What Really Happened on Mars?

Real-time software in the Mars Pathfinder spacecraft suffered from an issue known as priority inversion. One technique to address this issue is to use the Priority Ceiling Protocol.

In this problem, you will simulate the execution of multiple tasks according to this protocol. The tasks share a collection of resources, each of which can be used by only one task at a time. To ensure this, resources must be locked before use and unlocked after use. Each task is defined by a start time, a unique *base priority*, and a sequence of instructions. Each task also has a *current priority*, which may change during execution. Instructions come in three types:

- *compute* – perform a computation for one microsecond
- *lock k* – lock resource *k* (which takes no processor time)
- *unlock k* – unlock resource *k* (which takes no processor time)

After locking a resource, a task is said to *own* the resource until the task unlocks it. A task will unlock only the owned resource it most recently locked, will not lock a resource it already owns, and will complete with no owned resources.

Each resource has a fixed *priority ceiling*, which is the highest base priority of any task that contains an instruction to lock that resource.

There is a single processor that executes the tasks. When the processor starts, it initializes its clock to zero and then runs an infinite loop with the following steps:

Step 1.

Identify *running* tasks. A task is running if its start time is less than or equal to the current processor clock and not all of its instructions have been executed.

Step 2.

Determine the current priorities of the running tasks and which of the running tasks are *blocked*. A running task T is blocked if the next instruction in T is to lock resource k and either resource k is already owned or at least one other task owns a resource ℓ whose priority ceiling is greater than or equal to the current priority of T. If T is blocked, it is said to be blocked by every task owning such k or ℓ. The current priority of a task T is the maximum of T's base priority and the current priorities of all tasks that T blocks.

Step 3.

Execute the next instruction of the non-blocked running task (if any) with the highest current priority. If there was no such task or if a compute instruction was executed, increment the processor clock by one microsecond. If a lock or unlock instruction was executed, do not increment the clock.

The Priority Ceiling Protocol defined above has the following properties:

- Current priority is defined in terms of current priority and blocking, and blocking is defined in terms of current priority. While this may appear circular, there will always be a unique set of current priorities that satisfy the definitions.

- All tasks will eventually complete.
- There will never be a tie in step 3.

Input

Multiple test cases. Please process until EOF is reached. For each test case:

The first line of the input contains two integers t ($1 \le t \le 20$), which is the number of tasks, and r ($1 \le r \le 20$), which is the number of resources. This is followed by t lines, where the ith of these lines describes task i. The description of a task begins with three integers: the task's start time s ($1 \le s \le 10\,000$), its base priority b ($1 \le b \le t$), and an integer a ($1 \le a \le 100$). A task description is concluded by a sequence of a strings describing the instructions. Each string is a letter (`C` or `L` or `U`) followed by an integer. The string `C` n ($1 \le n \le 100$) indicates a sequence of n compute instructions. The strings `L` k and `U`k ($1 \le k \le r$) indicate instructions locking and unlocking resource k respectively.

No two tasks have the same base priority.

Output

For each test case:

For each task, display the time it completes execution, in the same order that the tasks are given in the input.

Example

Input:

```
3 1
50 2 5 C1 L1 C1 U1 C1
1 1 5 C1 L1 C100 U1 C1
70 3 1 C1
3 3
5 3 5 C1 L1 C1 U1 C1
3 2 9 C1 L2 C1 L3 C1 U3 C1 U2 C1
1 1 9 C1 L3 C3 L2 C1 U2 C1 U3 C1
```

Output:

```
106
107
71
8
15
```