A. Kalevitch and Chess

time limit per test: 2 seconds
memory limit per test: 64 megabytes
input: standard input
output: standard output

A famous Berland's painter Kalevitch likes to shock the public. One of his last obsessions is chess. For more than a thousand years people have been playing this old game on uninteresting, monotonous boards. Kalevitch decided to put an end to this tradition and to introduce a new attitude to chessboards.

As before, the chessboard is a square-checkered board with the squares arranged in a 8×8 grid, each square is painted black or white. Kalevitch suggests that chessboards should be painted in the following manner: there should be chosen a horizontal or a vertical line of 8 squares (i.e. a row or a column), and painted black. Initially the whole chessboard is white, and it can be painted in the above described way one or more times. It is allowed to paint a square many times, but after the first time it does not change its colour any more and remains black. Kalevitch paints chessboards neatly, and it is impossible to judge by an individual square if it was painted with a vertical or a horizontal stroke.

Kalevitch hopes that such chessboards will gain popularity, and he will be commissioned to paint chessboards, which will help him ensure a comfortable old age. The clients will inform him what chessboard they want to have, and the painter will paint a white chessboard meeting the client's requirements.

It goes without saying that in such business one should economize on everything — for each commission he wants to know the minimum amount of strokes that he has to paint to fulfill the client's needs. You are asked to help Kalevitch with this task.

Input

The input file contains 8 lines, each of the lines contains 8 characters. The given matrix describes the client's requirements, `W` character stands for a white square, and `B` character — for a square painted black.

It is guaranteed that client's requirments can be fulfilled with a sequence of allowed strokes (vertical/column or horizontal/row).

Output

Output the only number — the minimum amount of rows and columns that Kalevitch has to paint on the white chessboard to meet the client's requirements.

Sample test(s)

input

```
WWWBWWBW
BBBBBBBB
WWWBWWBW
WWWBWWBW
WWWBWWBW
WWWBWWBW
WWWBWWBW
WWWBWWBW
```

output

```
3
```

input

```
WWWWWWWW
BBBBBBBB
WWWWWWWW
WWWWWWWW
WWWWWWWW
WWWWWWWW
WWWWWWWW
WWWWWWWW
```

output

```
1
```

B. Memory Manager

time limit per test: 1 second
memory limit per test: 64 megabytes
input: standard input
output: standard output

There is little time left before the release of the first national operating system BerlOS. Some of its components are not finished yet — the memory manager is among them. According to the developers' plan, in the first release the memory manager will be very simple and rectilinear. It will support three operations:

- `alloc n` — to allocate n bytes of the memory and return the allocated block's identifier x;
- `erase x` — to erase the block with the identifier x;
- `defragment` — to defragment the free memory, bringing all the blocks as close to the beginning of the memory as possible and preserving their respective order;

The memory model in this case is very simple. It is a sequence of m bytes, numbered for convenience from the first to the m-th.

The first operation `alloc n` takes as the only parameter the size of the memory block that is to be allocated. While processing this operation, a free block of n successive bytes is being allocated in the memory. If the amount of such blocks is more than one, the block closest to the beginning of the memory (i.e. to the first byte) is prefered. All these bytes are marked as not free, and the memory manager returns a 32-bit integer numerical token that is the identifier of this block. If it is impossible to allocate a free block of this size, the function returns `NULL`.

The second operation `erase x` takes as its parameter the identifier of some block. This operation frees the system memory, marking the bytes of this block as free for further use. In the case when this identifier does not point to the previously allocated block, which has not been erased yet, the function returns `ILLEGAL_ERASE_ARGUMENT`.

The last operation `defragment` does not have any arguments and simply brings the occupied memory sections closer to the beginning of the memory without changing their respective order.

In the current implementation you are to use successive integers, starting with 1, as identifiers. Each successful `alloc` operation procession should return following number. Unsuccessful `alloc` operations do not affect numeration.

You are to write the implementation of the memory manager. You should output the returned value for each `alloc` command. You should also output `ILLEGAL_ERASE_ARGUMENT` for all the failed `erase` commands.

Input

The first line of the input data contains two positive integers t and m ($1 \le t \le 100; 1 \le m \le 100$), where t — the amount of operations given to the memory manager for processing, and m — the available memory size in bytes. Then there follow t lines where the operations themselves are given. The first operation is `alloc n` ($1 \le n \le 100$), where n is an integer. The second one is `erase x`, where x is an arbitrary 32-bit integer numerical token. The third operation is `defragment`.

Output

Output the sequence of lines. Each line should contain either the result of `alloc` operation procession , or `ILLEGAL_ERASE_ARGUMENT` as a result of failed `erase` operation procession. Output lines should go in the same order in which the operations are processed. Successful procession of `alloc` operation should return integers, starting with 1, as the identifiers of the allocated blocks.

Sample test(s)

input

```
6 10
alloc 5
alloc 3
erase 1
alloc 6
defragment
alloc 6
```

output

```
1
2
NULL
3
```

C. Line

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

A line on the plane is described by an equation $Ax + By + C = 0$. You are to find any point on this line, whose coordinates are integer numbers from $-5 \cdot 10^{18}$ to $5 \cdot 10^{18}$ inclusive, or to find out that such points do not exist.

Input
The first line contains three integers A, B and C ($-2 \cdot 10^9 \le A, B, C \le 2 \cdot 10^9$) — corresponding coefficients of the line equation. It is guaranteed that $A^2 + B^2 > 0$.

Output
If the required point exists, output its coordinates, otherwise output `-1`.

Sample test(s)

input
2 5 3
output
6 -3

D. Palindrome Degree

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

String s of length n is called k-palindrome, if it is a palindrome itself, and its prefix and suffix of length $\lfloor n/2 \rfloor$ are $(k - 1)$-palindromes. By definition, any string (even empty) is 0-palindrome.

Let's call the palindrome degree of string s such a maximum number k, for which s is k-palindrome. For example, "`abaaba`" has degree equals to 3.

You are given a string. Your task is to find the sum of the palindrome degrees of all its prefixes.

Input

The first line of the input data contains a non-empty string, consisting of Latin letters and digits. The length of the string does not exceed $5 \cdot 10^6$. The string is case-sensitive.

Output

Output the only number — the sum of the polindrome degrees of all the string's prefixes.

Sample test(s)

input

```
a2A
```

output

```
1
```

input

```
abacaba
```

output

```
6
```

E. Defining Macros

time limit per test: 3 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

Most C/C++ programmers know about excellent opportunities that preprocessor #define directives give; but many know as well about the problems that can arise because of their careless use.

In this problem we consider the following model of #define constructions (also called macros). Each macro has its name and value. The generic syntax for declaring a macro is the following:

#define *macro_name macro_value*

After the macro has been declared, "macro_name" is replaced with "macro_value" each time it is met in the program (only the whole tokens can be replaced; i.e. "macro_name" is replaced only when it is surrounded by spaces or other non-alphabetic symbol). A "macro_value" within our model can only be an arithmetic expression consisting of variables, four arithmetic operations, brackets, and also the names of previously declared macros (in this case replacement is performed sequentially). The process of replacing macros with their values is called substitution.

One of the main problems arising while using macros — the situation when as a result of substitution we get an arithmetic expression with the changed order of calculation because of different priorities of the operations.

Let's consider the following example. Say, we declared such a #define construction:

#define sum x + y

and further in the program the expression "2 * sum" is calculated. After macro substitution is performed we get "2 * x + y", instead of intuitively expected "2 * (x + y)".

Let's call the situation "suspicious", if after the macro substitution the order of calculation changes, falling outside the bounds of some macro. Thus, your task is to find out by the given set of #define definitions and the given expression if this expression is suspicious or not.

Let's speak more formally. We should perform an ordinary macros substitution in the given expression. Moreover, we should perform a "safe" macros substitution in the expression, putting in brackets each macro value; after this, guided by arithmetic rules of brackets expansion, we can omit some of the brackets. If there exist a way to get an expression, absolutely coinciding with the expression that is the result of an ordinary substitution (character-by-character, but ignoring spaces), then this expression and the macros system are called correct, otherwise — suspicious.

Note that we consider the "/" operation as the usual mathematical division, not the integer division like in C/C++. That's why, for example, in the expression "a*(b/c)" we can omit brackets to get the expression "a*b/c".

Input

The first line contains the only number n ($0 \le n \le 100$) — the amount of #define constructions in the given program.

Then there follow n lines, each of them contains just one #define construction. Each construction has the following syntax:

#define *name expression*

where

- *name* — the macro name,
- *expression* — the expression with which the given macro will be replaced. An expression is a non-empty string, containing digits,names of variables, names of previously declared macros, round brackets and operational signs +-*/. It is guaranteed that the expression (before and after macros substitution) is a correct arithmetic expression, having no unary operations. The expression contains only non-negative integers, not exceeding 10^9.

All the names (#define constructions' names and names of their arguments) are strings of case-sensitive Latin characters. It is guaranteed that the name of any variable is different from any #define construction.

Then, the last line contains an *expression* that you are to check. This expression is non-empty and satisfies the same limitations as the expressions in #define constructions.

The input lines may contain any number of spaces anywhere, providing these spaces do not break the word "define" or the names of constructions and variables. In particular, there can be any number of spaces before and after the "#" symbol.

The length of any line from the input file does not exceed 100 characters.

Output

Output "OK", if the expression is correct according to the above given criterion, otherwise output "Suspicious".

Sample test(s)

input

```
1
#define sum x + y
1 * sum
```

output

```
Suspicious
```

input

```
1
#define sum  (x + y)
sum - sum
```

output

```
OK
```

input

```
4
#define sum  x + y
#define mul  a * b
#define div  a / b
#define expr sum + mul * div * mul
expr
```

output

```
OK
```

input

```
3
#define SumSafe   (a+b)
#define DivUnsafe  a/b
#define DenominatorUnsafe  a*b
((SumSafe) + DivUnsafe/DivUnsafe + x/DenominatorUnsafe)
```

output

```
Suspicious
```