Documentation for Find All Numbers Disappeared in an Array

Table of Contents

1. Problem Statement

We are given an array nums of length n, where each element in nums is within the range [1, n]. Some numbers in this range may be missing from nums.

Our task is to return an array of all missing numbers in the range [1, n] that do not appear in nums.

Example 1

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

Example 2

Input:nums = [1,1]
Output:[2]

2. Intuition

Since all numbers are within the range [1, n], we can efficiently determine missing numbers by leveraging index-based marking techniques instead of using extra space.

A number is missing from nums if its corresponding index has never been visited.

3. Key Observations

- Every number x in nums is in the range [1, n], so it can be mapped to index x - 1.
- If a number exists in the array, we mark its corresponding index as negative.
- After processing all numbers, indices that remain positive indicate missing numbers.

4. Approach

Step 1: Mark Visited Indices

- Iterate through nums.
- For each num, calculate its corresponding index: index = abs(num) - 1.
- Mark nums[index] as negative to indicate presence.

Step 2: Identify Missing Numbers

- After the first pass, any index i where nums[i] remains positive corresponds to a missing number i + 1.

Example Walkthrough

Input: nums = [4,3,2,7,8,2,3,1]

Processing Steps:

Step	nums State
Initial	[4,3,2,7,8,2,3,1]
Mark 4	[4,3,2,-7,8,2,3,1]
Mark 3	[4,3,-2,-7,8,2,3,1]
Mark 2	[4,-3,-2,-7,8,2,3,1]
Mark 7	[4,-3,-2,-7,8,2,-3,1]
Mark 8	[4,-3,-2,-7,8,2,-3,-1]
Mark 2 (again)	[4,-3,-2,-7,8,2,-3,-1]
Mark 3 (again)	[4,-3,-2,-7,8,2,-3,-1]
Mark 1	[-4,-3,-2,-7,8,2,-3,-1]

Step 2: Identify Missing Numbers

- The remaining positive values at indices 4 and 5 indicate that numbers 5 and 6 are missing.

5. Edge Cases

Case	Example	Expected Output
All numbers present	[1,2,3,4,5]	[]
All numbers missing except one	[1,1,1,1,1]	[2,3,4,5]
Single-element array	[1]	[]
Minimum input size	[1]	[]
All elements are duplicates	[2,2,2,2]	[1,3,4]

6. Complexity Analysis

Time Complexity

- O(n): We iterate through nums twice (once for marking, once for finding positives).

Space Complexity

- O(1): No extra space is used apart from the output list (modifies nums in-place).

7. Alternative Approaches

Approach	Time Complexity	Space Complexity	Notes
Using HashSet	O(n)	O(n)	Uses extra space
Sorting + Binary Search	O(n log n)	O(1)	Not optimal

8. Test Cases

```
def test_solution():
    sol = Solution()
    assert sol.findDisappearedNumbers([4,3,2,7,8,2,3,1]) == [5,6]
    assert sol.findDisappearedNumbers([1,1]) == [2]
    assert sol.findDisappearedNumbers([1,2,3,4,5]) == []
    assert sol.findDisappearedNumbers([2,2,2,2]) == [1,3,4]
    assert sol.findDisappearedNumbers([1]) == []
    print("All test cases passed!")

test_solution()
```

9. Final Thoughts

- This approach efficiently finds missing numbers with O(n) time and O(1) space.
- It works well with large inputs since it avoids extra memory usage.
- The downside is that it modifies the input array, so if the original order must be preserved, an alternative approach like using a HashSet should be used.