Problem E
File Compression

These days, computers get used more and more widely in our daily life, and the amount of information stored in memory devices or transmitted through communication lines is increasing in dramatic speed. The technique of "file compression" is very important in this context. It encodes a file into more compact form, thus saving memory space and transmission time.

One might argue that file compression is not very valuable now, since large-scale storage devices and high-speed transmission channels can easily and cheaply be obtained. This is not true. The amount of information stored in computers or exchanged between computers is increasing more rapidly than the evolution of the technology of memory devices or transmission channels. File compression methods of high performance are more strongly desired than, say, ten years ago.

There are many well-known methods for file compression. For example, techniques like run-length encoding, Huffman coding, etc. have been used. New methods are being studied intensively, especially in multi-media field where the sizes of files recording pictures, music, etc. are often very large.

In this problem, we consider a typical file-compression technique called the LZ method, and try to implement a very simple version of the method. The principle of the LZ method, after its inventors Ziv and Lempel, is quite simple. It tries to find a repeated string, and if found, replaces the second and subsequent occurrences of the string with the references to the first occurrence.

In order to discuss the method more rigorously, we introduce a few notations. Assume that the length of the file, that is the number of characters in the file, is n. We call characters in the text file $a[0]$, $a[1]$, ..., $a[n-1]$. Note that the first character in the file is numbered zero. Let us denote a string in the file as $a[s..t]$ ($0 <= s <= t <= n-1$). This means a contiguous portion of the file starting at position s and ending at position t, both inclusive.

At each position q in the file ($0 <= q <= n-1$), the method tries to find a string $a[p..p+r-1]$ which starts before the position q and is equal to $a[q..q+r-1]$ ($p < q$). If such a matching is found, it removes the string $a[q..q+r-1]$ from the file, and inserts an adequate representation of the pair (p, r) instead. The pair (p, r) is the position of the first string and the length of the matched string. Since the representation of the pair (p, r) would be shorter than r in usual cases, file space can be saved.

This method is quite effective for a file with many repeated patterns. It is not effective for a random file.

One of the key design issues of this method is the representation of the pair (p, r). Binary encoding of two numbers is usually used. However, in this problem, we stick to textual representation, since sending binary data to the judges' machine might introduce unexpected difficulties. The resulted file will be larger than the results of the usual file compression utilities, but this is not our concern here.

Input

The input is a text file which is a sequence of input segments. Each input segment should be compressed independently. In the above explanation, we discussed the compression of a "file", but in the following we do not use this term. Each "input segment" should be processed.

The input is a text file which contains only printable characters (ASCII codes 20 to 7E in hexadecimal) and newlines. No other control characters such as horizontal or vertical tab appear in the input. Note that the physical representation of newline is different from machine to machine. Some machines use two characters (bytes) to represent a single newline. Nonetheless, we regard a newline as a single character in the rest of this problem.

An input segment is a non-empty sequence of non-empty lines terminated by an empty line. The empty line is not part of the input segment. Here, a non-empty line is a non-empty sequence of printable characters followed by a single newline. An empty line consists of a single newline.

There is one more empty line after the last input segment. In other words, the input is terminated by two consecutive empty lines.

The total number of characters in an input segment does not exceed 30000.

Output

Each input segment in the input should be compressed and the result (an output segment) should be output. Each output segment should be followed by a line containing exactly 72 '+' characters and a single newline.

An output segment is a copy of the input segment unless repeated patterns longer than a certain length, to be given below, appear in the input segment. No characters or newlines in the input segment should be added or deleted.

If a string in the input segment $a[q..q+r\text{-}1]$ is equal to $a[p..p+r\text{-}1]$ ($q > p$), and r is larger than or equal to 7, $a[q..q+r\text{-}1]$ should be deleted from the output segment, and the following representation should be inserted instead.

> `%` [representation of p] `%` [representation of r] [newline]

We use radix-64 notation in the representation of numbers, p and r. Characters (meta-digits) used in the representation are as follows: Upper-case letters '`A`'-'`Z`' denote values 0-25, lower-case letters '`a`'-'`z`' denote 26-51, digits '`0`'-'`9`' denote 52-61, '`+`' denotes 62, and '`/`' denotes 63. For example, decimal 123 should be denoted by "`B7`". This number representation is known as "base64" encoding, which is regarded as a de-facto standard now. Use minimum number of meta-digits without preceding '`A`'s (the representation of 0 is "`A`" and this is an exception).

Sometimes, you may find two or more matchings for a string. That is, it may be the case that $a[q..q+r1\text{-}1] = a[p1..p1+r1\text{-}1]$ and $a[q..q+r2\text{-}1] = a[p2..p2+r2\text{-}1]$. In such cases, we prefer (1) the longest matching, and if the length of the matchings are equal (2) the earliest occurrence of the matched string. In other words, if $r1 > r2$ then $(p1, r1)$ should be output. If $r1 = r2$ and $p1 < p2$ then $(p1, r1)$ should be output.

This choice minimizes the file space in most cases, but not always. For example, if $r1 = r2+1$ and $p1 = 64*64*p2$, we should choose $(p2, r2)$ to minimize space. However, we do not take such complexity into account. The simple rule mentioned above should be used.

Even overlapping strings are candidates for the matching. For example, the first to the ninth characters of "`AAAAAAAAAA`" match the second to the tenth characters of the same string. In this case, the first '`A`' would be output, but the second to the tenth characters should be replaced by the reference to the string one position left.

The input segment should be scanned from left to right. That is, the lower-numbered strings should be processed before the higher-numbered strings (smaller q first). Once a string is replaced by the reference to the matched string, it is never cancelled, even if the cancellation would lead to a shorter compressed result. For example, consider a string "`(a)-1234567 (b):23456789 (c)=123456789`". The best result would be obtained by matching "`23456789`" of (c) with (b), skipping the first '`1`' of (c). But we do not consider such a replacement. Before trying to find a previous occurrence of "`23456789`", the search for "`1234567`" is performed and a matching string in (a) is found. "`1234567`" is replaced by an adequate representation, and the search for "`23456789`" is not performed.

As a similar but slightly different case, consider a string "`(a)-23456789 (b):1234567 (c)=123456789`". In this case also, "`1234567`" is matched with (b) and the matching of "`23456789`" is not tried.

After all the replacements have been made, the '`%`' (percent) characters contained in the original input segment should be doubled in the output segment, so that they could be discriminated from the '`%`'s in the replacement strings.

Note that the compression scheme discussed above is quite different from the LZ method used in common utilities. They usually assume certain upper bounds for the length of the matching *r* or the distance of two strings *q-p*. Utilities are designed to work in environments where only a small amount of memory is available, while we ignored such practical issues in this problem.

Sample Input

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

fifty % of fifty % is twenty-five %

89abcde--23456789==1234567::123456789abcde
```

Output for the Sample Input

```
a%A%x

%A%D/
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
fifty %% of %A%I
is twenty-five %%
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
89abcde--23456789==1234567::%T%H
%A%H

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

First Input Data

Your first input data is here.

The ACM ICPC