CMPUT 275 - Tangible Computing
Interview Problem: Makefile Dependencies

Description

As you know, with each `Makefile` target you should specify a number of dependencies. If any of these dependencies have changed (or do not exist because they have not been built), the target will be rebuilt when you try to `make` it. Some of these dependencies can themselves be Makefile targets, so a `make` command will recursively check if any dependencies need to be rebuilt as well before building the target.

In this problem, you will first be given a list of `Makefile` targets and their dependencies (the actual build commands, eg. `g++` commands, do not matter for this exercise). Such a target will appear like:

```
<targetname>:  <list of dependencies>
```

Here, the list of dependencies will be a space-separated list of names. Some files listed as a dependency to a target may themselves be targets. A file that is *only* listed as a dependency (but never a target) is one that the programmer has already developed and does not need to be built by a `make` command.

Next, you will be given a sequence of `make` commands that build a target. Here, *building* a target is a recursive process. The list of dependencies for the target are processed *in the same order they are listed as dependencies of that target.*

When processing a dependency, if it is not a file given by the programmer (i.e. it is a target in some other line of the Makefile), and it has not already been built, that dependency will be recursively built before processing the remaining dependencies of the original target. Once the dependency list of a target has been processed, the target is finally built.

Each `make` command appears as:

```
make <targetname>
```

For each such command, you should output the targets that will be built when the `<targetname>` target is built. See the output specification for more details. When a target is built either directly or recursively through some `make` command, it remains built. Thus, each target will be built *at most once* even if it is listed as a dependency for many other targets.

Input

The first line of input contains two integers $1 \leq n, m \leq 20,000$ indicating the number of targets in the Makefile and the number of make commands to process, respecitvely. Then n lines follow, each describing a target and a dependency. Each such line will take the form:

```
k <targetname>:  <list of dependencies>
```

where $k \geq 1$ is an integer specifying the number of dependencies of the target. Exactly one space will separate the different dependencies in the list and there is exactly one space

between the colon character following the target name and the first dependency. There is no space separating the target name and the colon.

The final m lines describe the make operations. Each is given simply as:

`make <targetname>`

You are given the following additional guarantees:

- The total number of all dependencies of all targets is at most 200,000 (i.e. the sum of all the k values on n target lines is at most 200,000).
- All file names consist of letters, digits, or the dot symbol (a period). No file name contains any space. All file names will have at most 12 characters.
- Different files will not have the same name. So if you see a file name listed twice among the targets and dependencies, it is the same file.
- No file will be listed as a target more than once.
- The target name in each `make` command will be an actual target listed among the n Makefile targets.
- No file is named `make`.
- There will be no "cycle" of dependencies. For example, if file1 depends on file2 and file2 depends on file 3, then file3 will not depend on file1. More generally, if we consider the directed graph over files where we include a directed edge from X to Y if Y is listed as a dependency of X, then the graph will not contain any cycles. This also means a file will not depend on itself.
- Any file name listed as a dependency but not as a target is a file given by the user and does not need to be built.
- None of the targets were already built before the first make command.

Output

For each of the m `make <targetname>` commands, you should output a single line. If the target was already built by some previous `make` command (either directly or when a previous `make` command built it recursively), you should output message:

`make: ‘<targetname>’ is up to date.`

Otherwise, the line should contain the names of all targets that were built (either directly or recursively) by this `make` command. These should appear in the order they were built according to the recursive process described above. In particular, this means the original target file `<targetname>` will be the last file in this list. There should be exactly one space between different files on this line.

Running Time

There is no specific $O()$ running time you must achieve on this weekly exercise. The ideal

running time is linear in the total size of the input (i.e. the number of targets + total number of dependencies). It will not be possible for a significantly slower algorithm to pass the larger test cases. If we further let t denote the total number of dependencies of all targets:

- 1/4 of the test cases we test your solution on will have $n, m, t \leq 100$.
- An additional 1/4 of the test cases will have $n, m, t \leq 1000$.
- The remaining test cases will be very large (but no larger than the guarantees mentioned above, of course).

Graph Class + Other Requirements

You are **not** required to use the graph class developed in the lectures. However, if you wish to do so to model the dependency relationships in this problem, you should put the class declaration and method implementations in the same source code file you use to solve this problem. This is because you are to submit only a single source code file (we will not regard this as bad style). You may make any modifications you wish to the graph class.

You may include any other libraries or create any helper functions you wish to solve this problem. There are no strict requirements on the names of the functions you use.

Comment

So far your Makefiles are very "shallow". Your targets have been executables, which may depend on object files that are targets. However, Makefiles can recurse deeper in other situations. For example, when building a C++ program for an Arduino, the dependencies have recursive depth 4 because there are additional translation steps in the compiling process. More complicated settings (like building and install software libraries on a machine) can have much larger depth.

Sample Input 1

```
5 6
3 project: main.o matrix.o fft.o
3 debug: main.o matrix.o fft.o
3 main.o: main.cpp matrix.h fft.h
2 matrix.o: matrix.h matrix.cpp
2 fft.o: fft.h fft.cpp
make matrix.o
make matrix.o
make debug
make fft.o
make project
make main.o
```

Sample Output 1

```
matrix.o
make: 'matrix.o' is up to date.
main.o fft.o debug
```

```
make: ‘fft.o’ is up to date.
project
make: ‘main.o’ is up to date.
```

Explanation
The first `make matrix.o` builds the file (the only dependencies are supplied by the programmer because they are not targets), but the second `make matrix.o` command does not need to build anything.With `make debug`, the targets `main.o` and `fft.o` were not yet built so they were built first (in this order because they were listed as dependencies in this order) before finally building `debug`. For `make fft.o`, the target was already built (recursively when `make debug` was run prior to this make). For `make project`, all dependencies were already built so only the target itself had to be built. Finally, `main.o` was already built in an earlier `make` command.

Sample Input 2

```
8 4
3 a: b c g
3 b: e f g
3 c: b e d
1 d: f
1 g: e
3 sss: xxx yyy zzz
2 xxx: yyy zzz
1 yyy: zzz
make g
make a
make sss
make d
```

Sample Output 2

```
g
b d c a
yyy xxx sss
make: ‘d’ is up to date.
```

Explanation
For `make g`, the only dependency is a file given by the programmer (i.e. it is not a target) so only `g` needs to be built. For `make a`, it first recursively builds `b` which does not depend on any file that is not already built (`e` and `f` are given files and `g` was built earlier). Then it goes on to build `c` but before doing this it recursively builds `d` because this was not built yet. After `d` is built, `c` is built. Lastly, `a` is built because its only remaining dependency (after `c`) was already built.

For `make www`, we notice `yyy` was built before `xxx` because when we recursively built `xxx` we

had to wait until `yyy` was built. Thus, when we are processing the dependencies for `www` we have that `yyy` was already built by the time it is processed as a dependency for `www`.

Finally, `make d` does not build anything because `d` was built in a previous make operation (even though it was a few commands ago).

Grading Comments
Despite the fact this appears similar to a morning problem, it will be graded like a weekly exercise. In particular:

- Style matters. Use appropriate comments, proper indentation, etc. Include a file header. Consult the style guide on eClass. **Recall**: We are going allowing you to include the class declaration and method implementations of a graph class (or any other class you deem useful) in the file.
- You must adhere exactly to the output specification: for example, if you output the lines in the wrong order or print extra whitespaces then you will receive a deduction. For full credit, the test centre must accept the output without any presentation error.
- You were only give a few test cases in the test centre files on eClass. We will test your solution on additional test cases that adhere to the input specification.
- Partial credit may be obtained if your solution works on some inputs (eg. small instances).
- Adhere closely to the submission instructions for the weekly exercise. See the eClass code submission link for details.