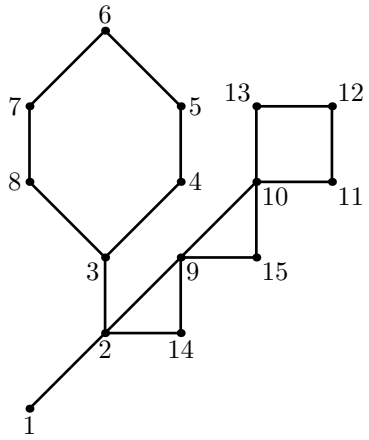
Problem C. Cactus Generator

Input file: `cactus.in`
Output file: `cactus.out`

NEERC featured a number of problems about *cactuses* — connected undirected graphs in which every edge belongs to at most one simple cycle. Intuitively, cactus is a generalization of a tree where some cycles are allowed.

In 2005, the first year where a problem about cactuses appeared, the problem was called simply "Cactus". In 2007 it was "Cactus Reloaded", in 2010 it was called "Cactus Revolution", and in 2013 it was called "Cactus Automorphisms". Here is an example of cactus that was used in those problems:



For four years judges had to generate test files for cactuses with thousands of vertices. Of course, a number of test generators of ever-increasing complexity were built, culminating with a domain-specific language called *CGL* — Cactus Generator Language. CGL can compactly define a big cactus for purposes of a test. In this problem you have to parse a simplified version of this language, which we call *SCGL* — Simple Cactus Generator Language, and output a resulting cactus.

A cactus has to be output by listing the minimal set of edge-distinct paths that cover the whole graph.

The syntax of SCGL cactus definition is represented by the *graph* non-terminal in the grammar that is given below using the Extended Backus-Naur Form:

$$
\begin{array}{rcl}
\mathit{graph} & = & \text{“}\texttt{c}\text{”} \\
& | & \text{“}\texttt{c(}\text{”}\ \mathit{list}\ \text{“}\texttt{)}\text{”} \\
& | & \text{“}\texttt{loop(}\text{”}\ \mathit{list}\ \text{“}\texttt{)}\text{”} \\
& | & \text{“}\texttt{t(}\text{”}\ \mathit{list}\ \text{“}\texttt{)}\text{”} \\
\\
\mathit{list} & = & \mathit{graph}\ \{\ \text{“}\texttt{,}\text{”}\ \mathit{graph}\ \} \\
& | & (\ \mathit{number} \mid \mathit{range} \mid \mathit{variable}\ )\ [\ \text{“}\texttt{,}\text{”}\ \mathit{graph}\ ] \\
\\
\mathit{number} & = & \mathit{nzdig}\ \{\ \text{“}\texttt{0}\text{”} \mid \mathit{nzdig}\ \} \\
\mathit{nzdig} & = & \text{“}\texttt{1}\text{”} \mid \text{“}\texttt{2}\text{”} \mid ... \mid \text{“}\texttt{8}\text{”} \mid \text{“}\texttt{9}\text{”} \\
\\
\mathit{range} & = & \text{“}\texttt{range(}\text{”}\ \mathit{variable}\ \text{“}\texttt{,}\text{”}\ \mathit{numvar}\ \text{“}\texttt{,}\text{”}\ \mathit{numvar}\ \text{“}\texttt{)}\text{”} \\
\mathit{variable} & = & \text{“}\texttt{A}\text{”} \mid \text{“}\texttt{B}\text{”} \mid ... \mid \text{“}\texttt{Y}\text{”} \mid \text{“}\texttt{Z}\text{”} \\
\mathit{numvar} & = & \mathit{number} \mid \mathit{variable}
\end{array}
$$

A *graph* production rule denotes a graph with two labeled vertices — the *first* and the *last*. Graphs definition rules have the following semantics:

- The basic building block c denotes a graph with just two vertices (one is the first and the other one is the last) and one edge.
- The $c(\sigma)$ rule connects a specified list of graphs σ from left to right into a chain, merging the last vertex of the first graph in the list with the first vertex of the second graph in the list, the last vertex of the second graph with the first of the third one, and so on. The resulting graph's first vertex is the first vertex of the first graph in the list, and the resulting graph's last vertex is the last vertex of the last graph in the list.
- The $loop(\sigma)$ rule connects a specified list of graphs σ from left to right, merging the last vertex of the first graph in the list with the first vertex of the second graph in the list, and so on like in $c(\sigma)$, while the last vertex of the last graph in the list is merged with the first vertex of the first graph in the list to form a loop. The resulting graph's first and last vertices are the first and the last vertices of the first graph in the list. Loop can be applied only to lists with more than one graph.
- The $t(\sigma)$ rule connects a specified list of graphs σ, merging their first vertices. The resulting graph's first and last vertices are the first and the last vertices of the first graph in the list.

The *list* of graphs is either specified explicitly, by a comma-separated list, or using a *list repetition* with a number, a range, or a variable, optionally followed by a comma and a graph. When a graph is not explicitly specified in a list repetition, then the given graph is assumed to be c.

The simplest list repetition is defined using a *number* non-terminal. It denotes a list of graphs with the specified integer number of copies of the given graph.

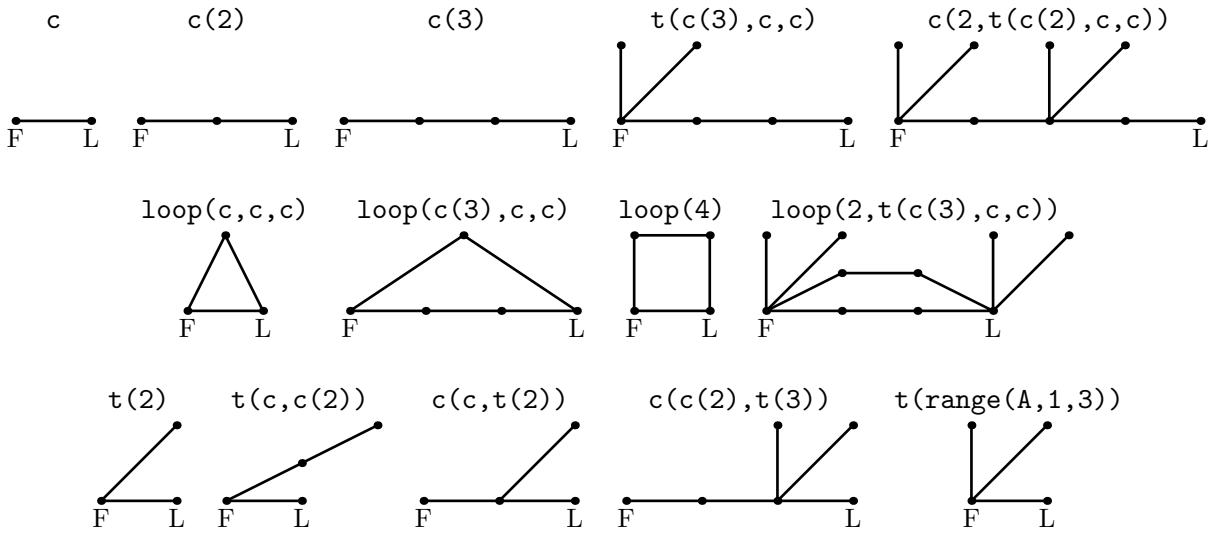
A *range* list repetition is defined by $range(\nu, \alpha, \beta)$ rule which has three components — a variable ν, and numbers α and β. If ξ character sequence is a *graph*, then $c|loop|t(range(\nu, \alpha, \beta), \xi)$ are called *range-enabled* rules and the variable ν is called a *bound variable* in ξ. In the context of a range-enabled rule, ξ is repeated $|\beta - \alpha| + 1$ times to form a list. Every occurrence of variable ν in ξ is replaced by consecutive integer numbers between α and β inclusive in ascending order. That produces a list of $|\beta - \alpha| + 1$ graphs, which are then connected according the specification of the corresponding range-enabled rule. The α and β themselves might refer to variables that are bound in the outer range-enabled rule.

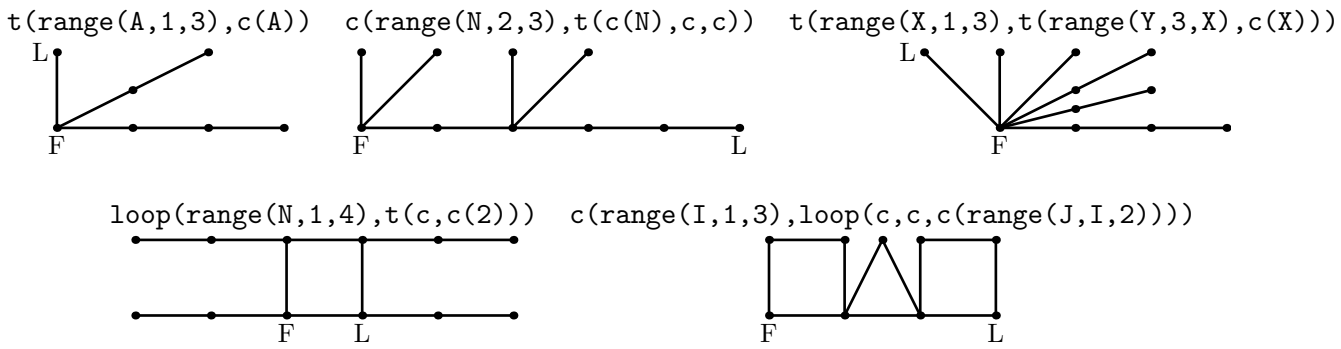
In a *well-formed* graph:

- each *variable* non-terminal (a letter from `A` to `Z`) occurs at most once as ν in $range(\nu, \alpha, \beta)$ rules;
- all other occurrences of *variable* non-terminal that are allowed by the grammar are bound.

Note, that if a character sequence ξ is a *graph*, then ξ, $c(\xi)$, $c(1, \xi)$, $t(\xi)$, and $t(1, \xi)$ all denote the same graph. On the other hand, neither $loop(\xi)$ nor $loop(1, \xi)$ are allowed.

The following examples illustrate these basic rules. The graphs have their first and last vertices marked with letters `F` and `L` correspondingly.





Input

The input file contains a single line with a well-formed cactus definition in SCGL. While the syntax and semantics of SCGL themselves do not guarantee that the resulting graph is a cactus, the input file for this problem always defines a cactus — every edge belongs to at most one simple cycle and there are no multiple edges between vertices. For example, neither `loop(3,loop(3))` nor `loop(2)` are possible.

The line in the input file is at most 1000 characters long and defines a cactus with at most 50 000 vertices. Integer numbers represented by *number* non-terminals do not exceed 50 000.

Output

The first line of the output file must contain two integer numbers n and m. Here n is the number of vertices in the graph. Vertices are numbered from 1 to n, where 1 is the number of the *first* vertex of the graph and n is the number of the *last* vertex of the graph. The other vertices can be numbered arbitrarily. Edges of the graph are represented by a set of edge-distinct paths, where m is the minimal number of such paths.

Each of the following m lines must contain a path in the graph. A path starts with an integer number k_i ($k_i \geq 2$) followed by k_i integers from 1 to n. These k_i integers represent vertices of a path. A path can go to the same vertex multiple times, but every edge must be traversed exactly once in the whole output file.

Sample input and output

cactus.in
`c(c,t(loop(3),c(c,loop(6))),loop(c,c,t(c,loop(4))))`
cactus.out
`15 1` `19 1 2 9 10 11 12 13 10 15 9 14 2 3 4 5 6 7 8 3`

cactus.in	cactus.out
`c`	`2 1` `2 1 2`
`c(2)`	`3 1` `3 1 2 3`
`c(3)`	`4 1` `4 1 2 3 4`

`t(c(3),c,c)`	`6 2`
	`2 1 2`
	`5 3 1 4 5 6`

`c(2,t(c(2),c,c))`	`9 3`
	`3 2 1 3`
	`3 4 5 6`
	`5 1 7 5 8 9`