Majority Element

https://www.youtube.com/watch?v=7pnhv842keE

Here's a rundown of the core concepts you'd pick up while working through the Majority-Element problem end-to-end:

1. **Array Traversal**
   – Scanning an array with a single `for`-loop.
   – Understanding how to access and process each element exactly once (linear pass).
2. **Hash-Map (Dictionary) Usage**
   – Choosing a map for "value → frequency" storage.
   – Methods like `get(key, default)` to safely initialize and update counts.
   – Insertion and lookup in O(1) average-time.
3. **Keeping a Running Maximum**
   – Tracking which element has the highest count so far (`res`) and its count (`maxCount`).
   – Using a simple `if count[x] > maxCount` or a ternary expression to update your candidate.
4. **Time-Complexity Analysis**
   – Recognizing that one pass plus constant-time map operations yields **O(n)** overall.
   – Noting occasional hash-table resizes but invoking **amortized O(1)** behavior.
5. **Space-Complexity Analysis**
   – Observing that storing counts for each *distinct* value uses **O(k)** extra space (worst-case **k=n**).
   – Weighing that against problem constraints and alternative approaches.
6. **Problem-Specific Guarantee**
   – Leveraging "there *is* always an element >n/2 times" to go beyond general "most frequent" logic.
7. **Boyer–Moore Voting Algorithm**
   – Learning an *in-place* linear-time, constant-space solution:
      1. Maintain a **candidate** (`res`) and a **vote counter** (`count`).
      2. On each new element, reset candidate if counter hits zero.
      3. "Vote" +1 for matches, –1 for mismatches.
         – Internalizing the "cancellation" intuition: non-majorities always get outweighed.
8. **Algorithm Correctness & Invariants**
   – Understanding why Boyer–Moore always ends on the true majority (the >n/2 element can never be fully cancelled).
   – Validating on small examples and edge cases.
9. **Edge-Case Handling**
   – Handling minimal inputs (`n=1`).
   – Confirming behavior when all elements are identical.
10. **Clean Code & Readability**
    – Writing self-documenting variable names (`count`, `res`).
    – Using concise constructs (ternary operators) without sacrificing clarity.

Mastering these gives you not only a fast solution for this one problem, but also transferable skills in data-structure choice, complexity analysis, and clever algorithm design.