Topic Introduction

Today, let's dive into a clever and interview-popular array manipulation technique: **Index Marking in Arrays of 1 to n**.

What Is It?

This is a pattern used in array problems where every element is guaranteed to be in the range 1 to n, with n being the length of the array. The trick? Use the array's own indices to "mark" which numbers have been seen, all **without extra space**.

How Does It Work?

Since every element is between 1 and n, each value naturally maps to a valid array index (after subtracting 1). By visiting or "marking" an index (often by negating the value at that index), we can track which numbers have been seen or are missing, all within the original array.

Why Is This Useful?

- **Saves space**: No need for extra hash sets or boolean arrays.
- **Efficient**: Most solutions run in O(n) time.
- **Interview favorite**: Shows deep understanding of in-place manipulation and array constraints.
- **Elegant**: Turns the array into its own bookkeeping device.

Quick Example (not one of our target problems):

Suppose you have `arr = [2, 3, 1, 5]`, and you want to know if number 3 is present. Since elements are 1 to n, just check `arr[3-1]` (that is, `arr[2]`). If you've marked visited numbers by negating them, you can instantly tell which numbers have been seen.

Why These Three Problems Together?

Today's problems are classic variations on this pattern. All involve arrays containing numbers from 1 to n and ask you to find missing or duplicate numbers. The trick in each is to use **index manipulation** to mark presence or absence, using the array itself.

- **Find All Duplicates in an Array**: Mark indices to spot duplicates.
- **Find All Numbers Disappeared in an Array**: Mark indices to find missing numbers.
- **First Missing Positive**: Place numbers at their "correct" index to find the smallest missing.

All three can be solved with smart in-place tricks, and mastering them will unlock a whole family of interview challenges.

Let's get started!

Problem 1: Find All Duplicates in an Array

Leetcode Link

Rephrased Problem Statement

Given an array of integers where each number is in the range 1 to n (n = array length), some numbers appear twice, and others appear once. Return an array of all numbers that appear exactly twice.

Example:

Input: `[4,3,2,7,8,2,3,1]`
Output: `[2,3]`

Here, the numbers 2 and 3 appear twice.

Step-by-step Thought Process

How do we spot duplicates *without* using extra space? Notice that since every value is 1 to n, each value x maps to index x-1. We can iterate through the array, and for each number, flip the sign of the number at its mapped index. If we see that a number at an index is already negative, that means the value's index has been visited before: a duplicate!

Try this with pen and paper to internalize the pattern!

Additional test case to try manually:

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

Brute-force Approach

Count the frequency of each number using a hash map or list.

- Time: O(n)
- Space: O(n) (needs extra storage)

Optimal Approach: In-Place Index Marking

Core Pattern:
For each element, use its value to index into the array (subtract 1), and flip the sign at that index. If you find the value already negative, the number is a duplicate.

Logic Breakdown

- Iterate through the array.
- For each number `num`, compute `index = abs(num) - 1`.
- If `nums[index]` is negative, you've seen this number before: add `abs(num)` to the answer.
- Otherwise, flip `nums[index]` to negative to mark it as seen.

Python Solution

```
def findDuplicates(nums):
    """
    Finds all numbers that appear exactly twice in the array.

    Args:
        nums: List[int] -- Array with numbers in 1..n.

    Returns:
        List[int] -- Numbers appearing twice.
    """
    res = []
    for num in nums:
        index = abs(num) - 1  # Map value to index
        if nums[index] < 0:
            # Already visited: duplicate found
            res.append(abs(num))
        else:
            # Mark as visited by negating
            nums[index] = -nums[index]
    return res
```

Time Complexity: O(n) (one pass through array)

Space Complexity: O(1) (ignoring output; modifies input in-place)

Code Walkthrough

- For each value, we use its absolute value to find its corresponding index.
- If that index is negative, we've seen this value before.
- Otherwise, we negate it to mark it as seen.

Trace Example

Input: `[4,3,2,7,8,2,3,1]`

Step-by-step:

- 4: index 3 → mark nums[3] negative.
- 3: index 2 → mark nums[2] negative.
- 2: index 1 → mark nums[1] negative.
- 7: index 6 → mark nums[6] negative.
- 8: index 7 → mark nums[7] negative.
- 2: index 1 → nums[1] is negative! Add 2 to result.
- 3: index 2 → nums[2] is negative! Add 3 to result.
- 1: index 0 → mark nums[0] negative.

Result: `[2,3]`

Try this one for practice:

Input: `[2,2,3,4,5,6,6,7]`
What should the output be?

Take a moment to solve this on your own before jumping into the solution!

Problem 2: Find All Numbers Disappeared in an Array

Leetcode Link

Problem Statement (Rephrased)

Given an array of size n with numbers in [1, n], some numbers may appear twice and others once. Find all the numbers from 1 to n that *do not* appear in the array.

Example:

Input: `[4,3,2,7,8,2,3,1]`
Output: `[5,6]`

How Is It Similar or Different?

Similar to the previous problem, but instead of finding duplicates, we need to find **missing** numbers.

Brute-force Approach

Create a set of numbers from 1 to n, and remove each number seen in the array.

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

Optimal Approach: In-Place Index Marking (Same core idea!)

Mark every number you see by negating the value at its mapped index. At the end, any index with a positive value means that index+1 was missing.

Pseudocode

```
for num in nums:
    index = abs(num) - 1
    if nums[index] > 0:
        nums[index] = -nums[index]

result = []
for i in range(len(nums)):
    if nums[i] > 0:
        result.append(i + 1)
return result
```

Step-by-step Example

Input: `[4,3,2,7,8,2,3,1]`

- After marking: `[ -4, -3, -2, -7, 8, 2, -3, -1 ]`
- Indices with positive values: 4 and 5 (0-based)
- Missing numbers: 5 and 6

Try this test case:

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

Trace of Example

Input: `[4,3,2,7,8,2,3,1]`

Pass 1: Mark indices for each number.

- 4: index 3 → mark negative.
- 3: index 2 → mark negative.
- 2: index 1 → mark negative.
- 7: index 6 → mark negative.
- 8: index 7 → mark negative.

- 2: index 1 → already negative.
- 3: index 2 → already negative.
- 1: index 0 → mark negative.

Resulting array: `[ -4, -3, -2, -7, 8, 2, -3, -1 ]`

Pass 2: Indices 4 and 5 are positive → missing numbers are 5 and 6.

Time Complexity: O(n)
Space Complexity: O(1) (modifies array in-place, output not counted)

Problem 3: First Missing Positive

Leetcode Link

Problem Statement (Rephrased)

Given an unsorted array of integers, find the smallest missing positive integer.

Example:

Input: `[3,4,-1,1]`
Output: `2`

What Makes This One Tricky?

- Not all elements are in 1 to n; some may be negative or out of range.
- We want the **smallest missing positive**.

Brute-force Approach

Add all positive numbers to a set, then check from 1 upwards for the first missing.

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

Optimal Approach: In-Place Placement

Core idea:
Place each number x in position x-1 if 1 <= x <= n. At the end, the first index where `nums[i] != i + 1` gives the answer.

Pseudocode

```
n = length of nums

for i in 0 to n-1:
    while 1 <= nums[i] <= n and nums[nums[i] - 1] != nums[i]:
        swap nums[i] and nums[nums[i] - 1]

for i in 0 to n-1:
    if nums[i] != i + 1:
        return i + 1

return n + 1
```

Example Input/Output:

Input: `[3, 4, -1, 1]`
Output: `2`

Step-by-step Walkthrough

- Initial: `[3,4,-1,1]`
- Place 3 at index 2: swap nums[0] and nums[2] → `[-1,4,3,1]`
- At index 0: -1 out of range, skip.
- At index 1: 4 at index 3, swap nums[1] and nums[3] → `[-1,1,3,4]`
- At index 1: 1 at index 0, swap nums[1] and nums[0] → `[1,-1,3,4]`
- All in place now.

Scan for first missing:

- i=0: nums[0]=1
- i=1: nums[1]=-1 (should be 2) → answer is 2

Try this additional test case:

Input: `[7,8,9,11,12]`
What should the answer be?

Time Complexity: O(n) (each number moved at most once)
Space Complexity: O(1)

Tip: Can you think of other clever ways to solve this? Try using a hash set and compare!

Summary and Next Steps

We just explored three classic array problems that all exploit the relationship between element values and their indices. By marking

or moving elements in place, you can solve these efficiently and elegantly, saving space and impressing interviewers.

Key Patterns:

- Use element value as an index.
- Mark visited by negation or swapping.
- At the end, scan for positive values or out-of-place elements.

Common Traps:

- Forgetting to use `abs()` when marking or checking.
- Not handling duplicates or out-of-range values correctly.
- Accidentally modifying input when not allowed (always check problem constraints).

Action List

- Try solving all three problems on your own, even the one with code provided.
- For Problems 2 and 3, attempt both in-place and set-based solutions — compare speed and elegance.
- Explore similar Leetcode problems, like "Set Mismatch" or "Missing Number", to reinforce this pattern.
- Dry-run your solutions with edge cases (arrays with all duplicates, all missing, negative numbers, etc).
- Discuss your code with a peer or on an online forum to spot improvements.
- Remember: the more you practice these patterns, the more naturally they'll come to you in interviews!

Happy coding!