Paxos
Problem ID: paxos

In this problem, you will be implementing a single-instance (or "Single Synod") version of the Paxos consensus algorithm. Instead of running the algorithm across multiple machines, you will instead simulate a distributed system composed of multiple machines connected with a basic (and deterministic) network.

Model

Our distributed system is composed of n computers, divided into two disjoint and non-empty sets P and A, each with n_P and n_A computers, respectively ($n_P + n_A = n$). The computers in P are the Proposers and the computers in A are the Acceptors. Computers in each set are numbered from 1 ($P = \{p_1, p_2, \ldots, p_{n_P}\}, A = \{a_1, a_2, \ldots, a_{n_A}\}$).

Each computer c has an attribute $c.\mathsf{failed} \in \{\texttt{true}, \texttt{false}\}$ which indicates whether it is operating correctly (`false`) or whether it has failed (`true`). This attribute is initially set to false. A machine that goes into a failed state may recover at a later time.

Besides the $c.\mathsf{failed}$ attribute, each computer can store an arbitrary amount of internal state (you will have to figure out, as part of this assignment, what internal state should be stored in each computer). This internal state is assumed to be stored in persistent storage, so it will be unaffected by failures.

Computers communicate by sending messages to one another. A message m has, at least, three attributes: $m.\mathsf{src}$ (the computer that is sending the message), $m.\mathsf{dst}$ (the computer for which the message is intended), and $m.\mathsf{type}$ (the type of message). The types of message are `PROPOSE`, `PREPARE`, `PROMISE`, `ACCEPT`, `ACCEPTED`, and `REJECTED`. You cannot define additional types of messages, but you can define any number of additional attributes for a given message type.

Messages are sent through a network N. All the computers are connected to this network. We model the network as a queue, and we define two operations on it:

- QUEUE-MESSAGE(N, m): Adds message m to the end of the queue.
- EXTRACT-MESSAGE(N): Finds the first message m in the queue (starting at the head of the queue) such that $m.\mathsf{src}.\mathsf{failed} = \texttt{false}$ and $m.\mathsf{dst}.\mathsf{failed} = \texttt{false}$. If such a message exists, m is removed from N and returned as the result of EXTRACT-MESSAGE. If no such message exists, a null value is returned.

Other than the behaviour specified in EXTRACT-MESSAGE, we assume that this network is perfectly reliable. Once a message is added to the network, it cannot be lost and its position in the queue cannot change.

Computers only carry out actions as the result of receiving a message. In other words, we assume that computers are, by default, "asleep" and they only wake up to do work in reaction to a message being delivered. Notice how, if a computer is in a failed state, it will not do any work because the network will prevent it from receiving any messages.

Thus, for each computer c, we define an operation DELIVER-MESSAGE(c, m). When this operation is invoked, c will wake up and will perform some actions (based on m and its internal state), and then go back to sleep. While awake, the computer can update its internal state and can call QUEUE-MESSAGE any number of times, but it *cannot* call EXTRACT-MESSAGE.

Finally, we model the passage of time in discrete increments of one time unit, which we will refer to as a "tick". Time always starts at zero. In a given tick, only the following can happen, in this order:

1. A set of machines can fail.
2. A set of (previously failed) machines can recover.
3. A single message can be delivered. In other words, during a given tick, only one computer can do any work. Two or more computers cannot work concurrently during a given tick.

The exact mechanism for simulating the passage of time and delivering messages is described in the following section.

Simulating the system

A simulation of this system is specified as a tuple $(n_P, n_A, t_{\text{max}}, E)$ where:

- n_P and n_A are the number of Proposers and Acceptors, as defined earlier.
- t_{max} is the maximum duration of the simulation (i.e., after this tick, the simulation is terminated).
- E is an ordered list of events $e_1, e_2, \ldots, e_{|E|}$.

An event e is a tuple (t, F, R, π_c, π_v) where:

- t indicates the tick at which this event happens ($0 \leqslant t \leqslant t_{\text{max}}$).
- F is a (possibly empty) set of computers that will fail at tick t.
- R is a (possibly empty) set of computers that will recover at tick t.
- π_c, π_v represent a request (by some external agent) for a new value to be proposed. $\pi_c \in P$ is the Proposer that must make the proposal, and π_v is the value that must be proposed. Both of these values can be null ($\varnothing$), indicating that no value is being proposed in tick t.

Events in E are ordered by increasing value of t. The list cannot contain two events with the same value of t. Note that this implies that, in a given tick, only one value can be proposed.

Pseudocode for running a simulation

The pseudocode for simulating the distributed system is given below. Notice how a request for a new value results in the generation of a `PROPOSE` message which is delivered directly to the specified Proposer (bypassing the network). When this happens, we do not check the network for any pending messages, since we are only allowed to deliver one message per tick.

function SIMULATE($n_P, n_A, t_{\text{max}}, E$)

/ Initialize Proposer and Acceptor sets, and create an empty network */*
$P \leftarrow \{p_1, p_2, \ldots, p_{n_P}\}$
$A \leftarrow \{a_1, a_2, \ldots, a_{n_A}\}$
$N \leftarrow \emptyset$

/ Step through all the ticks */*
for each $i \in [0 \ldots t_{\text{max}}]$ **do**

if $|N| = 0$ **and** $|E| = 0$ **then**
/ If there are no pending messages or events, we can end the simulation */*
return
end if

/ Process the event for this tick (if any) */*
$e \leftarrow$ Event in E with $t = i$
if $e \neq \varnothing$ **then**
Remove e from E
$t, F, R, \pi_c, \pi_v \leftarrow e$

for each $c \in F$ **do**
c.failed $\leftarrow$ `true`
end for

for each $c \in R$ **do**
c.failed $\leftarrow$ `false`
end for

if $\pi_c \neq \varnothing$ **and** $\pi_v \neq \varnothing$ **then**
$m \leftarrow$ new message

```
            m.type ← PROPOSE
            m.src ← ∅ /* PROPOSE messages originate from outside the system */
            m.dst ← π_c
            m.value ← π_v
            /* PROPOSE messages bypass the network and are delivered directly to the specified Proposer */
            DELIVER-MESSAGE(π_c, m)
          else
            m ← EXTRACT-MESSAGE(N)
            if m ≠ ∅ then
              DELIVER-MESSAGE(m.dst, m)
            end if
          end if
        end if
      end for
    end function
```

Assumptions

You can make the following assumptions in your implementation:

- There is a global proposal number that all Proposers have access to. This value is initialized to 1 and, when a Proposer wants to start a new proposal (i.e., a round or ballot), it uses the value of the global proposal number, and increments it by one. This guarantees that all Proposers always choose a proposal number that is larger than all previously used proposal numbers.
- Proposers are aware of how many Acceptors there are in the system.
- Computers cannot detect whether other computers have failed. Thus, a Proposer always sends a `PREPARE` message to all the Acceptors (since it cannot tell which ones are available and which ones are not). Then, regardless of the resulting quorum, the `ACCEPT` message is sent to all the Acceptors (not just to those which replied with a `PROMISE` message).
- A Proposer that receives a majority of `REJECT` messages from the Acceptors must attempt to pass another proposal from scratch.
- You do not have the implement the "learning" phase of the algorithm. However, this would be trivial to do, simply by having additional Learner computers and having the Proposers broadcast a `SUCCESS` message to all the learners when consensus is achieved.
- A simulation will never include an event such that π_c is in a failed state.

Input

The input to your program will be the specification of a single simulation.

The first line of the input contains the following values:

$$n_P \quad n_A \quad t_{\text{max}}$$

Where $0 < n_P, n_A \leq 10$ and $1 \leq t_{\text{max}} \leq 999$

Each subsequent line contains the specification of an event (there will always be at least one event) An event can be one of the following:

- t `PROPOSE` p_i v
- t `FAIL PROPOSER` p_i
- t `FAIL ACCEPTOR` a_i
- t `RECOVER PROPOSER` p_i

- t RECOVER ACCEPTOR a_i

Where $0 \leq t \leq t_{\text{max}}$, $1 \leq p_i \leq n_P$, $1 \leq a_i < n_A$, and $0 \leq v \leq 100$.
Take into account that there can be multiple lines specified for the same time t. In particular:

- PROPOSE must set π_c to p_i and π_v to v for the event at time t.
- FAIL PROPOSER adds p_i to the set F of computers that fail at time t.
- RECOVER PROPOSER adds p_i to the set R of computers that recover at time t.
- FAIL ACCEPTOR and RECOVER PROPOSER are defined similarly.

The end of the events is designated with a single line containing the following text:

```
0 END
```

Output

The output of the program is a bit more elaborate, and while we include a detailed specification below, it may be easier to simply start by looking at the sample output.

The output of your program will contain one line for every tick of the simulation. When a message is delivered in a given tick t, you must print the following:

$$t\texttt{: } src \texttt{ -> } dst \;\; message$$

Where src is the destination computer and dst is the destination computer (p_1 would be written P1 and a_1 would be written A1). Note that t must be printed with three characters, padded with zeroes to the left (e.g., $t = 7$ would be printed 007).

The possible messages are:

- PROPOSE v=v
- PREPARE n=*proposal*
- PROMISE n=*proposal* (Prior: None)
- PROMISE n=*proposal* (Prior: n=$proposal_{prior}$, v=v_{prior})
- ACCEPT n=*proposal* v=v
- ACCEPTED n=*proposal* v=v
- REJECTED n=*proposal*

When a computer c fails or recovers in a given tick t, you must print one of the following:

$$t\texttt{: ** } c \texttt{ FAILS **}$$

$$t\texttt{: ** } c \texttt{ RECOVERS **}$$

When no action is taken in a given tick t, you must simply print the following:

$$t\texttt{:}$$

When the simulation ends, you must print a blank line and then, for each proposer, one of the following:

- p_i did not reach consensus
- p_i has reached consensus (proposed v_{proposed}, accepted v_{accepted})

Sample Input 1

```
1 3 15
0 PROPOSE 1 42
0 END
```

Sample Output 1

```
000:    -> P1  PROPOSE v=42
001: P1 -> A1  PREPARE n=1
002: P1 -> A2  PREPARE n=1
003: P1 -> A3  PREPARE n=1
004: A1 -> P1  PROMISE n=1 (Prior: None)
005: A2 -> P1  PROMISE n=1 (Prior: None)
006: A3 -> P1  PROMISE n=1 (Prior: None)
007: P1 -> A1  ACCEPT n=1 v=42
008: P1 -> A2  ACCEPT n=1 v=42
009: P1 -> A3  ACCEPT n=1 v=42
010: A1 -> P1  ACCEPTED n=1 v=42
011: A2 -> P1  ACCEPTED n=1 v=42
012: A3 -> P1  ACCEPTED n=1 v=42

P1 has reached consensus (proposed 42, accepted 42)
```

Sample Input 2

```
2 3 50
0 PROPOSE 1 42
8 FAIL PROPOSER 1
11 PROPOSE 2 37
26 RECOVER PROPOSER 1
0 END
```

Sample Output 2

```
000:    -> P1  PROPOSE v=42
001: P1 -> A1  PREPARE n=1
002: P1 -> A2  PREPARE n=1
003: P1 -> A3  PREPARE n=1
004: A1 -> P1  PROMISE n=1 (Prior: None)
005: A2 -> P1  PROMISE n=1 (Prior: None)
006: A3 -> P1  PROMISE n=1 (Prior: None)
007: P1 -> A1  ACCEPT n=1 v=42
008: ** P1 FAILS **
009:
010:
011:    -> P2  PROPOSE v=37
012: P2 -> A1  PREPARE n=2
013: P2 -> A2  PREPARE n=2
014: P2 -> A3  PREPARE n=2
015: A1 -> P2  PROMISE n=2 (Prior: n=1, v=42)
016: A2 -> P2  PROMISE n=2 (Prior: None)
017: A3 -> P2  PROMISE n=2 (Prior: None)
018: P2 -> A1  ACCEPT n=2 v=42
019: P2 -> A2  ACCEPT n=2 v=42
020: P2 -> A3  ACCEPT n=2 v=42
021: A1 -> P2  ACCEPTED n=2 v=42
022: A2 -> P2  ACCEPTED n=2 v=42
023: A3 -> P2  ACCEPTED n=2 v=42
024:
025:
026: ** P1 RECOVERS **
026: P1 -> A2  ACCEPT n=1 v=42
027: P1 -> A3  ACCEPT n=1 v=42
028: A1 -> P1  ACCEPTED n=1 v=42
029: A2 -> P1  REJECTED n=1
030: A3 -> P1  REJECTED n=1
031: P1 -> A1  PREPARE n=3
032: P1 -> A2  PREPARE n=3
033: P1 -> A3  PREPARE n=3
034: A1 -> P1  PROMISE n=3 (Prior: n=2, v=42)
035: A2 -> P1  PROMISE n=3 (Prior: n=2, v=42)
036: A3 -> P1  PROMISE n=3 (Prior: n=2, v=42)
037: P1 -> A1  ACCEPT n=3 v=42
038: P1 -> A2  ACCEPT n=3 v=42
039: P1 -> A3  ACCEPT n=3 v=42
040: A1 -> P1  ACCEPTED n=3 v=42
041: A2 -> P1  ACCEPTED n=3 v=42
042: A3 -> P1  ACCEPTED n=3 v=42

P1 has reached consensus (proposed 42, accepted 42)
P2 has reached consensus (proposed 37, accepted 42)
```