Problem A. Additive Class

Input file:	`additive-class.in`
Output file:	`additive-class.out`
Time limit:	2 seconds
Memory limit:	256 mebibytes

In this problem, you have to find how many elements of the additive class of a and b belong to $[l, r]$.

An *additive class* of positive integers a and b is the set of numbers representable as a sum of zero or more numbers each of which is either a or b. Formally, it is the following set: $\{x \cdot a + y \cdot b \mid x, y \in \mathbb{Z},\ x, y \geq 0\}$.

Given a, b and two integers l and r ($l \leq r$), find how many elements e of the additive class of a and b satisfy the double inequality $l \leq e \leq r$.

Input

The input consists of one or more test cases. The first line of input contains an integer t, the number of test cases ($1 \leq t \leq 10\,000$). Then follow the test cases themselves.

Each test case is given on two lines. The first of these lines contains two positive integers a and b ($1 \leq a, b \leq 10^9$). The second line contains two integers l and r ($0 \leq l \leq r \leq 10^{18}$).

Output

For each test case, print the answer on a separate line. The answer for a test case is one integer: the number of elements of the additive class of a and b which belong to $[l, r]$.

Example

additive-class.in	additive-class.out
2 3 5 6 8 6 4 13 13	2 0

Explanation

The example consists of two test cases.

In the first test case, $a = 3$ and $b = 5$. The additive class of these numbers contains the numbers 0, 3, 5, $6 = 3 + 3$, $8 = 3 + 5$, $9 = 3 + 3 + 3$, ... Two of them belong to the segment $[6, 8]$.

In the second test case, $a = 6$ and $b = 4$. Obviously, their additive class contains only even numbers, so no element belongs to the segment $[13, 13]$.

Problem B. Board Blitz

Input file:	`board-blitz.in`
Output file:	`board-blitz.out`
Time limit:	2 seconds
Memory limit:	256 mebibytes

In this problem, you have to quickly find the final position in a game on a square board.

Consider a game on a square checkered board divided into 8×8 square cells. Initially, the board is empty. Two players take turns one after another. The first player controls white pieces, the second player controls black pieces. A turn consists in either a movement or a jump of the player's pieces.

To make a movement, a player chooses one of the four directions and shifts all her pieces simultaneously by one cell in that direction. If a piece moves off the board, it disappears. Furthermore, for each of the eight border cells where no piece could have been shifted to during a movement in this direction, the player may place a new piece of her color in that cell. If a piece appears on a cell which contains a piece of opposite color, that piece (of opposite color) disappears from the board.

To make a jump, a player takes all her pieces and simultaneously transfers them to cells which correspond to a 90-degree rotation around the board center. The rotation must be clockwise for white pieces and couter-clockwise for black pieces. Same as after a movement, if a piece appears on a cell which contains a piece of opposite color, that piece (of opposite color) disappears from the board.

Both players agreed to make turns according to the values obtained from a random number generator: the first random number determines the first player's turn, the second number gives the second player's turn, the third number specifies the next turn of the first player, and so on.

You are given the parameters a, c and s of a linear congruential pseudorandom number generator in a 64-bit signed integer data type. This generator works in the following way: to get the next random number, the assignment $s_{\text{new}} := s_{\text{old}} \cdot a + c$ is carried out, the result is truncated to a 64-bit signed integer data type, stored as the new value of s and returned as the next random number.

A turn is determined by the number p obtained as the highest 11 bits of the random number (in other words, p is the result of an integer division of s by 2^{53} rounded towards zero). If $p < 0$, the turn will be a jump. Otherwise, the turn will be a movement, bits 8 (multiplied by 1) and 9 (multiplied by 2) form the number of direction (0 is up, 1 is left, 2 is down, 3 is right), and bits 0 through 7 specify which of the eight cells will contain new pieces. The cells are numbered 0 through 7 from top to bottom or from left to right (depending on the direction).

Given numbers a, c and s, and also the number of half-turns n, find the final position on the board. A half-turn is a turn by one of the two players. It is guaranteed that n is even. Additionally, it is guaranteed that in all the judges' tests except the example, the triple of numbers a, c and s is chosen uniformly at random from all such triples that the linear congruential generator specified by them has a period of exactly 2^{64}.

Input

The first line of input contains an integer n, the number of half-turns in the game, and also integers a, c and s which are the parameters of the random number generator ($2 \le n \le 30\,000\,000$, n is even, $-2^{63} \le a, c, s < 2^{63}$, the period of the generator is exactly 2^{64}).

Output

Print eight lines with eight characters on each line: the state of the board after all turns have been completed. Denote empty cells with character "`.`", white pieces with character "`W`" and black pieces with character "`B`".

Example

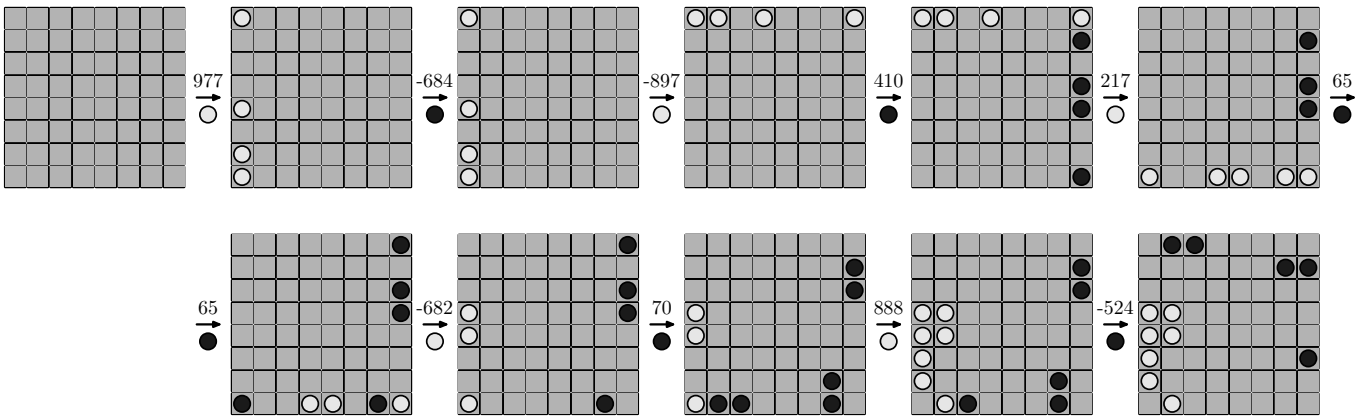
board-blitz.in	board-blitz.out
10 12345678901234565 23456789012345679 12345678	.BB..... ......BB .... WW...... WW...... W......B W....... .W......

Explanation

The following table shows the sequence of values of s, values of p and specific bits of p in the example.

s	**p**	**bit 10**	**bits 9–8**	**bits 7–0**
8800325836438854357	977	0	11	11010001
-6158326326699848968	-684	1	01	01010100
-8075422879926989273	-897	1	00	01111111
3701688810043832978	410	0	01	10011010
1958856781360031529	217	0	00	11011001
594250524398489244	65	0	00	01000001
-6137059809998356901	-682	1	01	01010110
634828852634288534	70	0	00	01000110
8001533107637205053	888	0	11	01111000
-4716194469178570240	-524	1	01	11110100

The pictures below show all the intermediate states of the board in the example.



Problem C. Divisor Count

Input file:	`divisor-count.in`
Output file:	`divisor-count.out`
Time limit:	5 seconds
Memory limit:	256 mebibytes

In this problem, you have to find the first n integers which have exactly k divisors.

A *divisor* of an integer a is an integer b such that the quotient $\frac{a}{b}$ is also an integer.

Given n and k, find the first n positive integers which have exactly k distinct positive integer divisors and are not greater than 10^{18}. If the total number of such integers is less than n, find all of them.

Input

The first line of input contains two integers n and k: the number of integers to find and the required number of divisors ($1 \le n, k \le 110\,000$).

Output

Print n integers, each on a separate line: the first n positive integers which have exactly k distinct positive integer divisors and are not greater than 10^{18}, in increasing order. If there are $m < n$ such integers, print the number -1 in each of the remaining $n - m$ lines.

Examples

divisor-count.in	divisor-count.out
5 4	6 8 10 14 15
4 29	268435456 22876792454961 -1 -1

Explanations

In the first example, you have to print the first five integers with exactly four divisors. These are 6 (divisors 1, 2, 3 and 6), 8 (divisors 1, 2, 4 and 8), 10 (divisors 1, 2, 5 and 10), 14 (divisors 1, 2, 7 and 14) and 15 (divisors 1, 3, 5 and 15).

In the second example, you have to find the four minimal integers having 29 divisors each. These are $2^{28} = 268\,435\,456$, $3^{28} = 22\,876\,792\,454\,961$, $5^{28} = 37\,252\,902\,984\,619\,140\,625$ and $7^{28} = 459\,986\,536\,544\,739\,960\,976\,801$. The latter two integers are strictly greater than 10^{18}, so print -1 instead of each.

Problem D. Domino Tiling

Input file:	`domino-tiling.in`
Output file:	`domino-tiling.out`
Time limit:	2 seconds
Memory limit:	256 mebibytes

In this problem, you have to transform one domino tiling of a rectangle into another by moving dominoes along cycles.

Consider a rectangular grid of size $h \times w$. A *domino piece* (or simply *domino*) is a rectangular block that can be placed on any two cells of the grid sharing a common side. A *tiling* of the grid is a collection of domino pieces such that each cell of the grid is covered by exactly one piece in the collection. Note that dominoes may not cross the border of the grid.

You are given two tilings of the grid: initial tiling and target tiling. Your task is to transform initial tiling into target tiling. In order to achieve that, you are allowed to make *moves*. A single move consists in picking a domino cycle and moving dominoes along that cycle (the formal definition is given below). The *cost* of each move is the number of cells in the respective domino cycle. The total cost of transformation is the sum of costs of all moves you make. Of course, the total cost must be minimal possible.

Now, to the details.

Let us define a *domino cycle* of positive length n on a given tiling. Consider a circular sequence of $2n$ distinct cells such that any two consecutive cells in the sequence share a common side. Here, circular means that the first and last cells of the sequence are also considered consecutive. Obviously, there are $2n$ pairs of consecutive cells in such a sequence. Cover n of them by n non-overlapping domino pieces and you get a domino cycle.

The process of *moving dominoes along a cycle* results in change of the tiling. First, the n domino pieces covering all the $2n$ cells which belong to the cycle are removed. After that, n new non-overlapping domino pieces are added to cover n other pairs of consecutive cells. One can easily see that the resulting collection of pieces is also a tiling. Recall that the cost of such move is $2 \cdot n$.

Input

The first line of input contains two integers h and w separated by a space: height and width of the rectangular grid ($1 \le h, w \le 100$).

The next h lines contain w characters each and describe initial tiling. Each of these characters is one of the following:

- character "`l`" denotes a left part of a domino piece,
- character "`r`" denotes a right part of a piece,
- character "`u`" denotes an upper part, and
- character "`d`" corresponds to a lower part.

Next goes a line containing w dash signs ("`-`").

The next h lines contain w characters each and describe target tiling in the same fashion.

It is guaranteed that the input is consistent:

- h and w are chosen in such a way that tiling an $h \times w$ grid is possible, and

- each character in each tiling is guaranteed to have the required neighbor in the required direction.

Output

On the first line of output, write two integers separated by a space: m, the number of moves, and p, the total cost of transformation. Note that as long as the cost p is minimal possible, the number of moves m can be arbitrary.

The next m lines should describe the moves, one per line. A move along a domino cycle of k cells should be written as $k\ r_1\ c_1\ r_2\ c_2\ \ldots\ r_k\ c_k$. Here, r_i and c_i are the coordinates of i-th cell of the cycle. Separate consecutive tokens on a line by one or more spaces. You can start a cycle at any point and traverse it in any direction.

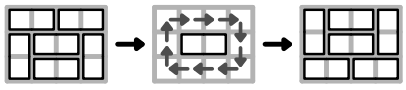
Examples

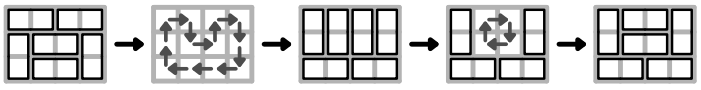
domino-tiling.in	domino-tiling.out
3 4 lrlr ulru dlrd ---- ulru dlrd lrlr	1 10 10 1 1 1 2 1 3 1 4 2 4 3 4 3 3 3 2 3 1 2 1
2 6 uuuuuu dddddd ------ lrlrlr lrlrlr	3 12 4 1 1 2 1 2 2 1 2 4 1 3 2 3 2 4 1 4 4 1 5 2 5 2 6 1 6

Explanation

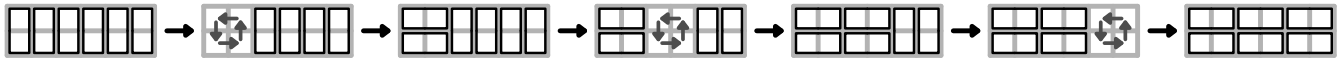
In the first example, the minimal total cost is 10. It can be achieved by just one move which involves the 10 border cells of the grid in cyclic order. Five dominoes covering cells $(1, 1)$ and $(1, 2)$, $(1, 3)$ and $(1, 4)$, $(2, 4)$ and $(3, 4)$, $(3, 3)$ and $(3, 2)$, $(3, 1)$ and $(2, 1)$ are removed. After that, new dominoes covering cells $(1, 2)$ and $(1, 3)$, $(1, 4)$ and $(2, 4)$, $(3, 4)$ and $(3, 3)$, $(3, 2)$ and $(3, 1)$, $(2, 1)$ and $(1, 1)$ are added.

The answer is illustrated below on the left. Note that there could be other ways of achieving the target tiling. To the right is another solution which is however not optimal.





In the second example, the minimal total cost is 12. It can be achieved by moving the dominoes along three small cycles each of which contains four cells. The process is illustrated below.

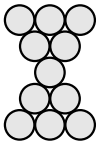


Problem E. Hourglass

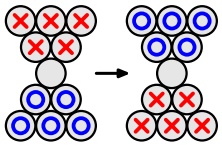
Input file:	`hourglass.in`
Output file:	`hourglass.out`
Time limit:	2 seconds
Memory limit:	256 mebibytes

In this problem, you have to solve a puzzle involving a planar board and pieces.

The Hourglass Puzzle of size n looks as follows. The board is drawn on a plane which consists of $(2 \cdot n + 1)$ rows. The rows are horizontal and lie one under the other. Each row consists of positions: the middle row contains one position, and each other row below or above the middle contains one more position than its neighbor towards the middle. The positions in neighboring rows are connected so that each position of the shorter row has exactly two neighboring positions in the longer one. The board for $n = 2$ is illustrated below.



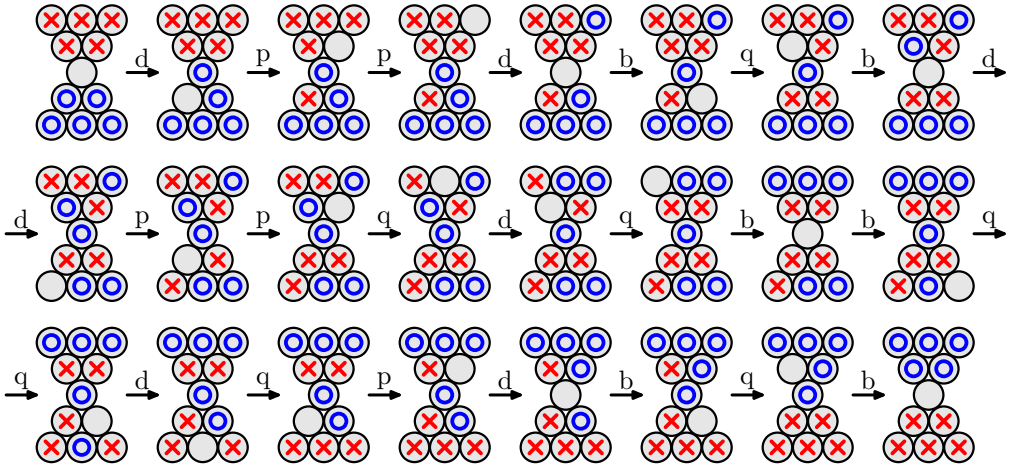
There are some pieces on the board. Initially, each position of the n top rows contains a red piece (sand), and each position of the n bottom rows contains a blue piece (air). The goal of the puzzle is to swap the pieces, that is, to fill n top rows by air and n bottom rows by sand. Pieces of the same color are considered the same. There can be at most one piece in each position at any given moment. The initial and final positions for $n = 2$ are illustrated below. Red pieces are denoted by "`X`" while blue pieces are denoted by "`O`".



Moving the pieces complies with the following rules.

- Red pieces can only move down, while blue pieces can only move up.
- On each move, exactly one piece changes its position.
- If for some piece, one of the two neighboring positions in the permitted direction (up-left or up-right for blue pieces, down-left or down-right for red ones) exists and is free, that piece can just move there.
- If that position is occupied by a piece of the other color, but the next position *in the same direction* (up-left, up-right, down-left or down-right) exists and is free, the piece can jump into that free position.
- Note that jumps are allowed only if they follow a straight line, have a length of two positions and go over a piece of the other color. Other kinds of jumps (for example, straight up or straight down by two rows) are not permitted.

The pictures below illustrate the sequence of moves allowing to solve the puzzle for $n = 2$.



The sequence of moves can be written down in the following way.

- Moving or jumping up-left is denoted by lowercase English letter «b».
- Moving or jumping up-right is denoted by lowercase English letter «d».
- Moving or jumping down-left is denoted by lowercase English letter «p».
- Moving or jumping down-right is denoted by lowercase English letter «q».

The intuitive meaning of this notation is that the tail of each letter shows the direction of a movement or jump.

It turns out that the information provided by such notation is enough to restore all intermediate states of the puzzle.

Given n, the size of the puzzle, print out its solution. Note that you don't have to minimize or maximize the number of moves.

Input

The first line of input contains one integer n: the size of the puzzle ($1 \le n \le 10$).

Output

Print one line containing a solution to the puzzle of the given size. This line must contain only lowercase English letters «b», «d», «p» and «q» and no other characters, in particular, no spaces. If there are several possible solutions, print any one of them.

Example

hourglass.in	hourglass.out
2	dppdbqbdppqdqbbqdqpdbqb

Explanation

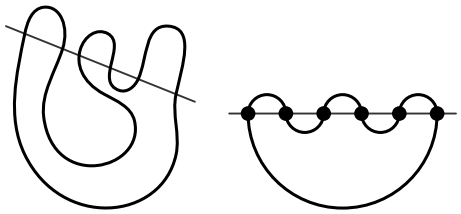
The pictures above correspond to the example.

Problem F. k-th Meander

Input file:	`kth-meander.in`
Output file:	`kth-meander.out`
Time limit:	5 seconds
Memory limit:	256 mebibytes

In this problem, you have to find the lexicographically k-th meander of order n.

Consider a straight line drawn on a plane. A *meander* of order n is a closed planar curve which intersects the straight line $2n$ times. One can imagine the meander as a twisting road loop which intersects a straight segment of a river via bridges. The intuitive definition above must however be refined with a few formal conditions: the curve can have no self-intersections or self-touchings, all $2n$ points where the curve intersects the line are different and are indeed intersections, not touchings, and furthermore, the curve and the line have no common points except the above-mentioned ones.



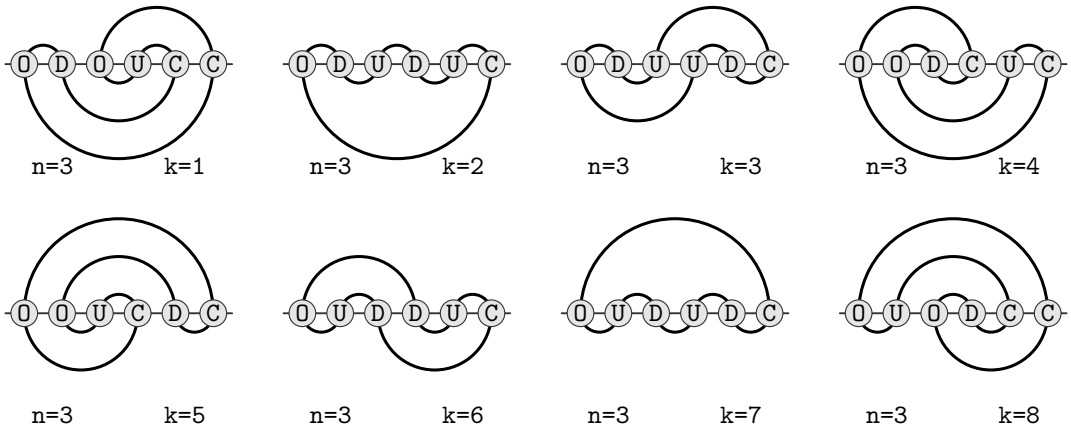
Let us define a convenient way to draw a meander. Without loss of generality, let the straight line be placed horizontally, and the intersection points put regularly with unit distances between them. Note that the intersection points divide the curve into $2n$ parts. Certain n parts are above the line, and the other n parts below it. We draw each of these parts as semicircles based on the corresponding segments of the line as diameters and located either above or below the line, as needed. One can prove that the picture we got is a meander: in particular, the curve constructed in this way has no self-intersections or self-touchings. The inverse is also true: every meander can be transformed to such picture by a coutinuous (homeomorphic) transformation.

Now, let us learn to write a meander down as a string. Consider a meander drawn according to the above procedure. Let us walk along the line from left to right and take a note every time we meet an intersection point. For each such point, there are two parts of the curve connected with it: one above and one below the line. Each of these parts is a semicircle going either to the left or to the right from the current intersection point. We will distinguish between four possible cases and label each case with an English letter:

- If both semicircles go to the right, we shall say that the curve *opens* at this intersection and denote it with the letter `O` (open).
- If both semicircles go to the left, we shall say that the curve *closes* at this intersection and denote it with the letter `C` (close).
- If the lower semicircle goes to the left and the upper one to the right, we shall say that the curve *goes up* at this intersection and denote it with the letter `U` (up).
- If the upper semicircle goes to the left and the lower one to the right, we shall say that the curve *goes down* at this intersection and denote it with the letter `D` (down).

Moving from left to right, write down the $2n$ letters which we used to label the intersections. It turns out that this information is enough to uniquely restore the drawing of a meander obtained by the procedure above. One can imagine this process as drifting with the current of the river (from left to right) and writing down the configuration of roads in the immediate vicinity of the bridges.

Two meanders are considered different if their notations are different. As an example, all eight different meanders of order 3 are presented below.



For a fixed order n, take all different meanders of this order, and then sort them lexicographically by their notations. Given the order n and the number k, find the notation of the lexicographically k-th meander of order n.

Input

The first line of input contains two integers n and k: the order and the number of a meander ($1 \le n \le 25$, $1 \le k \le 10^{11}$). It is guaranteed that a meander with the given parameters exists.

Output

Print a string consisting of $2n$ letters: the notation of the meander which has the number k in a lexicographically sorted list of meanders of order n.

Examples

kth-meander.in	kth-meander.out
3 1	ODOUCC
3 2	ODUDUC
3 3	ODUUDC
3 4	OODCUC
3 5	OOUCDC
3 6	OUDDUC
3 7	OUDUDC
3 8	OUODCC

Explanations

The pictures above correspond to the examples.

Problem G. Records and Cycles

Input file:	`records-and-cycles.in`
Output file:	`records-and-cycles.out`
Time limit:	2 seconds
Memory limit:	256 mebibytes

In this problem, you have to construct a bijection which maps every permutation of length n with k cycles to a permutation of length n with k records.

A *permutation* p of length n is a sequence of n integers $p(1)$, $p(2)$, ..., $p(n)$ such that each of the numbers 1, 2, ..., n occurs in that sequence exactly once.

A *cycle* in a permutation p is a sequence of integers c_1, c_2, ..., c_m such that $p(c_1) = c_2$, $p(c_2) = c_3$, ..., $p(c_m) = c_1$. A special case is a cycle of unit length: $p(i) = i$. It can be shown that each permutation can be represented as a collection of pairwise disjoint cycles.

A *record*, or record value, in a permutation p is its element p_j which is greater than all elements of the permutation preceding it: $p_j > p_i$ for each $i < j$. Note that p_1 is always a record by definition.

One can prove that for any given n and k, the number of permutations of length n with exactly k cycles is equal to the number of permutations of length n with exactly k records. Therefore, there exists a bijection which maps permutations of length n with k cycles to permutations of length n with k records and vice versa. Recall that a *bijection* is a function giving an exact pairing of the elements of two sets: every element of one set is paired with exactly one element from the other set, and every element of the other set is paired with exactly one element of the first set. Your task is to find and implement such a function.

Input

The first line of input contains an integer t, the number of permutations ($1 \le t \le 300\,000$).

Each of the next t lines contains description of a single permutation. A description starts with an integer n, the length of the permutation ($1 \le n \le 300\,000$), followed by a character which is either "`r`" or "`c`" and denotes the type of the permutation. Then follows a sequence of n integers which constitutes the permutation itself. Every integer from 1 to n occurs in this sequence exactly once.

Consecutive tokens on a line are separated by single spaces. The total length of all permutations in the input does not exceed 300 000.

Output

The output must be in the same format as the input.

The first line of output must contain the integer t, the number of permutations in the input.

After that, for each of the t given permutations, output the permutation corresponding to it on a separate line. If the type of the input permutation is "`r`", find the number of records in it and output the corresponding permutation of type "`c`" with the same number of cycles. If the type of the input permutation is "`c`", find the number of cycles in it and output the corresponding permutation of type "`r`" with the same number of records.

Each of the t lines must start with the integer n which is the length of the permutation, followed by the character denoting the type. After that must follow n integers which constitute the permutation itself. Separate consecutive tokens on a line by single spaces.

The correspondence must be consistent: if a permutation p of type "r" corresponds to a permutation q of type "c", the inverse correspondence must also be true, no other permutation of type "c" can correspond to p, and no other permutation of type "r" can correspond to q. Note however that your bijection does not need to be consistent across different outputs.

Example

records-and-cycles.in	records-and-cycles.out	alternative output
6 3 r 1 2 3 3 c 1 2 3 3 r 2 1 3 3 r 2 3 1 3 c 2 1 3 2 c 2 1	6 3 c 1 2 3 3 r 1 2 3 3 c 2 1 3 3 c 3 2 1 3 r 2 1 3 2 r 2 1	6 3 c 1 2 3 3 r 1 2 3 3 c 3 2 1 3 c 1 3 2 3 r 1 3 2 2 r 2 1

Explanation

In this example, the first five permutations have length $n = 3$. The first permutation has type "r" and contains three records. The second permutation has type "c" and contains three cycles. These two permutations correspond to each other. The third and the fourth permutations have type "r" and contain two records each. The fifth permutation has type "c" and contains two cycles. It corresponds to the third permutation. The fourth permutation corresponds to the permutation 3, 2, 1 of type "c" which is another permutation with two cycles.

The last permutation has length $n = 2$, type "c" and contains one cycle. It corresponds to a permutation 2, 1 of type "r" with one record.

Note that this is not the only possible bijection: one alternative output is given above. Here, for the third permutation in the input (which is 2, 1, 3 of type "r"), we pick 3, 2, 1 of type "c" as the corresponding permutation. Then, the fourth permutation (2, 3, 1 of type "r") must get a different corresponding permutation, too, and we take the permutation 1, 3, 2 of type "c". After that, for the fifth permutation (2, 1, 3 of type "c"), the only possible option is to pick 1, 3, 2 of type "r" since the other two permutations with two records already correspond to different permutations with two cycles.

Problem H. Rock-Paper-Scissors on a Plane

Input file:	`rps-walk.in`
Output file:	`rps-walk.out`
Time limit:	2 seconds
Memory limit:	256 mebibytes

In this problem, you have to travel from one corner of a checkered field to the opposite corner conforming to certain rules and avoiding making too much moves.

There is a checkered field on a plane consisting of m rows and n columns. Each cell of the field contains one of three items: rock, paper or scissors. We start moving from the cell $(1, 1)$ and want to reach the cell (m, n) by performing a sequence of moves. A single move consists of jumping into any cell that is no farther than d from the current cell. The distance is measured between cell centers: the distance between cells (x_1, y_1) and (x_2, y_2) is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Additionally, each move must lead to a cell containing an item which defeats the item on the previous cell. As in the game "Rock-Paper-Scissors", rock defeats scissors, scissors defeat paper, and paper defeats rock. When we arrive at a cell, we take the item from that cell, so it is forbidden to visit a single cell twice.

The field is fairly large, so the types of items in its cells are determined by use of a random number generator, more precisely, a linear congruential generator. This generator has a state which is an integer s from 0 to $2^{32} - 1$. In order to obtain the next random number from 0 to $r - 1$, we carry out the assignment $s := (s \cdot 1\,664\,525 + 1\,013\,904\,223) \bmod 2^{32}$ and return the remainder of dividing s by r. The initial value of s is given in the input.

The items in the cells are determined in the following way. For each cell, we obtain the next random number from the set $\{0, 1, 2\}$ (that is, $r = 3$). If this number is a zero, the item is rock, if it is one, paper, and in case it is two, scissors. Cells are considered row-by-row from top to bottom, cells in one row are considered from left to right. So, the order of cells is the following: $(1, 1)$, $(1, 2)$, ..., $(1, n)$, $(2, 1)$, $(2, 2)$, ..., $(2, n)$, ..., $(m, 1)$, $(m, 2)$, ..., (m, n).

An important additional requirement is that we must finish in time no greater than the time it would take to travel from cell $(1, 1)$ to cell (m, n) along a straight line with half the maximum speed, plus three extra moves. Formally, the number of moves must not exceed the real number $2 \cdot \frac{\sqrt{(m-1)^2+(n-1)^2}}{d} + 3$.

Given the dimensions of the field, the maximum travel distance for a single move and the initial state of the random number generator, find a path satisfying all the constraints.

Input

The first line of input contains integers m, n, d и s separated by spaces: the height and width of the field, the maximum travel distance for a single move and the initial state of the random number generator ($100 \le m, n \le 2^{16}$, $10 \le d \le 100$, $0 \le s \le 2^{32} - 1$). Note that the lower bounds for the numbers m, n and d are fairly large.

Output

On the first line, print the number of moves k in the path you found. In the next $(k+1)$ lines, print the coordinates of the visited cells in the order of traversal. Print the row first and the column second.

The first cell must be $(1, 1)$, and the last one must be (m, n). Each cell may be visited at most once. The item in each next cell must defeat the item in the previous cell. The distance between any two consecutive cells of the path must not exceed d, and the total number of moves k must not exceed the real number $2 \cdot \frac{\sqrt{(m-1)^2+(n-1)^2}}{d} + 3$.

If there are several possible paths, print any one of them. It is guaranteed that in all the judges' tests, there exists a path satisfying all constraints with length not exceeding the real number $1.5 \cdot \frac{\sqrt{(m-1)^2+(n-1)^2}}{d} + 3$.

Example

rps-walk.in	rps-walk.out
100 120 50 5	5 1 1 31 41 59 70 92 107 98 120 100 120

Explanation

The following table contains the values of s and the items for the cells visited in the example.

cell	**value of** s	$s \bmod 3$	**item**
$(1, 1)$	1022226848	2	scissors
$(31, 41)$	4062427704	0	rock
$(59, 70)$	11774611	1	paper
$(92, 107)$	860629922	2	scissors
$(98, 120)$	383195253	0	rock
$(100, 120)$	2533494757	1	paper

The value of the fraction $\frac{\sqrt{(m-1)^2+(n-1)^2}}{d}$ is $3.0959\ldots$, so in this example, it is required to make no more than nine moves, and the judges formally guarantee that there exists a path consisting of seven moves or less.

Problem I. Set Operations

Input file:	`set-operations.in`
Output file:	`set-operations.out`
Time limit:	2 seconds
Memory limit:	256 mebibytes

In this problem, you have to perform operations with sets according to the given expression.

Consider an expression involving three sets A, B and C and three set operations: union, intersection and complement.

For the purposes of this problem, sets can only consist of numbers $\{1, 2, \ldots, n\}$ for some fixed integer n. A *union* of sets X and Y is a set which contains all numbers present in at least one of the sets X and Y. An *intersection* of sets X and Y is a set which contains all numbers present in both of the sets X and Y. A *complement* of set X is a set which contains all numbers from the range $\{1, 2, \ldots, n\}$ which are not in X. For example, if $n = 5$, $X = \{3, 4\}$ and $Y = \{1, 3\}$, the union of X and Y is $\{1, 3, 4\}$, their intersection is $\{3\}$, and complement of X is $\{1, 2, 5\}$.

An *expression* is recursively defined as follows:

- `A`, `B` and `C` are expressions denoting the three given sets A, B and C respectively.
- If E is an expression, `~`E and `(`E`)` are also expressions.
- If E and F are expressions, E`|`F and E`&`F are also expressions.

Here, `~`X is the complement of set X, X`|`Y is the union of sets X and Y, and X`&`Y is the intersection of sets X and Y. Complement is evaluated before intersection, which in turn is evaluated before union. Parentheses play the usual role of prioritizing operations. So, for example, an expression `A|~B&C` is evaluated in the same way as `A|((~B)&C)`.

You are given one expression and a number of triples of sets A, B and C. Find and output the value of the expression for each of the given triples.

Input

The first line of input contains the expression. Its length is from 1 to 300 000 characters. It is guaranteed that the expression conforms with the recursive definition above. There are no spaces on the first line.

The second line contains two integers n and t separated by a space: the number of elements and the number of triples of sets ($1 \le n \le 20$, $1 \le t \le 10\,000$).

Each of the next t lines describes a triple of sets A, B and C in that order. A set is denoted by a list of integers contained in that set in strictly ascending order followed by a zero. Numbers are separated by one or more spaces.

Output

For each of the t given triples of sets, output a single line containing a single set: the result of evaluating the expression for the given A, B and C. A set must be written as a list of integers contained in that set in strictly ascending order followed by a zero. Separate numbers by one or more spaces.

Example

set-operations.in	set-operations.out
A|~B&C 5 2 1 3 0 3 4 0 2 3 5 0 0 1 3 5 0 0	1 2 3 5 0 0

Explanation

In this example, the expression is evaluated as `A|((~B)&C)`. For the first triple, `~B` $= \{1, 2, 5\}$, `~B&C` $= \{2, 5\}$ and the whole expression evaluates to the set $\{1, 2, 3, 5\}$. For the second triple, `~B` $= \{2, 4\}$, `~B&C` is empty and the result is also an empty set.