Section 1: Background

Many jobs have to be completed in steps, with some steps in a particular order and some more flexible. For example, to make toast with jam, one must:

1. Get a plate.
2. Get a jar of jam.
3. Get a spoon.
4. Get a loaf of bread.
5. Get a slice from the loaf.
6. Put the slice in the toaster.
7. Push down the handle.
8. Put the toasted bread on the plate.
9. Scoop jam with the spoon.
10. Using the spoon, spread the jam on the toast.
11. Put the jam away.
12. Put the loaf of bread away.
13. Put the spoon in the dishwasher.

(This list assumes that you know enough to open the jam before scooping, and to close it before putting it away, among a few other things.) One could get the plate at any point in the list, provided it was done before the toasted bread was ready to put down. Similarly, one could get the jam after toasting the bread and putting it on the plate, or get the spoon at any point before scooping with it. One could put the spoon in the dishwasher at any point after spreading the jam, but not before. Getting the spoon, or not, before to getting out the bread makes no difference. Either could be done first with no change in the outcome.

There are many other cases in which tasks have to be arranged so that prerequisite operations are completed before other operations which depend on them, but some tasks are independent of others. When registering for classes, one should take *Data Structures* before *Programming Languages*, and take *Geotechnical Engineering* before taking *Earth Retaining Systems*, but there is no particular reason why *Data Structures* should come before (or after) *Earth Retaining Systems*.

In computer science, arranging tasks so that each one is done before another task which depends on it is known as performing a *topological sort*.

Closely related to performing a topological sort is testing a proposed ordering. Your job is to write a program which will read in a list of jobs and their prerequisites, and then evaluate a number of proposed task arrangements to see whether they are valid topological orderings.

Our requirement for validity will be that those tasks which *are* attempted must have their prerequisites satisfied. Tasks which are not attempted are irrelevant. So, for example, if we decide to make plain toast with no jam, we won't bother getting a spoon or jam or spreading the jam we didn't get.

To keep the input and output simple, each task will be represented by a single upper-case letter. (Later, we can make up a chart which specifies that *A* is 'eat toast', and so forth. Here we are interested only in the automatic validation of proposed orderings.)

Section 2: Problem Specification

Your program must accept input in the following format:

1. A single integer, **N**, laying out the number of lines which describe the prerequisites.
2. **N** lines in the following format:
   (a) An upper-case letter, indicating which job's prerequisites are laid out on this line.
   (b) Zero or more letters, indicating which tasks must be completed before the first one listed on that line can be done.
   (c) A period ('.'), indicating the end of this line of data.
3. A single integer, **P**, indicating how many proposed arrangements have to be tested.
4. **P** lines in the following format:
   (a) One or more letters, indicating which tasks should be completed in which order.
   (b) A period ('.'), indicating the end of this line of data.

You do not need to do error-checking on the input. There will be nothing on the first line except a single integer; there will be no characters except for upper-case letters and periods on the lines describing the dependencies or on the lines which contain proposed orderings, each of which will end with a period. There will be no blank lines.

For example, this input:

```
4
A B .
B C .
D .
Q U .
6
A B C .
B A C .
C A B .
C B A .
D .
Z .
```

Means that the task dependency list has two lines: task **A** depends on task **B**, and task **B** depends on task **C**. (Note that **A** depends on **C** indirectly, even though **C** is not listed on the line for **A**.

The rest of the input indicates that there are seven lines of sample input. Each one has to be checked to see if the order of tasks is consistent with the dependencies described in the first part of the input. In this example, only the last three lines are valid orderings: line 4 because each task is completed before those which depend on it. Line 5 is valid because its only task, **D**, does not depend on anything, as is the case with line 6. Note that any task which is not specified as having dependencies is considered to have none. Note also that while **Q** is listed as depending on **U**, that doesn't mean that any of the proposed orderings *must* include either **Q** or **U**.

Section 3: Sample Data

1. Input:

```
1
A B C .
4
C B A .
B C A .
B A C .
A B C .
```

Output:

```
Line 1 is topologically sorted.
Line 2 is topologically sorted.
Line 3 is not topologically sorted: A depends on C.
Line 4 is not topologically sorted: A depends on B.
```

The first two lines are both valid orderings because while **A** depends on both **B** and **C**, neither **B** nor **C** depends on the other. Therefore, it doesn't matter whether **B** or **C** is completed first. The last two lines are not acceptable, for the reasons listed. (Note that Line 4 is invalid for two reasons: **A** also depends on **C**. In such cases, print out only one of the conflicts which is discovered; it does not matter which.)

2. Input:

```
4
A C .
A B .
C B .
Z X B .
3
C B A .
B C A .
A B C .
```

Output:

```
Line 1 is not topologically sorted: C depends on B.
Line 2 is topologically sorted.
Line 3 is not topologically sorted: A depends on B.
```

The first line is invalid because **A** depends on both **B** and **C**, and **C** depends on **B**. Line 3 is invalid for two separate reasons; listing either one is okay. Note that the dependency list has task **A** listed twice. Task **Z** depends on **X** and **B**; since none of the proposed orderings attempts task **Z**, it doesn't matter that none ever does **X**. (Task Z might involve jam, when in this case we only want plain toast.)

Section 4: Test Data

Run your program on these data sets, and print out screenshots showing that your output is correct. You may save the data sets in files, or enter them into a graphical text box, or use some other reasonable method making the data available to your program; do not hard-code the problem sets.

1.
```
7
D A B C .
E D B .
F D .
G E B .
H E .
I F .
E C .
6
B C A D F I G E .
C A B D F I E H .
A B C D E G H F I .
Z X Q U W Y L F D .
A C B D F E H G I .
A X C Z B Q D U F W E R H P G S I T .
```

2.
```
13
D C A B .
C B .
B A .
L K G F D .
K J I H .
F E .
W T P L .
T S Q O N M .
W V R .
V U T S .
X W .
Y X .
Z Y .
6
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z .
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A .
A B C E H F J I D P M O N K G L R S Q T U V W X Y Z .
A B C J I H E D G F D K S Q O N M L P T V R W X Y Z .
E U R O P A I O S I N O P E P A S I P H A E .
P A N D O R A T I T A N I A P E T U S P H O E B E .
```