Solution 1:

Understanding the problem.

Two walls (`n` cells each), start at left index 0. Each move takes 1s, then water rises by 1. Legal moves: up `x+1`, down `x-1`, jump to other wall `x+k`. Cells marked `'X'` are blocked. You escape if a generated move lands at `x ≥ n`.

Approach

Model it as reachability on a time-expanded graph and use **BFS**:

- State: `(wall, index, time)`.
- Pop `(w,x,t)`, generate neighbors at `t+1`.
- Enqueue only if safe, not visited, and won't be underwater after the move.
- If any neighbor has `x ≥ n`, print **YES**; else **NO** when the queue empties.

What went wrong

- Mixed a single array for **blocked** and **visited**, which blurred responsibilities.
- Applied the **flood rule** inconsistently (checked for some moves but not all; didn't skip popped states already underwater).
- Checked **escape** in the wrong place (on the current cell or after bounds), making it impossible to ever reach `x ≥ n`.
- Minor indexing/typos (e.g., pushing `x+1` on a left move), and ordering checks after indexing caused out-of-bounds risks.

Fixes

- **Separated concerns**: wall layout vs. visited.
- **Established invariants**:
  - On pop: skip if `x < t` (already underwater).
  - On push (every move): require `nx ≥ t+1`.
  - **Escape-first**: if `nx ≥ n`, succeed **before** any array access.

- Unified the neighbor-check pipeline: `escape? → bounds → not 'X' → not visited → not underwater`.

What would have prevented it

- Writing the three invariants up front (pop-skip, push-`nx ≥ t+1`, escape-first).
- A small per-move checklist applied identically to up/down/jump.
- Early manual trace on a tiny case to validate water timing and escape condition.
- Keeping **data roles separate** from the start.

Outcome

Clean BFS with simple data structures, correct under all cases: time handled as BFS depth, no underwater states enqueued, and escape detected on generation.