NOGA

Because the limits on the numbers of grasshoppers and jumps are large, a solution which simply simulates every jump is too slow. The line needs to be maintained in a cleverer way so that we don't need to go through all grasshoppers in every jump.

One way is to split the line into K blocks so that each block contains approximately N/K grasshoppers. If we keep the ordering of the grasshoppers inside each block in a linked list, and their heights in a priority queue, then each block can be processed faster.

With such a data structure, one jump is processed like this:

- The two blocks in which a jump starts and ends are processed in the simple way. The complexity is O(N/K).
- Each block completely jumped is processed by first looking at the element at the top of the priority queue (the tallest grasshopper). The grasshopper at one end of the block is sent to the adjacent block, and at the other we receive a grasshopper from the previous block. The complexity of this step for all blocks is O(K log N/K).

The overall complexity of processing a jump is O(N/K + K log N/K), we pick a K for which this function is close to the minimum. In the official source code, this value is 100.

POVEZ

The problem can be modeled with a finite state automaton, where each cup has its own state, and there are exactly two transitions from each state, one for each letter. When the game starts, we don't know which state the automaton is in, so we say that all states are possible. We need to find an input sequence which will make state 1 the only possible state.

If we choose a pair of states (A, B) and find a sequence of letters which takes the automaton from state A to state 1, and from state B to state 1, then that sequence of letters must reduce the number of possible states. We can always do this because, if the entire problem is solvable (guaranteed in the task description), then there is an input sequence which takes any pair of states to state 1.

The idea of the solution is to maintain the set of possible states, then repeatedly choose some two states, find an input sequence which merges those two states, and calculate the new set of possible states (which is surely smaller than previously). After some number of iterations only state 1 will be left.

The official solution always takes the smallest two possible states, one of which is state 1.

An input sequence that merges two states can be found using breadth-first search, or we can find such a sequence for each pair of states using one breadth-first search if we reverse all edges.