Problem A: Cut It Out!

Television's *Toddler Time*'s topic taught to toddler Tommy Tiwilliger today was triangles (and the letter T)! Tommy became so enamored by triangles that immediately after the show ended, he grabbed his safety scissors and the nearest sheets of paper he could find and started cutting out his own triangles. After about 15 minutes each paper had one triangular shaped hole cut out of it. But Tommy wasn't finished yet. He noticed he could divide each of the original triangles into two triangles with a single cut starting from one corner. He spent another 15 minutes doing just that. Things would have gone along swimmingly, except that Tommy's mother eventually came into the room and noticed that the original sheets of paper were part of a very important document for a legal case she was working on (involving a lover's triangle). After carefully removing Tommy from the room and counting slowly to 10 (a triangular number), she went about trying to reconstruct the pages after gathering together the now randomly scattered triangles. Your job is to help her by writing a little program to determine which triangles go where (and try to get it done before tomorrow's episode of *Toddler Time* on papier-mâchè).

Input

Each test case will start with an integer $n \leq 20$ indicating the number of holes cut out, followed by the coordinates of the holes, one hole per line. These holes are assumed to be numbered $1, 2, \ldots, n$. Following this will be the coordinates of the $2n$ triangles resulting from the bisections, one triangle per line. These triangles are assumed to be numbered $1, 2, \ldots, 2n$ and are listed in no particular order. The specification of any hole or triangle will have the form $x_1\ y_1\ x_2\ y_2\ x_3\ y_3$ where each x_i and y_i will be to the nearest thousandth and $|x_i|, |y_i| \leq 200$. No two holes will be congruent and no two triangles will be congruent. A value of $n = 0$ will terminate input.

Output

For each test case, you should output the case number and then n lines as follows:

```
Hole 1:  t1a, t1b
Hole 2:  t2a, t2b
...
Hole n:  tna, tnb
```

where `t1a, t1b` are the two triangles which fill hole 1, `t2a, t2b` are the two triangles which fill hole 2, etc. Always print the lower of the two numbers first on any line. Triangles should not be flipped over when filling a hole. Each test case will have a unique solution. Separate the output for each test case with a blank line. Note: when processing the triangles and checking for equality of lengths, angles or trigonometric values, you may assume that two items are equal if they differ by less than 0.01.

Sample Input

```
1
18.691 6.103 21.668 13.709 21.332 25.894
59.388 30.873 55.299 36.186 61.45 22.97
67.828 85.496 60.751 72.752 59.2 67.49
3
18.73 4.012 6.662 7.557 14.035 7.478
14.869 32.398 32.341 31.772 7.522 29.674
25.272 6.868 4.572 2.014 10.487 16.121
26.135 53.073 44.18 50.723 40.31 42.91
86.601 29.95 70.542 17.088 66.77 14.88
90.344 89.528 92.179 88.665 87.99 82.54
39.327 62.11 35.033 57.127 18.14 63.89
37.13 80.202 36.308 75.111 34.28 75.11
14.043 68.482 15.22 55.423 10.42 75.43
0
```

Sample Output

```
Case 1:
Hole 1: 1, 2

Case 2:
Hole 1: 3, 5
Hole 2: 2, 6
Hole 3: 1, 4
```

Problem G: Spy Cam

An overhead camera has snapped a photo of the papers on a bureaucrat's desk. Many of the papers overlap, but, because the bureaucrat in questions is a bit of a neatness freak, all papers are aligned with their edges parallel to the edges of the desk. As a preliminary step in analyzing these papers, other programs have used cues of paper color, edges, etc., to label each pixel in the image according to which sheet of paper is visible in that particular location. The result will look something like this

```
...aaaaaaaa.dd
...aaaaaaaa.ee
.ccaaaaaaaa...
.ccaaaabbaa...
.ccaaaabbaa...
.ccaaaabbaa...
.ccaaaaaaaa...
..............
```

A '.' denotes the desktop, and alphabetic labels 'a', 'b', 'c', etc., denote distinct pieces of paper. These labels are assigned in an arbitrary order but are dense (no letters are "skipped" during the labeling).

Assume that each paper is rectangular, and that no important information is being lost "between" the pixels. Assume also that the camera has recorded the entire desktop and that there are no papers that are completely hidden from view.

For each piece of paper (ordered by label), determine if the visual evidence proves that the entire sheet of paper is visible or if it is at least possible that a portion of the paper is hidden beneath another sheet.

Input

Input will consist of several snapshots.

Each snapshot begins with a line containing two integers, R and C, denoting the number of rows and columns of pixels in the snapshot. For each snapshot, $1 \leq R, C \leq 40$.

The R and C values are followed by R lines, each containing C characters. Those characters will be lower-case alphabetic characters or a period ('.'). As noted earlier, the alphabetic characters denote visible portions of distinct sheets of paper and the periods denote portions of the underlying desktop.

All snapshots in the input will correspond to a valid arrangement of rectangular sheets of paper. There will be no impossible arrangements.

The final snapshot will be followed by a line containing two zeros separated by a space.

Output

For each snapshot, print one line of output. That line will begin with the phrase “Uncovered:” followed by a single space. Then, on the same line, list the papers that are definitely completely visible, in order of character code, with no separating spaces.

Example

Input:

Given the input

```
3 3
..b
.cc
dca
8 14
...aaaaaaaa.dd
...aaaaaaaa..e
.ccaaaaaaaa...
.ccaagabbaa...
.ccafgffffa...
.ccaagabbaa...
.ccaaaaaaaa...
.cc...........
0 0
```

the output would be

Output:

```
Uncovered: a
Uncovered: cdg
```

Problem H: We've Got Chemistry, Babe

Barry Liam is a chemistry student at Uranium University (good ol' UU), and does pretty well except in one area — balancing chemical equations. As I'm sure you recall from your last chemistry class, a chemical reaction can be represented by a chemical equation, with the reactants on the left and the products on the right, separated by an arrow. For example, the chemical equation to form aluminum oxide out of aluminum and oxygen is

$$\mathrm{Al} + \mathrm{O_2} \rightarrow \mathrm{Al_2O_3}.$$

However, the above equation is not balanced - there is one aluminum (Al) atom on the left hand side and two on the right, and there are two oxygen (O) atoms on the left and three on the right. A properly balanced version of this equation would be:

$$4\mathrm{Al} + 3\mathrm{O_2} \rightarrow 2\mathrm{Al_2O_3}.$$

Note that not only are all the coefficients positive integers, but the greatest common divisor of all of them is 1 (two requirements when balancing). While Barry understands all this, he is flummoxed when asked to do it, especially when asked to balance equations such as:

$$\mathrm{Fe_2(SO_4)_3} + \mathrm{KSCN} \rightarrow \mathrm{K_3Fe(SCN)_6} + \mathrm{K_2SO_4}.$$

Notice here that atoms in parentheses are all multiplied by the number that follows (so in this equation there are 12 oxygen atoms on the left and six carbon (C) atoms on the right, for example). Also, a symbol appears at most once in any reactant or product. Barry would like to ask his roommate for help, but he's a Ballet major who thinks balancing equations have something to do with dancing en pointe (actually, he thought Barry said "Balanchine equations"). I guess Barry will have to look for a new major, unless you can help him.

Input

Each test case will consist of a single line starting with two positive integers r and p indicating the number of reactants and products (the maximum value for each is 10). There will then follow the r reactants. Each reactant will consist of one or more terms, where each term is either a single chemical symbol, a single chemical symbol followed by an integer ≥ 2, or a set of parentheses around a term, followed by an integer ≥ 2. All chemical symbols consist of one or two alphabetic characters, only the first of which is capitalized. Following the r reactants will be the p products which follow the same rules as the reactants. There will be no more than 20 different chemical symbols in any test case. A line containing two zeros will terminate the input.

Output

For each test case, you should output the appropriate coefficients needed to balance the equation, listed in the order that the reactants and products are given. Use a single space to separate the values, and preface each with the case number, using the format shown below. If an equation cannot be balanced correctly or uniquely (see the last sample input, which has solutions 1 2 1 1, 1 3 1 2, 1 4 1 3, etc.), you should output the word `No`.

Sample Input

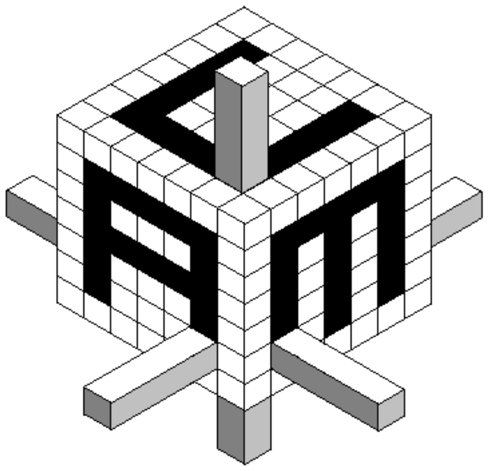
```
2 1 Al O2 Al2O3
2 2 Fe2(SO4)3 KSCN K3Fe(SCN)6 K2SO4
1 1 CO2 CO
2 2 A B AB B
0 0
```

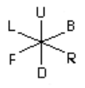
Sample Output

```
Case 1: 4 3 2
Case 2: 1 12 2 3
Case 3: No
Case 4: No
```

Problem C: Maze

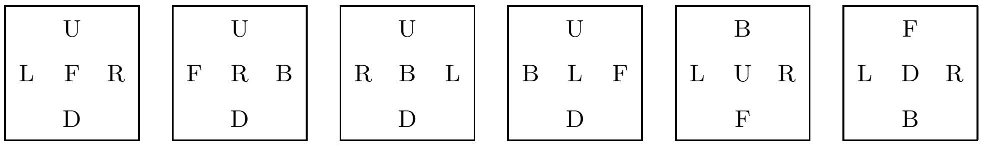
We are all familiar with conventional mazes laid out on a 2-D grid. A 3-D maze can be constructed as follows: Consider a hollowed out cube aligned along the x, y and z axes with one corner at (0,0,0) and the opposite corner at $(n-1, n-1, n-1)$. On each face of the cube is a 2-D maze made by removing a subset of $1 \times 1 \times 1$ cubes from the face (no edge cubes are removed). The object of the maze is to move a marker located inside the cube from an initial location of (1,1,1) to the final destination of $(n-2, n-2, n-2)$. However, attached to this marker are 6 rods, each protruding through one face of the cube. The movement of these rods is constrained by the 2-D mazes on the faces. The picture below gives an example of a $7 \times 7 \times 7$ maze. Note that this maze is not physically realizable since some faces (e.g., the front face containing the letter "A") have cubes that are disconnected from the edges of the face. Such mazes are allowed in this problem.





The black regions indicate open spaces where the rods can move. The figure to the right specifies the possible directions that the rods can move (Forward, Back, Left, Right, Up, Down) and also defines the labels for the six sides of the cube. In the maze above, the rods are shown in their initial position centered at (1,1,1). From here they can not move Forward, Backward, Right, Left or Down, but they can move Up (assuming there are open spaces for the two back rods to move to).

To specify a cube, a description of each face must be given. For this problem, the order and orientations of each face are given in the diagram below.



The first square represents the Forward face oriented so that the shared edge with the Up face is on top and the shared edge with the Right face is on the right, the second square represents the Right face oriented with the shared edge with the Up face on top and the shared edge with the Back face on the right, and so on. Your job is to solve such mazes in the minimum number of moves.

Input

Each test case will start with a single line containing the value of n, where n lies between 4 and 30, inclusive. Next will come descriptions of each face in the order and orientation shown above. The description of each face will consist of n lines each containing one string of length n. The characters in the string will be either a blank or `'X'`, indicating either an empty or full square, respectively. The last test case will be followed by a line containing 0.

Output

Output will consist of one line for each test case. Each line will contain a description of a minimum-move solution that moves the marker from cell (1,1,1) to cell $(n-2, n-2, n-2)$. Moves are either `F`, `B`, `L`, `R`, `U` or `D`. In case of a tie, choose the sequence of moves which is lexicographically first, where we consider `F` $<$ `B` $<$ `L` $<$ `R` $<$ `U` $<$ `D`. All mazes will have solutions.

Sample Input

```
7
XXXXXXX
X     X
X XXX X
X     X
X XXX X
X XXX X
XXXXXXX
XXXXXXX
X     X
X X X X
X X X X
X X X X
X X X X
XXXXXXX
XXXXXXX
X     X
X     X
X     X
X     X
X     X
XXXXXXX
```

(continued in next column)

```
XXXXXXX
X     X
X X X X
X     X
X X X X
X     X
XXXXXXX
XXXXXXX
X XXX X
X XXX X
X XXX X
X XXX X
X     X
XXXXXXX
XXXXXXX
X     X
X XXX X
X X X X
X XXX X
X     X
XXXXXXX
0
```

Sample Output

```
UULLLLUUBBBB
```

E

Problem C: Chain Code

Source: `chaincode.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

In a black and white (bi-level) image, collections of connected black pixels can be defined as foreground or objects, while white can be thought of as background. Each set of connected black pixels can be completely described by listing the positions of the pixels on its boundary in counterclockwise order, starting at some arbitrary point on the boundary. This list of pixels can, in turn, be represented simply as the direction to the next one in the list. This list of directions is called the chain code of the object, and describes the shape of the object precisely while being position independent.

There are 8 possible directions from one pixel to an adjacent pixel, and while assigning these numbers is arbitrary, **figure 1** shows the standard convention. The direction 0 means "to the right of", 2 "means immediately above", and 1 is at 45 degrees, bisecting 0 and 2, and so on.

Two black pixels are considered to be adjacent if the square of the distance between them is less than or equal to 2. This is based on a standard graphics coordinate system having a pixel at each integer coordinate. Two pixels are connected if a contiguous path of adjacent pixels can be traced between them. A connected region is a set of black pixels in which all members are connected to each other. A boundary pixel of a connected region (from now on just a region) is a pixel within the region that has at least one neighbor (in the four compass directions) that is not black. For this problem, you may assume that there are no “holes” in the region, so that there is only one boundary of the region.

The chain code of a region can start at any pixel on the boundary. It proceeds by finding the next adjacent pixel on the boundary in a counter-clockwise direction, saving the direction (0-7) in an output buffer, and then continuing the process from the new pixel. When we arrive at the starting pixel again, the chain code is complete. The output buffer contains a set of direction values which comprise the chain code itself, and from which the original set of pixels can be recreated starting at any pixel position in an image.

As an example, a chain code for the region in **figure 2** is `446567001232`. The chain code describes the shape of the region unambiguously, although its position is completely unknown. Shape related measures such as perimeter and area (number of pixels in the region) can be determined directly from the chain code description alone. You are to write a program that calculates the area of a connected region given only the chain code.

E

Input

The input will be a collection of chain code strings, one per line. Each chain code contains at most 1000000 characters. You may assume that each chain code describes a valid region, and does not describe a boundary that intersects itself.

Output

For each chain code in the input, the output will be the area of the region (i.e. the number of pixels belonging to it), each printed on its own line.

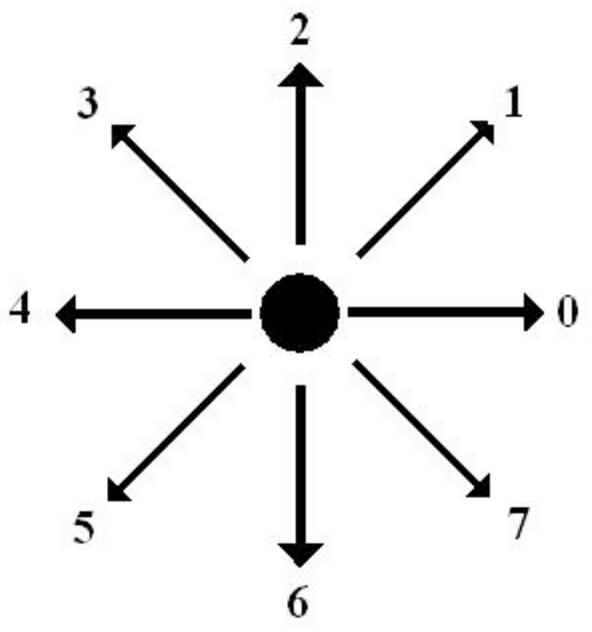


Figure 1: Directions.

E

Sample input

```
446567001232
6024
```

Sample Output

```
19
4
```

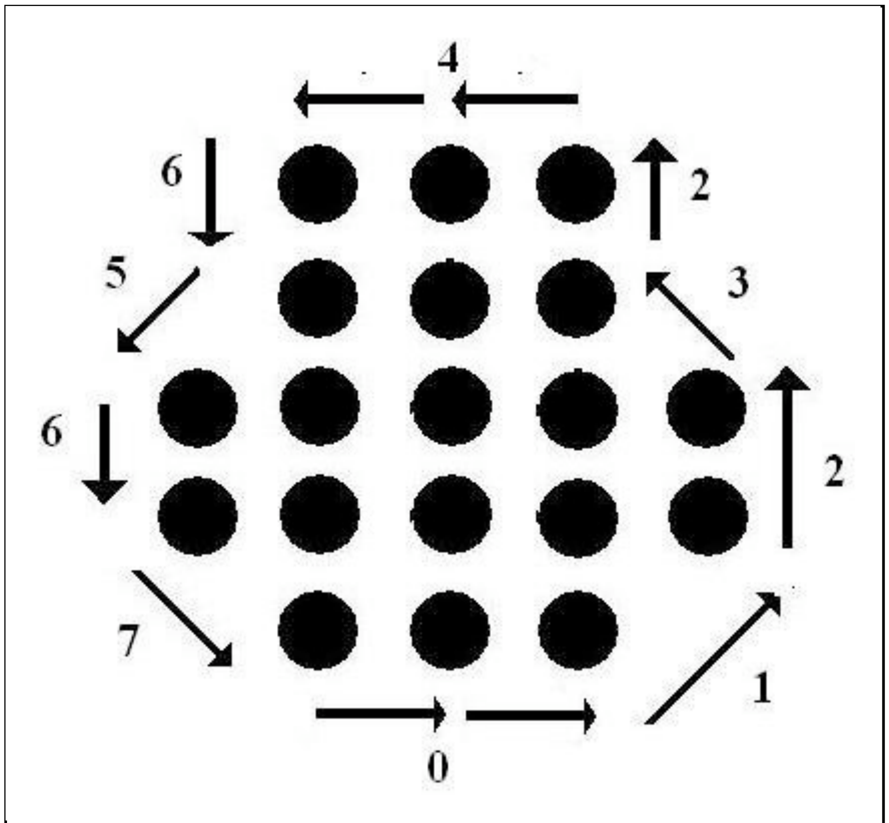


Figure 2: Example area.

E

Problem F: Pro-Test Voting

Old Bob Test is currently running for Mayor in the Hamlet of Kerning. Kerning is divided up into a number of precincts (numbered 0, 1, 2, ...), and after extensive polling by his crack staff, Bob knows the current percentage of voters in each precinct who plan to vote for him. Needless to say, he would like to increase these percentages in all precincts, but he has limited funds to spend. Based on past results, the effects of spending on any precinct obey the following equation:

$$F_p = I_p + \left(\frac{M}{10.1 + M}\right)\Delta$$

where I_p is the current percentage of pro-Test voters, Δ is the maximum increase in this percentage possible, M is the amount of money spent in the precinct, in integer multiples of \$1, and F_p is the final expected percentage. What Bob needs to know is the best way to spend his money to maximize the number of votes he can get.

Input

The first line of each test case contains two integers m and n, representing the amount of money Bob has to spend (in dollars) and the number of precincts. The maximum value for both of these is 100. After this will be n lines of the form N I_p Δ, all positive integers, which contain information on each precinct: N is the population of the precinct and I_p and Δ are as described above. The value of N will be less than 10000. The first of these lines refers to precinct 0, the next to precinct 1, and so on. A line containing 0 0 follows the last test case. NOTE: When calculating the number of pro-Test voters in a precinct, you should first perform a double calculation of F_p using the formula above, then multiply this percentage by the population N and round to get the final result.

Output

Output for each test case should consist of two lines. The first should contain the case number followed by the maximum number of votes Bob can obtain through optimum spending. The second line should list each precinct and the amount of money which Bob should spend there. The format for each precinct should be `precinctnum:money`, and each such pair should be separated by 1 blank. In the case where there is more than one way to spend Bob's money that yields the maximum number of votes, give the one that spends the most on precinct 0. If there is more than one with the same spent on precinct 0, take the one that spends the most on precinct 1, etc.

Sample Input

```
100 2
3000 45 15
2000 60 10
100 3
3000 45 15
2000 60 10
4000 20 8
100 3
3000 45 15
2000 60 10
4000 20 7
100 3
3000 45 15
2000 60 10
4000 20 6
100 3
3000 45 15
2000 60 10
4000 20 5
0 0
```

Sample Output

```
Case 1: 3095
0:64 1:36
Case 2: 4101
0:42 1:24 2:34
Case 3: 4070
0:45 1:27 2:28
Case 4: 4040
0:46 1:27 2:27
Case 5: 4011
0:46 1:27 2:27
```

Problem D: Not One Bit More

Start with an integer, N_0, which is greater than 0. Let N_1 be the number of ones in the binary representation of N_0. So, if $N_0 = 27$, $N_1 = 4$.

In general, let N_i be the number of ones in the binary representation of N_{i-1}. This sequence will always converge to one.

For any starting number, N_0, let $K(N_0)$ be the minimum i such that N_i is one. For example, if $N_0 = 31$, then $N_1 = 5$, $N_2 = 2$, $N_3 = 1$, so $K(31) = 3$.

Given a range of consecutive numbers, and a value X, how many numbers in the range have a $K(\ldots)$ value equal to X?

Input

There will be several test cases in the data file. Each test case will consist of three integers on a single line:

LO HI X

where LO and HI ($1 \leq \text{LO} \leq \text{HI} \leq 10^{18}$) are the lower and upper limits of a range of integers, and X ($0 \leq X \leq 10$) is the target value for $K(\ldots)$.

The data file will end with a line with three 0s.

Output

For each test case, output a line with a single integer, representing the number of integers in the range from LO to HI (inclusive) which have a $K(\ldots)$ value equal to X in the input.

Example

Input:

Given the input

```
31 31 3
31 31 1
27 31 1
27 31 2
1 4 1
1023 1025 1
1023 1025 2
0 0 0
```

the output would be

Output:

```
1
0
0
3
2
1
1
```

H

Problem A: Parenthesis

Source: `parenthesis.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

To a computer, there is no difference between the expression **`(((x)+(y))(t))`** and **`(x+y)t`**; but, to a human, the latter is easier to read. When writing automatically generated expressions that a human may have to read, it is useful to minimize the number of parentheses in an expression. We assume expressions consist of only two operations: addition (+) and multiplication (juxtaposition), and these operations act on single lower-case letter variables only. Specifically, here is the grammar for an expression **`E`**:

```
E : P | P '+' E
P : F | F  P
F : V | '(' E ')'
V : 'a' | 'b' | .. | 'z'
```

The addition (**`+`**, as in **`x+y`**) and multiplication (juxtaposition, as in **`xy`**) operators are associative: **`x+(y+z)=(x+y)+z=x+y+z`** and **`x(yz)=(xy)z=xyz`**. Commutativity and distributivity of these operations should not be assumed. Parentheses have the highest precedence, followed by multiplication and then addition.

Input

The input consists of a number of cases. Each case is given by one line that satisfies the grammar above. Each expression is at most 1000 characters long.

Output

For each case, print on one line the same expression with all unnecessary parentheses removed.

H

Sample input

```
x
(x+(y+z))
(x+(yz))
(x+y(x+t))
x+y+xt
```

Sample Output

```
x
x+y+z
x+yz
x+y(x+t)
x+y+xt
```

Problem I: Aronson

Source: **`aronson.{c,cpp,java}`**
Input: **`console {stdin,cin,System.in}`**
Output: **`console {stdout,cout,System.out}`**

Aronson's sequence a_k is a sequence of integers defined by the sentence "t is the first, fourth, eleventh, ... letter of this sentence.", where the ... are filled in appropriately so that the sentence makes sense. The first few values are 1, 4, 11, 16, 24, 29, 33, 35, 39, ... Note the non-letter characters and spaces are not considered in the formulation of the sequence. When $k \le 100000$, it turns out that $a_k \le 1000000$.

To formulate the sequence, you must be able to write the ordinal numbers in English. The ordinal numbers are first, second, third, ..., while the cardinal numbers are one, two, three, ... It is easiest to define the ordinals in terms of the cardinals, so we describe these first.

A cardinal number less than twenty is written directly from the first two columns of **table 1** (3→three, 17→seventeen, etc.). A cardinal number greater than or equal to twenty, but less than one hundred is written as the tens part, along with a nonzero ones part (40→ forty, 56→fifty six, etc). A cardinal number greater than or equal to one hundred, but less than one thousand, is written as the hundreds part, along with a nonzero remainder (100→one hundred, 117→one hundred seventeen, 640→six hundred forty, 999→nine hundred ninety nine). A cardinal number greater than or equal to one thousand, but less than one million, is written as the thousands part, along with a nonzero remainder (12345→twelve thousand three hundred forty five). An ordinal number is written as a cardinal number, but with the last word ordinalized using the columns three and four of **table 1**.

Some example ordinal numbers are 3^{rd}→third, 56^{th}→fifty sixth, 100^{th}→one hundredth, and 12345^{th}→twelve thousand three hundred forty fifth.

Input

The input consists of a number of cases. Each case is specified by a positive integer **`k`** on one line (**`1 ≤ k ≤ 100000`**). The sequence of **`k`** values will be non-decreasing. The input is terminated by a line containing a single **`0`**.

Output

For each **`k`**, print the value of a_k on one line. The values of a_k will be at most **`1000000`**.

Sample input

```
1
3
9
0
```

Sample Output

```
1
11
39
```

n	cardinal	nth	ordinal
1	one	1st	first
2	two	2nd	second
3	three	3rd	third
4	four	4th	fourth
5	five	5th	fifth
6	six	6th	sixth
7	seven	7th	seventh
8	eight	8th	eighth
9	nine	9th	ninth
10	ten	10th	tenth
11	eleven	11th	eleventh
12	twelve	12th	twelfth
13	thirteen	13th	thirteenth
14	fourteen	14th	fourteenth
15	fifteen	15th	fifteenth
16	sixteen	16th	sixteenth
17	seventeen	17th	seventeenth
18	eighteen	18th	eighteenth
19	nineteen	19th	nineteenth
20	twenty	20th	twentieth
30	thirty	30th	thirtieth
40	forty	40th	fortieth
50	fifty	50th	fiftieth
60	sixty	60th	sixtieth
70	seventy	70th	seventieth
80	eighty	80th	eightieth
90	ninety	90th	ninetieth
100	hundred	100th	hundredth
1000	thousand	1000th	thousandth

Table 1: Translation table.

I

Problem E: Abstract Extract

When writing articles, there is usually an abstract section which summarizes the entire article. We are experimenting with an automatic abstract generation algorithm. The algorithm reads in an article and prepares an abstract that summarizes the entire article. The abstract is formed by combining the "topic sentences" extracted from consecutive paragraphs.

For the purposes of this problem,

- An *article* consists of one or more paragraphs.
- A *paragraph* is a maximal sequence of non-empty lines.
- A *sentence* is a maximal sequence of characters within a paragraph that begins with a non-whitespace character, that ends with a '.' (period), '?', or '!', and that does not contain any other occurrences of '.', '?', or '!'.
- A *word* is a maximal sequence of alphabetic characters within a sentence.

The term *maximal* in the definitions above is intended to convey the idea that we are only interested in the longest sequences that match the definition, not in any of their subsequences. For example, the sentence "How now, brown cow?" contains four words. "now" is a word in this sentence, but "no", "ow", etc., are not words because they are not maximal – they are subsequences of a larger sequence of alphabetic characters.

A *topic sentence* for a paragraph is the single sentence in the paragraph that best describes the paragraph's content. For our purposes, we will select the earliest sentence in the paragraph that maximizes the number of distinct words in S that also occur in any following sentence within the same paragraph.

Paragraphs with fewer than three sentences are ignored and will not contribute to the abstract.

When comparing words for distinctness, changes in upper/lower case are ignored. For example, the sentence "See what I see." contains three distinct words, not four.

Input

Input will consist of one or more articles. Each article is terminated by a line containing only "***" or "******". The latter string ("******") indicates the end of the entire input set.

- Each article will contain one or more paragraphs. Each paragraph will consist of one or more non-empty lines, and is terminated by an empty line or by the "***" or "******" markers described above.
- No article will be longer than 500 lines; No line will contain more than 150 characters. No word will contain more than 50 characters.
- The only whitespace characters in the input will be blanks (ASCII 32) and line terminators.

Output

For each document, print the abstract followed by a line containing “======” (six equal signs).

Each abstract will be formed from the sequence of topic sentences, selected as described above, in the order that they occur in the input document. Each sentence shall be printed exactly as it appears in the input and should be followed by a line break.

Example

Input:

Given the input

```
Hello world!  Everyone should greet the entire world in a friendly
manner. Wouldn’t that make the world a more friendly place?

This is not a sentence
***
From: The Wright Brothers’ Aeroplane.

By Orville and Wilbur Wright

In the field of aviation there were two schools. The first,
represented by such men as Professor Langley and Sir Hiram Maxim,
gave chief attention to power flight; the second, represented by
Lilienthal, Mouillard, and Chanute, to soaring flight. Our sympathies
were with the latter school, partly from impatience at the wasteful
extravagance of mounting delicate and costly machinery on wings
which no one knew how to manage, and partly, no doubt, from the
extraordinary charm and enthusiasm with which the apostles of soaring
flight set forth the beauties of sailing through the air on fixed
wings, deriving the motive power from the wind itself.

We began our active experiments at the close of this period, in October,
1900, at Kitty Hawk, North Carolina. Our machine was designed to be
flown as a kite, with a man on board, in winds from 15 to 20 miles an
hour. But, upon trial, it was found that much stronger winds were
required to lift it. Suitable winds not being plentiful, we found it
necessary, in order to test the new balancing system, to fly the machine
as a kite without a man on board, operating the levers through cords
from the ground. This did not give the practice anticipated, but it
inspired confidence in the new system of balance.

In the summer of 1901 we became personally acquainted with Mr. Chanute.
When he learned that we were interested in flying as a sport, and not
with any expectation of recovering the money we were expending on it, he
gave us much encouragement. At our invitation, he spent several weeks
```

```
with us at our camp at Kill Devil Hill, four miles south of Kitty Hawk,
during our experiments of that and the two succeeding years. He also
witnessed one flight of the power machine near Dayton, Ohio, in October,
1904.
******
```

the output would be

Output:

```
Everyone should greet the entire world in a friendly
manner.
======
The first,
represented by such men as Professor Langley and Sir Hiram Maxim,
gave chief attention to power flight; the second, represented by
Lilienthal, Mouillard, and Chanute, to soaring flight.
Our machine was designed to be
flown as a kite, with a man on board, in winds from 15 to 20 miles an
hour.
When he learned that we were interested in flying as a sport, and not
with any expectation of recovering the money we were expending on it, he
gave us much encouragement.
======
```

J

K

Problem D: Task

Source: `task.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

In most recipes, certain tasks have to be done before others. For each task, if we are given a list of other tasks that it depends on, then it is relatively straightforward to come up with a schedule of tasks that satisfies the dependencies and produces a stunning dish. Many of us know that this can be solved by some algorithm called toplogical sort.

But life is not so easy sometimes. For example, here is a recipe for making pizza dough:

1. Mix the yeast with warm water, wait for 5 to 10 minutes.
2. Mix the the remaining ingredients 7 to 9 minutes.
3. Mix the yeast and the remaining ingredients together for 10 to 15 minutes.
4. Wait 90 to 120 minutes for the dough to rise.
5. Punch the dough and let it rest for 10 to 15 minutes.
6. Roll the dough.

In this case, tasks 1 and 2 may be scheduled after the first minute (we always spend the first minute to read the recipe and come up with a plan). The earliest task 3 may be started is at 8 minutes, and task 4 may start at 18 minutes after the start, and so on. This recipe is relatively simple, but if some tasks have many dependent tasks then scheduling can become unmanageable. Sometimes, the recipe may in fact be impossible to execute. For example, consider the following abstract recipe:

1. task 1
2. after task 1 but within 2 minutes of it, do task 2
3. at least 3 minutes after task 2 but within 2 minutes of task 1, do task 3

In this problem, you are given a number of tasks. Some tasks are related to another based on their starting times. You are asked to assign a starting time to each task to satisfy all constraints if possible, or report that no valid schedule is possible.

K

Input

The input consists of a number of cases. The first line of each case gives the number of tasks **n**, (**1 ≤ n ≤ 100**). This is followed by a line containing a non-negative integer **m** giving the number of constraints. Each of the next **m** lines specify a constraint. The two possible forms of constraints are:

```
task i starts at least A minutes later than task j
task i starts within A minutes of the starting time of task j
```

where **i** and **j** are the task numbers of two different tasks (**1 ≤ i, j ≤ n**), and **A** is a non-negative integer (**A ≤ 150**). The first form states that task **i** must start at least **A** minutes later than the start time of task **j**. The second form states that task **i** must start no earlier than task **j**, and within **A** minutes of the starting time of task **j**. There may be multiple constraints involving the same pair of tasks. Note that **at least** and **within** include the end points (i.e. if task 1 starts at 1 minute and task 2 starts at 4 minutes, then task 2 starts at least 3 minutes later than task 1, and within 3 minutes of the starting time of task 1).

The input is terminated by **n = 0**.

Output

For each case, output a single line containing the starting times of task 1 through task n separated by a single space. Each starting time should specify the minute at which the task starts. The starting time of each task should be positive and less than 1000000. There may be many possible schedules, and any valid schedule will be accepted. If no valid schedule is possible, print **Impossible.** on a line instead.

K

Sample input

```
6
10
task 3 starts at least 5 minutes later than task 1
task 3 starts within 10 minutes of the starting time of task 1
task 3 starts at least 7 minutes later than task 2
task 3 starts within 9 minutes of the starting time of task 2
task 4 starts at least 10 minutes later than task 3
task 4 starts within 15 minutes of the starting time of task 3
task 5 starts at least 90 minutes later than task 4
task 5 starts within 120 minutes of the starting time of task 4
task 6 starts at least 10 minutes later than task 5
task 6 starts within 15 minutes of the starting time of task 5
3
4
task 2 starts at least 0 minutes later than task 1
task 2 starts within 2 minutes of the starting time of task 1
task 3 starts at least 3 minutes later than task 2
task 3 starts within 2 minutes of the starting time of task 1
0
```

Sample Output

```
3 1 8 18 108 118
Impossible.
```

K