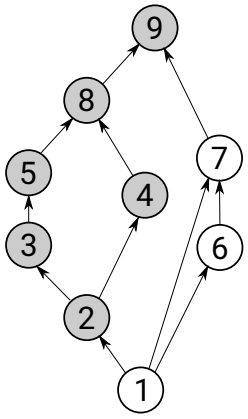
# Problem 4. bit gisect

| | |
|---|---|
| Input file: | `standard input stream` |
| Output file: | `standard output stream` |
| Time limit: | 3 seconds (6 seconds for Java) |
| Memory limit: | 256 mebibytes |

Vasya the programmer is using Bit, a version control system for storing his code. The history of changes in Bit can be viewed as an oriented acyclic graph. The nodes of this graph are called revisions. When Vasya wants to make changes in the code, he can take an arbitrary revision $A$ and change the code, resulting in a new revision $D$. In these case, we call the revision $D$ dependent of the revision $A$. Each of these dependencies in Bit is expressed as an arc from the revision node $A$ to the revision node $D$. Mergers are a special revision type in Bit: Vasya can join any two arbitrary revisions $A$ and $B$ and merge them into a new revision $D$. In this case, the revision $D$ will depend on two revisions: $A$ and $B$. The repository also contains a single "initial" revision: it is the only revision which does not depend on any other revision. All other revisions in Bit depend on one or more other revisions. Vasya has finished working on his project: there is strictly one revision which does not have any dependent revisions.

But one day, something went wrong, and the code became bug-ridden. Vasya is still trying to find them. Here is what Vasya's bugs do: if a bug appears in a revision $A$, it affects all revisions accessible from $A$ in the Bit graph: the revision $A$, all revisions dependent on $A$, all revisions dependent on revisions dependent on $A$, etc. To find the bug, Vasya can switch between revisions and check for the bug in the different revisions, thus narrowing down the suspect list. In the end, he will find the initial revision containing the bug.




Vasya is getting bug reports from the users, one after another, and he must find all these bugs, one by one. Help Vasya fix the code as quickly as possible.

The figure shows an example with 9 revisions. The revision `1` is the initial revision. The revisions `7`, `8`, and `9` are mergers, each depending on two other revisions. The revision `2`, which sprouted a bug, and all dependent revisions suffering from that bug are highlighted in gray.

## Interaction Protocol

This is an interactive problem. Instead of file input-output, you will be working with a special program "— the interactor. You will be interacting with this program through the standard input-output streams.

Upon the start, your program is fed information about revisions through the standard input stream. The first line contains an integer $N$ — the number of revisions in Bit ($1 \le N \le 1\,000$). The following $N$ lines contain the descriptions of revisions, each beginning with a revision number — a string of 6 hex digits (digits from `0` to `9` and lower case latin letters from '`a`' to '`f`'). It is guaranteed that this description does not contain a revision with the number `000000`. It is followed by an integer $k$ — the number of revisions from which the current revision depends ($0 \le k \le 2$), followed by $k$ space-separated revision numbers. It is guaranteed that by the moment of describing a new revision all descriptions of the revisions on which it depends have been provided.

Vasya must now find his bugs. It is guaranteed that there are no more than 10 000 bugs in the code.

A search for bugs begins with a line sent to your program through the input stream, containing the string "`bug`" and the number of the revision where the new bug has been located. If string "`000000`" is provided as the revision number, it means that all bugs have been found and the program must stop. Otherwise, the search for a bug begins.

Next, your program must send queries to the standard input stream. Each query must consist of a single line containing a command and the number of the revision upon which that command must be executed. The command can be one of the following:

- `check` — check the revision for bugs. The answer is provided as the line "`bad`" if the revision is bugged, and "`good`" if there are no bugs in the revision.
- `fix` — means that you have found a bug-ridden revision. After the command `fix` your program must start looking for the next bug.

For each bug, your program can make no more than 20 queries, otherwise the solution will receive the Wrong Answer verdict.

Make sure to print the line break symbol and clear the output stream buffer (the `flush` command of the language) after every printed query. Otherwise, the solution can get Timeout.

## Example

For ease of reading, the commands in the example are separated by blank lines.

| standard input stream | standard output stream |
|---|---|
| 9<br>000001 0<br>000002 1 000001<br>000003 1 000002<br>000004 1 000002<br>000005 1 000003<br>000006 1 000001<br>000007 2 000006 000001<br>000008 2 000004 000005<br>000009 2 000007 000008<br>bug 000008 | |
| bad | check 000004 |
| good | check 000001 |
| bad | check 000002 |
| bug 000009 | fix 000002 |
| bad | check 000007 |
| good | check 000006 |
| bug 000000 | fix 000007 |