# 2) Challenge: Reverse `"RWANDA"` with a stack

Below is a clear algorithmic sequence with the corresponding code lines and explanation **for each step**.

**Goal:** input `"RWANDA"` → output reversed `"ADNAWR"`.

### Algorithm (step-by-step)

1. Initialize an empty stack.
2. For each character `c` in the input string (left → right), push `c` onto the stack.
   - This puts the first character at the bottom and last character at the top.
3. Initialize an empty result buffer (list of chars).
4. While the stack is not empty, pop the top character and append it to the result buffer.
   - Popping yields characters in reverse order.
5. Join the result buffer into a string and return it.

# 3) Challenge: Queue vs Stack for handling voter lines — which is correct?

**Short answer:** Use a **queue (FIFO)** for voter lines. FIFO serves people in arrival order, which is the standard fairness model for a line.

### Algorithmic sequence + code lines (explicit mapping)

**Algorithm:**

1. Initialize an empty queue.
2. As each voter arrives, enqueue them at the back of the queue.
3. When a voting booth (or server) becomes free, dequeue the front voter and serve them.
4. **Code (with comments mapping to steps):**

# 4) Reflection (theoretical only)

# a) Why does a stack work for undo but not (by itself) for redo actions?

- **Undo** follows LIFO: the last action performed is the first action we want to reverse. A **single stack** holding action history fits undo perfectly — `pop()` returns the last action to undo.
- **Redo** requires remembering undone actions in their original order so you can re-apply them. If you simply pop actions off the undo-stack and lose them, you cannot redo. Therefore, most editors use **two stacks**:
    1. **Undo stack** — push actions as they happen.
    2. When you undo: pop from undo stack and push the popped action onto the **redo stack**.
    3. To redo: pop from the redo stack and reapply (and then push it back onto the undo stack).
- Important subtlety: if the user performs a **new action** after some undos, the redo stack is typically cleared — because the timeline changed and old redo actions are no longer valid.
- So: **stack alone** supports undo; **redo** requires additional state (usually a second stack) and care about when to clear redo history.

# b) Why FIFO creates fairness in elections?

- **FIFO (First-In, First-Out)** ensures **procedural fairness**: service order is determined only by arrival time, not by identity, influence, or other attributes.
- This reduces incentives to manipulate or “cut in line” and provides a predictable, auditable order. For voting, FIFO helps ensure that voters are served in a neutral, non-discriminatory way.
- Caveats:
    - FIFO is fair **relative to arrival time** — it does not compensate for differences in ability to arrive early (so additional accessibility measures may be needed).
    - Practical systems sometimes include exceptions (priority lanes for disabled/elderly, scheduled appointments), but the default impartial policy is FIFO for fairness and transparency.