Problem A. Bijection

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	512 mebibytes

Consider paths on a plane from $(0, 0)$ to (n, n) consisting of unit steps to the right ("`R`") and upwards ("`U`"). It is known that the number of distinct such paths is the binomial coefficient

$$\text{choose}(2n, n) = \frac{(2n)!}{n! \cdot n!}.$$

For example, when $n = 2$, there are six such paths: "`RRUU`", "`RURU`", "`RUUR`", "`URRU`", "`URUR`", "`UURR`".

A string U is a regular bracket sequence if it is an empty string, or a string of the form "`(`V`)`", or a concatenation of two strings of the form "VW", where V and W are regular bracket sequences. Consider regular bracket sequences containing n pairs of brackets. It is known that the number of distinct such sequences is the Catalan number which can be calculated, in particular, as follows:

$$C_n = \frac{1}{n+1} \cdot \text{choose}(2n, n).$$

For example, when $n = 2$, there are two such sequences: "`(())`", "`()()`".

Construct any bijection that reflects this fact. More specifically, given a path of n steps to the right and n steps upwards, construct a regular bracket sequence containing n pairs of brackets, and additionally memorize an integer k from 0 to n inclusive. Afterwards, given the sequence and the integer k, restore the original path.

Interaction Protocol

In this problem, your solution will be run twice on each test.

During the first run, the solution encodes the path. The first line contains the word "`path`". The second line contains an integer n: half of the path length ($1 \le n \le 300$). The third line contains a path of $2n$ steps: n letters "`R`" and n letters "`U`" in some order.

On the first line, print any regular bracket sequence containing n "`(`" characters and n "`)`" characters. On the second line, print any integer k ($0 \le k \le n$).

During the second run, the solution restores the path. The first line contains the word "`brackets`". The second line contains an integer n, the same as during the first run: half of the bracket sequence length ($1 \le n \le 300$). The third line contains a regular bracket sequence containing n "`(`" characters and n "`)`" characters. The fourth line contains an integer k ($0 \le k \le n$). The sequence and the integer are the ones printed during the first run.

On the first line, print the restored initial path: n letters "`R`" and n letters "`U`" in the same order as in the input during the first run.

During each run, each line of input including the last one is terminated by a newline.

Examples

On each test, the input during the second run depends on the solution's output during the first run.

Two runs of some solution on the first test are shown below.

standard input	standard output
path 2 RRUU	(()) 0
brackets 2 (()) 0	RRUU

Two runs of some solution on the second test are shown below.

standard input	standard output
path 3 RUURRU	(())() 3
brackets 3 (())() 3	RUURRU

Problem B. Rectangle Tree

Input file:	*standard input*
Output file:	*standard output*
Time limit:	6 seconds
Memory limit:	512 mebibytes

Mr. Peanutbutter has recently discovered a nice $n \times n$ field covered with various crops. Diane told Mr. Peanutbutter that for generations this particular field is planted with crops using tree-like rectangle method. While Mr. Peanutbutter got distracted by a bird, Diane continued.

A *combinatorial rectangle* in the field is a subset of squares of the field of the form $A \times B$ where A and B are subsets of the set $\{0, \ldots, n-1\}$.

A *rectangle tree* is a rooted binary tree with k vertices with the following properties. Each vertex v of the tree is labeled with a combinatorial rectangle $r(v) \subseteq \{0, \ldots, n-1\} \times \{0, \ldots, n-1\}$. If s is an inner node of the tree, and c_1 and c_2 are its direct descendants, then their combinatorial rectangles form a partition of $r(s)$: formally, $r(s) = r(c_1) \cup r(c_2)$ and $r(c_1) \cap r(c_2) = \varnothing$. A node cannot have only one direct descendant.

Let $\mathrm{Crop}(x, y)$ be the crop that grows on the square $(x, y) \in \{0, \ldots, n-1\} \times \{0, \ldots, n-1\}$ of the field. A rectangle tree T with the root Root *computes* the crop types of the field if $r(\mathrm{Root}) = \{0, \ldots, n-1\} \times \{0, \ldots, n-1\}$ and for each leaf ℓ, the combinatorial rectangle $r(\ell)$ has exactly one type of crop growing on it: that is, for any two $(x, y), (x', y') \in r(\ell)$, we have $\mathrm{Crop}(x, y) = \mathrm{Crop}(x', y')$.

The *depth* of tree T is the largest distance between the root of T and a leaf of T. Here, distance stands for the number of edges in the shortest path between the vertices.

The *size* of tree T is the number of vertices in it.

You are given a rectangle tree T computing crop types Crop. Let the size of T be S. Construct another rectangle tree T' computing Crop such that its depth is at most $3 \log_2 S$ and its size is at most $5S$.

Input

The first line contains a single integer n, the size of the field ($1 \le n \le 1000$). Each of the next n lines contain n integers describing the types of the crops. The j-th integer in the i-th row is the type of crop in the square (i, j). All types are positive integers not exceeding 10^7.

The next line contains a single integer S, the size of the rectangle tree ($1 \le S \le 10\,000$, $S \cdot n \le 10^6$). The i-th of the next S lines contains the description of the i-th vertex of the tree. It contains several space-separated integers: p, m_1, m_2, a_1, …, a_{m_1}, b_1, …, b_{m_2}. Here, $p \in \{0, 1, \ldots, S-1\}$ is the number of the parent of vertex i (if i is the root, then $p = i$), and the combinatorial rectangle corresponding to this vertex is $r(i) = \{a_1, \ldots, a_{m_1}\} \times \{b_1, \ldots, b_{m_2}\}$.

It is guaranteed that, if ℓ is a leaf of the tree, then all types of crops in $r(\ell)$ are the same. Additionally, for each inner vertex v with direct descendants c_1 and c_2, the rectangles $r(c_1)$ and $r(c_2)$ form a partition of $r(v)$: $r(v) = r(c_1) \cup r(c_2)$ and $r(c_1) \cap r(c_2) = \varnothing$. Finally, if i is the root, then $r(i) = \{0, \ldots, n-1\} \times \{0, \ldots, n-1\}$.

Output

Print a tree of depth at most $3 \log_2 S$ and of size at most $5S$ such that it is also a rectangle tree that computes the given crops. The tree should be printed in the same format as the one given in the input.

Example

standard input	standard output
3 1 1 2 1 1 2 2 2 2 5 0 3 3 0 1 2 0 1 2 0 3 2 0 1 2 0 1 0 3 1 0 1 2 2 1 2 2 0 1 0 1 1 1 2 2 0 1	7 0 3 3 0 1 2 0 1 2 0 1 3 2 0 1 2 1 1 2 2 0 1 1 1 1 2 2 0 2 3 0 1 0 1 2 4 2 1 0 1 2 4 2 2 0 1 0 1

Problem C. Integer Cow

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	512 mebibytes

A cow stands on an infinite plane in integer point (x_0, y_0). Grass grows in a disk centered in integer point (x_c, y_c) with integer radius r, and also on the disk border.

The cow can perform the following command an arbitrary number of times: move from its current integer point (x_1, y_1) to integer point (x_2, y_2). The time to perform such command is equal to the Euclidean distance between the points. The two points may coincide.

Find a sequence of commands which will bring the cow to an integer point with grass in minimum possible time.

Input

The first line contains an integer t, the number of test cases ($1 \le t \le 100$). The next t lines contain test cases, one per line. Each test case is defined by five integers x_c, y_c, r, x_0, y_0: the coordinates of the grass disk's center, its radius, and the initial coordinates of the cow ($-10^9 \le x_c, y_c, x_0, y_0 \le 10^9$, $1 \le r \le 10^9$).

Output

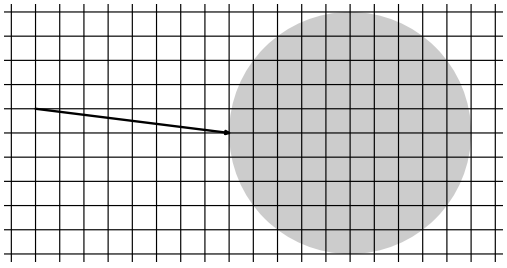
For each test case, print two lines. On the first one, print an integer k, the number of commands ($0 \le k \le 1\,000\,000$). On the second line, print $2(k + 1)$ integers, the cow's path: x_0 y_0 ... x_k y_k. If there are several optimal sequences, print any one of them.

Example

standard input	standard output
3 1 2 1 1 2 3 2 5 -10 3 0 0 1 10 0	0 1 2 1 -10 3 -2 2 3 10 0 5 0 5 0 1 0

Explanation

The picture corresponds to the second test case.



Problem D. Lost in Transfer

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	512 mebibytes

Dima has a set of n numbers. Dima wants to transmit this set to Katya. He takes numbers from the set, one by one, in any order he pleases, and enters them into the transmitter.

Katya receives numbers from the receiver in the order in which Dima enters them. However, the transmission channel is not ideal, so one of the numbers could have been lost in transfer. Nevertheless, it is very important for Katya to exactly reconstruct the set that Dima wanted to transmit.

Help Dima and Katya agree in advance how they transmit numbers so that Katya can always reconstruct Dima's set, even if one of its elements was lost in transfer.

Interaction Protocol

In this problem, your solution will be run twice on each test. Each test consists of separate test cases. Both in input and in output, adjacent numbers on a line are separated by spaces.

During the first run, the solution transmits sets as Dima. The first line contains the word "`transmit`". The second line contains an integer t, the number of test cases ($1 \le t \le 1000$). Each of the next t lines describes a single test case. Such line starts with an integer n, the number of elements in the set ($20 \le n \le 100$). Then follow n pairwise distinct integers $a_1, a_2, \ldots, a_n$, the elements of the set ($1 \le a_i \le 500$).

Print t lines, one for each test case. On each line, print the respective integers $a_1, a_2, \ldots, a_n$, each of them exactly once, in any order you like.

During the second run, the solution reconstructs sets as Katya. The first line contains the word "`recover`". The second line contains an integer t, the number of test cases, same as during the first run ($1 \le t \le 1000$). Each of the next t lines describes a single test case. Such line starts with an integer m, the number of integers received by Katya ($19 \le m \le 100$). Then follow m pairwise distinct integers $b_1, b_2, \ldots, b_m$, the integers received by Katya themselves. These are the integers sent by Dima during the first run, given in the order of transmission. However, one of the integers might be omitted (and then m is one less than the respective n during the first run).

Print t lines, one for each test case. On each line, print the integers $a_1, a_2, \ldots, a_n$ from the respective test case, each of them exactly once, in any order you like.

Note

In this problem, the tests are generated using a pseudorandom number generator. In each test, the number of test cases t and the size of the set n in each test case are chosen in advance. After that, each set of size n is selected randomly with equal probability among all possible sets of size n consisting of integers from 1 to 500. The elements of the set are given in random order.

Additionally, in each test case, it is decided in advance which number will be lost in transfer. For a set of size n, a position p is selected randomly with equal probability among all possible integers from 1 to $n + 1$. If $p \le n$, it means that p-th printed number will be lost. When $p = n + 1$, every number will be transmitted successfully.

Example

On each test, the input during the second run depends on the solution's output during the first run. Two runs of some solution on the first test are shown below.

standard input
transmit 2 20 97 388 459 467 32 99 98 296 403 325 330 271 87 333 378 267 405 58 426 374 20 125 481 451 150 495 136 444 192 118 26 68 281 120 61 494 339 86 292 100 32
standard output
405 97 87 58 374 98 271 296 330 267 99 32 378 333 325 467 388 403 459 426 494 68 481 61 120 125 281 444 150 86 339 26 32 118 451 136 495 100 292 192

standard input
recover 2 19 97 87 58 374 98 271 296 330 267 99 32 378 333 325 467 388 403 459 426 20 494 68 481 61 120 125 281 444 150 86 339 26 32 118 451 136 495 100 292 192
standard output
97 87 58 374 98 271 296 330 267 99 32 378 333 325 467 388 403 459 426 405 494 68 481 61 120 125 281 444 150 86 339 26 32 118 451 136 495 100 292 192

Problem E. Maze with a Hint

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	512 mebibytes

This is an interactive problem.

Thor had boasted to dwarves that he can go through any maze without a single drop of sorcery, using only a small torch. The dwarves decided to give Thor a trial. They are fast and skilled builders, and their new maze will be large and tricky. If an adventurer walks inside for too long, the torch will go out, and the dwarves will laugh at the Ace. After taking a look on the new construction, Thor decided that he has to win, no matter the price, and asked Loki for help.

Loki is a resourceful trickster, and he will be able to obtain the map of the maze as soon as it is finished. But he won't be able to just give the map to Thor: the dwarves will surely see through such deception. Thor can only get a short hint from Loki...

Help Thor and Loki to prepare for passing the hint, so that Thor would then be able to pass the maze before the torch goes out.

Maze structure

The maze which is being built by dwarves can be drawn as a square board consisting of $n \times n$ squares. Between each two squares adjacent horizontally or vertically, there is either a passage or a wall. Additionally, the whole maze is surrounded by a wall. The entrance is in the bottom left square, and the exit is in the top right square.

The dwarves build walls as follows. They consider all possible positions for the walls inside the maze in random order, each position exactly once. In each such position, they erect a wall if after that, it is still possible to move from every square of the maze to every other square.

In text form, the maze is given by $2n + 1$ lines, each containing $2n + 1$ characters. The even rows and columns, if counted from one, correspond to squares, and the odd ones to the walls between them. The "`.`" character corresponds to a square or a passage, and the "`#`" character to a wall or a joint between walls. In particular, a character in an even row and an even column is always a dot (it's a square), and a character in an odd row and an odd column is always a hash (it's a joint between walls).

A map of a 5×5 maze using the described notation can be seen in the example below. Additionally, to make local testing easier, you can download all mazes from tests with odd numbers. They can be found under "Samples ZIP" in the testing system interface.

Interaction Protocol

In this problem, your solution will be run twice on each test.

During the first run, the solution obtains a map and writes the hint as Loki. The first line contains the word "`view`". The second line contains an integer n, the size of the maze ($5 \le n \le 200$). Each of the next $2n + 1$ lines contains $2n + 1$ characters. Together these lines constitute the map of the maze.

The solution has to print one line: the hint that Loki will send to Thor. This hint has to be composed from digits `0` and `1` and have length from 0 to 1000 characters.

During the second run, the solution obtains the hint and walks through the maze as Thor. This run is interactive. The first line contains the word "`walk`". The second line contains the hint given by Loki to Thor: the one printed by the solution during the first run. The third line contains the integer n, the size of the maze, same as during the first run.

After that, the solution will get a piece of the map seen by Thor with the help of his torch, and in response, it should print the direction of his next step. Each piece of the map is given as three lines containing three characters each: the square of the maze where Thor is, along with its surroundings.

If the solution is sure that Thor passed the maze and stands at the exit, the solution should simply terminate gracefully. Otherwise, it has to print a line containing the direction of Thor's next step: "`N`" for a step to the North (up on the map), "`W`" for a step to the West (left on the map), "`S`" for a step to the South (down on the map), or "`E`" for a step to the East (right on the map). After that, the solution should flush the output buffer: this can be done by calling, for example, `fflush (stdout)` in C or C++, `System.out.flush ()` in Java, or `sys.stdout.flush ()` in Python.

If the passage is clear, Thor moves to an adjacent square in the requested direction and is given the next piece of the map seen by him at the moment. If there is a wall in the given direction, the process is terminated with "`Wrong Answer`" outcome.

The solution passes the maze if it terminated gracefully when Thor stood at the exit, and made at most 6000 steps before that.

Example

On each test, the input during the second run depends on the solution's output during the first run. In the example, We will consider solution which forms the hint by simply printing the required sequence of steps: `0` for a step to the East and `1` for a step to the North. Sure enough, this solution does not always work.

Two runs of this solution on the first test are shown below. In the second run, blank lines are added to show the sequence of events.

standard input	standard output
view 5 ########### #.#.......# #.###.##### #.#.....#.# #.#.###.#.# #...#.....# ###.####### #.........# ###.####### #.........# ###########	10001011
walk 10001011 5 ### #.. ### #.# ... ### #.# ... #.# #.# ..# #.# ### #.. #.# #.# ... ### ### ... #.# ### ... ### ### ..# ###	 E N N N E N E E

Problem F. Maharajas are Going Home

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	512 mebibytes

Jack and Jill have a chessboard which is infinite upwards and to the right. Rows and columns are numbered by integers from 1 to infinity. They play the following game on the chessboard. Initially, there are k stones on the chessboard. During their move, a player takes a single stone and moves it one or more positions to the left, downwards, diagonally (the same number of positions to the left and downwards), or using a knight's move that reduces the row and column numbers (one position downwards and two to the left, or two positions downwards and one to the left). The stone must not leave the chessboard, but may jump over other stones or occupy the same position as other stones. The player who can not make a move loses. Jack and Jill move in turns, Jack moves first.

Help Jack find a winning move, or determine that there exists a winning strategy for Jill.

Input

The first line contains an integer t, the number of test cases ($1 \le t \le 100$).

Each test case spans several lines. The first of them contains an integer k: the number of stones in the initial position ($1 \le k \le 10$). The next k lines determine the initial positions of the stones, one per line. Each position is given by two integers r and c: the numbers of row and column respectively ($1 \le r, c \le 2000$).

Output

For each test case, print a line containing three space-separated integers: i, r, and c. They mean that, when both players play optimally, Jack's winning move is to move i-th stone to position (r, c). The stones are numbered starting from 1.

If there are several possible solutions, print the lexicographically smallest one: the solution with the minimum number of stone, in case of ambiguity the solution with the minimum row number, in case of ambiguity the solution with the minimum column number.

If Jack does not have a winning strategy, print "`-1 -1 -1`" (without quotes).

Example

standard input	standard output
3 5 2 3 3 2 3 3 3 3 3 3 1 2 4 2 1 1 3 2	3 1 1 -1 -1 -1 2 1 1

Problem G. Ook

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	512 mebibytes

The librarian has a ticket with a string consisting of letters "o" and "k". Additionally, he has a pattern from the grocery store. The pattern contains a string which consists of characters "o", "k", and "?".

The librarian can perform an arbitrary number of any of the following operations in any order:

1. Put the pattern onto the ticket and cut the respective piece of the ticket: the number of letters on the piece has to be equal to the number of characters in the pattern. The remaining parts of the ticket can be used to cut more pieces, but only separately (they are not glued together). Any or both remaining parts can contain no more letters (in that case, obviously, the pattern can not be put onto them).
2. Take a piece of the ticket obtained by the first operation to the grocery store and exchange it for bananas.

When exchanging a piece of the ticket for bananas, the shopkeeper acts as follows. He starts by putting bananas into a heap. Initially, the heap is empty. The shopkeeper looks over the piece of the ticket, from left to right. For every letter "o" on it, he adds o bananas to the heap, and for every letter "k", he adds k bananas.

After that, the shopkeeper compares two strings from left to right, character by character: the one on the pattern and the one on the piece of the ticket. During the comparison, a "?" character in the pattern is considered equal to any letter. If the shopkeeper discovers a mismatch in a certain position, he gets very angry, and because of that, his hunger intensifies. As a result, he divides the heap into two parts such that the difference in the number of bananas is at most one, and then eats the part which is not less than the other. After that, the shopkeeper continues comparing the strings until either he compares the last pair of characters or the heap of bananas becomes empty.

All bananas left in the heap after comparison are given to the librarian in exchange for the part of the ticket.

Find the maximum possible number of bananas the librarian can obtain.

Input

The first line contains two integers o and k ($0 \le o, k \le 5000$). The second line contains a string S consisting of letters "o" and "k": the one printed on the ticket. The third line contains a string P consisting of characters "o", "k", and "?": the one printed on the pattern. It is guaranteed that $1 \le |P| \le |S| \le 250\,000$.

Output

Print the maximum possible number of bananas the librarian can obtain.

Examples

standard input	standard output
2 1 ookookook koo	10
1 3 kooooooook ?	13
1000 0 kookoo ook	2000
21 1 ooo kkk	7

Problem H. Pi Approximation

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	512 mebibytes

Vasya has a robot which can calculate the number of distinct right triangles that can be constructed from a given collection of sticks: one can take three sticks and use them as three sides of a triangle. Unfortunately, the robot can distinguish only angles of a triangle and does not distinguish its sides. So, the robot's calculations assume that similar triangles are equal. Two triangles are similar if one can be obtained from the other using uniform scaling, translation, rotation, and reflection.

Vasya's friend Petya discovered an amazing fact. If the robot is given n sticks with integer lengths from 1 to n, and the result of the robot's calculations is then divided by n, the number we obtain is a good approximation for the irrational number $\frac{1}{2\pi}$.

Vasya used the idea proposed by Petya for all integers n from $n_{\min}$ to $n_{\max}$ inclusive, and wrote down the results. Time has passed, and Vasya lost the results of the experiment, while his robot broke. Help him to once again find the best approximation of π that can be obtained if we use the idea for all integers n from $n_{\min}$ to $n_{\max}$. Here, x is a better approximation to π than y if $|\pi - x| < |\pi - y|$.

Input

The first line contains an integer t, the number of test cases ($1 \le t \le 100$). The next t lines contain test cases, one per line. Each test case is denoted by two integers $n_{\min}$ and $n_{\max}$ ($5 \le n_{\min} \le n_{\max} \le 200\,000\,000$, $n_{\max} - n_{\min} < 100$).

Output

For each test case, print a line containing a reduced fraction which is the best approximation of π that could be found using Petya's idea. Separate numerator, the division sign, and divisor by spaces.

Example

standard input	standard output
5 5 6 5 13 14 17 91 100 99999901 100000000	3 / 1 13 / 4 17 / 6 47 / 15 99999967 / 31830978

Explanation

In the third test case, among four approximations $\{\frac{7}{2}, \frac{15}{4}, 4, \frac{17}{6}\}$, fraction $\frac{17}{6}$ is the best one because its value is closest to π.

Problem I. Partition of Queries

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	512 mebibytes

Richard Roe has just implemented a brand new data structure S. It is able to perform only two operations: "`add`" and "`query`".

- "`add`" adds one item to S. You can assume that this operation takes zero seconds.
- "`query`" makes some request to S. This operation takes x seconds, where x equals to the number of previously added items.

Other details related to these operations are not important for this problem.

Suddenly Richard understood that he can optimize this structure by rebuilding it from time to time. So he implemented a new function named "`rebuild`". This new function works as follows: when "`rebuild`" is called, S "forgets" about items added before rebuilding. More precisely, after adding this operation:

- "`add`" adds one item to S and takes zero seconds.
- "`query`" makes some request to S and takes x seconds, where x equals to the number of items added after the last call of "`rebuild`" (here, assume that a "`rebuild`" was also called before all queries).
- "`rebuild`" takes y seconds.

You are given a sequence of "`add`" and "`query`" operations with S.

Your task is to insert "`rebuild`" operations in some positions in such a way that the number of seconds all operations will take is minimized.

Input

The first line of input contains two space-separated integers n and y ($1 \le n \le 10^6$, $0 \le y \le 10^6$).

The second line contains one string q of length n. Each character in q is either "`+`" or "`?`" (without quotes). Here, "`+`" means one call of "`add`", and "`?`" means one call of "`query`". These operations are performed according to the order in q.

Output

Output a single integer t: the minimum total time in seconds it will take for S to process all queries after possibly adding some calls of "`rebuild`".

Examples

standard input	standard output
6 5 ++??+?	6
6 8 ++??+?	7
5 1 +++++	0

Explanations

In the first example, the most optimal way is to place "`rebuild`" before the first "`?`".

In the second example, no placing of "`rebuild`" operations can decrease the total time.

In the third example, you also cannot decrease the total time because there are no "`query`" operations at all.

Problem J. Random Chess Game

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	512 mebibytes

This is an interactive problem.

One a boring Tuesday evening, Jack and Jill decided to play a game of chess. Since Jack is a very mediocre chessplayer, Jill promised to play random moves on her turns. Formally, if on Jill's turn, there are n legal moves, then Jill will choose each move with probability $1/n$. Also, Jill likes black color very much. So, she played all games with black pieces. After losing several games, Jack asked you to write a program which will help him to beat Jill's random strategy.

Chess rules

This section is based on the Wikipedia article about chess.

Chess game pieces are divided into white and black sets. Each set consists of 16 pieces: one king, one queen, two rooks, two bishops, two knights, and eight pawns. The game is played on a square board of eight rows and eight columns. The 64 squares alternate in color and are referred to as light and dark squares. The chessboard is placed with a light square at the right-hand corner nearest to each player. Thus, each queen starts on a square of its own color (the white queen on a light square; the black queen on a dark square).

White moves first, after which players alternate turns, moving one piece per turn (except for castling, when two pieces are moved). A piece is moved to either an unoccupied square or one occupied by an opponent's piece, which is captured and removed from play. With the sole exception of en passant, all pieces capture by moving to the square that the opponent's piece occupies. Moving is compulsory, it is illegal to skip a turn. A player may not make any move that would put or leave the player's own king in check. If the player to move has no legal move, the game is over; the result is either checkmate (a loss for the player with no legal move) if the king is in check, or stalemate (a draw) if the king is not. Each piece has its own way of moving:

- The king moves one square in any direction. The king also has a special move called castling that involves also moving a rook.
- A rook can move any number of squares along a rank or file, but cannot leap over other pieces. Along with the king, a rook is involved during the king's castling move.
- A bishop can move any number of squares diagonally, but cannot leap over other pieces.
- The queen combines the power of a rook and bishop and can move any number of squares along a rank, file, or diagonal, but cannot leap over other pieces.
- A knight moves to any of the closest squares that are not on the same rank, file, or diagonal. (Thus the move forms an L-shape: two squares vertically and one square horizontally, or two squares horizontally and one square vertically.) The knight is the only piece that can leap over other pieces.
- A pawn can move forward to the unoccupied square immediately in front of it on the same file, or on its first move it can advance two squares along the same file, provided both squares are unoccupied. A pawn can capture an opponent's piece on a square diagonally in front of it on an adjacent file, by moving to that square. A pawn has two special moves: the *en passant* capture and *promotion*.

Once in every game, each king can make a special move, known as castling. Castling consists of moving the king two squares along the first rank toward a rook that is on the player's first rank and then placing the rook on the last square that the king just crossed. Castling is permissible if the following conditions are met:

- Neither the king nor the rook has previously moved during the game.
- There are no pieces between the king and the rook.
- The king cannot be in check, nor can the king pass through any square that is under attack by an enemy piece, or move to a square that would result in check. (Note that castling is permitted if the rook is under attack, or if the rook crosses an attacked square.)

When a pawn makes a two-step advance from its starting position and there is an opponent's pawn on a square next to the destination square on an adjacent file, then the opponent's pawn can capture it *en passant* ("in passing"), moving to the square the pawn passed over. This can be done only on the very next turn; otherwise the right to do so is forfeited.

When a pawn advances to the eighth rank, as a part of the move it is promoted and must be exchanged for the player's choice of queen, rook, bishop, or knight of the same color. Usually, the pawn is chosen to be promoted to a queen, but in some cases another piece is chosen; this is called underpromotion. There is no restriction on the piece promoted to, so it is possible to have more pieces of the same type than at the start of the game (for example, two or more queens).

When a king is under immediate attack by one or two of the opponent's pieces, it is said to be in check. A move in response to a check is legal only if it results in a position where the king is no longer in check. This can involve capturing the checking piece; interposing a piece between the checking piece and the king (which is possible only if the attacking piece is a queen, rook, or bishop and there is a square between it and the king); or moving the king to a square where it is not under attack. *Castling* is not a permissible response to a check.

The object of the game is to checkmate the opponent; this occurs when the opponent's king is in check, and there is no legal way to remove it from attack. It is never legal for a player to make a move that puts or leaves the player's own king in check.

There are several ways games can end in a draw:

- *Stalemate*: The player whose turn it is to move has no legal move and is not in check.
- *Threefold repetition*: This most commonly occurs when neither side is able to avoid repeating moves without incurring a disadvantage. In this situation, either player can claim a draw. The three occurrences of the position need not occur on consecutive moves for a claim to be valid. Two positions are considered same or equal if all occupied squares and kind of pieces (not necessarily the same piece) they occupy are the same, the castling rights for both sides did not change, and no *en passant* capture was possible during the first occurrence, even if obviously not played.
- *Fifty-move rule*: If during the previous 50 moves (100 half-moves) no pawn has been moved and no capture has been made, either player can claim a draw. A half-move is a turn by either White or Black.

In this problem we assume that both players claim a draw whenever it is possible.

Standard Algebraic Notation (SAN)

This section is based on the Wikipedia article about algebraic notation in chess.

Each square of the chessboard is identified by a unique coordinate pair: a letter and a number. The vertical columns of squares, called files, are labeled `a` through `h` from White's left (the queenside) to right (the kingside). The horizontal rows of squares, called ranks, are numbered `1` to `8` starting from White's side of the board. Thus each square has a unique identification of file letter followed by rank number.

Each piece type (other than pawns) is identified by an uppercase letter (`K` for king, `Q` for queen, `R` for rook, `B` for bishop, and `N` for knight). Pawns are not identified by uppercase letters, but rather by the absence of one. Distinguishing between pawns is not necessary for recording moves, since only one pawn can move to a given square.

Each move of a piece is indicated by the piece's uppercase letter, plus the coordinate of the destination square. For example, `Qg4` (move a queen to $g4$). For pawn moves, a letter indicating pawn is not used, only the destination square is given. For example, `e4` (move a pawn to $e4$).

When a piece makes a capture, an `x` is inserted immediately before the destination square. For example, `Bxa6` (bishop captures the piece on $a6$). When a pawn makes a capture, the file from which the pawn departed is used to identify the pawn. For example, `fxe7` (pawn on the f-file captures the piece on $e7$). *En passant* captures are indicated by specifying the capturing pawn's file of departure, the `x`, and the destination square (not the square of the captured pawn). For example, `exf6` (pawn on the e-file captures the pawn on $f5$).

When two (or more) identical pieces can move to the same square, the moving piece is uniquely identified by specifying the piece's letter, followed by (in descending order of preference):

1. the file of departure (if they differ), for example, `Nbc3`,
2. the rank of departure (if the files are the same but the ranks differ),
3. both the file and rank (if neither alone is sufficient to identify the piece, which occurs only in rare cases where one or more pawns have promoted, resulting in a player having three or more identical pieces able to reach the same square).

As above, an `x` can be inserted to indicate a capture. For example, `N1xc3` (white knight on $b1$ captures black piece on $c3$ when another white knight is located on $b5$).

When a pawn moves to the last rank and promotes, an equals sign and the piece promoted to is indicated at the end of the move notation, for example: `h8=R` (promoting to rook).

Castling is indicated by the special notations `O-O` (for kingside castling) and `O-O-O` (queenside castling). Note that uppercase letter `O` is used.

A move that places the opponent's king in check has the character `+` appended. If check is also a mate then the character `+` is replaced by the character `#`.

Interaction Protocol

Each line of input describes one input command. Each command consists of command type and command argument separated by a colon and a single space. There are three different types of input commands:

`black_move:` *<last-black-move>*
`white_moves:` *<move-list>*
`result:` *<verdict>*

The game starts with a `white_moves` command listing the initially possible moves: *<move-list>* is a space-separated list of legal white moves in some order.

On each turn, your program should choose one legal move for White from the given *move-list* and output it on a single line.

After that, if the game has ended after your program's move, the `result` command is sent. Otherwise, a `black_move` command is sent describing the Black move (recall that it is chosen uniformly at random from all legal moves).

After that, if the game has ended after the Black move, the `result` command is sent. Otherwise, a `white_moves` command is sent again, listing all currently available moves in some order, and then it is your program's turn again.

Your program should terminate after receiving the `result` command. There are six different types of verdict (game termination status):

<verdict>	*result*	*<verdict>*	*result*
`White won by checkmate`	**OK**	`Illegal move`	**PE**
`Game drawn by stalemate`	**WA**	`Game drawn by repetition`	**WA**
`Game drawn by fifty-move rule`	**WA**	`Black won by checkmate`	**WA**

After printing each line, flush the output buffer, or you will get the outcome `Idleness Limit Exceeded`: this can be done by calling, for example, `fflush (stdout)` in C or C++, `System.out.flush ()` in Java, or `sys.stdout.flush ()` in Python.

There are 150 different tests. In each test, the initial state of pseudorandom generator used to generate the moves is fixed in advance. Test 1 corresponds to the example.

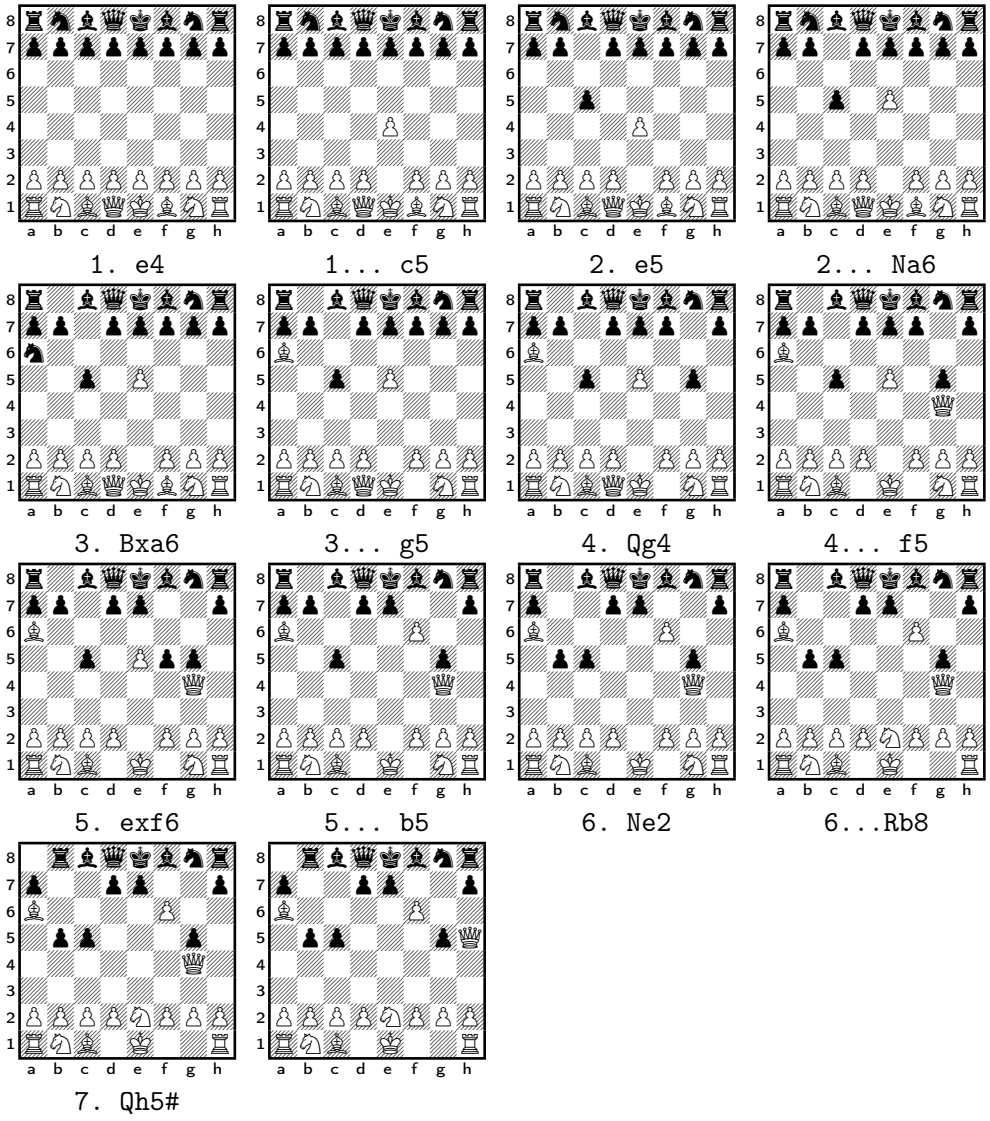
Example

standard input	standard output
white_moves: a3 a4 b3 b4 c3 c4 d3 d4 e3 e4 f3 f4 g3 g4 h3 h4 Nh3 Na3 Nc3 ←Nf3	
	e4
black_move: c5 white_moves: a3 a4 b3 b4 c3 c4 d3 d4 f3 f4 g3 g4 h3 h4 e5 Bc4 Nc3 Ba6 Qh5 ←Nf3 Ke2 Na3 Bb5 Qe2 Ne2 Nh3 Bd3 Be2 Qf3 Qg4	
	e5
black_move: Na6 white_moves: a3 a4 b3 b4 c3 c4 d3 d4 f3 f4 g3 g4 h3 h4 e6 Bc4 Nc3 Bxa6 Qh5 ←Nf3 Ke2 Na3 Bb5 Qe2 Ne2 Nh3 Bd3 Be2 Qf3 Qg4	
	Bxa6
black_move: g5 white_moves: a3 a4 b3 b4 c3 c4 d3 d4 f3 f4 g3 g4 h3 h4 e6 Bc4 Nc3 Qh5 Be2 ←Ke2 Na3 Bb5 Kf1 Qe2 Ne2 Nh3 Bd3 Bf1 Bxb7 Nf3 Qf3 Qg4	
	Qg4
black_move: f5 white_moves: a3 a4 b3 b4 c3 c4 d3 d4 f3 f4 g3 h3 h4 e6 exf6 Nc3 Qa4 Qd4 ←Qe2 Ne2 Qf3 Qb4 Qxg5 Na3 Qf4 Qc4 Kf1 Qd1 Qg3 Qh5# Qxf5 Bb5 Qe4 Bd3 Qh4 Bf1 ←Be2 Qh3 Ke2 Nh3 Kd1 Bxb7 Nf3 Bc4	
	exf6
black_move: b5 white_moves: a3 a4 b3 b4 c3 c4 d3 d4 f3 f4 g3 h3 h4 fxe7 f7+ Nc3 Qa4 Qd4 ←Qe2 Ne2 Qf3 Qb4 Qxg5 Na3 Qf4 Qc4 Kf1 Qe6 Qd1 Qg3 Qh5# Qf5 Bxb5 Qe4 Qxd7+ ←Qh4 Bxc8 Qh3 Ke2 Nh3 Kd1 Bb7 Nf3	
	Ne2
black_move: Rb8 white_moves: a3 a4 b3 b4 c3 c4 d3 d4 f3 f4 g3 h3 h4 fxe7 f7+ O-O Nbc3 Nec3 ←Qa4 Qd4 Nd4 Qf3 Qb4 Rg1 Qxg5 Na3 Qf4 Qc4 Kf1 Ng1 Qe6 Qg3 Qh5# Qf5 Bxb5 Ng3 ←Qe4 Rf1 Qxd7+ Qh4 Bxc8 Qh3 Nf4 Kd1 Bb7	
	Qh5#
result: White won by checkmate	

The "←" characters and blank lines were inserted into example to enhance readability. Real input doesn't contain such characters, and there are no empty lines in the input.

Explanation

In the example above, you could see an *en passant* capture (`5. exf6`). Also, on white move 7, there are examples of *disambiguating moves* (`Nbc3` and `Nec3`) and *castling* move (`O-O`).



Problem K. What? Subtasks? Again?

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	512 mebibytes

Vasya conducts programming contests. A total of n participants have registered for the upcoming round. Unfortunately, the testing system is stable only when the number of participants is at most m. If nothing is done, the contest will most probably have issues, and the round will become unrated.

Vasya doesn't have time to buy more servers or rewrite the testing system in another programming language for a performance gain. Nevertheless, he can enable features which some of the participants don't like at all, to the point that they will not take part in the contest. In particular, Vasya can:

1. disable HTTPS connections
2. postpone the round by 10 minutes
3. set the time limits in all problems to 100 milliseconds
4. divide problems into subtasks
5. honestly announce that the round will quite possibly be unrated

Help Vasya find a set of features which will allow him to conduct a contest without issues for the maximum possible number of participants.

Input

The first line contains three integers, n, m, and k ($1 < m < n \le 100\,000$, $0 \le k \le 100\,000$). The next k lines contain pairs of integers c_i ($1 \le c_i \le n$) and f_i ($1 \le f_i \le 5$) which mean that participant c_i will not take part in the contest if Vasya enables feature numbered f_i. Some of the pairs (c_i, f_i) can be equal.

Output

If it is not possible to have a contest without problems with at most m participants, print the phrase "`Round will be unrated`" (without quotes). Otherwise, print one integer: the maximum possible number of participants a rated contest can have.

Examples

standard input	standard output
10 7 10 2 1 3 5 2 1 4 1 9 5 5 4 6 4 7 4 8 4 10 4	6
10 9 0	Round will be unrated
5 4 3 4 1 4 2 1 2	4
2 1 2 1 1 2 1	0

Explanations

In the first example, the optimal strategy for Vasya is to enable the first and the fifth features. Then participants 2, 3, 4, and 9 will not take part in the contest.

In the third example, the optimal strategy for Vasya is to enable the first feature. Then participant 4 will not take part in the contest.

Problem L. The Five Bishops

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	512 mebibytes

This is an interactive problem.

There are five White Bishops and one Black King on an infinite chessboard. You are playing White. Your task is to **checkmate or stalemate** the King or to determine whether the King could avoid the checkmate or stalemate. White moves first.

You will be given 50 moves to do that. If after 50-th move of White side, the King is still not checkmated/stalemated, the answer will be considered incorrect.

The initial coordinates of pieces will be not greater than 10^6 by an absolute value. During the game, all coordinates must not exceed 10^9 by an absolute value.

Interaction Protocol

The first line of the input will contain only one single integer t, the number of problem instances to solve ($1 \le t \le 1000$).

Each instance starts with six pairs of integers x_i and y_i on a single line: the first five are coordinates of Bishops, the last pair is the coordinates of the King. All these integers do not exceed 10^6 by an absolute value.

Then the game starts. Your move consists of four integers x_1 y_1 x_2 y_2 on a single line: the initial and final coordinates of a Bishop. The interactor responds with four integers x_1 y_1 x_2 y_2 on a single line: the initial and final coordinates of the King. When the King is checkmated/stalemated, the interactor responds with four zeros, followed by the next problem instance if there is one. If your program decides that there is no way to checkmate/stalemate the King, it must output four zeros instead of the first move, in this case the interactor immediately responds with the next problem instance if there is one. The coordinates of moving pieces must not exceed 10^9 by an absolute value.

See sample interaction for more details.

If the checkmate/stalemate is possible, your program must do it in no more than 50 moves, otherwise the answer will be considered invalid (remember fifty-move rule). You don't need to find the optimal solution, but it is guaranteed that in every case where the solution exists it is possible to achieve the goal in no more than 50 moves.

If the checkmate/stalemate is not possible, your program should immediately respond with the line of four zeros (no moves allowed), otherwise the answer will be considered incorrect.

It is always guaranteed that every problem instance is correct, that is, no two pieces occupy the same cell and the King is not in check.

After each printed line, flush the output buffer: this can be done by calling, for example, `fflush (stdout)` in C or C++, `System.out.flush ()` in Java, or `sys.stdout.flush ()` in Python.

Example

standard input	standard output
4 1 1 2 2 3 3 4 4 5 5 7 5 1 1 1 2 1 4 1 5 2 2 3 5 0 0 0 0 1 2 2 3 4 5 5 6 6 7 5 3 3 2 4 2 3 6 4 6 6 4 3 4 0 0 0 0	 0 0 0 0 2 2 1 3 0 0 0 0 6 4 5 5

Explanation

Empty lines are added only to show the sequence of events.

In the first and third test cases there are five same-color bishops. In the second test case the checkmate could be achieved in one move. In the fourth case the stalemate could be achieved in one move.