Array and Hashing

1. 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)
```

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]
                # return [complement, num] -> returns the actual number

            # 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)
```

```python
    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.

---

2. Contains Duplicate

Pattern: Arrays & Hashing

Problem Statement

Given an integer array `nums`, return `true` if any value appears at least twice in the array, and return `false` if every element is distinct.

Sample Input & Output

```
Input: nums = [1,2,3,1]
Output: true
Explanation: The number 1 appears twice.
```

```
Input: nums = [1,2,3,4]
Output: false
Explanation: All elements are unique.
```

```
Input: nums = [1]
Output: false
Explanation: Single element cannot have duplicates.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:
        # STEP 1: Initialize hash set to track seen numbers
        #   - Set gives O(1) lookup; tracks uniqueness efficiently
        seen = set()

        # STEP 2: Iterate through each number
        #   - If num already in set → duplicate found → return True
        for num in nums:
            if num in seen:
                return True
            seen.add(num)  # Add to set for future checks

        # STEP 3: No duplicates found during iteration
        #   - Return False only after full traversal
        return False
        # -> below mentioned code also do the trick
        # return True if len(set(nums)) != len(num) else False
```

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

    #  Test 1: Normal case - duplicate exists
    assert sol.containsDuplicate([1,2,3,1]) == True, "Test 1 Failed"
    print(" Test 1 passed: [1,2,3,1] → True")

    #  Test 2: Edge case - single element
    assert sol.containsDuplicate([1]) == False, "Test 2 Failed"
    print(" Test 2 passed: [1] → False")

    #  Test 3: Tricky/negative - all unique, large input
    assert sol.containsDuplicate([i for i in range(1000)]) == False, \
        "Test 3 Failed"
    print(" Test 3 passed: 0..999 (all unique) → False")
```

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

Example Walkthrough

Let's walk through `nums = [1, 2, 3, 1]` step by step:

Initial State:
- `seen = set()` — empty
- `nums = [1, 2, 3, 1]`

Step 1: Process `num = 1`
- Check: Is `1 in seen`? → No
- Action: Add `1` to `seen` → `seen = {1}`
- Why? We record that we've seen `1` so future `1`s will trigger duplicate.

Step 2: Process `num = 2`
- Check: Is `2 in seen`? → No
- Action: Add `2` → `seen = {1, 2}`
- Why? Still building our uniqueness tracker.

Step 3: Process `num = 3`
- Check: Is `3 in seen`? → No
- Action: Add `3` → `seen = {1, 2, 3}`
- Why? No duplicate yet — continue.

Step 4: Process `num = 1` (again)
- Check: Is `1 in seen`? → Yes!
- Action: Immediately `return True`
- Why? We found a duplicate — no need to process further.
- Pattern Insight: Hashing lets us detect reoccurrence in O(1), breaking early saves time.

Final Output: `True`

This exemplifies the **Arrays & Hashing** pattern:
→ Use a hash set to track what we've seen.
→ Leverage O(1) average lookup to detect duplicates on the fly.
→ Early termination optimizes best-case performance.

Complexity Analysis

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

  We iterate through the array once. Each `in` check and `add` operation on the set is O(1) average case. Worst-case total = O(n).

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

  In the worst case (all elements unique), we store all `n` elements in the set. Space scales linearly with input size.

3. Majority Element

Pattern: Arrays & Hashing

Problem Statement

Given an array `nums` of size `n`, return the majority element.

The majority element is the element that appears more than `n / 2` times. You may assume that the majority element always exists in the array.

Sample Input & Output

```
Input: nums = [3,2,3]
Output: 3
Explanation: 3 appears 2 times > 3/2 = 1 → majority.
```

```
Input: nums = [2,2,1,1,1,2,2]
Output: 2
Explanation: 2 appears 4 times > 7/2 = 3 → majority.
```

```
Input: nums = [1]
Output: 1
Explanation: Single element is always majority by definition.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        # STEP 1: Initialize hash map to count occurrences
        #   - Tracks frequency of each element for quick lookup
        count = {}

        # STEP 2: Iterate and count each element
        #   - Invariant: after each step, count[x] = freq of x so far
        for num in nums:
            count[num] = count.get(num, 0) + 1

            # Early exit: if any count exceeds n//2, return immediately
            #   - Optimization: avoids full scan if found early
            if count[num] > len(nums) // 2:
                return num

        # STEP 3: Return fallback (problem guarantees existence)
        #   - Should never reach here per problem constraint
        return nums[0]

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

    #  Test 1: Normal case
    result1 = sol.majorityElement([3, 2, 3])
    print(f"Test 1 - Input: [3,2,3] → Output: {result1} (Expected: 3)")
    assert result1 == 3, "Test 1 Failed"

    #  Test 2: Edge case - single element
    result2 = sol.majorityElement([1])
    print(f"Test 2 - Input: [1] → Output: {result2} (Expected: 1)")
    assert result2 == 1, "Test 2 Failed"

    #  Test 3: Tricky case - long sequence, majority late
    result3 = sol.majorityElement([1, 1, 1, 2, 2, 2, 2])
    print(f"Test 3 - Input: [1,1,1,2,2,2,2] → Output: {result3} "
          f"(Expected: 2)")
    assert result3 == 2, "Test 3 Failed"
```

```
    print("  All 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 = [2,2,1,1,1,2,2]` step by step:

Initial State:
- `count = {}` (empty dictionary)
- `len(nums) = 7`, so majority threshold = `7//2 = 3`

Step 1: `num = 2`
→ `count.get(2, 0) + 1 = 0 + 1 = 1`
→ `count = {2: 1}`
→ `1 > 3?` → No → continue

Step 2: `num = 2`
→ `count[2] = 1 + 1 = 2`
→ `count = {2: 2}`
→ `2 > 3?` → No → continue

Step 3: `num = 1`
→ `count[1] = 0 + 1 = 1`
→ `count = {2:2, 1:1}`
→ `1 > 3?` → No → continue

Step 4: `num = 1`
→ `count[1] = 1 + 1 = 2`
→ `count = {2:2, 1:2}`
→ `2 > 3?` → No → continue

Step 5: `num = 1`
→ `count[1] = 2 + 1 = 3`
→ `count = {2:2, 1:3}`
→ `3 > 3?` → No → continue (note: must be >, not)

Step 6: `num = 2`
→ `count[2] = 2 + 1 = 3`

→ `count = {2:3, 1:3}`
→ `3 > 3?` → No → continue

Step 7: `num = 2`
→ `count[2] = 3 + 1 = 4`
→ `count = {2:4, 1:3}`
→ `4 > 3?` → YES → **return 2**

Why each step matters: - **Counting with dict**: Lets us track exact frequencies — core of hashing pattern. - **Early return**: Minor optimization — not required, but good practice. - **No fallback needed**: Problem guarantees majority exists, so we *will* hit return in loop.

Pattern Insight:
This is classic **Arrays & Hashing** — use a hash map to trade space for time, avoiding nested loops. Instead of $O(n^2)$, we get O(n) by storing what we've seen.

Complexity Analysis

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

  Single pass through array. Each dictionary `get` and assignment is O(1) average case.

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

  In worst case, all elements are distinct until majority is found — hash map stores up to n keys.
  (Note: Boyer-Moore algorithm can solve this in O(1) space — but that's a different pattern.) -> stick with hashing for cleaner implementation.

4. Valid Anagram

Pattern: Arrays & Hashing

Problem Statement

Given two strings `s` and `t`, return `true` if `t` is an anagram of `s`, and `false` otherwise.

An **anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Sample Input & Output

```
Input: s = "anagram", t = "nagaram"
Output: true
Explanation: All characters in 't' are a rearrangement of 's'.
```

```
Input: s = "rat", t = "car"
Output: false
Explanation: 't' has different characters than 's'.
```

```
Input: s = "", t = ""
Output: true
Explanation: Two empty strings are trivially anagrams.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        # STEP 1: Initialize structures
        #   - Use two hash maps (dicts) to count char frequencies in s and t.
        #   - More flexible than array - handles any Unicode char.
        if len(s) != len(t):
            return False

        count_s, count_t = {}, {}
```

```python
        # STEP 2: Main loop - populate both hash maps
        #   - Traverse by index since strings are same length.
        #   - Use .get() to safely handle missing keys.
        for i in range(len(s)):
            count_s[s[i]] = 1 + count_s.get(s[i], 0)
            count_t[t[i]] = 1 + count_t.get(t[i], 0)

        # STEP 3: Update state / bookkeeping
        #   - No incremental update needed - we build full maps first.
        #   - Final comparison validates anagram property.

        # STEP 4: Return result
        #   - Direct dict equality check - Python compares keys & values.
        return count_s == count_t

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

    #  Test 1: Normal case
    assert sol.isAnagram("anagram", "nagaram") == True, "Normal case failed"
    print(" Test 1 passed: 'anagram' vs 'nagaram'")

    #  Test 2: Edge case - empty strings
    assert sol.isAnagram("", "") == True, "Empty strings should be anagrams"
    print(" Test 2 passed: empty strings")

    #  Test 3: Tricky/negative - different letters
    assert sol.isAnagram("rat", "car") == False, "Mismatched letters"
    print(" Test 3 passed: 'rat' vs 'car'")
```

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

Example Walkthrough

Let's walk through `s = "anagram"`, `t = "nagaram"`:

Initial State: `count_s = {}`, `count_t = {}`, and `len(s) == len(t)` → proceed.

After i=0 (`s[0]='a'`, `t[0]='n'`):
→ `count_s['a'] = 1 + 0 → {'a': 1}`
→ `count_t['n'] = 1 + 0 → {'n': 1}`

After i=1 (`s[1]='n'`, `t[1]='a'`):
→ `count_s['n'] = 1 + 0 → {'a':1, 'n':1}`
→ `count_t['a'] = 1 + 0 → {'n':1, 'a':1}`

Continue to end:
→ `'a'` appears 3x in both → count_s['a']=3, count_t['a']=3
→ `'g'`,`'r'`,`'m'`,`'n'` each appear 1x → all match

Final Comparison:
→ `count_s == {'a':3, 'n':1, 'g':1, 'r':1, 'm':1}`
→ `count_t == {'n':1, 'a':3, 'g':1, 'r':1, 'm':1}`
→ Dictionaries are equal → return `True`

Why necessary?
→ Length check avoids unnecessary work and index errors.
→ `.get(key, 0)` prevents KeyError — critical for robust hashing.
→ Direct `dict1 == dict2` leverages Python's deep equality — clean and readable.
→ This exemplifies **frequency counting with hash maps** — a foundational Arrays & Hashing technique.

Complexity Analysis

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

  Single pass over both strings (n steps), then O(1) dict comparison (since alphabet size is bounded — even for Unicode, in practice it's limited per input).

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

  Where k = number of unique characters. In worst case, k = n (all chars distinct). For lowercase English, k 26 → effectively O(1). For general Unicode, O(k) where k scales with input diversity.

5. Group Anagrams

Pattern: Arrays & Hashing

Problem Statement

Given an array of strings `strs`, group the anagrams together. You can return the answer in **any order**.

An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Constraints: - `1 <= strs.length <= 10^4` - `0 <= strs[i].length <= 100` - `strs[i]` consists of lowercase English letters.

Sample Input & Output

```
Input: strs = ["eat","tea","tan","ate","nat","bat"]
Output: [["eat","tea","ate"],["tan","nat"],["bat"]]
Explanation: Words with same frequency signature
grouped together.

Input: strs = [""]
Output: [[""]]
Explanation: Single empty string forms its own group.

Input: strs = ["a"]
Output: [["a"]]
Explanation: Single letter trivially grouped.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
from collections import defaultdict

class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        # STEP 1: Create hashmap to group by freq signature
        anagram_map = defaultdict(list) # -> defaultdict(list, {})
```

```python
        for word in strs:
            # STEP 2: Count frequency of 26 letters
            freq = [0] * 26
            for ch in word:
                freq[ord(ch) - ord('a')] += 1

            # STEP 3: Use tuple of freq as key
            key = tuple(freq)
            anagram_map[key].append(word)

        # STEP 4: Return grouped anagrams
        return list(anagram_map.values())

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

    #  Test 1: Normal case
    strs1 = ["eat","tea","tan","ate","nat","bat"]
    print(sol.groupAnagrams(strs1))
    # Expected: groups like [["eat","tea","ate"],["tan","nat"],["bat"]]

    #  Test 2: Edge case - single empty string
    strs2 = [""]
    print(sol.groupAnagrams(strs2))
    # Expected: [[""]]

    #  Test 3: Tricky case - single letter
    strs3 = ["a"]
    print(sol.groupAnagrams(strs3))
    # Expected: [["a"]]
```

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

Detailed Example Walkthrough

Pick input: `["eat", "tea", "tan", "ate", "nat", "bat"]`.

1. **Initialize**: `anagram_map = {}` (empty defaultdict). Tracks groups keyed by letter-frequency tuple.
2. **Word = "eat"**
   - freq = `[1,0,0,0,1,0,...,1,...]`
   - key = `(1,0,0,0,1,0,...,1,...)`
   - Add → `anagram_map[key] = ["eat"]`.
3. **Word = "tea"**
   - Same frequency tuple as `"eat"`.
   - Append → `["eat","tea"]`.
4. **Word = "tan"**
   - freq different → new key.
   - Add → `["tan"]`.
5. **Word = "ate"**
   - Same key as `"eat"`.
   - Append → `["eat","tea","ate"]`.
6. **Word = "nat"**
   - Same key as `"tan"`.
   - Append → `["tan","nat"]`.
7. **Word = "bat"**
   - New frequency key → `["bat"]`.
8. **Final Output**: `[["eat","tea","ate"],["tan","nat"],["bat"]]`.

Pattern Insight: We reduce each string into a **hashable signature** (frequency tuple). Hashing ensures `O(1)` group lookup, which is the core of **Arrays & Hashing** mastery.

Complexity Analysis

- **Time Complexity**:
  - For each word (n words), count letters (k length).
  - Total: `O(n * k)`.
- **Space Complexity**:
  - Hashmap stores up to `n` groups.
  - Each key is a 26-length tuple.
  - Total: `O(n * k)` for output + `O(26)` per key.

---

6. Longest Consecutive Sequence

Pattern: Arrays & Hashing

Problem Statement

> Given an unsorted array of integers `nums`, return the length of the longest consecutive elements sequence.
>
> You must write an algorithm that runs in `O(n)` time.

Sample Input & Output

```
Input: nums = [100, 4, 200, 1, 3, 2]
Output: 4
Explanation: The longest consecutive sequence is [1, 2, 3, 4] → length 4.
```

```
Input: nums = [0, 3, 7, 2, 5, 8, 4, 6, 0, 1]
Output: 9
Explanation: Longest sequence is [0,1,2,3,4,5,6,7,8] → length 9.
```

```
Input: nums = []
Output: 0
Explanation: Empty input → no sequence → length 0.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        # STEP 1: Initialize structures
        #   Use a set for O(1) lookups. Avoid duplicates.
        num_set = set(nums)
        max_len = 0

        # STEP 2: Main loop / recursion
        #   Only start counting if 'n' is the start of a sequence
        #   (i.e., n-1 not in set). Avoids recounting same seq.
        for n in num_set:
            if n - 1 not in num_set:
                curr_num = n
                curr_len = 1

                # STEP 3: Update state / bookkeeping
                #   Extend sequence as long as next number exists
                while curr_num + 1 in num_set:
                    curr_num += 1
                    curr_len += 1

                # Update global max
                max_len = max(max_len, curr_len)

        # STEP 4: Return result
        #   Handles edge case: empty input → max_len remains 0
        return max_len

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
```

```python
sol = Solution()

#   Test 1: Normal case
result1 = sol.longestConsecutive([100, 4, 200, 1, 3, 2])
print(f"Test 1 - Expected: 4, Got: {result1}")
assert result1 == 4, "Test 1 Failed"

#   Test 2: Edge case
result2 = sol.longestConsecutive([])
print(f"Test 2 - Expected: 0, Got: {result2}")
assert result2 == 0, "Test 2 Failed"

#   Test 3: Tricky/negative
result3 = sol.longestConsecutive([0, -1, 1, -2, 2, -3, 3])
print(f"Test 3 - Expected: 7, Got: {result3}")
assert result3 == 7, "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 = [100, 4, 200, 1, 3, 2]`.

Initial state:
- `num_set = {100, 4, 200, 1, 3, 2}`
- `max_len = 0`

Step-by-step:

1. **n = 100**
   → Check: Is `99` in set? No → start sequence.
   → `curr_num = 100`, `curr_len = 1`
   → Check `101`? Not in set → sequence ends.
   → `max_len = max(0, 1) → 1`
2. **n = 4**
   → Is `3` in set? Yes → skip (not start of sequence).

3. **n = 200**
   → `199`? No → start sequence.
   → `curr_len = 1`, `201`? No → update `max_len = max(1,1) → 1`
4. **n = 1**
   → `0`? No → start sequence.
   → `curr_num = 1`, `curr_len = 1`
   → `2`? Yes → `curr_num=2`, `curr_len=2`
   → `3`? Yes → `curr_num=3`, `curr_len=3`
   → `4`? Yes → `curr_num=4`, `curr_len=4`
   → `5`? No → stop.
   → `max_len = max(1,4) → 4`
5. **n = 3** → `2` exists → skip.

6. **n = 2** → `1` exists → skip.

Final `max_len = 4`.

Why skip if `n-1` exists?
→ Avoids recounting the same sequence multiple times.
→ Ensures each sequence is counted only from its minimum element.
→ Critical for achieving O(n) — each element visited at most twice.

Pattern Insight:
Hashing lets us check existence in O(1).
By anchoring sequences at their start, we avoid nested loops → linear time.

Complexity Analysis

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

  Each element is visited at most twice: once in outer loop, once in while loop. Set lookups are O(1). Total operations bounded by 2n → O(n).

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

  We store all elements in a set. No recursion or auxiliary structures beyond that. Space scales linearly with input size.

---

7. Subarray Sum Equals K

Pattern: Arrays & Hashing

Problem Statement

Given an array of integers `nums` and an integer `k`, return the total number of subarrays whose sum equals `k`.

A subarray is a contiguous non-empty sequence of elements within an array.

Sample Input & Output

```
Input: nums = [1, 1, 1], k = 2
Output: 2
Explanation: Subarrays [1,1] (indices 0-1) and [1,1] (indices 1-2)
both sum to 2.
```

```
Input: nums = [1, 2, 3], k = 3
Output: 2
Explanation: Subarrays [1,2] (indices 0-1) and [3] (index 2) both sum to 3.
```

```
Input: nums = [1], k = 0
Output: 0
Explanation: Only one element, no way to get sum 0.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        # STEP 1: Initialize structures
        #   - prefix_sum: running total from start to current index
        #   - count: total valid subarrays found
        #   - prefix_freq: tracks how many times each prefix sum occurred
        prefix_sum = 0
        count = 0
        prefix_freq = {0: 1}  # crucial: empty prefix has sum 0

        # STEP 2: Main loop / recursion
        #   - For each num, update running sum
        #   - Check if (current_sum - k) was seen before → means
        #     there's a subarray ending here with sum = k
        for num in nums:
            prefix_sum += num

            # STEP 3: Update state / bookkeeping
            #   - If (prefix_sum - k) exists, add its frequency to count
            target = prefix_sum - k
            if target in prefix_freq:
                count += prefix_freq[target]

            # Record current prefix_sum occurrence
            prefix_freq[prefix_sum] = (
                prefix_freq.get(prefix_sum, 0) + 1
            )

        # STEP 4: Return result
        #   - All edge cases handled by initializing {0:1} and using
        #     .get() with default 0
        return count

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

    #  Test 1: Normal case
    result1 = sol.subarraySum([1, 1, 1], 2)
    print(f"Test 1 - [1,1,1], k=2 → {result1}")  # Expected: 2
```

```
#   Test 2: Edge case (no matches)
result2 = sol.subarraySum([1], 0)
print(f"Test 2 - [1], k=0 → {result2}")      # Expected: 0

#   Test 3: Tricky/negative (negative numbers)
result3 = sol.subarraySum([-1, -1, 1], 0)
print(f"Test 3 - [-1,-1,1], k=0 → {result3}") # Expected: 1
```

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

Example Walkthrough

Let's trace `nums = [1, 1, 1], k = 2` — step by slow step.

Initial State (Before Loop):

- `prefix_sum = 0`
- `count = 0`
- `prefix_freq = {0: 1}` ← This lets us count subarrays starting at index 0!

Step 1: Process first element → num = 1

- Execute: `prefix_sum += 1` → now `prefix_sum = 1`
- Calculate: `target = 1 - 2 = -1`
- Check: Is `-1` in `prefix_freq`? → No → `count` unchanged (still 0)
- Update map: `prefix_freq[1] = 0 + 1 = 1`
- Map now: `{0:1, 1:1}`

State after Step 1:

- `prefix_sum = 1`
- `count = 0`
- `prefix_freq = {0:1, 1:1}`

Step 2: Process second element → num = 1

- Execute: `prefix_sum += 1` → now `prefix_sum = 2`
- Calculate: `target = 2 - 2 = 0`
- Check: Is `0` in `prefix_freq`? → Yes! Value is `1`
- So: `count += 1` → `count = 1`
- Update map: `prefix_freq[2] = 0 + 1 = 1`
- Map now: `{0:1, 1:1, 2:1}`

State after Step 2: - `prefix_sum = 2` - `count = 1` - `prefix_freq = {0:1, 1:1, 2:1}`

Step 3: Process third element → num = 1

- Execute: `prefix_sum += 1` → now `prefix_sum = 3`
- Calculate: `target = 3 - 2 = 1`
- Check: Is `1` in `prefix_freq`? → Yes! Value is `1`
- So: `count += 1` → `count = 2`
- Update map: `prefix_freq[3] = 0 + 1 = 1`
- Map now: `{0:1, 1:1, 2:1, 3:1}`

Final State:

- `prefix_sum = 3`
- `count = 2`
- `prefix_freq = {0:1, 1:1, 2:1, 3:1}`

Function returns `2`.

Why This Works — Pattern Insight:

Arrays & Hashing Pattern: We trade space for time. Instead of checking every subarray $O(n^2)$, we store prefix sums and "ask the past" — *"Have I seen a sum that, if subtracted from now, gives me k?"*

The key equation:
`sum[i:j] = prefix[j] - prefix[i] = k`
→ `prefix[i] = prefix[j] - k`

`{0:1}` handles subarrays starting at index 0 — because `prefix[j] - 0 = k` → subarray from start to j.

Complexity Analysis

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

  One pass through the array. Each dictionary lookup/insertion is average O(1).

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

  In worst case (all prefix sums unique), we store n+1 keys in the hashmap.

---

8. Contiguous Array

Pattern: Arrays & Hashing

Problem Statement

> Given a binary array **`nums`**, return the maximum length of a contiguous subarray with an equal number of `0` and `1`.

Sample Input & Output

```
Input: nums = [0,1]
Output: 2
Explanation: [0, 1] is the longest contiguous subarray with equal 0s and 1s.
```

```
Input: nums = [0,1,0]
Output: 2
Explanation: [0, 1] or [1, 0] - both length 2.
No longer valid subarray exists.
```

```
Input: nums = [0,0,0,0,0]
Output: 0
Explanation: No 1s → no balanced subarray possible.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def findMaxLength(self, nums: List[int]) -> int:
        # STEP 1: Initialize structures
        #   - Use hashmap to store first occurrence of prefix balance.
        #   - Key: running balance (count_1 - count_0), Value: index.
        #   - Start with balance 0 at index -1 to handle full-array case.
        balance_map = {0: -1}
        max_len = 0
        balance = 0  # tracks (count of 1s) - (count of 0s)

        # STEP 2: Main loop / recursion
        #   - Traverse each element. Treat 0 as -1, 1 as +1.
        #   - If same balance seen before → subarray between indices
        #     has net zero → equal 0s and 1s.
        for i in range(len(nums)):
            # Update balance: +1 for 1, -1 for 0
            balance += 1 if nums[i] == 1 else -1

            # STEP 3: Update state / bookkeeping
            #   - If balance seen → calculate length from first index.
            #   - Else, record first occurrence of this balance.
            if balance in balance_map:
                length = i - balance_map[balance]
                if length > max_len:
                    max_len = length
            else:
                balance_map[balance] = i

        # STEP 4: Return result
        #   - Handles edge cases: all 0s, all 1s, empty → returns 0.
```

```python
        return max_len

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

    #   Test 1: Normal case
    result1 = sol.findMaxLength([0, 1, 0, 1])
    print(f"Test 1 - Input: [0,1,0,1] → Output: {result1}")
    assert result1 == 4, f"Expected 4, got {result1}"

    #   Test 2: Edge case - all zeros
    result2 = sol.findMaxLength([0, 0, 0, 0])
    print(f"Test 2 - Input: [0,0,0,0] → Output: {result2}")
    assert result2 == 0, f"Expected 0, got {result2}"

    #   Test 3: Tricky/negative - single element
    result3 = sol.findMaxLength([1])
    print(f"Test 3 - Input: [1] → Output: {result3}")
    assert result3 == 0, f"Expected 0, got {result3}"

    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 = [0, 1, 0, 1]` step by step.

Initial State
- `balance_map = {0: -1}` ← We start assuming balance 0 occurred "before" index 0.
- `max_len = 0`
- `balance = 0`

Step 1: i = 0, nums[0] = 0
- `balance += -1` → `balance = -1`
- Is `-1` in `balance_map`? No → record it: `balance_map[-1] = 0`

→ *State*: `balance_map = {0: -1, -1: 0}`, `max_len = 0`, `balance = -1`

Step 2: i = 1, nums[1] = 1
- `balance += +1` → `balance = 0`
- Is `0` in `balance_map`? Yes → at index `-1`
- Length = `1 - (-1) = 2` → update `max_len = 2`

→ *State*: `balance_map unchanged`, `max_len = 2`, `balance = 0`

Step 3: i = 2, nums[2] = 0
- `balance += -1` → `balance = -1`
- Is `-1` in `balance_map`? Yes → at index `0`
- Length = `2 - 0 = 2` → not greater than `max_len` (still 2)

→ *State*: unchanged `max_len`, `balance = -1`

Step 4: i = 3, nums[3] = 1
- `balance += +1` → `balance = 0`
- Is `0` in `balance_map`? Yes → at index `-1`
- Length = `3 - (-1) = 4` → update `max_len = 4`

→ *State*: `max_len = 4`, `balance = 0`

Final Output: `4`

The entire array `[0,1,0,1]` is balanced → 2 zeros, 2 ones.

Complexity Analysis

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

  Single pass through the array. Hashmap lookups and inserts are O(1) average.

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

  In worst case, we store every unique balance encountered → up to `n+1` entries (including initial 0).
