Spring 1994 Denison Computer Programming Contest
Problem 1
What's on the Grille?

An interesting encryption method is to use a grille, which is an n by n square of cardboard with one quarter of the squares cut out. To encrypt a message, the characters of the message are written on an n by n grid, where n^2 is the smallest perfect square no smaller than the message length. Leftover spots on the grid are filled in with the letters of the alphabet, in alphabetical order. The grille is then placed over the grid where the cutouts reveal certain letters. The revealed letters are then read off, row by row, giving exactly one fourth of the letters. The grille is then rotated 90° counterclockwise and the process is repeated. Do this two more times and the message is encrypted. Of course, in order to get each letter of the grid exactly once, the grille must be carefully constructed.

For example, suppose the grille looked like

```
XXOX
XXOX
OXXX
OXXX
```

where `O` is an opening. The message `SPRING IS HERE!` would be placed on the grid as

```
SPRI
NGIS
HERE
ABCD
```

Notice that only alphabetic data is placed on the grid. It would then be encrypted as `RIHANGCDISEBSPRE`. Your job here is to find the *original* message when given the encrypted text (by original message we mean both the original text and and characters used to fill out the message).

Input will consist of one problem instance. The first line will consist of the encrypted text all uppercase letters terminated by end-of-line. Maximum input size will be 64 characters. The remaining lines of input will be the rows of the grille as given in the format used above.

Output should be the original text, written with no spaces on one line.

Examples

On input:

```
LPPEHEHRLMONEHDA
XXOO
XXXO
XOXX
XXXX
```

the output should look like:

```
HELPHELPMERHONDA
```

Spring 1994 Denison Computer Programming Contest
Problem 2
Trip the Lights Fantastic

Fast Phil works the late shift and leaves the parking lot at precisely 2:00 AM every morning. His drive home is over a straight road over which he must pass through n lights. Phil has always wondered the following: given the locations and cycles of each of the n lights, are there velocities he can travel home without ever having to speed up or slow down on account of a red light? You are to write a program to answer this question.

Input will consist of one problem instance. The first line will contain the value n. The greatest value of this will be 6. The next n lines will be of the form

```
l g y r
```

where `l`$<$ 200 is a real number indicating the location of each light (in miles from the parking lot) and `g`, `y` and `r` are the lengths of time (in secs) of the green, yellow and red periods of the traffic light. Phil has learned (from an informant in the Highway Commission) that all n traffic lights start their green period precisely at 2:00 AM.

Your program should find all *integer* speeds (in miles per hour) which can be used for Phil's ride home, i.e. each is a constant speed he can maintain the moment he leaves the parking lot at 2:00 AM until he arrives home (we'll assume Phil has a long driveway to decelerate in) such that he never passes through a red light. He is allowed to pass through a light at the exact moment a yellow light turns to red or a red light turns to green. Since Phil is a relatively law-abiding citizen, you need only consider speeds $\leq$ 100 mph.

Output should consist of a list of all valid integer speeds. Consecutive integer speeds should be specified in interval notation of the form `l-h`, where `l` and `h` are the lowest and highest speeds for the interval. Intervals of the form `l-l` (i.e. an interval of length 1) should just be written as `l`. Intervals should be separated by commas. If there are no valid speeds, your program should output the phrase `You'll have to use the brake, Phil`.

Example

On input:

```
4
1.25 20 10 20
2.3 20 9 15
2.5 15 10 19
4.0 20 8 25
```

the output should look like:

```
6,18,22,29,37,45,60-62,80-90
```

Spring 1994 Denison Computer Programming Contest
Problem 3
All Decked Out

A perfect shuffle is one in which you you first cut the deck exactly in half (if the deck has an even number of cards) and then shuffle the two halves so that the cards alternate between the top half and the bottom half of the deck. For instance, let's suppose our deck has eight cards. Let's identify them by numbering them 0 through 7, starting from the top. First we cut the deck in half. The top half is 0, 1, 2, 3 and the bottom half is 4, 5, 6, 7. Now we shuffle them, in this example 0 is the top card, yielding the deck in order 0, 4, 1, 5, 2, 6, 3, 7. If we repeat this process, the deck would be in order 0, 2, 4, 6, 1, 3, 5, 7. If we repeat this process enough times we will return the deck to its original order.

The example above is called an *outer shuffle* since the top card came from the top half of the deck. As you might guess, an *inner shuffle* is one where the top card comes from the bottom half of the deck. In our example of eight cards, one inner shuffle results in a deck with order 4, 0, 5, 1, 6, 2, 7, 3.

If the deck has an odd number of cards, then one of the halves will contain one more card than the other. The half with more cards will be the one that gives the top card in the shuffle. This will be the top half in the case of an outer shuffle and the bottom half in the case of an inner shuffle. For instance, a deck of five results in an order of 0, 3, 1, 4, 2, with an outer shuffle and 2, 0, 3, 1, 4 with an inner shuffle.

The problem here is: *Given the size of the deck and the type of shuffle to be used (either outer or inner), what is the minimum number of shuffles necessary to return the deck to its original order?*

Input will consist of multiple problem instances. Each instance will be on one line of input of the form n s, where n is an integer from 2 to 100 and is the size of the deck and s is either `i` or `o`, indicating an inner or outer shuffle. There will be exactly one space between n and s. Input terminates on end-of-file.

Output for each problem instance should be one line containing a single integer, giving the number of shuffles necessary.

Examples

On input:

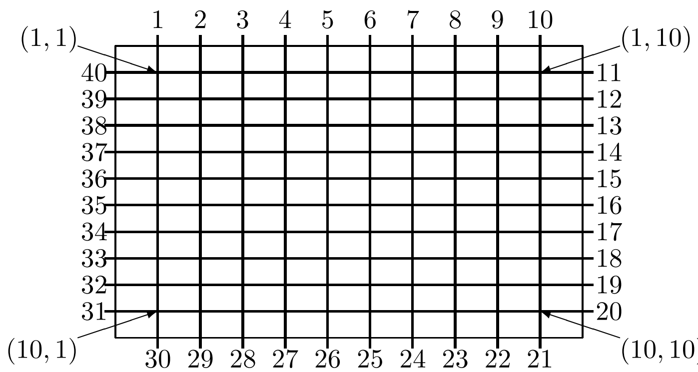
```
4 o
14 o
14 i
```

the output should look like:

```
2
12
4
```

Spring 1994 Denison Computer Programming Contest
Problem 4
Rays to the Limit

The diagram below shows a solid box with 10 vertical and 10 horizontal hollow shafts drilled through it:



The 40 shaft openings and 100 shaft intersections are labeled as shown in the diagram. You should write a program to solve the following problem: a set of balls will be placed at random shaft intersections in the box, and a single light ray will be shown in one of the shaft openings. The light ray will pass through the shaft until it either leaves the box or strikes a ball. When it strikes a ball it splits into two beams which leave the ball at right angles to the incident ray. These rays may themselves be split if they hit another ball, and so on. Given a set of ball locations and an initial light ray, you should list all the shaft openings in which light rays will leave the box.

Input will consist of several problem instances. Each instance will consist of 11 lines of input. The first ten lines will specify the location of the balls; each line will consist of 10 characters, either `'0'` or `'1'`, indicating the absence or presence (respectively) of a ball. A `'1'` in line n, position m, indicates a ball in shaft intersection (n, m). The final line of each problem instance will contain an integer `s` indicating the shaft opening which the initial light ray is shown through. End-of-file will terminate input.

Output should consist of a list of all shaft openings which have light escaping from them. Shaft openings should be listed in increasing order, separated by commas. A blank line should separate output for each problem instance.

Example

On input:

```
0000000000
0000000000
0000001000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0100001000
38
```

the output should look like:

```
2,7,20,29
```

Spring 1994 Denison Computer Programming Contest
Problem 5
Bearing, See?

The object of backgammon is to move your 15 pieces around the board until they are all located somewhere within the last 6 locations on the board (which we will label 6 though 1). At this point the player must *bear off*, that is move the pieces off the board. This is done by the successively rolling two dice and moving the pieces to lower positions based on the values rolled. For the purposes of this problem, the movement of the pieces are determined by the application of the rules below (note that these rules may not be the "best" rules). For each die value, you use the first rule which is applicable. Assuming that the die value if n, then the rules are:

1. remove a piece from the board if it is at position n.
2. move a piece to the lowest possible open position (i.e. a position on the board which has no pieces on it) such that you do not create a new empty position in the original location. In other words, if the empty location is i, then there must be at least 2 piece at location $i+n$ before you can apply this rule and move one piece from $i+n$ to i.
3. move a piece to the lowest possible open position even if you do create a new open position.
4. move the piece in the highest position n positions (which might move it off the board, thus bearing off that piece).

One other rule of backgammon which is important: if doubles are rolled (i.e. both die equal the same value) then it is treated as if *four* dice have been rolled with that value. The following examples may help to clarify the application of these rules:

LOCA-TIONS	ROLL	MOVES	RULES USED	LOCA-TIONS	ROLL	MOVES	RULES USED
6, 5, 1	6-5	remove 6	Rule 1	6, 5, 5	1-5	move from 5 to 4	Rule 2
		remove 5	Rule 1			remove 5	Rule 1
6, 5, 5	5-4	remove 5	Rule 1	4, 3, 2	5-6	remove 4	Rule 4
		move from 5 to 1	Rule 3			remove 3	Rule 4
6, 6, 5, 5	5-4	remove 5	Rule 1	6, 5, 4, 2	3-3	move from 4 to 1	Rule 3
		move from 6 to 2	Rule 2			move from 6 to 3	Rule 3
6, 5, 1	2-3	move from 5 to 3	Rule 3			remove 3	Rule 1
		remove 3	Rule 1			move from 5 to 2	Rule 4

Notice that in the last example, the double 3's were treated as four 3's.

Input will consist of one problem instance. The first line will contain the six values $n_6, n_5, \ldots, n_1$, corresponding to the number of pieces on locations 6 through 1. The sum of these 6 values will always be 15. The remaining lines in the input file will contain the dice rolls, one per line. Each dice roll line will contain two values in the range 1 through 6, inclusive, and will be in no particular order. End-of-file will terminate input.

Your program should read in one dice roll at a time and move the pieces on the board according to the rules listed above. You should use the values in the order they are read in, e.g. if the dice roll is `1 5` then you should apply the `1` first, and then the `5`. If at some point all of the pieces have been moved off of the board, all subsequent dice rolls (if any) should be ignored.

Output should consist of a description of the board after the application of all of the dice rolls. This line will have the same format as the first line of input, i.e. it will contain 6 integers $m_6, \ldots, m_1$ which will indicate the number of pieces remaining on each position. These values should be separated by at least one space.

Example

On input:

```
6 7 0 1 1 0
3 2
2 3
4 1
1 5
1 1
4 4
2 1
```

the output should look like:

```
3 2 0 0 2 0
```