Array and Hashing

Two Sum

Pattern: Arrays & Hashing

Problem Statement

Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Sample Input & Output

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: nums[0] + nums[1] == 9, so we return [0, 1].
```

```
Input: nums = [3,3], target = 6
Output: [0,1]
Explanation: Two identical elements at different indices are valid.
```

```
Input: nums = [1,2,3], target = 7
Output: [] (or raises; but problem guarantees one solution)
Explanation: Edge - guaranteed solution per constraints, so this case won't occur.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        # STEP 1: Initialize hash map to store value → index
        #   - Why? To check in O(1) if complement (target - num) exists
        seen = {}

        # STEP 2: Iterate through array with index
        #   - Why index? We need to return positions, not values
        for i, num in enumerate(nums):
            complement = target - num

            # STEP 3: Check if complement already seen
            #   - If yes, we found our pair: current index + stored index
            if complement in seen:
                return [seen[complement], i]

            # STEP 4: Store current number and index for future lookup
            #   - Why here? To avoid using same element twice
            seen[num] = i

        # STEP 5: Return empty if no solution (per constraints, won't happen)
        #   - Included for safety / clarity
        return []

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    result1 = sol.twoSum([2, 7, 11, 15], 9)
    print(f"Test 1: {result1} → Expected: [0, 1]")
    assert result1 == [0, 1], "Test 1 Failed"

    #  Test 2: Edge case - duplicate values
    result2 = sol.twoSum([3, 3], 6)
    print(f"Test 2: {result2} → Expected: [0, 1]")
    assert result2 == [0, 1], "Test 2 Failed"

    #  Test 3: Tricky - negative numbers
    result3 = sol.twoSum([-1, -2, -3, -4, -5], -8)
    print(f"Test 3: {result3} → Expected: [2, 4]")
    assert result3 == [2, 4], "Test 3 Failed"

    print(" All inline tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's walk through `nums = [2, 7, 11, 15]`, `target = 9`.

Initial state:
`seen = {}` — empty hash map.
We'll iterate with index `i` and value `num`.

Step 1 — i=0, num=2:
- `complement = 9 - 2 = 7`
- Is 7 in `seen`? No → skip return
- Store `seen[2] = 0` → `seen = {2: 0}`
→ *Why store?* So if later we see 7, we know 2 was at index 0.

Step 2 — i=1, num=7:
- `complement = 9 - 7 = 2`
- Is 2 in `seen`? Yes → at index 0
- Return `[seen[2], 1]` → `[0, 1]`

→ *Why not store 7 first?*
Because we check *before* storing — this ensures we never use same index twice.

→ *Pattern insight*:
We trade space (hash map) for time — instead of nested loops O(n^2), we do one pass O(n). Hashing lets us "remember" what we've seen and instantly find complements.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We traverse the list once. Each hash map lookup and insertion is O(1) average case.

- **Space Complexity**: `O(n)`

  In worst case, we store n-1 elements in the hash map before finding the solution.
