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".

Examples

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
```