Codeforces Beta Round #96 (Div. 2)

A. HQ9+

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

HQ9+ is a joke programming language which has only four one-character instructions:

- "H" prints "Hello, World!",
- "Q" prints the source code of the program itself,
- "9" prints the lyrics of "99 Bottles of Beer" song,
- "+" increments the value stored in the internal accumulator.

Instructions "H" and "Q" are case-sensitive and must be uppercase. The characters of the program which are not instructions are ignored.

You are given a program written in HQ9+. You have to figure out whether executing this program will produce any output.

Input

The input will consist of a single line p which will give a program in HQ9+. String p will contain between 1 and 100 characters, inclusive. ASCII-code of each character of p will be between 33 (exclamation mark) and 126 (tilde), inclusive.

Output

Output "YES", if executing the program will produce any output, and "NO" otherwise.

Sample test(s)

input
Hi!
output
YES

input
Codeforces
output
NO

Note

In the first case the program contains only one instruction — "H", which prints "Hello, World!".

In the second case none of the program characters are language instructions.

B. Unary

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

Unary is a minimalistic Brainfuck dialect in which programs are written using only one token.

Brainfuck programs use 8 commands: "+", "-", "[", "]", "<", ">", "." and "," (their meaning is not important for the purposes of this problem). Unary programs are created from Brainfuck programs using the following algorithm. First, replace each command with a corresponding binary code, using the following conversion table:

- ">" $\longrightarrow$ 1000,
- "<" $\longrightarrow$ 1001,
- "+" $\longrightarrow$ 1010,
- "-" $\longrightarrow$ 1011,
- "." $\longrightarrow$ 1100,
- "," $\longrightarrow$ 1101,
- "[" $\longrightarrow$ 1110,
- "]" $\longrightarrow$ 1111.

Next, concatenate the resulting binary codes into one binary number in the same order as in the program. Finally, write this number using unary numeral system — this is the Unary program equivalent to the original Brainfuck one.

You are given a Brainfuck program. Your task is to calculate the size of the equivalent Unary program, and print it modulo 1000003 $(10^6 + 3)$.

Input

The input will consist of a single line p which gives a Brainfuck program. String p will contain between 1 and 100 characters, inclusive. Each character of p will be "+", "-", "[", "]", "<", ">", "." or ",".

Output

Output the size of the equivalent Unary program modulo 1000003 $(10^6 + 3)$.

Sample test(s)

input
,.
output
220

input
++++[>,.<-]
output
61425

Note

To write a number n in unary numeral system, one simply has to write 1 n times. For example, 5 written in unary system will be 11111.

In the first example replacing Brainfuck commands with binary code will give us 1101 1100. After we concatenate the codes, we'll get 11011100 in binary system, or 220 in decimal. That's exactly the number of tokens in the equivalent Unary program.

C. 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.

Sample test(s)

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 $(0 - 18) \bmod 256 = 238$, where $a \bmod b$ is the remainder of division of a by b.

D. 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.

Sample test(s)

input

```
2 10
12
43
```

output

```
1
```

input

```
3 12
1423
6624
6625
```

output

```
6
```

input

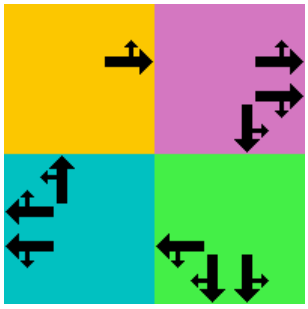
```
5 9
10345
23456
34567
45678
56789
```

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.

E. 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.

Sample test(s)

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.