A. Turing Tape

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

INTERCAL is the oldest of esoteric programming languages. One of its many weird features is the method of character-based output, known as Turing Tape method. It converts an array of unsigned 8-bit integers into a sequence of characters to print, using the following method.

The integers of the array are processed one by one, starting from the first. Processing i-th element of the array is done in three steps:

1. The 8-bit binary notation of the ASCII-code of the previous printed character is reversed. When the first element of the array is processed, the result of this step is considered to be 0.

2. The i-th element of the array is subtracted from the result of the previous step modulo 256.

3. The binary notation of the result of the previous step is reversed again to produce ASCII-code of the i-th character to be printed.

You are given the text printed using this method. Restore the array used to produce this text.

Input

The input will consist of a single line $text$ which contains the message printed using the described method. String $text$ will contain between 1 and 100 characters, inclusive. ASCII-code of each character of $text$ will be between 32 (space) and 126 (tilde), inclusive.

Output

Output the initial array, which was used to produce $text$, one integer per line.

Examples

input

```
Hello, World!
```

output

```
238
108
112
0
64
194
48
26
244
168
24
16
162
```

Note

Let's have a closer look at the beginning of the example. The first character is "H" with ASCII-code $72 = 01001000_2$. Its reverse is $00010010_2 = 18$, and this number should become the result of the second step of processing. The result of the first step is considered to be 0, so the first element of the array has to be $(0 - 18)$ mod $256 = 238$, where a mod b is the remainder of division of a by b.

B. Piet

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

Piet is one of the most known visual esoteric programming languages. The programs in Piet are constructed from colorful blocks of pixels and interpreted using pretty complicated rules. In this problem we will use a subset of Piet language with simplified rules.

The program will be a rectangular image consisting of colored and black pixels. The color of each pixel will be given by an integer number between 0 and 9, inclusive, with 0 denoting black. A block of pixels is defined as a rectangle of pixels of the same color (not black). It is guaranteed that all connected groups of colored pixels of the same color will form rectangular blocks. Groups of black pixels can form arbitrary shapes.

The program is interpreted using movement of instruction pointer (IP) which consists of three parts:

- current block pointer (BP); note that there is no concept of current pixel within the block;
- direction pointer (DP) which can point left, right, up or down;
- block chooser (CP) which can point to the left or to the right from the direction given by DP; in absolute values CP can differ from DP by 90 degrees counterclockwise or clockwise, respectively.

Initially BP points to the block which contains the top-left corner of the program, DP points to the right, and CP points to the left (see the orange square on the image below).

One step of program interpretation changes the state of IP in a following way. The interpreter finds the furthest edge of the current color block in the direction of the DP. From all pixels that form this edge, the interpreter selects the furthest one in the direction of **CP**. After this, BP attempts to move from this pixel into the next one in the direction of DP. If the next pixel belongs to a colored block, this block becomes the current one, and two other parts of IP stay the same. It the next pixel is black or outside of the program, BP stays the same but two other parts of IP change. If CP was pointing to the left, now it points to the right, and DP stays the same. If CP was pointing to the right, now it points to the left, and DP is rotated 90 degrees clockwise.

This way BP will never point to a black block (it is guaranteed that top-left pixel of the program will not be black).

You are given a Piet program. You have to figure out which block of the program will be current after n steps.

Input

The first line of the input contains two integer numbers m ($1 \le m \le 50$) and n ($1 \le n \le 5 \cdot 10^7$). Next m lines contain the rows of the program. All the lines have the same length between 1 and 50 pixels, and consist of characters `0`-`9`. The first character of the first line will not be equal to 0.

Output

Output the color of the block which will be current after n steps of program interpretation.

Examples

input

```
2 10
12
43
```

output

```
1
```

input

```
3 12
1423
6624
6625
```

output

```
6
```

input

```
5 9
10345
23456
34567
45678
56789
```

output

```
5
```

Note

In the first example IP changes in the following way. After step 1 block 2 becomes current one and stays it after two more steps. After step 4 BP moves to block 3, after step 7 — to block 4, and finally after step 10 BP returns to block 1.

The sequence of states of IP is shown on the image: the arrows are traversed clockwise, the main arrow shows direction of DP, the side one — the direction of CP.

C. Logo Turtle

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

A lot of people associate Logo programming language with turtle graphics. In this case the turtle moves along the straight line and accepts commands "T" ("turn around") and "F" ("move 1 unit forward").

You are given a list of commands that will be given to the turtle. You have to change exactly n commands from the list (one command can be changed several times). How far from the starting point can the turtle move after it follows **all** the commands of the modified list?

Input

The first line of input contains a string $commands$ — the original list of commands. The string $commands$ contains between 1 and 100 characters, inclusive, and contains only characters "T" and "F".

The second line contains an integer n ($1 \le n \le 50$) — the number of commands you have to change in the list.

Output

Output the maximum distance from the starting point to the ending point of the turtle's path. The ending point of the turtle's path is turtle's coordinate after it follows **all** the commands of the modified list.

Examples

input

```
FT
1
```

output

```
2
```

input

```
FFFTFFF
2
```

output

```
6
```

Note

In the first example the best option is to change the second command ("T") to "F" — this way the turtle will cover a distance of 2 units.

In the second example you have to change two commands. One of the ways to cover maximal distance of 6 units is to change the fourth command and first or last one.

D. Constants in the language of Shakespeare

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

Shakespeare is a widely known esoteric programming language in which programs look like plays by Shakespeare, and numbers are given by combinations of ornate epithets. In this problem we will have a closer look at the way the numbers are described in Shakespeare.

Each constant in Shakespeare is created from non-negative powers of 2 using arithmetic operations. For simplicity we'll allow only addition and subtraction and will look for a representation of the given number which requires a minimal number of operations.

You are given an integer n. You have to represent it as $n = a_1 + a_2 + \ldots + a_m$, where each of a_i is a non-negative power of 2, possibly multiplied by -1. Find a representation which minimizes the value of m.

Input

The only line of input contains a positive integer n, written as its binary notation. The length of the notation is at most 10^6. The first digit of the notation is guaranteed to be 1.

Output

Output the required minimal m. After it output m lines. Each line has to be formatted as "+2^x" or "-2^x", where x is the power coefficient of the corresponding term. The order of the lines doesn't matter.

Examples

input

```
1111
```

output

```
2
+2^4
-2^0
```

input

```
1010011
```

output

```
4
+2^0
+2^1
+2^4
+2^6
```

E. Bits of merry old England

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

Another feature of Shakespeare language is that the variables are named after characters of plays by Shakespeare, and all operations on them (value assignment, output etc.) look like a dialog with other characters. New values of variables are defined in a rather lengthy way, so a programmer should try to minimize their usage.

You have to print the given sequence of n integers. To do this, you have m variables and two types of operations on them:

- `variable=integer`
- `print(variable)`

Any of the m variables can be used as `variable`. Variables are denoted by lowercase letters between "a" and "z", inclusive. Any integer number can be used as `integer`.

Let's say that the penalty for using first type of operations equals to the number of set bits in the number `integer`. There is no penalty on using second type of operations. Find and output the program which minimizes the penalty for printing the given sequence of numbers.

Input

The first line of input contains integers n and m ($1 \le n \le 250$, $1 \le m \le 26$). The second line contains the sequence to be printed. Each element of the sequence is an integer between 1 and 10^9, inclusive. The sequence has to be printed in the given order (from left to right).

Output

Output the number of lines in the optimal program and the optimal penalty. Next, output the program itself, one command per line. If there are several programs with minimal penalty, output any of them (you have only to minimize the penalty).

Examples

input

```
7 2
1 2 2 4 2 1 2
```

output

```
11 4
b=1
print(b)
a=2
print(a)
print(a)
b=4
print(b)
print(a)
b=1
print(b)
print(a)
```

input

```
6 3
1 2 3 1 2 3
```

output

```
9 4
c=1
print(c)
b=2
print(b)
a=3
print(a)
print(c)
print(b)
print(a)
```