Majority Element II

https://www.youtube.com/watch?v=Eua-UrQ_ANo

By working through this "Majority Element II" problem end-to-end, you're really touching on a handful of core algorithmic and coding concepts. Here's what you'd learn or reinforce:

1. **Pigeonhole Principle & Frequency Thresholds**
    - Recognizing that if you need elements appearing > $\lfloor n/3 \rfloor$ times, there can be at most two such elements (because 3×(more than n/3) would exceed n).
    - Translating "> $\lfloor n/3 \rfloor$" into code via integer division (`len(nums)//3`).
2. **Boyer–Moore Voting Paradigm (Generalized)**
    - Understanding the idea of "cancelling out" votes between different candidates so that only true heavy hitters survive.
    - Generalizing from the classic "> n/2" majority vote to maintaining two candidate slots (for the "> n/3" case).
3. **Two-Pass vs. One-Pass with Verification**
    - First pass to identify up to two possible candidates (with cancellation).
    - Second pass to **verify** true counts against the threshold.
    - Appreciating why you can't trust the provisional vote counts until you re-scan.
4. **Constant Extra Space Analysis**
    - Designing an algorithm that only keeps a fixed number of variables or small dict entries, regardless of input size.
    - Contrasting with a naïve hash-map solution that might store O(n) keys.
5. **`defaultdict(int)` Convenience**
    - Automating zero-initialization for unseen keys so you can write `count[x] += 1` without a guard.
    - Understanding how `defaultdict` differs from a normal `dict`.
6. **Pruning Technique**
    - Implementing the "prune when you have three distinct keys" step by subtracting one vote from each and dropping zeros.
    - Seeing how this directly embodies the cancellation logic in code.
7. **Pythonic Loop Constructs & Conditionals**
    - Writing clear `for` loops, `if/elif/else` ladders to cover all cases (match candidate, assign new slot, or cancel).
    - Proper use of `continue`, inner loops, and dict comprehensions or manual rebuilds.
8. **Time Complexity Trade-offs**
    - Ensuring O(n) behavior by keeping all operations per element O(1) (pruning over ≤ 3 keys is constant work).
    - Understanding the cost of `nums.count(cand)` in the verification pass (up to 2 full rescans), yet still O(n) overall.
9. **Edge-Case Handling**
    - Very small arrays (n = 1 or 2) where $\lfloor n/3 \rfloor = 0$, so every distinct element qualifies.
    - Arrays with no majority (> n/3), exactly one, or exactly two.
10. **Putting It All Together: Correctness Proof Sketch**

- Convincing yourself (or a reviewer) that no potential > n/3 element can be pruned away permanently—because it outvotes all cancellations over the full scan.