One important problem in concurrent programming is to ensure exclusive access to shared resources by multiple threads. It is also known as Mutual Exclusion protocol. A code that needs to be protected from concurrent execution is called *critical section* (*CS*). In order to coordinate access to CS, application threads use a set of shared variables to send information to each other. These shared variables are distinct from all the variables that are used by application code. In practice, mutual exclusion protocol is implemented as two methods — *enterCS* and *exitCS*. When application needs to execute some code in CS, it calls enterCS, then executes CS, then calls exitCS.

For theoretical analysis of mutual exclusion protocol one must consider running application as a whole. Each thread of application is represented as an infinite loop that repeatedly performs some work unrelated to CS, which is called *non-critical section* (*NCS*), then calls enterCS, then executes CS, then calls exitCS, then the loop repeats. The code inside NCS and CS is not relevant; it is considered to perform no operations related to the protocol and does not modify shared variables used by the protocol.

We consider a system with two concurrently running threads. Threads use a set of shared one-bit variables to implement mutual exclusion protocol. Each variable can store a value of zero or one that can be read or written by a single instruction. Shared variables are initialized to zero. Each thread has a local pointer to the instruction (*IP*) that it is going to execute next. Execution starts from the top of the code. During each step of execution one of the threads is arbitrarily chosen, it executes one instruction, and then changes its IP to the next instruction to execute. This infinite sequence of steps is called *history*. A history is called *legal* if either both threads execute infinitely many steps or just one thread does, while the other thread, having taken a finite number of steps, stops with IP at NCS.

The table below contains several algorithms in pseudo-code that attempt to implement mutual exclusion protocol. In this pseudo-code *id* is 0 for the first thread and 1 for the second. Variables *want*[0], *want*[1], and *turn* are shared between threads to implement mutual exclusion protocol. Lines marked with '+' implement enterCS, lines marked with '−' implement exitCS. Lines NCS() and CS() are placeholders for some code that works inside non-critical and critical sections respectively and is not relevant for this problem.

Algorithm 1	Algorithm 2	Algorithm 3

`loop forever`
`  NCS()`
`+ loop while`
`+  (turn == 1 - id)`
`  CS()`
`- turn <- (1 - id)`
`end loop`
	`loop forever`
	`  NCS()`
	`+ want[id] <- 1`
	`+ loop while`
	`+  (want[1 - id] == 1)`
	`  CS()`
	`- want[id] <- 0`
	`end loop`
		`loop forever`
		`  NCS()`
		`+  want[id] <- 1`
		`+  turn <- (1 - id)`
		`+  loop while`
		`+  (want[1 - id] == 1 and`
		`+  turn == 1 - id)`
		`  CS()`
		`-  want[id] <- 0`
		`end loop`

The task is to figure out if the given algorithm satisfies three important properties:

- The algorithm satisfies *mutual exclusion* if in any legal history CS is not executed concurrently by two threads (that is, there is no step where IP of both threads is at CS).
- The algorithm satisfies *deadlock freedom* if any legal history has infinitely many executions of CS.
- The algorithm satisfies *starvation freedom* if in any legal history a thread that executes infinitely many steps has infinitely many executions of CS.

The property of mutual exclusion is trivial. The algorithm that simply loops forever doing nothing will satisfy it. The sample algorithms above all satisfy mutual exclusion, but the first two fail to achieve deadlock freedom. The algorithm 3 (originally created by Gary Peterson) satisfies all three properties.

Input

The input file contains several test cases, each of them as described below.

The input starts with a line with two integer numbers — m_1 and m_2, where m_i is the number of lines of code for i-th thread ($2 \le m_i \le 9$). It is followed by m_1 lines with the code for the first thread and m_2 lines with the code for the second thread.

The code for each thread contains one instruction per line. Instruction starts with an integer line number from 1 to m_i (lines are numbered in ascending order and are included to aid readability), followed by instruction mnemonic, followed by a list of instruction arguments, all separated by spaces. The last arguments of instruction represent line numbers of the next instructions to execute (NIP — from 1 to m_i). There are three variables shared between threads — '`A`', '`B`', and '`C`'. Instruction mnemonics are:

- `NCS` — non-critical section placeholder. Its single argument is NIP.
- `CS` — critical section placeholder. Its single argument is NIP.
- `SET` — write value to the shared variable. It has three arguments v, x, and g, where v is the variable to write ('`A`', '`B`', or '`C`'), x is the value to write (0 or 1), and g is NIP.
- `TEST` — read and test the value of the shared variable. It has three arguments v, g_0, and g_1 where v is the variable to read ('`A`', '`B`', or '`C`'), g_0 is NIP if the value of the variable is zero, and g_1 is NIP if the value of the variable is one.

`NCS` and `CS` appear in the code for each thread exactly once. The code may or may not represent a simple loop, but is guaranteed to alternate executions of `CS` and `NCS` by one thread, that is, in every legal history two executions of `CS` by one thread always have `NCS` execution by the same thread in between and, vice versa, two executions of `NCS` by one thread have `CS` execution by the same thread in between.

Output

For each test case, write to the output a string of three letters on a line by itself.

Letters represent properties of mutual exclusion, deadlock freedom, and starvation freedom. Write letter '`Y`' if the corresponding property is satisfied and '`N`' otherwise.

Note:

Three first samples below represent algorithms 1–3 from the problem statement.

The fourth one is an algorithm (originally created by Leslie Lamport) that uses just two shared bits (A and B) and satisfies mutual exclusion and deadlock freedom, but is not free from starvation.

Last two are trivial algorithms. First one never executes CS nor NCS and thus guarantees mutual exclusion, but does not have deadlock freedom, nor starvation freedom properties. Second one loops between NCS and CS, thus fails to achieve mutual exclusion, but is free from deadlock and starvation.

Sample Input

```
4 4
1 NCS 2
2 TEST C 3 2
3 CS 4
4 SET C 1 1
1 NCS 2
2 TEST C 2 3
3 CS 4
4 SET C 0 1
5 5
1 NCS 2
2 SET A 1 3
3 TEST B 4 3
4 CS 5
5 SET A 0 1
1 NCS 2
2 SET B 1 3
3 TEST A 4 3
4 CS 5
5 SET B 0 1
7 7
1 NCS 2
2 SET A 1 3
3 SET C 1 4
4 TEST B 6 5
5 TEST C 6 4
6 CS 7
7 SET A 0 1
1 NCS 2
2 SET B 1 3
3 SET C 0 4
4 TEST A 6 5
5 TEST C 4 6
6 CS 7
7 SET B 0 1
5 7
1 NCS 2
2 SET A 1 3
3 TEST B 4 3
4 CS 5
5 SET A 0 1
1 NCS 2
2 SET B 1 3
3 TEST A 6 4
4 SET B 0 5
5 TEST A 2 5
6 CS 7
7 SET B 0 1
3 3
1 SET A 0 1
2 CS 2
3 NCS 3
1 TEST A 1 1
2 CS 2
3 NCS 3
2 2
1 CS 2
2 NCS 1
1 NCS 2
2 CS 1
```

Sample Output

```
YNN
YNN
YYY
YYN
YNN
NYY
```