Problem E: Street Directions

According to the Automobile Collision Monitor (ACM), most fatal traffic accidents occur on two-way streets. In order to reduce the number of fatalities caused by traffic accidents, the mayor wants to convert as many streets as possible into one-way streets. You have been hired to perform this conversion, so that from each intersection, it is possible for a motorist to drive to all the other intersections following some route.

You will be given a list of streets (all two-way) of the city. Each street connects two intersections, and does not go through an intersection. At most four streets meet at each intersection, and there is at most one street connecting any pair of intersections. It is possible for an intersection to be the end point of only one street. You may assume that it is possible for a motorist to drive from each destination to any other destination when every street is a two-way street.

Input

The input consists of a number of cases. The first line of each case contains two integers n and m. The number of intersections is n ($2 \leq n \leq 1000$), and the number of streets is m. The next m lines contain the intersections incident to each of the m streets. The intersections are numbered from 1 to n, and each street is listed once. If the pair $i\ j$ is present, $j\ i$ will not be present. End of input is indicated by $n = m = 0$.

Output

For each case, print the case number (starting from 1) followed by a blank line. Next, print on separate lines each street as the pair $i\ j$ to indicate that the street has been assigned the direction going from intersection i to intersection j. For a street that cannot be converted into a one-way street, print both $i\ j$ and $j\ i$ on two different lines. The list of streets can be printed in any order. Terminate each case with a line containing a single '#' character.

Note: There may be many possible direction assignments satisfying the requirements. Any such assignment is acceptable.

Sample Input

```
7 10
1 2
1 3
2 4
3 4
4 5
4 6
5 7
```

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

Sample Output

```
1

1 2
2 4
3 1
3 6
4 3
5 2
5 4
6 4
6 7
7 5
#
2

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

Problem F: Parallel Deadlock

A common problem in parallel computing is establishing proper communication patterns so that processors do not deadlock while either waiting to receive messages from other processors, or waiting for the sending of messages to other processors to complete. That is, one processor will not complete sending a message until it is received by the destination processor. Likewise, a receive cannot complete until a message is actually sent.

There are two modes of communication: blocking and non-blocking. A blocking send will not complete until a matching receive is performed at the destination processor. Likewise, a blocking receive will not complete until the matching send is performed by the source processor. Non-blocking actions will "return" immediately (i.e., allow the program to continue), but will not actually complete until the matching action is performed at the target. The matching action of a send is a receive (either blocking or non-blocking), and similarly, the matching action of a receive is a send (either blocking or non-blocking).

At the start of each timestep, each processor that is not blocked starts to run its next instruction. Processors that execute blocking instructions become blocked. Messages can be received at the end of the timestep in which they are sent, but may need to wait several timesteps until the recipient performs a matching receive. If the recipient of a message is waiting to receive from the sender, then the message is received in the same timestep. Messages are received in the order that they are sent. If all of the actions for a particular blocking instruction complete at the end of the timestep, then the processor that ran the instruction will be unblocked before the next timestep.

A correct program will terminate only when all of its actions have completed. Pending non-blocking operations must be completed before a program can terminate.

Your program will take in a list of processors and actions (no more than 100 for each processor), and determine if each processor finishes its program. If a given processor does not finish, it must print out which other processors are preventing it from finishing.

Input

The first line will be a single positive integer that tells how many processors will be listed. For each processor there will be one line containing the name of the processor (a single capital letter) followed by a positive integer, N. The following N lines will contain the instructions that comprise the program for that processor.

An instruction is of the form "*Mode Action Target(s)*" where "*Mode*" can be "B" or "N", for blocking or non-blocking, respectively. "*Action*" can be "S" or "R", for send or receive, respectively. "*Target(s)*" will be one or more processor names to which the action is to be addressed. No processor will be listed twice and a processor will never attempt any sort of communication with itself . A send to multiple targets will not complete until matching receives

have been performed by all of the targets (and vice versa).

Output

Given that instruction 1 occurs at $t = 1$, your program will output at which timestep each processor finishes. If a processor does not finish, you must output which processor are preventing that processor from finishing. Processors should be listed in alphabetical order, both for the list of processors and the sets of processors that prevent a processor from finishing. The list of processors preventing termination should list processors at most once and separate multiple processors with "and".

Sample Input

```
4
I 5
B S B P C
N S B P C
N R B
B R P
B R C
B 2
B R I
B S I
P 3
N S I
N R I
B R I
C 4
N S I
B R I
B S P
B R I
```

Sample Output

```
B finishes at t=4
C never finishes because of P
I never finishes because of B and C
P finishes at t=5
```

Notice how C's final blocking receive would be matched by a send on I if both instructions were executed. However, it never gets executed because it is stuck in the blocking send to P (that has no matching receive on P), therefore causing deadlock on I.

Problem G: DNA Sorting

One measure of "unsortedness" in a sequence is the number of pairs of entries that are out of order with respect to each other. For instance, in the letter sequence "DAABEC", this measure is 5, since D is greater than four letters to its right and E is greater than one letter to its right. This measure is called the number of inversions in the sequence. The sequence "AACEDGG" has only one inversion (E and D)—it is nearly sorted—while the sequence "ZWQM" has 6 inversions (it is as unsorted as can be—exactly the reverse of sorted).

You are responsible for cataloguing a sequence of DNA strings (sequences containing only the four letters A, C, G, and T). However, you want to catalog them, not in alphabetical order, but rather in order of "sortedness", from "most sorted" to "least sorted". All the strings are of the same length.

Input

The first line contains two integers: a positive integer n ($0 < n \leq 50$) giving the length of the strings; and a positive integer m ($0 < m \leq 100$) giving the number of strings. These are followed by m lines, each containing a string of length n.

Output

Output the list of input strings, arranged from "most sorted" to "least sorted". Since two strings can be equally sorted, there may be several different correct outputs.

Sample Input

```
10 6
AACATGAAGG
TTTTGGCCAA
TTTGGCCAAA
GATCAGATTT
CCCGGGGGGA
ATCGATGCAT
```

Sample Output

```
CCCGGGGGGA
AACATGAAGG
GATCAGATTT
ATCGATGCAT
TTTTGGCCAA
TTTGGCCAAA
```

Problem H: Numbers That Count

"Kronecker's Knumbers" is a little company that manufactures plastic digits for use in signs (theater marquees, gas station price displays, and so on). The owner and sole employee, Klyde Kronecker, keeps track of how many digits of each type he has used by maintaining an inventory book. For instance, if he has just made a sign containing the telephone number "5553141", he'll write down the number "5553141" in one column of his book, and in the next column he'll list how many of each digit he used: two 1s, one 3, one 4, and three 5s. (Digits that don't get used don't appear in the inventory.) He writes the inventory in condensed form, like this: "21131435".

The other day, Klyde filled an order for the number 31123314 and was amazed to discover that the inventory of this number is the same as the number—it has three 1s, one 2, three 3s, and one 4! He calls this an example of a "self-inventorying number", and now he wants to find out which numbers are self-inventorying, or lead to a self-inventorying number through iterated application of the inventorying operation described below. You have been hired to help him in his investigations.

Given any non-negative integer n, its inventory is another integer consisting of a concatenation of integers $c_1 d_1 c_2 d_2 \ldots c_k d_k$, where each c_i and d_i is an unsigned integer, every c_i is positive, the d_i satisfy $0 \leq d_1 < d_2 < \ldots < d_k \leq 9$, and, for each digit d that appears anywhere in n, d equals d_i for some i and d occurs exactly c_i times in the decimal representation of n. For instance, to compute the inventory of 5553141 we set $c_1 = 2$, $d_1 = 1$, $c_2 = 1$, $d_2 = 3$, etc., giving 21131435. The number 1000000000000 has inventory 12011 ("twelve 0s, one 1").

An integer n is called self-inventorying if n equals its inventory. It is called self-inventorying after j steps ($j \geq 1$) if j is the smallest number such that the value of the j-th iterative application of the inventory function is self-inventorying. For instance, 21221314 is self-inventorying after 2 steps, since the inventory of 21221314 is 31321314, the inventory of 31321314 is 31123314, and 31123314 is self-inventorying.

Finally, n enters an inventory loop of length k ($k \geq 2$) if k is the smallest number such that for some integer j ($j \geq 0$), the value of the j-th iterative application of the inventory function is the same as the value of the $(j+k)$-th iterative application. For instance, 314213241519 enters an inventory loop of length 2, since the inventory of 314213241519 is 412223241519 and the inventory of 412223241519 is 314213241519, the original number (we have $j = 0$ in this case).

Write a program that will read a sequence of non-negative integers and, for each input value, state whether it is self-inventorying, self-inventorying after j steps, enters an inventory loop of length k, or has none of these properties after 15 iterative applications of the inventory function.

Input

A sequence of non-negative integers, each having at most 80 digits, followed by the terminating

value -1. There are no extra leading zeros.

Output

For each non-negative input value n, output the appropriate choice from among the following messages (where n is the input value, j is a positive integer, and k is a positive integer greater than 1):

> n is self-inventorying
> n is self-inventorying after j steps
> n enters an inventory loop of length k
> n can not be classified after 15 iterations

Sample Input

```
22
31123314
314213241519
21221314
111222234459
-1
```

Sample Output

```
22 is self-inventorying
31123314 is self-inventorying
314213241519 enters an inventory loop of length 2
21221314 is self-inventorying after 2 steps
111222234459 enters an inventory loop of length 2
```