Array Rotation Trick

https://www.youtube.com/watch?v=BHr381Guz3Y

When you work through "rotate an array by k steps," you'll naturally touch on a handful of core algorithm-and-data-structure concepts. Here's a structured list of those concepts, with a brief explanation of how each shows up in this problem:

1. **Array Indexing and Zero-Based Positions**
   - Understanding that `nums[0]` is the first element and `nums[n-1]` is the last.
   - Realizing that rotating "right by 1" means the element at `nums[n-1]` wraps around to index 0, and everything else shifts one slot to the right.
2. **Modular Arithmetic (`%`) for Circular Shifts**
   - Noticing that rotating by any multiple of n returns you to the original array (so `k % n` is enough).
   - Computing `(i + k) % n` to find the new position of element i if you use an auxiliary array, or computing `(start + i) % n` if you "simulate" the rotation by index offset.
3. **In-Place Reversal Technique (Three-Step Reversal Pattern)**
   - Learning that you can accomplish a right-rotation by:
4. Reversing the entire array,
5. Reversing the first k elements,
6. Reversing the last n−k elements.

- Seeing how each reversal "mirrors" two elements at once, and why three reversals in that order end up with the correct rotated order.

4. **Two-Pointer (Head/Tail) Reversal Pattern**
   - Practicing the classic "swap `nums[l]` with `nums[r]` and then `l++`, `r--`, stop when `l >= r`."
   - Counting exactly $\lfloor m/2 \rfloor$ swaps to reverse any subarray of length m, and recognizing that each swap is O(1).
5. **Time Complexity (Big-O) Analysis**
   - Observing that each reversal pass visits ~m/2 pairs, so reversing m elements is $\Theta(m)$ time.
   - Summing $\Theta(n) + \Theta(k) + \Theta(n-k) = \Theta(2n) = \Theta(n)$ total for the three reversals, so the algorithm is O(n).
6. **Space Complexity and In-Place Constraints**
   - Distinguishing between an O(n) auxiliary-array solution (copy-and-write-back) versus the O(1) "three reversals" approach that uses only a few integer variables.
   - Understanding why physically rearranging n elements must cost at least $\Omega(n)$ in time, and that you can't genuinely move all elements in O(1) time.
7. **Edge Case Handling**
   - Realizing that if `k == 0` or `k % n == 0`, you should leave the array untouched (one of the first checks in code).
   - Verifying behavior when `n = 1`, or when `k > n` (hence `k = k % n`).
8. **Problem Decomposition and Pattern Recognition**

   - Breaking the problem into smaller sub-problems (e.g., “how do I rotate by 1?”, “how do I rotate by k?”, “can I do it without extra space?”).
   - Spotting the “reverse-all, reverse-first-k, reverse-remaining” pattern as a reusable trick for many array-rotation or cyclic-shift tasks.
9. **Alternative In-Place Approaches (like the Juggling Algorithm)**
   - Knowing that there’s also a GCD-based “cycle” method (sometimes called the juggling algorithm) that moves elements in cycles of length gcd(n, k), visiting each position exactly once.
   - Comparing why the reversal method tends to be simpler to code and just as efficient (O(n)/O(1)).
10. **Simulating a Rotation Without Physically Moving (Index Offsetting)**
    - Learning that if all you need is to “answer queries” as if the array were rotated (rather than actually rewrite it), you can store a single integer offset “start = (n − k) % n,” and answer “rotated[i]” by looking up `nums[(start + i) % n]`.
    - Seeing the difference between “logical” rotation (O(1) per lookup, no array change) versus “physical” rotation (O(n) to modify in place).
11. **Consistency Checks via Examples**
    - Tracing small examples by hand (e.g. `[1,2,3,4,5]`, k=2) to confirm that each step of “reverse all → reverse first 2 → reverse last 3” produces exactly the desired result.
    - Ensuring your implementation matches those hand-computed examples.
12. **Writing Clean, Modular Code**
    - Recognizing that “reverse a subarray” can be factored into its own helper function (e.g. `def reverse(nums, left, right): …`).
    - Writing each of the three reversal calls as `reverse(nums, 0, n-1)`, `reverse(nums, 0, k-1)`, `reverse(nums, k, n-1)`.
    - Documenting why `k = k % n` is necessary inside your main `rotate(...)` function.

Putting It All Together
When you solve “rotate array” in place, you end up practicing:

- Array indexing and boundary checks
- Modulo arithmetic for wrapping around
- The two-pointer reversal pattern
- Counting swaps to analyze Θ(n) time
- Distinguishing O(n) vs. O(1) extra space
- Edge cases (`n=1`, `k≥n`, `k=0`)
- Problem decomposition into reversible sub-arrays
- Potential alternative techniques (auxiliary array, juggling/GCD cycles, simulated offset lookup)

By understanding each of those pieces, you build a small but solid toolkit that applies not only to array rotations but to many other array-manipulation tasks in coding interviews and real-world coding.