7th South African Regional ACM Collegiate Programming Competition

Sponsored by IBM

Problem C – Green balloon Compactor

Dictionary-based compression algorithms, like those belonging to the LZ77 family, typically build up a table of codes representing strings (or *patterns*) that appeared recently in the input. The idea is that when a pattern is encountered again in the input stream, the algorithm can output the code for that pattern, instead of the pattern itself. If the codes are constructed correctly, the code (representing the pattern) will be shorter than the pattern itself, thus achieving compression.

It just so happens that you have already written the hard part of the algorithm that constructs the table of codes and their corresponding patterns. All that is left to do is to parse the input stream optimally, so that, given more than one possible match from your code table, you choose the permutation that minimises the cost (or length) of the output stream.

Consider the following table of patterns, with their associated costs (note that the encodings of the actual codes have been omitted, since they are not relevant to this part of the algorithm):

Pattern	**Cost**
abc	4
ab	1
bcaa	3
a	2
b	3
c	3

Thus, given the table above, consider two possible ways in which the input stream

```
abcaaabbaabc
```

could be parsed. If your algorithm were to implement the "greedy" strategy, the following parsing would result:

```
(abc)(a)(a)(ab)(b)(a)(abc)
```

with a total cost of 4 + 2 + 2 + 1 + 3 + 2 + 4 = 18. By careful inspection, it is possible to see that a better parsing of this input stream is possible, namely

```
(a)(bcaa)(ab)(b)(a)(abc)
```

with a total cost of 2 + 3 + 1 + 3 + 2 + 4 = 15.

Your objective is to implement an algorithm that will determine the optimal parsing of a given input stream, for a given code table. You must print the optimal cost as output.

Input

Each input record will start with the number of codes in the table, *n*. This is followed by *n* pairs of the form
<cost> <pattern>
After the *n* pairs, a single string, <data>, follows.

You may assume that the <pattern> and <data> elements contain only characters in the set 'a' through 'z'.

Your input may contain any number of records of this form, with a value of *n = -1* indicating the end of the input stream.

Output

For each input record, you must output the cost of the optimal parsing of <data> given the code table.

Sample Input

```
6
4 abc
1 ab
3 bcaa
2 a
3 b
3 c
abcaaabbaabc
-1
```

Sample Output

```
15
```