Topic Introduction

Today, we're diving into a fundamental pattern in coding interviews: **element frequency and counting problems**. These problems ask us to analyze how often elements appear in a list or array, and then use that information to answer specific questions.

What does "element frequency" mean?

Element frequency simply means: *How many times does each value occur?* Think of it as tallying votes in an election or counting how many people ordered each menu item. In programming, we often use a **hash map** (also called a dictionary) to keep track of these counts.

Why is this concept useful?

- **Interviewers love it:** Counting elements is at the heart of many real-world problems, from data analytics to fraud detection.
- **Unlocks efficiency:** When we count smartly, we avoid slow brute-force solutions and cut down our time complexity.
- **Opens up patterns:** Once you see a problem as "about counts," you can often spot a direct, elegant approach.

How does it work?

Imagine you're given this array: `[2, 3, 2, 4, 2, 3]`.
A frequency count would look like:

- 2: appears 3 times
- 3: appears 2 times
- 4: appears 1 time

In Python, you might use a Dictionary:

```
counts = {}
for num in arr:
    counts[num] = counts.get(num, 0) + 1
```

This basic pattern is the backbone of today's problems.

A simple example (not one of our main problems):

Suppose you're asked: _"Find the element that appears most often in `[5,1,2,2,5,5,3]`."_

- Count everything.
- See that 5 appears 3 times, which is more than any other number.
- Return 5.

Why These Problems?

Today’s trio is about **finding majority and "lonely" elements**:

- **Majority Element:** Find the element that appears more than half the time.
- **Majority Element II:** Find all elements that appear more than a third of the time.
- **Find All Lonely Numbers:** Find elements that appear exactly once and whose neighbors are missing.

All three rely on counting, but each has its own twist. Let’s get started!

Problem 1: Majority Element

LeetCode #169

Problem Statement (in my own words):

Given a list of numbers, return the element that appears **more than half** the time. You can assume such an element always exists.

Example:

Input: `[3, 2, 3]`
Output: `3`
(3 appears twice, which is more than half of the list length)

Let’s walk through it:

Suppose your input is `[2, 2, 1, 1, 1, 2, 2]`.

- Length = 7, so majority = appears > 3 times.
- 2 appears 4 times, 1 appears 3 times.
- Return 2.

Try this test case by hand:

Input: `[4, 4, 4, 2, 2, 4, 2]`
What’s the answer?

Brute-force approach:

- For each number, count how many times it appears.
- Return the one whose count > n/2.

But this is slow: **O(n^2)** time if you check every element against every other.

Optimal approach: Hash Map Counting

Let's use the **frequency counting** pattern:

- Initialize an empty dictionary.
- Loop through the list, counting each number.
- After counting, loop through the dictionary to find which number appears more than n/2 times.

Time Complexity: O(n)

Space Complexity: O(n)

Alternatively, there's a famous O(1) space algorithm called **Boyer-Moore Voting**, but let's focus on counting to build intuition.

Python Solution

```
def majorityElement(nums):
    # Step 1: Count frequencies
    counts = {}
    for num in nums:
        # If num not in counts, default to 0, then add 1
        counts[num] = counts.get(num, 0) + 1

    # Step 2: Find element with count > n/2
    n = len(nums)
    for num, count in counts.items():
        if count > n // 2:
            return num
    # Given the problem promise, we'll always return inside the loop
```

Explanation:

- `counts` is a dictionary where keys are numbers and values are their counts.
- For each number, we increase its count.
- Finally, we look for the number with count greater than half the list length.

Example trace:

Input: `[2, 2, 1, 1, 1, 2, 2]`

- After counting: `{2: 4, 1: 3}`
- 2 appears 4 times (more than 3.5). Return 2.

Try this test case on your own:

Input: `[5, 5, 5, 2, 2, 5, 5]`

Take a moment to solve this yourself before reading further!

Problem 2: Majority Element II

LeetCode #229

Problem Statement (in my own words):

Given a list of numbers, find **all** elements that appear **more than n/3 times** (where n is the length of the list). You can return them in any order.

Example:

Input: `[1, 2, 3, 1, 2, 1]`
n = 6, so threshold is more than 2.
1 appears 3 times. 2 and 3 appear 2 and 1 times.
Output: `[1]`

Another test case:

Input: `[1, 2, 3, 4, 5, 6]`
No element appears more than twice, so output: `[]`

How is this different from Problem 1?

- There can be **zero, one, or two** elements that qualify.
- The threshold is n/3, not n/2.

Brute-force:

- For each element, count how many times it appears (O(n^2)).
- But we can use the same hash map counting trick.

Optimal approach: Counting with a Hash Map

- Create an empty dictionary.
- Count frequencies for each number.
- Return all numbers with count > n/3.

Time Complexity: O(n)
Space Complexity: O(n)

Python-like Pseudocode

```
function majorityElementII(nums):
    counts = empty map
    for num in nums:
        if num not in counts:
            counts[num] = 0
        counts[num] += 1

    result = empty list
    threshold = floor(len(nums) / 3)
    for num, count in counts:
        if count > threshold:
            add num to result
    return result
```

Example trace:

Input: `[3, 3, 4, 2, 4, 4, 2, 4, 4]`
n = 9, threshold = 3
Counts: {3:2, 4:5, 2:2}
Only 4 occurs more than 3 times.

Output: `[4]`

Try this case:

Input: `[1,1,1,3,3,2,2,2]`
What should the output be?

Time and Space Complexity:

- **Time:** O(n)
- **Space:** O(n)

Note: There is a more advanced solution using extended Boyer-Moore Voting for O(1) space, but the counting approach is easier to understand and sufficient for most interviews.

Problem 3: Find All Lonely Numbers in the Array

LeetCode #2150

Problem Statement (in my own words):

Given a list of numbers, return all numbers that:

- Appear **exactly once** in the list,
- And **neither (num-1) nor (num+1) appear** in the list.

Return the lonely numbers in any order.

Example:

Input: `[10, 6, 5, 8]`

- Counts: {10:1, 6:1, 5:1, 8:1}
- 6: 5 is present (so not lonely)
- 5: 6 is present (not lonely)
- 8: 7 not present, 9 not present, and only appears once. Lonely!
- 10: 9 not present, 11 not present, and only appears once. Lonely!

Output: `[8, 10]`

Another test case for you:

Input: `[1, 3, 5, 7]`
What are the lonely numbers?

How is this different?

Now, in addition to **counting**, we must check if the neighbors exist.

Brute-force:

- For each number, check its count is 1.
- Then scan the array to see if num-1 or num+1 exist.
- O(n^2) time.

Optimal approach: Combine Counting and Set Lookup

Let’s use two structures:

- A hash map to count occurrences.
- A set to quickly check if neighbors exist.

Pseudocode

```
function findLonely(nums):
    counts = empty map
```

```
    for num in nums:
        counts[num] += 1

    num_set = set(nums)
    result = []
    for num in nums:
        if counts[num] == 1 and (num-1 not in num_set) and (num+1 not in num_set):
            add num to result
    return result
```

Example trace:

Input: `[4, 4, 3, 5, 7, 8]`

- Counts: {4:2, 3:1, 5:1, 7:1, 8:1}
- 3: count=1, neighbors 2 (not in set), 4 (in set) — not lonely
- 5: count=1, neighbors 4 (in set), 6 (not in set) — not lonely
- 7: count=1, neighbors 6 (not in set), 8 (in set) — not lonely
- 8: count=1, neighbors 7 (in set), 9 (not in set) — not lonely

No lonely numbers in this case.

Try this:

Input: `[1, 2, 4, 5, 7]`
Which numbers are lonely?

Time and Space Complexity:

- **Time:** O(n)
- **Space:** O(n)

Challenge:

Can you think of a way to solve this using only a single pass or less space? Try it out!

Summary and Next Steps

You've just tackled three classic **element frequency and counting** problems:

- **Majority Element:** Find the one that appears more than half the time.
- **Majority Element II:** Find all that appear more than a third of the time.
- **Find All Lonely Numbers:** Find those that appear once and have no neighbors.

Key patterns and insights:

- Use a hash map (dictionary) to quickly count element frequencies.
- Use a set for fast existence checks (especially for neighbors).
- Always check the problem's threshold: n/2, n/3, *exactly* once, etc.
- For problems with strict space constraints, explore more advanced algorithms (like Boyer-Moore).

Common mistakes:

- Off-by-one errors with thresholds (e.g., > n/3 vs >= n/3).
- Forgetting to check both neighbors in the lonely numbers problem.
- Not handling empty lists or lists with no qualifying elements.

Action List

- Solve all three problems on your own, including the one with code!
- For Problem 2, try to implement the O(1) space Boyer-Moore variation.
- For Problem 3, experiment with different approaches — can you do it in one pass?
- Find more LeetCode problems that use counting or frequency analysis.
- Review and compare your code with the solutions — pay attention to clean style and edge case handling.
- If you get stuck, that's totally normal. The more you practice these patterns, the faster you'll spot them next time!

Happy coding, and see you in the next PrepLetter!