Rujia Liu's Present 5

Developing

Simplified Softwares

January 15, 2012
UVa Online Judge

Problems

My apologies go to non-Chinese people, who can find quite a few alien words in the problem titles. If you're interested, "Shi Xiong Bang Bang Mang" means "Help me, Brother!", "Jiandan" means easy, "Kingdie" means something like "tricky".

I know that some of these problems are very tricky or complex, so I decided to provide some additional data that make your life (a little bit) easier. You can download them on the contest website.

Thanks Mingjing Xiaoliu for problem G, and H, Yechen Li for problem B, Peichao Zhang for problem F, Zhuohua Chen for problem A, G and I, Feng Chen for problem J, Youzhi Bao for problem A, C, E, and the Mua series.

Problem D, F and H are adapted from previous Chinese Olympiad in Informatics contests, with test data greatly enhanced. Thanks for the authors of the original problems and reference materials (Yifei Zhang, Cailiang Liu, Shi Li, Rong Ge, Tiancheng Lou, Weidong Hu). And finally, the authors of the Lua programming language: that's my favorite embedded language ***J***

Hello, everyone! My name is Rujia Liu. I used to do a lot of problem solving and problemsetting, but after graduated from Tsinghua University, I'm spending more and more time on my company **L**

(You may realized that the paragraph above is copied from the texts of my 3rd and 4th contest, but that's me, lazy me.)

This time, my contest is all about developing software's. Well, I know that it's very difficult to finish a good software within contest time, but I'm trying to allow the contestants to enjoy the happiness of coding, not just algorithms.

As usual, don't hesitate to write emails to me (rujia.liu@gmail.com) during the contest for any reason (e.g. the problems are unclear to you, or you suspect that the judge data might be wrong). Please remember that my goal is to help you learn more, not prevent you from solving problems.

This contest is about software developing, so don't forget that you can always use google. I even gave some hints the problems that contain external resources for you to learn. So... try your best and solve these problems!

Best regards,
Rujia Liu

A. A Typical Homework
(a.k.a Shi Xiong Bang Bang Mang)

Hi, I am an undergraduate student in institute of foreign languages. As you know, C programming is a required course in our university, even if his/her major is far from computer science. I don't like this course at all, as I am not good at computer and I don't wanna even have a try of any programming!! But I have to do the homework in order to pass :(Sh… Could you help me with it? Please keep secret!! I know that you won't say NO to a poor little girl, boy. :)

Task

Write a Student Performance Management System (SPMS).

Concepts

In the SPMS, there will be at most 100 students, each has an SID, a CID, a name and four scores (Chinese, Mathematics, English and Programming).

- SID (student ID) is a 10-digit number
- CID (class ID) is a positive integer not greater than 20.
- Name is a string of no more than 10 letters and digits, beginning with a capital letter. Note that a name cannot contain space characters inside.
- Each score is a non-negative integer not greater than 100.

Main Menu

When you enter the SPMS, the main menu should be shown like this:

```
Welcome to Student Performance Management System (SPMS).

1 - Add
2 - Remove
3 - Query
4 - Show ranking
5 - Show Statistics
0 - Exit
```

Adding a Student

If you choose 1 from the main menu, the following message should be printed on the screen:

```
Please enter the SID, CID, name and four scores. Enter 0 to finish.
```

Then your program should wait for user input. The input lines are always valid (no invalid SID, CID, or name, exactly four scores etc), but the SID may already exist. In that case, simply ignore this line and print the following:

```
Duplicated SID.
```

On the other hand, multiple students can have the same name.

You should keep printing the message above until the user inputs a single zero. After that the main menu is printed again.

Removing a Student

If you choose 2 from the main menu, the following message should be printed on the screen:

```
Please enter SID or name. Enter 0 to finish.
```

Then your program should wait for user input, and remove all the students matching the SID or name in the database, and print the following message (it's possible xx=0):

```
xx student(s) removed.
```

You should keep printing the message above until the user inputs a single zero. After that the main menu is printed again.

Querying Students

If you choose 3 from the main menu, the following message should be printed on the screen:

```
Please enter SID or name. Enter 0 to finish.
```

Then your program should wait for user input. If no student matches the SID or name, simply do nothing, otherwise print out all the matching students, in the same order they're added to the database. The format is similar to the input format for "adding a student", but 3 more columns are added: rank (1st column), total score and average score (last two columns). The student with highest total score (considering all classes) received rank-1, and if there are two rank-2 students, the next one would be rank-4.

You should keep printing the message above until the user inputs a single zero. After that the main menu is printed again.

Showing the Ranklist

If you choose 4 from the main menu, the following message should be printed on the screen:

```
Showing the ranklist hurts students' self-esteem. Don't do that.
```

Then the main menu is printed again.

Showing Statistics

If you choose 5 from the main menu, show the statistics, in the following format:

```
Please enter class ID, 0 for the whole statistics.
```

When a class ID is entered, print the following statistics. Note that "passed" means to have a score of at least 60.

```
Chinese
Average Score: xx.xx
Number of passed students: xx
Number of failed students: xx

Mathematics
Average Score: xx.xx
Number of passed students: xx
Number of failed students: xx
```

```
English
Average Score: xx.xx
Number of passed students: xx
Number of failed students: xx

Programming
Average Score: xx.xx
Number of passed students: xx
Number of failed students: xx

Overall:
Number of students who passed all subjects: xx
Number of students who passed 3 or more subjects: xx
Number of students who passed 2 or more subjects: xx
Number of students who passed 1 or more subjects: xx
Number of students who failed all subjects: xx
```

Then, the main menu is printed again.

Exiting SPMS

If you choose 0 from the main menu, the program should terminate.

Note that course scores and total score should be formatted as integers, but average scores should be formatted as a real number with exactly two digits after the decimal point.

Input

There will be a single test case, ending with a zero entered in the main menu screen. The entire input will be valid. The size of input does not exceed 10KB.

Output

Print out everything as stated in the problem description. You should be able to play around this little program in your machine, with a keyboard and a screen. However, both the input and output may look silly when they're not mixed, as in the keyboard-screen case.

Sample Input

```
1
0011223344 1 John 79 98 91 100
0022334455 1 Tom 59 72 60 81
0011223344 2 Alice 100 100 100 100
2423475629 2 John 60 80 30 99
0
3
0022334455
John
0
5
1
2
0011223344
0
5
0
4
```

```
0
```

Output for Sample Input

```
Welcome to Student Performance Management System (SPMS).

1 - Add
2 - Remove
3 - Query
4 - Show ranking
5 - Show Statistics
0 - Exit

Please enter the SID, CID, name and four scores. Enter 0 to finish.
Please enter the SID, CID, name and four scores. Enter 0 to finish.
Please enter the SID, CID, name and four scores. Enter 0 to finish.
Duplicated SID.
Please enter the SID, CID, name and four scores. Enter 0 to finish.
Please enter the SID, CID, name and four scores. Enter 0 to finish.
Welcome to Student Performance Management System (SPMS).

1 - Add
2 - Remove
3 - Query
4 - Show ranking
5 - Show Statistics
0 - Exit

Please enter SID or name. Enter 0 to finish.
2 0022334455 1 Tom 59 72 60 81 272 68.00
Please enter SID or name. Enter 0 to finish.
1 0011223344 1 John 79 98 91 100 368 92.00
3 2423475629 2 John 60 80 30 99 269 67.25
Please enter SID or name. Enter 0 to finish.
Welcome to Student Performance Management System (SPMS).

1 - Add
2 - Remove
3 - Query
4 - Show ranking
5 - Show Statistics
0 - Exit

Please enter class ID, 0 for the whole statistics.
Chinese
Average Score: 69.00
Number of passed students: 1
Number of failed students: 1

Mathematics
Average Score: 85.00
Number of passed students: 2
Number of failed students: 0

English
Average Score: 75.50
```

```
Number of passed students: 2
Number of failed students: 0

Programming
Average Score: 90.50
Number of passed students: 2
Number of failed students: 0

Overall:
Number of students who passed all subjects: 1
Number of students who passed 3 or more subjects: 2
Number of students who passed 2 or more subjects: 2
Number of students who passed 1 or more subjects: 2
Number of students who failed all subjects: 0

Welcome to Student Performance Management System (SPMS).

1 - Add
2 - Remove
3 - Query
4 - Show ranking
5 - Show Statistics
0 - Exit

Please enter SID or name. Enter 0 to finish.
1 student(s) removed.
Please enter SID or name. Enter 0 to finish.
Welcome to Student Performance Management System (SPMS).

1 - Add
2 - Remove
3 - Query
4 - Show ranking
5 - Show Statistics
0 - Exit

Please enter class ID, 0 for the whole statistics.
Chinese
Average Score: 59.50
Number of passed students: 1
Number of failed students: 1

Mathematics
Average Score: 76.00
Number of passed students: 2
Number of failed students: 0

English
Average Score: 45.00
Number of passed students: 1
Number of failed students: 1

Programming
Average Score: 90.00
Number of passed students: 2
```

```
Number of failed students: 0

Overall:
Number of students who passed all subjects: 0
Number of students who passed 3 or more subjects: 2
Number of students who passed 2 or more subjects: 2
Number of students who passed 1 or more subjects: 2
Number of students who failed all subjects: 0

Welcome to Student Performance Management System (SPMS).

1 - Add
2 - Remove
3 - Query
4 - Show ranking
5 - Show Statistics
0 - Exit

Showing the ranklist hurts students' self-esteem. Don't do that.
Welcome to Student Performance Management System (SPMS).

1 - Add
2 - Remove
3 - Query
4 - Show ranking
5 - Show Statistics
0 - Exit
```

Hint

When formatting a floating-point number such as Average Score, a good way to prevent floating-point errors is to add a small number (like 1e-5 in this problem). Otherwise, 80.315 would be printed as 80.31 if the floating-point error makes it 80.31499999…

B. Big Decimal Calculator

Languages like Java have Big Decimal libraries supporting basic arithmetic operations like additions, subtractions, multiplications and divisions. However, scientific problems usually requires mathematical functions like sin, cos, etc. In this problem, you're to write a Big Decimal Calculator.

There are 15 commands:

- Function with two arguments: add, sub, mul, div, pow, atan2
- Functions with only one argument: exp, ln, sqrt, asin, acos, atan, sin, cos, tan

For trigonometric functions, angles are always in radians.

Input

There will be at most 100 lines. Each line begins with the function name, followed by arguments, then the precision p ($1 <= p <= 50$). Each argument is formatted as one or more digits, followed by a dot ".", then by one or more digits. The integer part cannot be omitted, but the last two parts can be omitted together. There can be an optional negative sign before an argument. Each input number contains at most 20 digits. In function pow, exp, ln and sqrt, all the arguments are strictly positive; In function asin and acos, the integer part of the arguments are always zero.

Output

For each line, print the answer, rounded to p decimal places (Don't use scientific notation!). It is guaranteed that the result will be a finite number, and its integer part will not exceed 10 digits.

Sample Input

```
add 1.357 4.6279 10
sub 1.357 4.6279 10
mul 1.357 4.6279 10
div 1 103 30
pow 12.2 12.15 20
atan2 2.45 1.77 30
exp 10.98 50
ln 21.065 50
sqrt 2 40
asin 0.81 30
acos 0.47 35
atan 0.618 40
sin 3.1415 25
cos 2.0113 50
tan 1.78987 30
```

Output for Sample Input

```
5.9849000000
-3.2709000000
6.2800603000
0.00970873786407766990291262135 9
15822384813181.618723820016834840 36
0.94516227746721596739490262805 2
58688.554274617556019463290914429885325512373423266514 23
3.04761289543097985660178308429069456872534888053139
1.41421356237309504880168872420969807856 97
0.94415211515415595047769777565 3
```

```
1.08150554878078090500864808815790029
0.553549764032731654457264234348267164633 1
0.000092653589660671440566 2
-0.426395110189181767033110065367874038710859211613 47
-4.491415179046604916096895094786
```

Note

You may notice that this problem is not *language-neutral*. I mean, some programming languages have advantages over some others. This is intentional: real-world software development is like this. Choosing programming languages, libraries and the overall architectures can be vital.

There are quite a lot of literatures on this topic (for example, *Fast multiprecision evaluation of series of rational numbers* by Bruno Haible , Thomas Papanikolaou), but that's overkill for *this* problem. The time limit for this problem is rather large, and the test cases are quite gentle: the goal of this problem is to write a *working* program, not a *perfect* one, so try to write a concise code, which is usually faster to write and easier to debug.

For a much more practical literature, look at this: http://www.tc.umn.edu/~ringx004/sidebar.html, that small article presents how to translate arguments to make the series expansion converge faster. Though you will not find the whole solution, you'll have some nice ideas.

C. Calculating Yuan Fen

Yuanfen (http://en.wikipedia.org/wiki/Yuanfen) is a Chinese term that is hard to understand for people in other countries. Roughly speaking, yuanfen means the pre-determined "binding force" that links two people (usually two lovers) together. Although it is a blind faith, many people, especially girls like to calculate it.

Unfortunately, my girlfriend is one of them. One day, she asked me, "Sweetie, shall we find out our yuanfen?" Oh, I really hate that question, but I cannot reject it... Luckily, I'm a programmer, so the only thing I need to do is to find a seemingly good algorithm and write a yuanfen calculator. After several hours' searching in the web, I decided to implement the following popular yuanfen algorithm:

Step 1: Pick up the name abbreviations of the couple and concatenate them. For example, if the couple named Jiang Yun Fan and Tang Yu Rou, the concatenation of abbreviations is JYFTYR.

Step 2: Replace each letter with a number string. For some predefined *positive* integer ST, replace A with ST, and B with ST+1, C with ST+2, …, Z with ST+25. For example, if ST=81, A should be replaced with 81, B should be replaced with 82, …, Z will be replaced by 106. In the case above, JYFTYR will be replaced by 901058610010598.

Step 3: Repeat the following: add up each pair of consecutive digits, and write down the last digit of each sum. It's not difficult to see that each time we perform this action, the number of digits is decreased by 1. When the number string is exactly 100, or has no more than 2 digits, the process ends. The current number is the yuanfen between the couple. In the case above, the process is as follows:

```
901058610010598
91153471011547
0268718112691
284589923850
02937815135
2120596648
332545202
65799722
1268694
384453
12898
3077
374
01
```

So if ST=81, Jiang Yun Fan and Tang Yu Rou's yuanfen is only 1!

Too bad! I know my girlfriend very well. I know that even the result is as high as 99, she'll still be unhappy. Could you find the value of ST such that the yuanfen between my sweetheart and I is 100?

Input

There will be at most 50 test cases. Each case contains a string of at least four and at most ten capital letters.

Output

For each test case, print the smallest *positive* integer ST (note that ST should not be zero). If it does not exist or larger than 10000, print a string "`:(`" (without quotes.

Sample Input

```
JYFTYR
ABCDEF
YTHHLS
YTHLML
LYXM
JYFLY
CBTZX
LXYZLE
LXYLYR
QWERTY
```

Output for Sample Input

```
148
634
:(
910
96
4284
631
850
149
2277
```

Disclaimer

Don't be sad if the result of you and your sweetie is larger than 10000. That's no big deal.

D. Digit Patterns

We construct R-expression in the following way:

1. 0, 1, 2, ..., 9 and 0*, 1*, ..., 9* are R-expressions
2. if A and B are R-expressions, so do (A), A+B, AB and (A)*.
   - (*union*) A+B matches all the strings *s* such that either A or B (or both) matches *s*.
   - (*concatenation*) AB matches all the strings in the form s_1s_2 (the concatenation of s_1 and s_2), where A matches s_1 and B matches s_2.
   - (*closure*) (A)* matches all the strings in the form $s_1s_2s_3...s_k$ (k>=0), where A matches every s_i. (Note s_1, s_2, ... don't have to be equal to each other)

R-expressions can only be constructed by rule 1 and 2. In this problem, an R-expressions will not match the empty string. Note that "concatenation" has higher priority than "or", so 11+22 is interpreted as (11)+(22), not 1(1+2)2.

Given a text T, you're to find all position "matching point" *p*, such that R matches a substring of T, ending at position *p* (positions start from 1). For example, if R = "1(2+3)*4", T = "012345", there is exactly one matching point 5, because T matches 1234, which is ending at position 5.

Input

There are at most 40 test cases. Each test case begins with an integer n (1<=n<=10) and an R-expressions (length not exceeding 500). The next line contains the text (length not exceeding 10^7). Both the R-expression and the text only uses the first *n* digits (i.e., 0, 1, ..., *n*-1). It is guaranteed that the R-expression will not match the empty string. The size of input does not exceed 20MB.

Output

For each test case, output a single line containing the matching points, in ascending order, in one line. It is guaranteed that there is at least one matching point for each test case.

Sample Input

```
6 1(2+3)*4
012345
2 00*(10+100)*
00100
```

Output for Sample Input

```
5
1 2 4 5
```

Hint

This problem is hard. You need to know some theory behind regular expressions, not just how to use them. Please make sure your program can pass the test cases in the gift package in the contest website.

E. Excessive Space Remover

How do you remove consecutive spaces in a simple editor like notepad in Microsoft Windows? One way is to repeatedly "replace all" two consecutive spaces with one space (we call it an action). In this problem, you're to simulate this process and report the number of such "replace all" actions.

For example, if you want to remove consecutive spaces in "A very big   joke.", you need two actions:

"A very big   joke."**à**"A very big joke."**à**"A very big joke."

Input

The input contains multiple test cases, one in a separate line. Each line contains letters, digits, punctuations and spaces (possibly leading spaces, but no trailing spaces). There will be no TAB character in the input. The size of input does not exceed 1MB.

Output

For each line, print the number of actions that are required to remove all the consecutive spaces.

Sample Input

```
A very  big   joke.
         Goodbye!
```

Output for Sample Input

```
2
4
```

Explanation

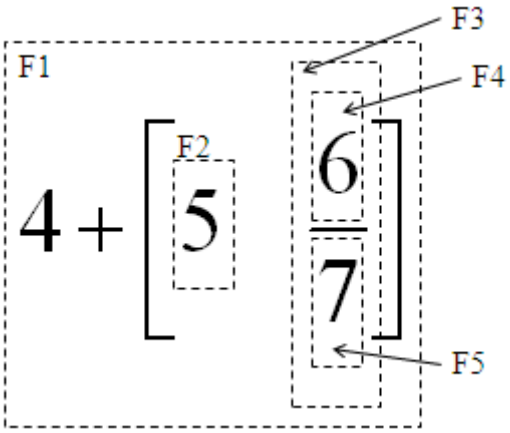
If you can't see clearly, here is the sample input, after replacing spaces with underscores:

```
A*very**big***joke.
*********Goodbye!
```

F. Formula Editor

In this problem, you're to write a formula editor. Technically, a formula is an expression, which is a sequence of elements. There are 3 kinds of elements: basic elements (arithmetic operator, parenthesis, digit and letters), matrices and fractions (discussed below).

The editor builds an invisible box for the expression being edited. Since the cells of a matrix are also expressions, each cell has a box enclosing it. Similarly, the denominator and numerator of a fraction are both expressions, so each of them has a box.



In the expression above, there are 5 boxes. F1 is encloses the whole expression, F2 and F3 enclose two matrix cells, F4 encloses the numerator and F5 encloses the denominator.

It's not difficult to see that boxes are nested. If box A directly contain box B, we say box A is the parent box of box B (for example, F1 is the parent of F2 and F3, and F3 is the parent of F4 and F5; If box A and box B have the same parent box, we say they're sibling boxes (for example, F4 and F5 are siblings and F2 and F3 are siblings, too).

Cursor Movement

At any time, the cursor is always directly contained in a box. It might be to the left of all the elements in the box (i.e. at head position), to the right of all the elements (i.e. at tail position), or between two consecutive elements. If the cursor is between element x and y, and x is to the left (so y is to the right), we say the cursor's immediate left element is x, and immediate right element is y.

The cursor supports six kinds of movements: HOME, END, and four arrow directions. Let A be the inner-most box that contains the cursor, then

Home(End): place the cursor at the head(tail) position of A (it's still directly in A!).
Up(Down): If A has a sibling box to its up(down) direction, place the cursor at the head position of B, otherwise check A's parent box (if A's parent box has a sibling...). If none of A's ancestor boxes have such a sibling box, do nothing for this command.

Left(Right): There are four cases.

- If the cursor is at the head(tail) position of A, then place the cursor at the tail(head) position of A's left(right) sibling B. If there is no such B, place the cursor directly in the A's parent box C, to the immediate left(right) of A.
- If the cursor's immediate left(right) element is a fraction, place the cursor at the tail(head) position of the numerator.

- If the cursor's immediate left(right) element is a matrix with n rows and m rows, then place the cursor to the tail(head) position of the box at row [n/2] and column 1(column m).
- If the cursor's immediate left(right) element is a basic element, then place the cursor to the immediate left(right) of that element.

Output Formatting

The formula is output in ASCII format. Each box is formatted as an ASCII rectangle (though most of them are spaces), which is the horizontal concatenation of the ASCII rectangles of all its elements (inner rectangles). Inner rectangles are aligned with their base lines (will be explained shortly). There are no spaces between consecutive elements, and there are no extra lines or columns between the inner rectangles and the boundary of the outer rectangle.

Each element is also formatted as an ASCII rectangle:

- Basic elements occupy exactly one line, which is also the base line. We use three characters " - " (that is, one space at each side) to represent the subtraction operator, and all other elements is formatted as a single character.
- A matrix is formatted as follows: first, format all the boxes of the matrix cells, and then arrange them into a matrix. The boxes in the same row are aligned according to the base lines, and the boxes in the same column are aligned horizontally. Consecutive rows are separated by an empty line, and consecutive columns are separated by an empty column. Finally, a pair of square brackets is added to the both sides of base lines for each row. The base line of the whole matrix is the base line of the center row if there are an odd number of rows. Otherwise, the base line of the whole matrix is the center empty line.
- A fraction is formatted as follows: first, format the denominator's box and the numerator's box, then draw a horizontal line (a sequence of '-'characters) between these two, which is also the base line of the whole fraction. The width of the line is the larger value of two widths, plus 2 (i.e. one more '-'on both sides). The denominator and the numerator are aligned horizontally.

Note that when aligning rectangles horizontally, we fix the position of the widest rectangle, and try to move other rectangles so that their center columns have the same horizontal position of the widest rectangle. When this is not possible (i.e. for rectangles whose width has a different parity from the widest one), we can move these troublesome rectangles 0.5 unit to the left of the ideal position, like this:

```
 XXXX
------
 XXX
```

There is a special case: if the expression is empty, the formatted ASCII rectangle is an empty line, i.e. its width is zero, but its height is one. Naturally, the empty line is the base line.

Input Handling

The input of the editor is already converted to a sequence of command strings. For each string:

- If it contains a single character, it must be a basic element. Insert that element at the cursor, and move the cursor to its immediate right.
- If it is `Matrix` of `Fraction`, then insert an 1x1 matrix or an empty fraction at the cursor, then move the cursor right once. Note that, before moving the cursor right, the new matrix/fraction is to immediate right of the cursor.
- If it is `AddRow` or `AddCol`, insert a row/column before the box that *directly* contains the cursor and place the cursor in the new row/column of the same column/row. If the box is not a matrix cell, check its parent box, until a matrix cell is found. If there is no matrix cell containing the cursor, ignore the command.
- If it is one of Home, End, Left, Right, Up, Down, follow the cursor movement rules above.

Input

There will be several test cases. Each test case contains a series of command strings, one in each line, ending with command **Done** followed by an empty line. Note that the resulting formula is not necessarily mathematically correct. There will be at most 1000 commands in each test case, and the whole input size does not exceed 200KB.

Output

For each test case, print the final formula in ASCII form. Print the case number in the first line, then an empty line after the formula. Don't print any trailing spaces, but don't omit empty lines (e.g. there is an empty line at the end of sample output 3).

Sample Input

```
-
5
Fraction
1
Down
6
Right
*
Matrix
AddCol
AddCol
1
Right
2
Right
3
Right
*
Matrix
AddRow
AddRow
1
Down
2
Down
3
Done

1
+
Fraction
1
Down
1
+
Fraction
1
Down
1
+
Fraction
1
```

```
Down
x
Up
Up
Right
Right
Home
Up
Done

Fraction
a
Done

Matrix
Fraction
a
Down
b
Matrix
Fraction
c
Down
d
AddRow
e
Done
```

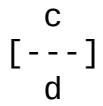
Output for Sample Input

```
Case 1:
                [1]
     1
 - 5---*[1 2 3]*[2]
     6
                [3]

Case 2:
       1
1+-----------
        1
   1+-------
         1
      1+---
         x

Case 3:
 a
---


Case 4:
    a
[--------]
   [ e ]
  b
```



Hint

This problem is tricky. Please make sure your program can pass the test cases in the gift package in the contest website. Don’t forget that clarification requests to my email address are always welcome.

G. Game of 999

In this problem, you're to implement an automatic solver, to the game of 999: *Nine Hours, Nine Persons, Nine Doors*. However, the rules might be a bit different from the original game, so please read carefully!

Disclaimer: *No no no, please do NOT read the wiki for an introduction, if you want to try this awesome game yourself, because that page contains spoilers!!! However, don't skip this problem! The information below has nothing to do with the plot, and will be given to the players at the beginning of the game, so you can still enjoy this game after solving this problem.*

There is a mysterious maze, with n rooms and m corridors connecting them. Some corridors have a big door with a digit (1~9) written on it. Each corridor is one-way, so each door can be opened in exactly one side. Nine people, who numbered 1 to 9, are standing in room 1 when the game begins. The goal of this game is to escape the maze from the exit, which is located in room n.

There are some rules to follow:

- Each door can be used exactly once. When the door is closed again, it's locked forever.
- Each door can be opened by a group of 3~5 people. However, the sum of the numbers of these people should have a digit root equal to the digit written on the door (The digit root of an integer can be obtained by repeatedly sum up all its digits, until it becomes a single digit). For example, people 3, 5, 6, 8 can open door 4, because the digit root of 3+5+6+8=22 is 2+2=4.

Please maximize the number of people who reached room n (the exit). Note that it's allowed to visit a room multiple times, including room n. However, once you escaped, you cannot go back into the maze again.

Input

There are multiple test cases. Each test case begins with two integers n, m ($2 <= n <= 10$, $1 <= m <= 10$). Each of the next m lines contains 3 integers u, v and d to describe a corridor from room u to room v, with a door of digit d. If $d=0$, that means there is no door in this corridor. There can be multiple corridors connecting the same ordered pair of rooms, but no corridor can connect a room to itself.

Output

For each test case, print the maximum number of people who can escape, followed by all possible combinations of the people who escaped. Each combination is string of digits, representing the escaped people. The digits within each combination should be sorted in ascending order, and all the combinations should also be sorted in ascending order (lexicographically).

Sample Input

```
2 1
1 2 9
5 10
1 2 4
1 2 5
2 3 3
2 3 7
2 3 8
3 4 1
3 4 2
3 4 6
4 5 9
4 5 9
3 3
1 2 1
2 3 2
3 2 0
3 4
1 2 1
2 3 2
2 3 2
3 2 0
4 3
1 2 1
2 3 2
3 4 3
4 3
1 2 1
2 3 2
3 4 6
```

Output for Sample Input

```
5 12348 12357 12456 12789 13689 14589 14679 15678 23589 23679 24579
24678 34569
34578
9 123456789
4 1235 1379 1469 2369 2459 2567 3467
5 12358 12367 13789 23689 25678
0
3 123 249 267
```

Hint

If you got time limit exceeded, that usually means you should change your algorithm or use some techniques to avoid unnecessary calculations. Don’t make your solution too complicated, though. It’s not intended to be a hard problem. Only very conventional techniques are involved.

H. Heap Manager

In this problem, you're to implement a heap manager.

There are n memory units in the heap, whose address range from 0 to n-1. Each memory unit is either free or occupied. If there are k consecutive free memory units a, a+1, a+2, ..., a+k-1, we call it a free memory slice of length k, starting from address a.

There will be some processes which use the manager to allocate memory. We use a triple (t, m, p) to represent a process who requests for a memory slice of length m at time t, and needs p time units to run. Let t' be the time when this process successfully allocated memory, then the memory slice will be free at time t'+p.

Processes will be sorted in ascending order of t, and no two processes are allocating memory simultaneously. For each process (t, m, p), we do the following:

- If there is a free memory slice of length m at time t, it is allocated to the process. If there are more than one such slice, use the one with smallest starting address.
- If there is no such slice, place the process in a waiting queue. *Whenever* (for example, just after freeing some memory) there is a suitable memory slice for the first process in the queue, we remove the process from the queue and allocate memory for it. Other processes in the queue will not be considered until they become the first process in the queue. Note that new requests are processed only when the first process in the queue cannot allocate memory (or the queue is empty).

Input

There are multiple test cases. Each test case begins with two integers n ($10 <= n <= 10^9$) and b ($0 <= b <= 1$), where n is the number of memory cells, and b is whether or not the events should be printed. There will be no more than 200,000 lines followed, each containing three integers t, m, p ($m <= n$, $0 <= t <= 10^9$, $0 < p <= 10^9$). The processes are sorted in increasing order of t, and no two processes have identical t. The process list in each test case terminates with three zeros. The size of input does not exceed 10MB.

Output

For each test case, first print all the events, in the order they happened, if b=1 (if b=0, the events should not be printed). Each event is formatted as "T i a", that means at time T, the i-th process (counting from 1) has successfully allocated a memory slice beginning at a. Then two lines contains two integers (one for each line), the time that all the processes have finished, and the number of processes that are ever placed in the queue. Print an empty line after each test case.

Sample Input

```
10 1
1 3 10
2 4 3
3 4 4
4 1 4
5 3 4
0 0 0
4 1
0 3 5
1 1 4
2 2 2
```

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

Output for Sample Input

```
1 1 0
2 2 3
4 4 7
5 3 3
8 5 7
12
2

0 1 0
1 2 3
5 3 0
5 4 2
5 6 3
7 5 0
8
3

0 1 0
1 2 3
3 4 3
5 3 0
5 5 2
6 6 2
9
3
```

I. Item-Based Recommendation

In this problem, you're to implement a simple recommendation system. There are *n* users, each of them rated some of the *m* movies.

For example, *n*=7, *m*=6, the ratings are shown in the following table:

	M1	M2	M3	M4	M5	M6
U1	2.5	3.5	3.0	3.5	2.5	3.0
U2	3.0	3.5	1.5	5.0	3.5	3.0
U3	2.5	3.0		3.5		4.0
U4		3.5	3.0	4.0	2.5	4.5
U5	3.0	4.0	2.0	3.0	2.0	3.0
U6	3.0	4.0		5.0	3.5	3.0
U7		4.5		4.0	1.0	

We can see that there are 3 movies that user 7 haven't watched: M1, M3 and M6.
Question: Which one do we recommend?

One of the most popular methods to solve this is collaborative filtering. For example, we can:

- Step 1: Look for users who share the same rating patterns with the active user (the user whom the prediction is for).
- Step 2: Use the ratings from those like-minded users found in step 1 to calculate a prediction for the active user.

This is known as user-based collaborative filtering. Alternatively, item-based collaborative filtering invented by Amazon.com (users who bought x also bought y), proceeds in an item-centric manner:

- Step 1: Build an item-item matrix determining relationships between pairs of items.
- Step 2: Using the matrix, and the data on the current user, infer his taste.

This is what we use in this problem.

Building the Matrix

The similarity matrix has *m* rows and *m* columns. The element in row *i* and column *j*, denoted by S(*i*,*j*), is the similarity of movie *i* and movie *j*. To calculate S(*i*,*j*), we first calculate the "rating difference" for each user who rating both movie *i* and movie *j*, then let *x* be the sum of the squares of these differences, then S(*i*,*j*)=1/(1+*x*).

For example, to calculate S(2,3), we first find out the rating differences: |3.5-3.0|=0.5, |3.5-1.5|=2.0, |3.5-3.0|=0.5, |4.0-2.0|=2.0, then $x=0.5^2+2.0^2+0.5^2+2.0^2=8.5$, so $S_{2,3}=1/(1+8.5)=0.105$.

The complete similarity matrix of the example above is (the matrix is symmetric so we omit some elements):

	M1	M2	M3	M4	M5	M6
M1	1	0.222	0.222	0.091	0.400	0.286
M2		1	0.105	0.167	0.051	0.182
M3			1	0.065	0.182	0.154
M4				1	0.053	0.103
M5					1	0.148
M6						1

Making Recommendations

Once we have the similarity matrix, it's easy to make recommendations. Suppose we want to make recommendations for user *u*, then the (predicted) score for each "unwatched" movie is simply the *weighted average* of his ratings of the "watched" movies.

To be more specific, suppose user *u* has watched *k* movies m_1, m_2, …, m_k, then the score of an unwatched movie *i* equals to:

$$(S(i,m_1)*rating(m_1) + S(i,m_2)*rating(m_2) + \ldots + S(i,m_k)*rating(m_k)) / (S(i,m_1)+ S(i,m_2)+\ldots+ S(i,m_k))$$

For example, the score of movie 6 for user 7 is:

$$(S(6,2)*4.5+S(6,4)*4.0*S(6,5)*1.0) / (S(6,2)+S(6,4)+S(6,5)) = 3.183$$

Actually, this score is higher than movie 1 and movie 3, so we recommend movie 6 to user 7.

Note that in the formula above, if the denominator is zero, which means movie *i* is not at all similar to any movie he watched, we should not recommend this movie.

Input

There will be only a single test case. The first line contains 3 integers, *n*, *m* and *c* (1<=*n*<=50, 1<=*m*<=200, 1<=*c*<=*nm*). Each of the next c lines contains two integers *i*, *j* (1<=*i*<=*n*, 1<=*j*<=*m*), and a real number *r* between 0 and 5, that means user *i*'s rating of movie j is *r*. Nobody will rate the same movie twice. Then there will be several lines, each containing an integer *u*. That means we need to recommend 10 movies to user *u*. Every user has a least one unwatched movie that is somewhat similar to his watched movie.

Output

For each request, print the top 10 recommended movies: the movie number, and the score (to three digits after the decimal point), in descending order to score. The input is carefully designed so that the final output will not be changed due to floating-point errors. If there are less than 10 unwatched movies that are somewhat similar to his watched movies, simply display them all (but still sorted). Print a blank line after each user.

Sample Input

```
7 6 35
1 1 2.5
1 2 3.5
1 3 3.0
1 4 3.5
1 5 2.5
1 6 3.0
2 1 3.0
2 2 3.5
2 3 1.5
2 4 5.0
2 6 3.0
2 5 3.5
3 1 2.5
3 2 3.0
3 4 3.5
3 6 4.0
```

```
4 2 3.5
4 3 3.0
4 6 4.5
4 4 4.0
4 5 2.5
5 1 3.0
5 2 4.0
5 3 2.0
5 4 3.0
5 6 3.0
5 5 2.0
6 1 3.0
6 2 4.0
6 6 3.0
6 4 5.0
6 5 3.5
7 2 4.5
7 5 1.0
7 4 4.0
7
7
```

Output for Sample Input

```
Recommendations for user 7:
6 3.183
3 2.598
1 2.473

Recommendations for user 7:
6 3.183
3 2.598
1 2.473
```

J. (Jiandan) Mua(I) - Lexical Analyzer

In this problem-series, you're to implement a subset of the Lua language (version 5.1), called mini-lua (mua). This is one of Rujia Liu's experimental languages, mainly for implementing algorithms, not real-world programs.

This is the first problem in the series, which requires you to write a lexical analyzer (lexer), i.e. split the input program into a stream of **tokens** (defined below).

There are six kinds of tokens in mini-lua:

RESERVED: The reserved words, listed below. Note that mini-lua is case-sensitive, so AND is not a reserved word. Note that not all of them are actually used in mini-lua, but we want any valid mini-lua program to be valid in Lua.

```
and       break     do        else      elseif
end       false     for       function  if
in        local     nil       not       or
repeat    return    then      true      until     while
```

NUMBER: There are two kinds of numbers:

- *Integers:* Decimal integers consist of one or more digits (0-9), and hexadecimal integers consists of a prefix (0x or 0X), then followed by one or more hexadecimal digits(0-9, a-f, case-insensitive). Note that leading zeros are simply ignored (e.g. 0123 is the same as 123).
- *Floating-point numbers:* Always in decimal notation, e.g. 1.23. Scientific notation may be used by adding e or E followed by a decimal exponent (e.g. 1.23e2, which has the value 123.0). Either a decimal point or an exponent is required, but the integer part can be omitted (like .2e3). If the integer part is omitted, the decimal point and at least one digit after the decimal point are required (so .e2 is illegal). Hexadecimal floating-point numbers are not supported. Note that the exponent can be prefixed by "+" or "-", e.g. 1e+10 and 4e-3 are legal.

Note that "negative numbers" consists of two tokens: the unary "minus" operator, and the absolute number. For example, -34 has two tokens. Similarly, +7e8 also has two tokens.

STRING: A string enclosed by `""` or `''`. Only four escaped characters are supported: `\" \' \\ \n`. A string cannot contain a real newline character.

SYMBOL: Symbols that have special meanings, listed below:

```
+       -       *       /       %       ^       #
==      ~=      <=      >=      <       >       =
(       )       {       }       [       ]
;       :       ,       .       ..      ...
```

NAME: an identifier starting with a letter and followed by letters, digits and underscores. Note that reserved words cannot be used as names.

EOL: End of a physical line.

COMMENT: Start with `--`, and may not cross a line (the newline character following the comment is part of it. It is a separate **EOL** token). Note that white spaces output any string constants are always ignored, so `1+1` and `1 +1` have no differences for the lexer.

The lexer should be greedy. i.e. if there are more than one way to split the input into tokens, always maximize the length of the first token, then the second one, etc. For example, the only way to tokenize "`abc123<=x`" is "`abc123`", "`<=`" and "`x`".

In order to simplify the problem, a mini-lua program can only contain ASCII characters, regardless of your current locale (if you couldn't understand this, just ignore it).

Input

A valid mini-lua program.

Output

Non-comment tokens, one for each line. formatted as "[TYPE] token", where "token" is printed as in the input. If type is EOL, print an empty text.

Sample Input

```
print("Hello".." ".."--\"World\"--")
x=-3+4
this_1s_a_variable = 0xabcF-.012e-56+7.e8+9e+10--8
if 1 then
  y=x
end
print (y)
```

Output for Sample Input

```
[NAME] print
[SYMBOL] (
[STRING] "Hello"
[SYMBOL] ..
[STRING] " "
[SYMBOL] ..
[STRING] "--\"World\"--"
[SYMBOL] )
[EOL]
[NAME] x
[SYMBOL] =
[SYMBOL] -
[NUMBER] 3
[SYMBOL] +
[NUMBER] 4
[EOL]
[NAME] this_1s_a_variable
[SYMBOL] =
[NUMBER] 0xabcF
[SYMBOL] -
[NUMBER] .012e-56
[SYMBOL] +
[NUMBER] 7.e8
[SYMBOL] +
[NUMBER] 9e+10
[EOL]
[RESERVED] if
[NUMBER] 1
[RESERVED] then
[EOL]
[NAME] y
[SYMBOL] =
[NAME] x
[EOL]
[RESERVED] end
[EOL]
[NAME] print
[SYMBOL] (
[NAME] y
[SYMBOL] )
[EOL]
```

K. (Kengdie) Mua(II) - Expression Evaluation

In this problem-series, you're to implement a subset of the Lua language (version 5.1), called mini-lua (mua). This is one of Rujia Liu's experimental languages, mainly for implementing algorithms, not real-world programs.

This is the second problem in the series, which requires you to write an expression evaluator. The grammar below is described in extended BNF, in which {x} means "x appears one or more times", [x] means "x appears 0 or 1 time", and x | y means "either x or y (but not both) appear exactly 1 time".

Types

mini-lua is a dynamically typed language. This means that variables do not have types; only values do. There are no type definitions in the language. All values carry their own type.

There are six basic types in Lua: nil, boolean, number, string, function, table.

- Nil is the type of the value nil, whose main property is to be different from any other value; it usually represents the absence of a useful value.
- Boolean is the type of the values false and true.
- Number represents real (double-precision floating-point) numbers. Internally, numbers are represented by IEEE-754 numbers (conventional double variable in C/C++ or Java). Note that integers are also stored in this way. It's no big deal, because IEEE-754 can represent integers precisely, as long as you don't perform inaccurate operations (i.e. divisions).
- String represents arrays of ASCII (8-bit) characters.
- Function is function. It doesn't have a name (though the variable holding the function has a name).
- Table implements associative arrays, that is, arrays that can be indexed not only with numbers, but with any value (except nil). Tables can be heterogeneous; that is, they can contain values of all types (except nil).

Tables and functions values are objects: variables do not actually contain these values, only references to them. Assignment, parameter passing, and function returns always manipulate references to such values; these operations do not imply any kind of copy.

To lua programmers: Lua has two more basic types: userdata and thread, both are not supported in Mini-Lua. Mini-lua has very limited support for FP (functional programming). For example, lambda expression is not supported, and you cannot return a function or closure in a function.

Variables

A variable defined as follows:

```
var -> NAME { `.' NAME | `[' expr `]' }
```

Here "`expr`" means any valid expression, because tables can be indexed with any value. Note that the "dot" syntax is a syntactic sugar that makes the expression "look like accessing object member". For example, `a.name` is equivalent to `a["name"]`.

Variables have values of nil by default. If you assignment nil to a variable, you're removing that variable. Similarly, you can assign a nil to an element in a table to remove it from the table.

To lua programmers: Lua provides more syntactic sugars, `NAME `:' NAME args`. Let's ignore these.

Simple Expressions

An expression consists of so-called "simple expressions":

```
simpleexp -> NUMBER | STRING | nil | true | false | `{' '}' | var |
functioncall
```

Here `{' '}' means an empty table.

To lua programmers: Lua provides more convenient ways to construct tables. However, they add complexity to the language so we don't use them. Moreover, mini-lua does not support lambda expressions.

Function calls

The functioncall expression in the last section is defined this way:

```
functioncall -> var `(' [ expr {`,' expr } ] `)'
```

To lua programmers: In mini-lua, the only way to call a function is directly specifying the variable holding that function. For example `(print)(1)` won't work, but `a = (f)` is allowed, because `(f)` is a valid expression (see below). In Lua, if you call a function with a string constant or table constructor, the parenthesis' could be omitted. This syntax is not supported in mini-lua.

Expressions

Now we have all the building blocks. We use operators to combine simple expressions into complex expressions (i.e. "expr"):

```
expr -> simpleexp | expr binop expr | unop expr | `(' expr `)'
```

Note that this grammar does not consider operator precedence's, which is summarized in the following table (from low to high):

```
or
and
<     >     <=    >=    ~=    ==
..
+     -
*     /     %
not   #     - (unary)
^
```

Most of them are common-sense, but there are several things to mention:

- "not equal" is not "<>" or "!=". it is "~="
- "%" (mod) is defined on real numbers. it's equivalent to `a - math.floor(a/b)*b`
- "^" (power) is the only right-associative operator.
- "`not`", "`and`" and "`or`" always returns false or true. Only "`and`" and "`or`" are short-cut operators.
- Concatenation is "`..`", not "+".

- operator # is getting the length of a table or a string. The length of a table `T` is the maximal non-negative integer `n` such that `T[1], T[2], ..., T[n]` all exists (i.e. not nil). In this way, each table `T` can be regarded as an array, having `#T` elements.

To lua programmers: In lua, concatenation is right associative (See: http://lua-users.org/wiki/AssociativityOfConcatenation), but for this problem, whether it is left or right associative doesn't affect the result. Morever, "and" and "or" do not always return boolean values in Lua, but we restrict the result to boolean values in mini-lua in order to simplify the language. Maybe the most import change is: No automatic conversions between strings and numbers (i.e. no coercion).

Library functions

For testing purpose, here are some functions you should implement (extracted from http://www.lua.org/manual/5.1/manual.html , with simplifications):

- `tonumber(e)`

Tries to convert its argument to a number. If the argument is already a number or a string convertible to a number, then tonumber returns this number; otherwise, it returns nil. Note that there is no optional parameter as in Lua.

- `tostring(e)`

Receives an argument of any type and converts it to a string in a reasonable format. In mini-lua, if `e` is a function or a table, return `"function"` or `"table"` instead. If e is a number, convert it to a string using the traditional C format string `"%.14g"` (it's the default format string in lua 5.1, see luaconf.h). Don't worry you're using other programming languages and do not know the exact semantics of the format string, you can choose your own way. We're not testing the strict format in the judge data.

- `print(e)`

Prints `tostring(e)` to standard output, followed by a newline. Note that in mini-lua, this function can only print one value.

- `string.rep(s,n)`

Returns a string that is the concatenation of `n` copies of the string `s`.

- `string.sub(s, i, [,j])`

Returns the substring of `s` that starts at `i` and continues until `j`; `i` and `j` can be negative. If j is absent, then it is assumed to be equal to -1 (which is the same as the string length). In particular, the call string.sub(s,1,j) returns a prefix of s with length j, and string.sub(s, -i) returns a suffix of s with length i.

- `table.concat(table, [,sep])`

Given an array where all elements are strings, returns `table[1]..table[2]..` ... `table[#table]`. The default value for `sep` is the empty string.

- `table.sort (table [, comp])`

Sorts table elements in a given order, in-place, from `table[1]` to `table[n]`, where `n` is the length of the table. If `comp` is given, then it must be a function that receives two table elements, and returns `true` when the first is less than the second (so that no `comp(a[i+1],a[i])` will be true after the sort). If `comp` is not given, then the standard operator `<` is used instead. Note that the sort algorithm is not stable, but we're carefully designing the test data so that how the sort is implemented does not affect the final.

- `math.abs(x), math.floor(x), math.ceil(x)`

Returns the absolute value of `x`, the largest integer smaller than or equal to `x`, and the smallest integer larger than or equal to `x`, respectively.

- `math.sqrt(x), math.exp(x), math.log(x), math.log10(x)`

Returns the square root of `x`, the value e^x, $\ln(x)$ and $\log_{10}(x)$.

- `math.pi, math.rad(x), math.deg(x)`

`math.pi` is a variable holding the value of pi, while the other two functions converts between degrees and radians.

- `math.acos(x), math.asin(x), math.atan(x), math.atan2(y,x)`

Returns the value of math functions, in radians.

- `math.cos(x), math.sin(x), math.tan(x)`

Returns the value of math functions, assuming `x` is in radians.

- `math.min(x,y), math.max(x,y)`

Returns the smaller/larger value of `x` and `y`.

Input

There will be multiple mini-lua programs. Each program consists of lines. Each line is either in the form `print(expr)`, or in the form `var = expr` (indicates an assignment). All the expressions will obey the rules above. An empty line terminates a program (all the variables should be reset to nil). The expressions will be correct and evaluates to a reasonable value (for example, you don't have to handle NaN or arithmetic overflows, and you will not be asked to compare a number with a string). We will not re-assign these functions/variables, though we may assign them to new variables. There will be no comments in the program.

Output

For each line in the form `print(expr)`, print the expression. When printing numbers, print as many digits as you like, as long as the relative error OR the absolute error is no more than 1e-9.

Sample Input

```
print(1+2*3/4)
f = math.sin
print(f(1.57))
print(1<2 and 4+5==9)
print(math.max(3,-math.min(5*7,-4)))
a={}
a[1]={}
a[1][2]={}
a[1][3]="hehe\'\\"
a[1][1]=a
print(#a)
print(#a[1])
print(#a[1][3])
print(a)
print(f)
print(a[1][1][1][3] .. "\n" .. "..")
```

Output for Sample Input

```
2.5
0.99999968293183
true
4
1
3
6
table
function
hehe'\
..
```

Hint

If you want to learn from the source code of official lua compiler, download the 5.1.4 version and go straight to the "subexpr" function in lparser.c (you can see the precedence table right before it).

L. (Last) Mua (III) - Full Interpreter

In this problem-series, you're to implement a subset of the Lua language (version 5.1), called mini-lua (mua). This is one of Rujia Liu's experimental languages, mainly for implementing algorithms, not real-world programs.

This is the third (and last!) problem in the series, which requires you to write a full interpreter. Make sure you've solved the first two problems before attempting this problem. You may have trouble understanding this problem if you haven't done so.

Chunk

A chunk (a program or a piece of code that are run as a whole) is a single block:

```
block -> {stat EOL} [laststat EOL]
laststat -> return [expr] | break
```

Note that you can use break only as the last statement in a block (to keep things simple, lua even doesn't support continue!)

To lua programmers: In lua, you can write two statements in one line, even *without* a semicolon. In mini-lua, you can't write multiple statements in a line. Actually semicolon cannot be used as a statement delimiter. Moreover, a block can only return one value.

Simple Statements

The following types of statements are easy to understand:

- Empty statement: `stat ->`
- Function call, then discard the result: `stat -> functioncall`
- Assignment: `stat -> var `=' expr`
- Block statement: `stat -> do block end`

To lua programmers: Multiple assignment (like `a,b=b,a`) is not supported in mini-lua. No tail-recursion optimization is necessary in this problem.

Control Flows

While, repeat and if statements all use conditionals. Both nil and false make a condition false; any other value makes it true. These statements are defined as follows:

- While statement: `stat -> while expr do block end`
- Repeat statement: `stat -> repeat block until expr`
- If statement: `stat -> if expr then block { elseif expr then block } [else block] end`

And there are two kinds of for loops. The first one is:

```
stat -> for NAME `=' expr `,' expr [`,' expr] do block end
```

The three expressions in this loop are the initial value, the upper limit (when step > 0) / lower limit (when step <= 0), and the step (default: 1). All three expressions are evaluated exactly once, converted to number (using `tonumber(e)` function), before the loop starts. All three expressions must all result in numbers (`tonumber(e)` should not return nil). Note that `NAME` is local (you cannot access it after the loop ends), and you can use `break` to exit the loop, but there is no `continue` statement.

The second one is a more general iteration, looping for all the keys in a table:

```
stat -> for NAME in iterator do block end
```

Here `iterator` is either `ipairs(table)` or `pairs(table)`(of course the actual variable being iterated can have other names other than `table`). The difference is: `ipairs` loops from `1` to `#table`, but `pairs` loops for all the keys in table, in no particular order.

Function definition

Given other building blocks discussed above, function definitions are quite simple:

```
stat -> function NAME `(' [NAME {`,' NAME} ] `)' block end
```

Note that all the parameters are local variables. As discussed before, you can use `return` statement to exit a function, carrying a single return value if you like. You can only use `return` in the last statement of a block, so if you want to exit a function in the middle, you can wrap it in a block, like `do return end`. It means to exit from the inner-most function that includes the block.

Scoping and Visibility

You can declare a local variable this way:

```
stat -> local NAME [ `=' expr ]
```

Like Lua, mini-Lua is a lexically scoped language. The scope of variables begins at the first statement *after* their declaration and last until the end of the innermost block that includes the declaration. If there is another variable having the same name in an outer block, that variable is shadowed (it still exists, but you cannot access it). Only after the inner block ends, we regain the access to it, because the inner variable no longer exists. Recall that a chunk is also a block, so you can also declare local variables in a trunk.

Note that the local variable is accessible only after the declaration, so you can use `local x = x` to declare a local variable called `x`, initialized with the value of outside variable whose name is also `x`.

Note that in the **repeat-until** loop, the inner block does not end at the **until** keyword, but only after the condition. So, the condition can refer to local variables declared inside the loop block.

Because of the lexical scoping rules, local variables can be freely accessed by functions defined inside their scope. However, to keep things simple, all the functions will be declared globally (not nested in another function or a block), and local variables in the chunk (if any) are always declared after the functions.

Input

There will be multiple mini-lua programs. Each program starts with a line starting with `--PROGRAM` (it will not appear inside a program).

Output

For each program, print the test case number and the output from the program. Print an empty after each test case.

Sample Input

```
-- PROGRAM: Eight-Queen problem solver in Mini-Lua
```

```
function dfs(d)
  if d == n then
    cnt = cnt + 1
  else
    for i = 1, n do
      if (not vis[i]) and (not vis2[d-i]) and (not vis3[d+i]) then
        vis[i] = true
        vis2[d-i] = true
        vis3[d+i] = true
        dfs(d+1)
        vis[i] = nil
        vis2[d-i] = nil
        vis3[d+i] = nil
      end
    end
  end
end

vis = {}
vis2 = {}
vis3 = {}
cnt = 0
n = 8
dfs(0)
print(cnt)

-- PROGRAM: Scoping and Visibility rules
x = 10
do
  local x = x
  print(x)
  x = x+1
  do
    local x = x+1
    print(x)
  end
  print(x)
end
print(x)
```

Output for Sample Input

```
Program 1:
92

Program 2:
10
12
11
10
```

Hint

Haven’t you noticed that mua is in LL(1)? A recursive decent parser is enough for this problem.