound.

Note, that if a character sequence ξ is a *graph*, then ξ, $c(\xi)$, $c(1, \xi)$, $t(\xi)$, and $t(1, \xi)$ all denote the same graph. On the other hand, neither $loop(\xi)$ nor $loop(1, \xi)$ are allowed.

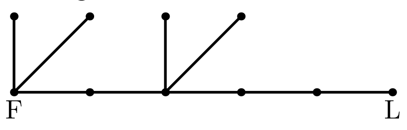
The following examples illustrate these basic rules. The graphs have their first and last vertices marked with letters `F` and `L` correspondingly.



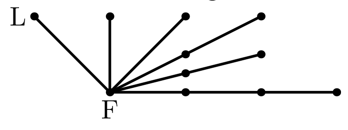
`t(range(A,1,3),c(A))`



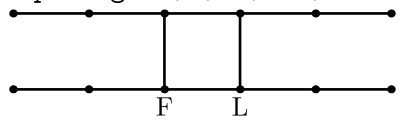
`c(range(N,2,3),t(c(N),c,c))`



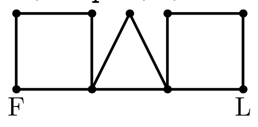
`t(range(X,1,3),t(range(Y,3,X),c(X)))`



`loop(range(N,1,4),t(c,c(2)))`



`c(range(I,1,3),loop(c,c,c(range(J,I,2))))`



Input

The input file contains a single line with a well-formed cactus definition in SCGL. While the syntax and semantics of SCGL themselves do not guarantee that the resulting graph is a cactus, the input file for this problem always defines a cactus — every edge belongs to at most one simple cycle and there are no multiple edges between vertices. For example, neither `loop(3,loop(3))` nor `loop(2)` are possible.

The line in the input file is at most 1000 characters long and defines a cactus with at most 50 000 vertices. Integer numbers represented by *number* non-terminals do not exceed 50 000.

Output

The first line of the output file must contain two integer numbers n and m. Here n is the number of vertices in the graph. Vertices are numbered from 1 to n, where 1 is the number of the *first* vertex of the graph and n is the number of the *last* vertex of the graph. The other vertices can be numbered arbitrarily. Edges of the graph are represented by a set of edge-distinct paths, where m is the minimal number of such paths.

Each of the following m lines must contain a path in the graph. A path starts with an integer number k_i ($k_i \geq 2$) followed by k_i integers from 1 to n. These k_i integers represent vertices of a path. A path can go to the same vertex multiple times, but every edge must be traversed exactly once in the whole output file.

Sample input and output

cactus.in
`c(c,t(loop(3),c(c,loop(6))),loop(c,c,t(c,loop(4))))`
cactus.out
`15 1`
`19 1 2 9 10 11 12 13 10 15 9 14 2 3 4 5 6 7 8 3`

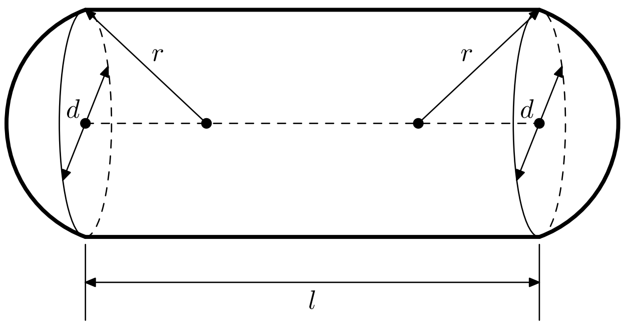
cactus.in	cactus.out
`c`	`2 1`
	`2 1 2`
`c(2)`	`3 1`
	`3 1 2 3`
`c(3)`	`4 1`
	`4 1 2 3 4`
`t(c(3),c,c)`	`6 2`
	`2 1 2`
	`5 3 1 4 5 6`
`c(2,t(c(2),c,c))`	`9 3`
	`3 2 1 3`
	`3 4 5 6`
	`5 1 7 5 8 9`

Problem D. Damage Assessment

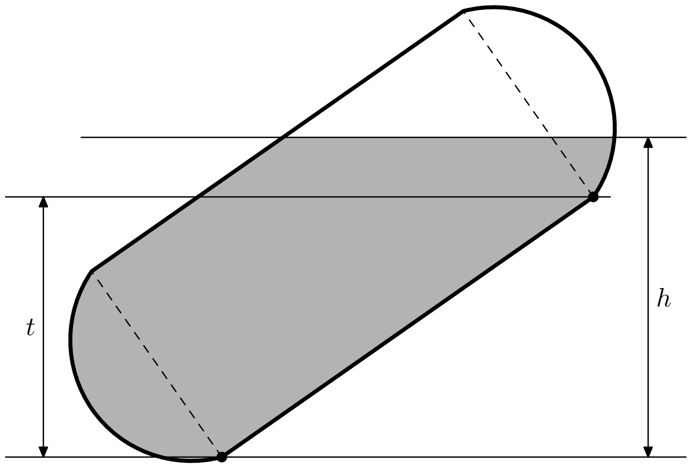
Input file:	`damage.in`
Output file:	`damage.out`

A tank car that transports gasoline via rail road has a shape of cylinder with two spherical caps at the sides. The cylinder has a diameter d and a length l. The spherical caps have a radius r ($2r \ge d$). There was the rail road accident and the tank car had derailed. It now lies on the ground and some of the stored gasoline had flown out. The damage assessment must be performed. The location of the tank on the ground was established by measuring its tilt as the height difference t from the bottom points of the *cylinder* on its left and right sides ($0 \le t \le l$). The level of gasoline in the tank was established by measuring the height difference h from the bottom point of the *cylinder* and the top level of gasoline. For the purpose of this problem, the top level of gasoline always intersects the cylinder part of the tank ($0 \le h \le t + d\sqrt{1-(t/l)^2}$).

Your task is to figure out how much gasoline was left in the tank.



Tank schematic drawing



Tank location and gasoline level

Input

The input file consists of a single line with five integer numbers — d, l, r, t and h, which represent the diameter and the length of the tank's cylinder part, the radius of its spherical caps, tilt and gasoline level measurements. They are all expressed in *millimeters* (1 millimeter = 10^{-3} meters), they satisfy all constraints expressed in the problem statement and $d, l \ge 100$, $d, l, r \le 10\,000$.

Output

Write a single real number to the output file — the volume of gasoline in the tank in *liters* (1 liter = 10^{-3} cubic meters). The absolute error of the answer must not exceed 0.1 liters.

Sample input and output

`damage.in`	`damage.out`
`3000 6000 1600 0 3000`	`50974.56`
`3000 6000 1600 3441 4228`	`40728.90`

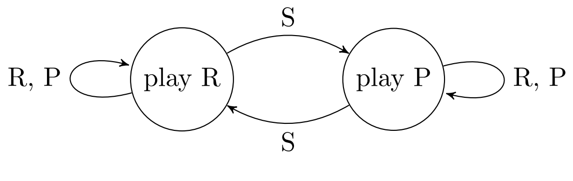
Problem E. Epic Win!

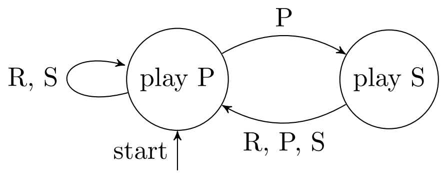
Input file:	`epic.in`
Output file:	`epic.out`

A game of rock-paper-scissors is played by two players who simultaneously show out their moves: *Rock*, *Paper*, or *Scissors*. If their moves are the same, it's a draw. Otherwise, *Rock* beats *Scissors*, *Paper* beats *Rock*, and *Scissors* beat *Paper*.

The described procedure can be repeated many times. In this problem, two Finite State Machines (FSMs) will compete in a series of rounds. (Formally speaking, by FSMs we mean Moore machines in this problem.)

An FSM for playing rock-paper-scissors has finitely many states. Each state is described by the following: what move the FSM will make in the upcoming round, and what will be the new state in case of its opponent playing *Rock*, *Paper*, and *Scissors*.





Fortunately, you know your opponent FSM — the entire scheme except for one thing: you *do not know* the initial state of that FSM.

Your task is to design your own FSM to fight the given one. Your FSM must beat the opponent in at least 99% of the first 1 billion rounds. That's what we call an epic win!

Input

The input file contains a description of the opponent FSM in the following format.

The first line contains an integer n ($1 \le n \le 100$) — the number of states in the FSM. States are numbered from 1 to n. Each of the following n lines contains a description of the state: a character c_i denoting the move made by FSM and integers r_i, p_i, s_i denoting the next state in case of seeing *Rock*, *Paper*, or *Scissors* respectively (c_i can be "R", "P", or "S"; $1 \le r_i, p_i, s_i \le n$).

Output

Write to the output the description of your FSM in the same format.

The initial state of your FSM is the first state.

The number of states may not exceed 50 000.

Sample input and output

epic.in	epic.out
2 R 1 1 2 P 2 2 1	2 P 1 2 1 S 1 1 1

The picture in the problem statement illustrates the opponent FSM given in the above sample input and a possible solution of yours given in the sample output.

Opponent FSM keeps playing *Rock* or *Paper* (depending on its initial state) until it sees *Scissors* — seeing *Scissors* triggers a change in its behaviour.

One way to beat such FSM is to play *Paper*. If your opponent keeps playing *Rock*, just continue playing *Paper* and thus win. If the opponent FSM is playing *Paper*, trigger it to playing *Rock* by playing *Scissors* once, and then it'll keep playing *Rock* and you'll keep beating it with your *Paper*.

Problem F. Filter

Input file:	`filter.in`
Output file:	`filter.out`

You are working on a new high-performance database engine — Instant Compression and Processing Codec (ICPC). ICPC stores user activity records. Each user activity record has an integer *user identifier*. The records are stored in a number of data files. Each data file is compressed and can contain records from multiple users, however ICPC has to process queries that look for a specific subsets of users. In order to do so, there has to be a way to quickly determine which data files may contain records for a specific user before attempting to decompress them, which may be a long and CPU-consuming process.

ICPC uses an algorithm called *Bloom Filter*. The way it is implemented in ICPC is described below. For each ICPC database the following integer parameters are chosen:

- m is the number of bits in the filter;
- f is the number of hash functions in the filter;
- a_i are the parameters for hash functions for $0 \le i < f$.

A value of the bloom filter is computed for each data file. The data file's bloom filter is a vector of m bits. A bit number j $(0 \le j < m)$ is set to one if and only if there is a record in this data file for some user identifier u_k, such that for some hash function i $(0 \le i < f)$ the following equality holds:

$$j = (u_k \cdot a_i) \bmod m \tag{1}$$

Your task is to implement ICPC filtering logic. You are given filter parameters and values for a number of data files and a set of user identifiers. Your task is determine which data files may contain record with at least one user identifier from the specified set. A data file may contain a record with a user identifier u_k if and only if for all i $(0 \le i < f)$ all the bits j given by equality (1) in its filter value are set to one.

Input

The first line of the input file contains filter parameters — integer numbers m, f, and a_i for $0 \le i < f$ $(1 \le m \le 1000,\ 1 \le f \le 100,\ 1 \le a_i < 2^{31})$.

The second line of the input file contains an integer n — the number of data files $(1 \le n \le 1000)$. Each of the following n lines contains bloom filter value of the corresponding file in hexadecimal form. Each value is represented by a string of $\lceil m/4 \rceil$ hexadecimal digits (one of `0123456789abcdef`). The first digit of the string represents bits 0–3 of the value (stored in order from the least significant bit of a hexadecimal digit to the most significant bit), the second digit — bits 4–7, the third — 8–11, etc. When $m \bmod 4 \ne 0$, then the last hexadecimal digit represents the last $m \bmod 4$ bits of the value in its least significant bits.

The following line of the input file contains an integer q — the number of user identifiers in a query $(1 \le q \le 1000)$, followed by q integers u_k — the set of distinct user identifiers in the query $(1 \le u_k < 2^{31})$.

Output

Write a line with the integer number s to the output file — the number of data files that may contain a record with at least one user identifier from the specified set, followed by s numbers d_t $(0 \le d_t < n)$ — the 0-based numbers of the corresponding data files in ascending order.

Sample input and output

filter.in	filter.out

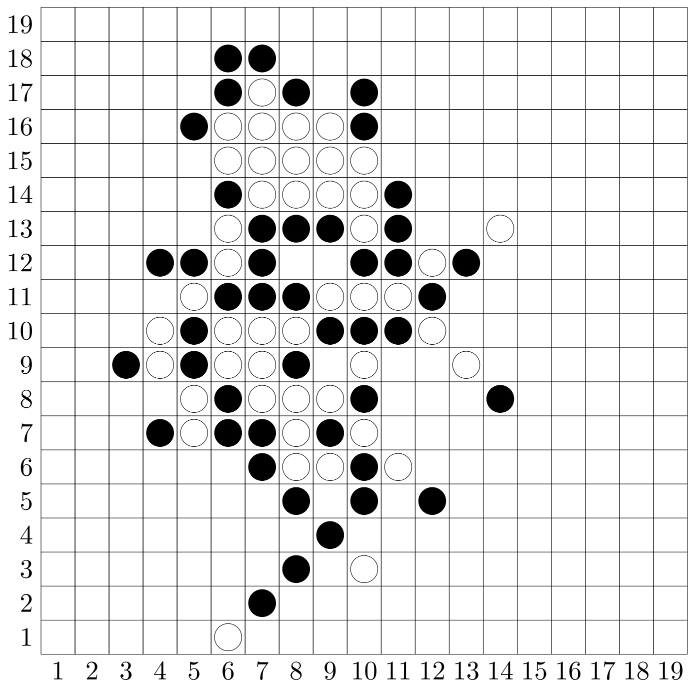
```
23 4 3 5 7 11
3
effde7
c07902
0800c1
3 2 4 6
```
	`2 0 2`

Problem G. Gomoku

Input file:	`standard input`
Output file:	`standard output`

This is an interactive problem.

Gomoku is a two-player game on a two-dimensional grid. Each cell of the grid can be either empty, contain the first player's mark (black), or contain the second player's mark (white), but not both. Initially the entire grid is empty. Two players make alternating moves, starting with the first player. At each move, a player can put her mark into exactly one empty cell. The first player to have her five adjacent marks in a single row wins. The winning row can be either vertical, horizontal or diagonal.



Position where the second player (white marks) had won.

The players use a 19×19 grid in this problem. If the entire grid gets filled with marks but no player has won, the game is declared a draw.

The first player uses the following strategy: as the first move, she puts her mark into the center cell of the grid. At every other move, she picks such a move that maximizes the score of the resulting position.

In order to find the score of a position, the first player considers all possible places where the winning combination might eventually form — in other words, all horizonal, vertical and diagonal rows of five consecutive cells on the board (of course, they may overlap each other). If such a row contains both the first player's marks and the second player's marks, it is disregarded. If such a row contains no marks, it is disregarded as well. For each row with exactly k $(1 \le k \le 5)$ marks of the first player and no marks of the second player, add 50^{2k-1} to the score of the position. For each row with exactly k marks of the second player and no marks of the first player, subtract 50^{2k} from the score of the position. Finally, add a random integer number between 0 and $50^2 - 1$ to the score. This random number is chosen uniformly.

In case when several moves of the first player have equal scores (such ties are quite rare because of the random addition mentioned above), the first player picks the one with the smallest x-coordinate, and in case of equal x-coordinates, the one with the smallest y-coordinate.

Your task is to write a program that plays the second player and beats this strategy.

Your program will play 100 games against the strategy described above, with different seeds of random generator. Your program must win all these games.

Interaction protocol

On each step, your program must:

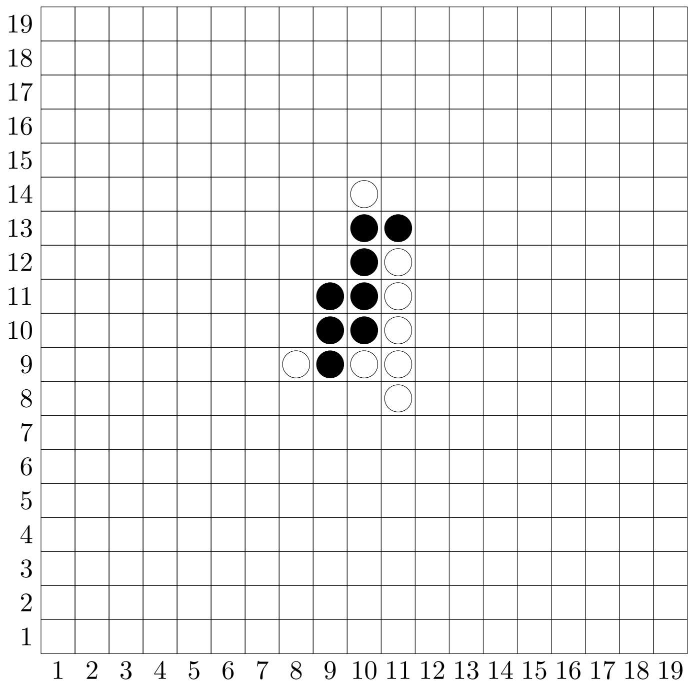
1. Read numbers x and y from the input.
2. If both these numbers are equal to -1 then the game is over and your program must exit.
3. Otherwise these numbers are the coordinates of the first player's move ($1 \le x, y \le 19$).
4. Print the coordinates of the move of the second player, followed by line end. Don't forget to flush the output buffer.

Sample input and output

In the example below the first player **does not** use the strategy from the problem statement. The example is given only to illustrate the interaction format.

standard input	standard output

10 10
10 11
10 12
10 13
9 10
9 11
9 9
11 13
-1 -1
	11 10
	11 11
	10 9
	10 14
	8 9
	11 9
	11 12
	11 8



Final position from the example.

Note

There are many variations of Gomoku rules in the world. Please only consider the rules described in this problem statement.

Problem H. Hidden Maze

Input file:	`hidden.in`
Output file:	`hidden.out`

Helen and Henry are fans of the TV show "Hidden Maze", which is very popular in Hiddenland. During this show two participants (usually a married couple) are rushing through the maze consisting of n halls connected by some tunnels. Each tunnel connects two different halls and there can't be more than one tunnel connecting any pair of halls.

In the beginning of the show, two participants are placed in two different halls. Then they need to move very quickly to meet each other before the time runs out. To pass through each tunnel, the participant needs to find a clue which is some positive integer number written on a small piece of paper.

If the participants finally meet in some tunnel before the time runs out, and successfully find a clue contained in the tunnel where they met, they are considered winners. The value of their prize is calculated by sorting all the clues found by them and taking the median value. The game is always set up in such a way, that the number of clues they find is odd.

Helen and Henry saw a large number of episodes of the show, and now they understand a lot about the mechanics of it. They noticed that the maze doesn't change between episodes, and they drew a complete map of the maze. Shortly after, Helen and Henry have discovered that the maze is built in such a manner that if you visit any tunnel at most once, then there is exactly one path between any two halls.

Helen and Henry have been wondering how this great maze is created, and not so long time ago they have seen an interview with Hillary, who worked for the company which had built the maze. Hillary has told that to make the show fair, the maze had been created using the following randomized algorithm:

1. Pick the number of halls n. Build n halls enumerated from 1 to n.
2. Choose at random two integers i and j, each of them uniformly distributed between 1 and n.
3. If halls i and j are the same or are already connected with some path of tunnels, then go to step 2.
4. Build the tunnel between i and j. If there is a path of tunnels from any hall to any other one, stop the process, otherwise go to step 2.

Helen and Henry have also noticed that each tunnel contains exactly one clue and its value never changes from episode to episode. However, they don't know what algorithm was used to generate clue values. Helen and Henry have written on their map the value of the clue for each tunnel.

It always takes 1 minute to find a clue and to run through the tunnel from one hall to another. It takes half a minute to run from the hall to the center of the tunnel when the participants meet in the center of the tunnel at the end. The time given to participants is only enough to meet each other if they act optimally, that is they just run to each other via the shortest possible path, never make mistakes when finding clues, and never turn into any other tunnels that do not belong to the shortest path. To make the participants meet in the center of some tunnel, they are placed in the beginning of the show in such a way that the length of the shortest path between the halls where they are placed is odd.

Helen and Henry want to participate in the show. They know the maze by heart and they are pretty sure that they will succeed in moving optimally to each other and finding all clues in time. Provided that the pair of initial halls is selected uniformly from all pairs with an odd-length shortest path between them, they need to know the expected value of the prize they win. Your task is to help them find this expected value.

Input

The first line of the input contains an integer number n ($2 \le n \le 30\,000$) — the number of halls. Each of the following $n - 1$ lines contains three integers: u_i, v_i, c_i ($1 \le u_i, v_i \le n$, $1 \le c_i \le 10^6$), describing the i-th tunnel connecting the halls u_i and v_i, containing the clue with the value c_i. The maze is always created by the randomized algorithm that is specified in the problem statement.

Output

Write to the output file a single real number — the expected value of the prize. The absolute or relative error of the answer must not exceed 10^{-9}.

Sample input and output

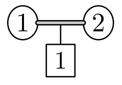
`hidden.in`	`hidden.out`

`2`
`2 1 1`
	`1`

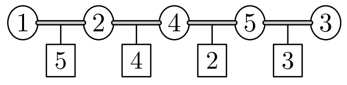
`5`
`2 4 4`
`1 2 5`
`5 4 2`
`5 3 3`
	`3.50`

`5`
`4 1 2`
`5 3 2`
`4 2 3`
`5 4 7`
	`3.1666666667`

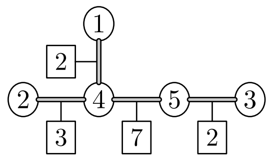
You can look at the pictures of mazes from the sample inputs below. The halls are shown as circles, the tunnels are gray lines and the clues are squares.



Sample input 1



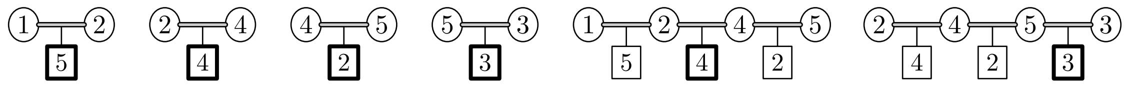
Sample input 2



Sample input 3

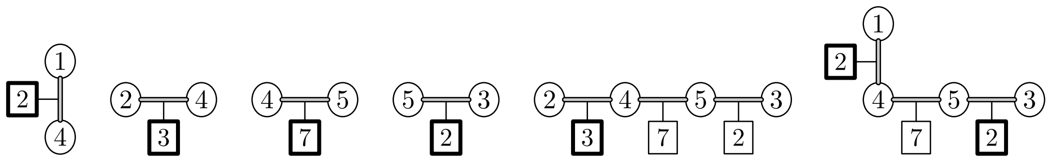
There are six possible pairs of initial halls in the second sample maze. They are shown on the picture below, the clue used to determine the prize is marked with bold frame. The expected value is

$$(4 + 5 + 2 + 3 + 4 + 3)/6 = 21/6 = 3.5.$$



All six possible initial pairs for the third maze are shown on the picture below, the expected value is calculated as

$$(2 + 3 + 7 + 2 + 3 + 2)/6 = 19/6 = 3.16666666666\ldots$$



Note that in this case there are two clues with number 2 for the pair $(1, 3)$, which is shown on the rightmost picture. Any one of them can be selected as the median (they are both marked with bold frame) but this obviously does not change the value of the prize.

Problem I. Improvements

Input file:	`improvements.in`
Output file:	`improvements.out`

Son Halo owns n spaceships numbered from 1 to n and a space station. They are initially placed on one line with the space station so the spaceship i is positioned x_i meters from the station and all ships are on the same side from the station ($x_i > 0$). All x_i are distinct. Station is considered to have number 0 and x_0 is considered to be equal to 0.

Every two spaceships with consequent numbers are connected by a rope, and the first one is connected to the station. The rope number i (for $1 \le i \le n$) connects ships i and $i-1$. Note, that the rope number 1 connects the first ship to the station.

Son Halo considers that the rope i and the rope j intersect when the segments $[x_i^{min}, x_i^{max}]$ and $[x_j^{min}, x_j^{max}]$ have common internal point but neither one of them is completely contained in the other, where $x_k^{min} = \min(x_{k-1}, x_k)$, $x_k^{max} = \max(x_{k-1}, x_k)$. That is:

$$\left[\begin{array}{l} x_i^{min} < x_j^{min} \ \& \ x_j^{min} < x_i^{max} \ \& \ x_i^{max} < x_j^{max} \\ x_j^{min} < x_i^{min} \ \& \ x_i^{min} < x_j^{max} \ \& \ x_j^{max} < x_i^{max} \end{array}\right.$$

Son Halo wants to rearrange spaceships in such a way, that there are no rope intersections. Because he is lazy, he wants to rearrange the ships in such a way, that the total number of ships that remain at their original position x_i is maximal. All the ships must stay on the same side of the station and at different positions x_i after rearrangement. However, ships can occupy any real positions x_i after rearrangement.

Your task is to figure out what is the maximal number of ships that can remain at their initial positions.

Input

The first line of the input file contains n ($1 \le n \le 200\,000$) — the number of ships. The following line contains n distinct integers x_i ($1 \le x_i \le n$) — the initial positions of the spaceships.

Output

The output file must contain one integer — the maximal number of ships that can remain at their initial positions in the solution of this problem.

Sample input and output

`improvements.in`	`improvements.out`

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

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

Note

In the first sample Son Halo can move the second spaceship in the position between the first and the third to solve the problem while keeping 3 other ships at their initial positions.

In the second sample there are no rope intersections, so all 4 ships can be left at their initial positions.

Problem J. Jokewithpermutation

Input file:	`joke.in`
Output file:	`joke.out`

Joey had saved a permutation of integers from 1 to n in a text file. All the numbers were written as decimal numbers without leading spaces.

Then Joe made a practical joke on her: he removed all the spaces in the file.

Help Joey to restore the original permutation after the Joe's joke!

Input

The input file contains a single line with a single string — the Joey's permutation without spaces.

The Joey's permutation had at least 1 and at most 50 numbers.

Output

Write a line to the output file with the restored permutation. Don't forget the spaces!

If there are several possible original permutations, write any one of them.

Sample input and output

joke.in	joke.out
`4111109876532`	`4 1 11 10 9 8 7 6 5 3 2`

Problem K. Knockout Racing

Input file:	`knockout.in`
Output file:	`knockout.out`

The races became more popular than ever at Pandora planet. But these races are quite unusual. There are n cars participating in a race on the long straight track. Each car moves with a speed of 1 meter per second. Track has coordinates in meters.

The car number i moves between two points on the track with coordinates a_i and b_i starting at the second 0 in the point a_i. The car moves from a_i to b_i, then from b_i to a_i, then from a_i to b_i again, and so on.

Handsome Mike wants to knock some cars out of the race using dynamite. Thus he has m questions. The question number j is: what is the number of cars in the coordinates between x_j and y_j inclusive after t_j seconds from the start?

Your task is to answer Mike's questions.

Input

The first line of the input file contains two integers n and m ($1 \le n, m \le 1000$) — the number of cars in the race and the number of questions.

Each of the following n lines contains a description of the car: two integers a_i and b_i ($0 \le a_i, b_i \le 10^9$, $a_i \ne b_i$) — the coordinates of the two points between which the car i moves.

Each of the following m lines contains a description of the question: three integers x_j, y_j, and t_j ($0 \le x_j \le y_j \le 10^9$, $0 \le t_j \le 10^9$) — the coordinate range and the time for the question j.

Output

Write m lines to the output file. Each line must contain one integer — the answer to the corresponding question in order they are given in the input file.

Sample input and output

knockout.in	knockout.out

5 5
0 1
0 2
2 3
3 5
4 5
0 5 0
0 1 2
0 2 1
2 5 2
2 5 3
	5
	1
	2
	4
	3