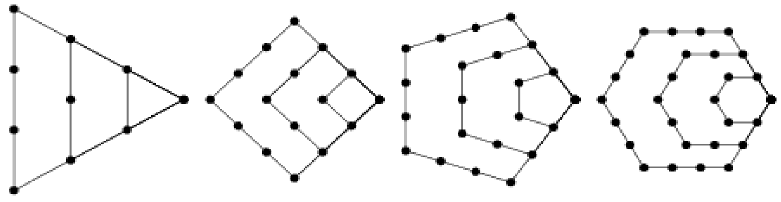
Problem A Poly-polygonal Numbers

A *polygonal* number is a number which can be represented by a regular geometrical arrangement of equally spaced points, where the arrangement forms a regular polygon. Some examples are shown in the figure below.



The first figure shows the first 4 *triangular* numbers 1, 3, 6, 10. The next three show the first four *square, pentagonal* and *hexagonal* numbers, respectively. In general, *k-gonal* numbers are those whose points define a regular k-gon (hence triangular numbers are 3-gonal, square numbers are 4-gonal, etc.). We will define k as an *index* of the polygonal number. For this problem, you are to find numbers which are k-gonal for two or more values of k. We will call these numbers *poly-polygonal*.

Input

Input will consist of multiple problem instances. Each instance will consist of 3 lines. The first line will be a non-negative integer $n \leq 50$ indicating the number of types of polygonal numbers of interest in this problem. Note that this line may be longer than 80 characters. The next line will contain n integers indicating the indices of these polygonal numbers (all distinct and in increasing order). For example, if the first line contained the value 3, and the next line contained the values `3 6 10`, then that problem instance would be interested in 3-gonal, 6-gonal and 10-gonal numbers. Each index k will lie in the range $3 \leq k \leq 1000$. The last line of the problem instance will consist of a single positive integer $s \leq 10000$, which serves as a starting point for the search for poly-polygonal numbers. A value of $n = 0$ terminates the input.

Output

For each problem instance, you should output the next 5 poly-polygonal numbers which are greater than or equal to s. Each number should be on a single line and conform to the following format:

```
num:k1 k2 k3 ...
```

where `num` is the poly-polygonal number, and `k1, k2, k3` ... are the indices (in increasing order) of the poly-polygonal number equal to `num`. A single space should separate each index, and you should separate each problem instance with a single blank line. The judges input will be such that the maximum value for any poly-polygonal number will fit in a **long** variable.

Sample Input

```
10
6 7 8 9 10 11 12 13 14 15
1000
5
3 4 13 36 124
1
0
```

Sample Output

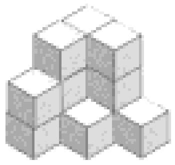
```
1216:9 12
1540:6 10
1701:10 13
2300:11 14
3025:12 15

1:3 4 13 36 124
36:3 4 13 36
105:3 36
171:3 13
1225:3 4 124
```

Problem B Stacking Cubes

Consider the following pattern of positive integers:

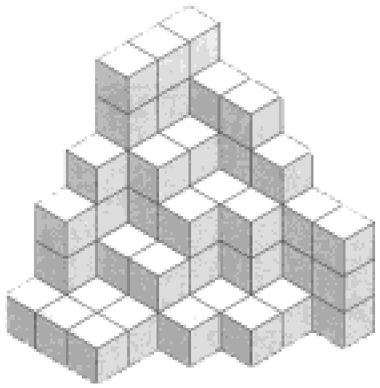
```
3 3 1
3 1
2
```

Note that each row is left-justified and no longer than its preceding row. Also, the entries in each row, when read left to right, are non-increasing and the entries in each column, when read top to bottom are non-increasing. We will call such a pattern a *stacking pattern* (SP) because such a pattern can represent a way of stacking cubes in a corner in the following way: if you consider placing the topmost row and leftmost column against walls, then the SP gives a bird's-eye view of how many cubes are stacked vertically. The SP above represents the following corner stacking:



We will call the wall against the topmost row the *right wall*, and the wall against the leftmost column the *left wall.* Here is another SP and the corner stacking it represents:

```
6 5 5 4 3 3
6 4 3 3 1
6 4 3 1 1
4 2 2 1
3 1 1
1 1 1
```



Note that if you rotate a corner stacking so the left wall becomes the floor and the floor becomes the right wall, you still have a corner stacking. (We will call this a *left rotation.*) Likewise, you would still have a corner stacking if you rotate so the right wall becomes the floor and the floor becomes the left wall. (We will call this a *right rotation.*) So the SP of the left and right rotations of the first SP given above are

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

You should check that both the left and right rotations of the second example SP are identical to the original SP.

Input

This problem will consist of multiple problem instances. Each problem instance will consist of a positive integer $n \leq 11$ indicating the number of rows in the SP that follows. ($n = 0$ indicates the end of input.) The rows of the SP will follow, one per line with entries separated by single spaces, delimited by a trailing 0. (The trailing 0 is, of course, not part of the input data proper and you may assume that each row given has at least one cube.) Each entry in the pattern proper will be a positive integer less than or equal to 20 and there will be no more than 20 entries in any row.

Output

For each input SP you should produce two stacking patterns corresponding to the left rotation and the right rotation (in that order). Rows of the SP should be left-justified with entries separated by a single space. One blank line should separate the left and right rotations of the given SP and two blank lines should separate output for different problem instances.

Sample Input

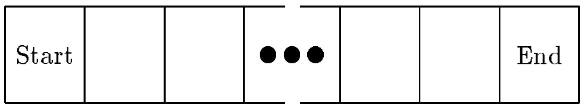
```
3
3 3 1 0
3 1 0
2 0
6
6 5 5 4 3 3 0
6 4 3 3 1 0
6 4 3 1 1 0
4 2 2 1 0
3 1 1 0
1 1 1 0
0
```

Sample Output

```
3 2 1
2 1 1
2 1

3 3 2
2 1 1
1


6 5 5 4 3 3
6 4 3 3 1
6 4 3 1 1
4 2 2 1
3 1 1
1 1 1

6 5 5 4 3 3
6 4 3 3 1
6 4 3 1 1
4 2 2 1
3 1 1
1 1 1
```

Problem C To Bet or Not To Bet

Alexander Charles McMillan loves to gamble, and during his last trip to the casino he ran across a new game. It is played on a linear sequence of squares as shown below.



A chip is initially placed on the Start square. The player then tries to move the chip to the End square through a series of turns, at which point the game ends. In each turn a coin is flipped: if the coin is heads the chip is moved one square to the right and if the coin is tails the chip is moved two squares to the right (unless the chip is one square away from the End square, in which case it just moves to the End square). At that point, any instruction on the square the coin lands on must be followed. Each instruction is one of the following:

1. Move right n squares (where n is some positive integer)
2. Move left n squares (where n is some positive integer)
3. Lose a turn
4. No instruction

After following the instruction, the turn ends and a new one begins. Note that the chip only follows the instruction on the square it lands on after the coin flip. If, for example, the chip lands on a square that instructs it to move 3 spaces to the left, the move is made, but the instruction on the resulting square is ignored and the turn ends. Gambling for this game proceeds as follows: given a board layout and an integer T, you must wager whether or not you think the game will end within T turns.

After losing his shirt and several other articles of clothing, Alexander has decided he needs professional help—not in beating his gambling addiction, but in writing a program to help decide how to bet in this game.

Input

Input will consist of multiple problem instances. The first line will consist of an integer n indicating the number of problem instances. Each instance will consist of two lines: the first will contain two integers m and T ($1 \leq m \leq 50$, $1 \leq T \leq 40$), where m is the size of the board *excluding* the Start and End squares, and T is the target number of turns. The next line will contain instructions for each of the m interior squares on the board. Instructions for the squares will be separated by a single space, and a square instruction will be one of the following: `+n`, `-n`, `L` or `0` (the digit zero). The first indicates a right move of n squares, the second a left move of n squares, the third a lose-a-turn square, and the fourth indicates no instruction for the square. No right or left move will ever move you off the board.

Output

Output for each problem instance will consist of one line, either

```
Bet for. x.xxxx
```

if you think that there is a greater than 50% chance that the game will end in T or fewer turns, or

```
Bet against. x.xxxx
```

if you think there is a less than 50% chance that the game will end in T or fewer turns, or

```
Push. 0.5000
```

otherwise, where `x.xxxx` is the probability of the game ending in T or fewer turns rounded to 4 decimal places. (Note that due to rounding the calculated probability for display, a probability of `0.5000` may appear after the `Bet for.` or `Bet against.` message.)

Sample Input

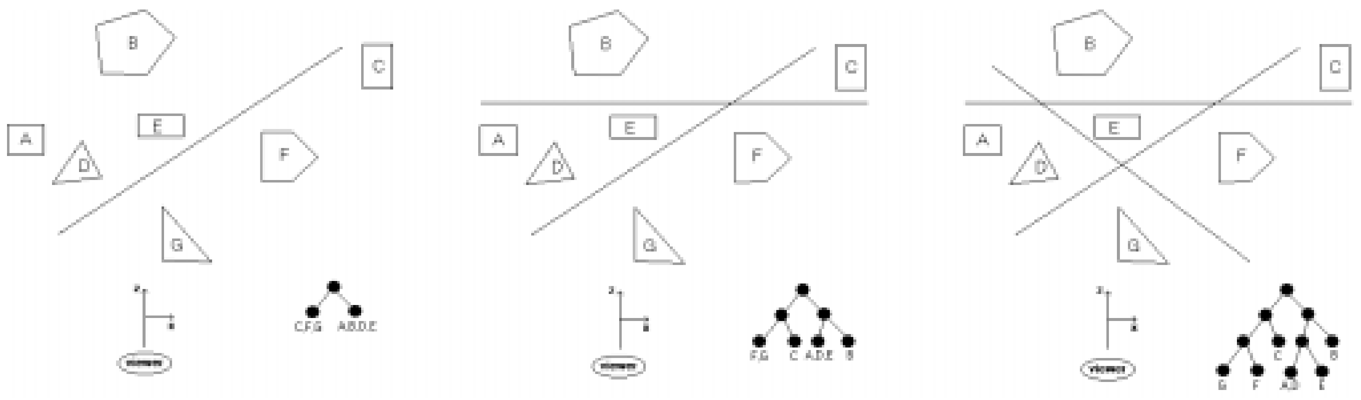
```
5
4 4
0 0 0 0
3 3
0 -1 L
3 4
0 -1 L
3 5
0 -1 L
10 20
+1 0 0 -1 L L 0 +3 -7 0
```

Sample Output

```
Bet for. 0.9375
Bet against. 0.0000
Push. 0.5000
Bet for. 0.7500
Bet for. 0.8954
```

Problem D BSP Trees

When rendering a scene with multiple objects onto a screen, the order in which the objects are drawn is very important. In general, the farther an object is from the screen, the earlier it should be drawn allowing later, closer objects to be drawn on top of them. If two objects do not overlap, the order of drawing is immaterial. A binary space-partitioning (BSP) tree is one type of data structure which attempts to simplify the determination of the ordering of objects. It works as follows. Assume that the screen lies in the xy-plane centered on the z-axis and that the z-axis points away from the user looking at the screen. (For our purposes, assume the user lies near $-\infty$ on the z-axis.) We also assume that all the objects lie on the opposite side of the screen ($z > 0$). The BSP tree is built by placing a series of planes parallel to the y-axis. The first plane divides space into two regions: a region containing the viewer and a region not containing the viewer. We partition all objects in space according to which of these two regions they lie in, and observe that all objects in the region containing the viewer should be drawn after all the objects in the other region. The BSP tree can be viewed at this point as a root with only two children, each child containing one of the partitions. We can now add a second plane, which subdivides the space again. We split each of the two partitions from the first plane in two, making a total of 4 partitions, and the resulting BSP tree now has three levels, with the partitions in the leaves (note that some of these partitions may contain several objects and some may contain none). This process is continued until each partition has at most one object in it, or until some predetermined number of planes has been used. The diagram below gives an example of using 1, 2 and 3 planes (looking down along the y-axis). For simplicity we assume that all objects lie parallel to the z-axis, so we need only deal this 2-d image to determine the BSP tree.



Assuming you have split the partitions correctly, a simple traversal of the BSP tree will give you an appropriate ordering for which to render the objects in the scene. Note in the example above that once a node contains just one object it need not be split as additional planes are added.

Input

Input will consist of one problem instance. The first line will contain a positive integer $n \leq 20$ indicating the number of objects in the scene. The next n lines will contain a description of these objects using the format m $x1$ $z1$ $x2$ $z2 \ldots xm$ zm, where m is the number of vertices in the object and the remaining values are the vertices of the intersection of the object with the xz-plane. All objects will have between 3 and 6 vertices. Objects are assumed to be labeled "A", "B", "C", ... in the order they are defined. Next in the input file will be a positive integer $p \leq 10$ indicating the number of planes used to create the BSP tree. The last p input lines will contain a description of each plane of the form $x1$ $z1$ $x2$ $z2$ representing two points on the intersection line of the plane and the xz-plane. You may assume that no line will intersect any object (including edges and vertices) and that no plane is parallel to the z-axis. All coordinates will be integers.

Output

Output will consist of a single line containing the names of the objects in the order that they should be rendered for the specified BSP tree. In the case when some partition contains two or more objects, you should list the objects in alphabetical order.

Sample Input

```
10
3 65 5 66 5 65 6
3 65 123 66 123 65 124
3 122 176 123 176 122 177
3 56 23 57 23 56 24
3 11 49 12 49 11 50
3 167 111 168 111 167 112
3 57 123 58 123 57 124
3 130 6 131 6 130 7
3 100 85 101 85 100 86
3 11 28 12 28 11 29
10
159 165 -131 -177
-153 -192 -197 158
-77 -86 -98 30
-177 59 146 63
192 -117 92 43
121 -67 -62 -134
41 -81 130 196
95 -185 -89 154
-163 -179 93 175
113 41 -92 -28
```

Sample Output

```
BCGEJFIHDA
```

Problem E Double Trouble

Alice Catherine Morris and her sister Irene Barbara frequently send each other e-mails. Ever wary of interceptions and wishing to keep their correspondence private, they encrypt their messages in two steps. After removing all nonalphabetic characters and converting all letters to upper case, they: 1) replace each letter by the letter s positions after it in the alphabet ($1 \leq s \leq 25$)—we call this a *shift* by s—and then, 2) divide the result of step 1 into groups of m letters and reverse the letters in each group ($5 \leq m \leq 20$). If the length of the message is not divisible by m, then the last k (less than m) letters are reversed. For example, suppose $s = 2$ and $m = 6$. If the plaintext were

```
Meet me in St. Louis, Louis.
```

after removing unwanted characters and changing to upper case we get

```
MEETMEINSTLOUISLOUIS
```

We will call this the *modified plaintext*. We then shift each letter by 2 (`Y` would be replaced with `A` and `Z` would be replaced by `B`, here), getting the intermediate result:

```
OGGVOGKPUVNQWKUNQWKU
```

And finally reverse every group of 6 letters:

```
GOVGGOQNVUPKWQNUKWUK
```

Note the last two letters made up the last reversed group. As is customary, we write the result in groups of 5 letters. So the ciphertext would be:

```
GOVGG OQNVU PKWQN UKWUK
```

Alas, it's not so hard to find the values for s and m when the ciphertext is intercepted. In fact it's even easier if you know a *crib*, which is a word in the modified plaintext. In the above example, `LOUIS` would be a crib. Your job here is to find s and m when presented with a ciphertext and a crib.

Input

Input will consist of multiple problem instances. The first line of input will contain a positive integer indicating the number of problem instances. The input for each problem will consist of multiple lines. The first line of input for a problem will contain the integer n ($20 \leq n \leq 500$) which is equal to the number of characters in the ciphertext. The following lines will contain the ciphertext, all upper case in groups of 5 letters separated by a single space. (The last group of letters may contain fewer than 5 letters.) There will be 10 groups of letters per line, except possibly for the last line of ciphertext. The input line following the last line of ciphertext will contain the crib; a single word consisting of between 4 and 10 (inclusive) upper case characters.

Output

Output will be two integers, `s` and `m` on a line, separated by a single space, indicating the encryption key that produces the crib, where `s` is the shift and `m` is the reversed group size. If there is more than one solution, output the one with smallest `s`. If there is more than one with the same `s`, output the one with smallest `m`. If no such `s` and `m` exist, output the message `Crib is not encrypted`.

Sample Input

```
4
83
FIQMF IISFN QMFIB EOPFH FNQMV PSFIU IZNGP UPEUS BFPEP PEPPE
PPEPN QMFIP EOPIS FIQMF IBSFN QMFBE OPI
RHONDA
105
VDBMN DQDGS LNQEM ZLZRZ RNGVX ZALNA TERZV CZDGD MZQHZ GENKK
KONSC DJHKC KKZAD RZAXZ SNMRH GBHGV RZVDG XZRNS XZKOS ZCNNF
SHFMH
BOMBAY
50
QFNWX YQFNW YSAQX FYNWY XQFNW SXYQF FXNYS AXYQF NASXY QFNAX
HEAVEN
20
GOVGG OQNVU PKWQN UKWUK
LOUIS
```

Sample Output

```
1 6
25 6
Crib is not encrypted.
2 6
```