Problem 1 – Basic Language

Dancho likes only odd programming languages (the one he loves the most is called MSIL, of course). Few days ago he invented a new programming language. He called it BL (Basic Language).

The BL consists of only 3 commands: **PRINT**, **FOR** and **EXIT**. All commands in a BL program are executed consecutively one by one following the rules explained bellow.

The **PRINT** command has the following syntax: **PRINT(**<*text to print*>**);** where <*text to print*> is a consequence of symbols (all symbols are allowed except for the symbol “)”) with length between 0 and 100 000, inclusive. The **PRINT** command outputs the text inside the circle brackets.

Note that the BL language ignores all spaces and line breaks because spaces and line breaks do not affect the semantics of the language. Of course, the spaces and the line breaks inside **PRINT** command should not be ignored.

The **FOR** command represents a loop in BL. It has 2 forms:

- **FOR(a)** – The next command will be executed exactly **a** number of times. **a** will be positive integer between 0 and 2 000 000 000, inclusive.
- **FOR(a, b)** – The next command will be executed exactly **b – a + 1** number of times. **a** will always be less than **b**. **a** and **b** will be integer numbers between -2 000 000 000 and 2 000 000 000, inclusive.

Loops in the BL can be nested. For example, the following code: **FOR(-2,-1) FOR(100) PRINT(x);** will output 200 times the letter **x**.

The **EXIT;** command means the end of every BL program. When **EXIT;** is reached the program ends immediately.

Dancho does not know C# (because C# is not one of the strangest programming languages), so help Dancho execute his own programming language (BL) by writing a program in C# that reads a valid BL code from the console, executes it and writes the BL code output on the console.

Input

The input data should be read from the console.

The input will consist of valid code written in Basic Language (BL), always ending with the **EXIT;** command.

The input data will always be valid and in the format described. There is no need to check it explicitly.

Output

The output data should be printed on the console.

You must write on the console the output result from executing the BL code given in the input.

At the end of your output you must print a new line character.

Constraints

- The input data will be no longer than 100 000 symbols.
- The input data will always be valid Basic Language syntax.
- None combination of **FOR** loops in the BL code will produce more than 10^{18} iterations.

- The output data will be no longer than 1 000 000 symbols.
- Allowed working time for your program: 0.2 seconds.
- Allowed memory: 16 MB.

Examples

Input example	Output example

`PRINT(Black and yellow, );`
`FOR(0,1)PRINT(black and yellow, );`
`PRINT(black and yellow...);`
`EXIT;`
	`Black and yellow, black and yellow, black and yellow, black and yellow...`

`FOR ( 1 , 5 ) PRINT (ha) ;` `FOR(2)FOR(2,3)PRINT(xi);PRINT(i);EXIT;`	`hahahahahaxixixixii`

Problem 2 – Messages in a Bottle

In a warm, sunny day Andrew found a bottle near the sea. It was a very special bottle, containing not one, but two messages. The first message contained a big sequence of digits (0-9). "This must be a secret code", Andrew thought. And he was right. After seeing the second message everything became clearer. The second message said something like this: "**A123C11B98**". Another idea struck Andrew: "Hmm may be this is the cipher, used for creating the secret code". And again he was right.

An alphabetical message, containing only capital English letters, is encoded with the given cipher. For every letter in the original message it is replaced by the given sequence of digits in the cipher.

For example an original message **ABC** with a cipher **A123C11B98** will be encoded as **1239811**.

Write a program that for a given secret code from the first bottle message and a given cipher from the second bottle message finds all possible original messages that can be encoded to the given secret code.

Input

The input data should be read from the console.

On the first input line there will be the secret message (the sequence of digits).

On the second input line there will be the cipher used for encoding the message in the given format: "{LetterX}{CodeForX}{LetterY}{CodeForY}…" where every `LetterX` from the original message will be encoded with `CodeForX` in the secret code. One letter will have only one unique representation.

The input data will always be valid and in the format described. There is no need to check it explicitly.

Output

The output data should be printed on the console.

In the first console line you must print the number of all possible original messages that can be encoded to the given secret code. And these messages should be printed in the next output lines sorted alphabetically. See the examples bellow.

Constraints

- The given secret code will contain no more than 12 digits.
- The cipher will be no longer than 180 symbols, containing only capital English letters and digits.

- The number of original messages (the answers) will be no more than 2048.
- Allowed working time for your program: 0.1 seconds. Allowed memory: 16 MB.

Examples

Input example	Output example
1122 A1B12C11D2	3 AADD ABD CDD
778 Z123A7787X666Y234	0

Problem 3 – Cooking

Krisi is such an amazing cook that she can easily be mistaken with a chef. Today Krisi is cooking a special meal she has never done before. To get it right she needs to follow her grandmother’s recipe. Unfortunately, she has already started cooking and was interrupted by a phone call, while pouring some of the ingredients. Now she remembers what products she has used so far, their quantity and has the recipe in front of her. **Help her finish her dish by writing a program that finds which products from the recipe need to be added, and their quantity**. Bear in mind that Krisi has partly (less than they were required in the recipe) put some of the ingredients.

To make the meal even better, Krisi uses more products than the products given in the recipe and sometimes she uses one product more than once. Her recipe also may contain the same product more than once.

The products are given by their name and quantity in different cooking measurement units. The relation between the measurement units is given in the table below. The synonym of the measurement unit (if any) is given in the brackets. The original names and their synonyms mean the same unit.

1 tablespoons (tbsps) = 3 teaspoons (tsps)	1 gallons (gals) = 4 quarts
1 liters (ls) = 1000 milliliters (mls)	1 pints (pts) = 2 cups
8 fluid ounces (fl ozs) = 1 cups	1 quarts (qts) = 2 pints (pts)
1 teaspoons = 5 milliliters (mls)	1 cups = 48 teaspoons (tsps)

All products from the input and in the output should be in this format: “`{Quantity}:{Measurement unit}:{Product}`”. All measurement units will be given and should be outputted in plural forms. All numbers will be between 1 and 125 000, inclusive and all decimal points will be presented as “.” (point).

Input

On the first line you will be given the number **N** of the products in the recipe. On each of next **N** lines you will be given one product in the described format. On the next line you will be given the number **M** of the products Krisi used so far. From each of the next **M** lines you should read the products Krisi has already used. The case of the letters doesn’t matter (for example: “milk” and “MILK” are the same products).

The input data will always be valid and in the format described. There is no need to check it explicitly.

Output

You should output the ingredients from the recipe that Krisi hasn't used so far and the ones she has already used but their quantity does not match the recipe (it is less than required). The products must be ordered as they appear in the recipe, measured in the same measurement units as given in the recipe for the first time and with the same names as the first product appearance in the recipe. See the examples below.

All numbers must be rounded and printed with 2 digits after the decimal point (XXX.XX).

Constraints

- **N** will be between 1 and 1000, inclusive. **M** will be between 1 and 1000, inclusive.
- Product names length will be between 1 and 50 and will contain only Latin letters and spaces.
- Allowed working time for your program: 0.5 seconds. Allowed memory: 16 MB.

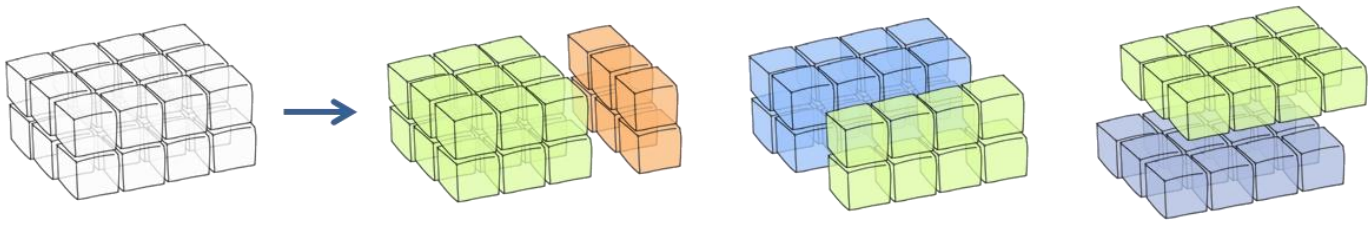
Examples

Input example	Output example
2 1:cups:Sugar 1.006:ls:Old milK 2 800:mls:old MILK 1.5:cups:sugar	0.21:ls:Old milK

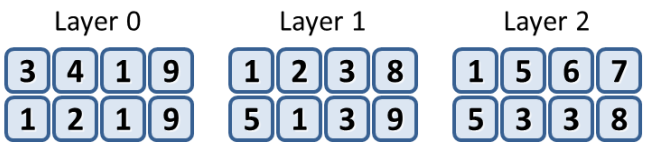
Input example	Output example
2 12:ls:water 12000:mls:Water 1 12:cups:coffee	24.00:ls:water

Problem 4 – 3D Slices

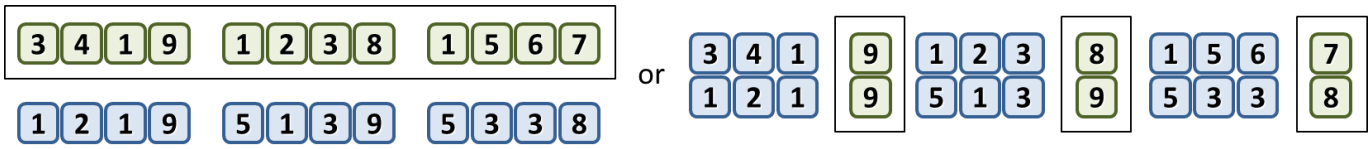
You are given a **rectangular cuboid** of size **W** (width), **H** (height) and **D** (depth) consisting of **W** * **H** * **D** cubes, each containing an integer number. A cuboid can be **split into two sub-cuboids** by slicing it over some of the planes {x, y}, {x, z} or {y, z}. For example a cuboid of size {4 x 3 x 2} could be split into sub-cubes {4 x 3 x 1} + {4 x 3 x 1} or into {1 x 3 x 2} + {3 x 3 x 2} or by few other ways. The figure below shows few examples how we can slice a cube into two non-empty sub-cubes:



The cuboid is given as layers of matrices holding integer numbers. The figure below shows a cuboid of size 4 x 2 x 3 (width = 4, height = 2, depth = 3):



Your task is to write a program that finds in how many ways we can split the cuboid into two non-empty sub-cuboids such that the sums of the numbers in the obtained sub-cuboids are equal. For example the cuboid at the figure could be split into **equal-sum sub-cuboids** as follows:



Input

The input data should be read from the console. At the first line 3 integers **W**, **H** and **D** are given separated by a space. These numbers specify the width, height and depth of the cuboid. At the next **H** lines the colors of the cubes in the cuboid are given as **D** sequences of exactly **W** integers. Each of these sequences consists of **W** integers separated by a single space. The sequences of **W** integers are separated one from another by " **|** " (space + vertical line + space).

The input data will be correct and there is no need to check it explicitly.

Output

The output data should be printed on the console.

On the first line of the output print the **total number of splits of the cuboid into equal-sum sub-cuboids**.

Constraints

- The numbers **W**, **H** and **D** are all integers in the range [1…100].
- The integers in the cuboid are in the range [-1000…1000]
- Allowed work time for your program: 0.5 seconds.
- Allowed memory: 16 MB.

Examples

Input	Output
4 2 3 3 4 1 9 | 1 2 3 8 | 1 5 6 7 1 2 1 9 | 5 1 3 9 | 5 3 3 8	2

Input	Output
2 2 2 1 2 | 3 4 5 6 | 7 8	0

Problem 5 – Secret Language

Little Cecilia and her friends were dismayed to learn that their parents were reading all of their private communications. They decided to invent a new language that would allow them to talk freely. What they finally came up with was a language where sentences are built using a special method.

All the valid words that can be used in the new language are given on the second input line as a list of strings wrapped by a double quotes (") and separated by a comma and a space (see the examples below). A sentence is a concatenation (with no spaces) of a sequence of valid words. Each valid word can appear 0 or more times in the sentence. What makes the language special is that each word can be transformed by rearranging its letters before being used. The cost to transform a word is defined as the number of character positions where the original word and the transformed word differ. For example,

"abc" can be transformed to "abc" with a cost of 0, to "acb", "cba" or "bac" with a cost of 2, and to "bca" or "cab" with a cost of 3.

Although several different sequences of valid words can produce the same sentence in this language, only the sequence with the least total transformation cost carries the meaning of the sentence. The advantage of the new language is that the parents can no longer understand what the kids are saying. The disadvantage is that the kids themselves also do not understand. They need your help.

Given a sentence, return the total cost of transformation of the sequence of valid words which carries the meaning of the sentence, or **-1** if no such sequence exists.

If a word is used multiple times in a sentence, each occurrence can be transformed differently.

Input

The input data should be read from the console. On the first input line you should read the sentence.

On the second input line you should read the list of the valid words in the format described above.

The input data will always be valid and in the format described. There is no need to check it explicitly.

Output

The output data should be printed on the console.

On the only output line you should print the total cost of transformation of the sequence of valid words which carries the meaning of the sentence, or -1 if no such sequence exists.

Constraints

- The sentence will contain between 1 and 50 lowercase Latin letters ('a'-'z'), inclusive.
- The list of valid words will contain between 1 and 50 elements, inclusive.
- Each valid word will contain between 1 and 50 lowercase letters ('a'-'z'), inclusive.
- Allowed working time for your program: 0.1 seconds. Allowed memory: 16 MB.

Examples

Input example	Output example	Explanation
`neotowheret` `"one", "two", "three", "there"`	8	The following transformations can be made: • "one" -> "neo" with a cost of 3 • "two" -> "tow" with a cost of 2 • "three" -> "heret" with a cost of 3 • "there" -> "heret" with a cost of 5 So the sequence {"one", "two", "three"} is the one carrying the meaning of "neotowheret". Its total transformation cost is 3 + 2 + 3 = 8
`sepawaterords` `"this", "is", "meaningful"`	`-1`	You are only allowed to rearrange letters within words, and not in the entire sentence.