SOLVED PROBLEMS

🧩 LeetCode 1 — Two Sum

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

Example 1:

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

```
def twoSum(nums, target):
    seen = {}

    for i, num in enumerate(nums):
        complement = target - num

        if complement in seen:
            return [seen[complement], i]

        seen[num] = i
```

🧠 Thought Process

1. For each number `x`, compute its **complement**:

   ```
   complement = target - x
   ```

2. Use a **dictionary** for O(1) lookup.
3. Dictionary stores:

   ```
   number → index
   ```

   because we must return **indices**.
4. For each element:
   - **check first** if complement exists

     (prevents using same element twice)
   - if found → return indices
   - otherwise → store current number with its index

This is exactly how interviewers expect you to think.

🗒️ LINE-BY-LINE EXPLANATION (Notion Ready)

```
def twoSum(nums, target):
```

- Function receives a list of numbers and a target sum.

```
seen = {}
```

- Dictionary to store numbers already visited.
- Format: `{number : index}`

```
for i, num in enumerate(nums):
```

- Loop through the list with both **index** and **value**.

```
complement = target - num
```

- Calculate the number needed to reach the target.

```
if complement in seen:
```

- Check if the required number has already been seen.
- Dictionary lookup happens in **O(1)** time.

```
return [seen[complement], i]
```

- If found, return the index of the complement and current index.

```
seen[num] = i
```

- Store the current number and its index for future checks.

⏱️ Time & Space Complexity

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

Much better than brute force `O(n²)`.

🧩 LeetCode 242 — Valid Anagram

📌 Problem Statement

Given two strings `s` and `t`, return `True` if `t` is an **anagram** of `s`, otherwise return `False`.

👉 An anagram means:

- Same characters
- Same frequency
- Order does **not** matter

🧠 Key Observations

1. If lengths of `s` and `t` are different → **cannot** be anagrams.
2. Order does not matter → sorting is not required.
3. What matters is **frequency of each character**.
4. Dictionaries (hashmaps) are ideal for frequency counting.

💡 Approach (Hash Map / Frequency Count)

1. Check if `len(s) != len(t)` → return `False`
2. Create two dictionaries:
    - One for frequency of characters in `s`
    - One for frequency of characters in `t`
3. Compare both dictionaries:
    - If equal → anagram
    - Else → not an anagram

✅ Python Code

```
def isAnagram(s, t):
    if len(s) != len(t):
        return False

    freq_s = {}
    freq_t = {}

    for ch in s:
        freq_s[ch] = freq_s.get(ch, 0) + 1

    for ch in t:
        freq_t[ch] = freq_t.get(ch, 0) + 1

    return freq_s == freq_t
```

🧾 Line-by-Line Explanation

```
def isAnagram(s, t):
```

Defines the function with two input strings.

```
if len(s) != len(t):
    return False
```

If lengths differ, they cannot be anagrams.

```
freq_s = {}
freq_t = {}
```

Two dictionaries to store character frequencies.

```
for ch in s:
    freq_s[ch] = freq_s.get(ch, 0) + 1
```

Counts how many times each character appears in `s`.

```
for ch in t:
    freq_t[ch] = freq_t.get(ch, 0) + 1
```

Counts how many times each character appears in `t`.

```
return freq_s == freq_t
```

Dictionaries are equal only if:

- Same keys
- Same values

  Order does not matter → perfect for anagrams.

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Frequency map using dictionary
- Early exit using length check
- Dictionary equality comparison
- Strings → HashMap transformation

🧩 LeetCode 217 – Contains Duplicate

📌 Problem Statement

Given an integer array `nums`, return `True` if **any value appears at least twice** in the array, otherwise return `False`.

🧠 Key Observations

1. We only need to know **whether a duplicate exists**, not how many times it occurs.
2. The moment a number repeats, we can **stop immediately** and return `True`.
3. Using nested loops would be inefficient.
4. A dictionary (hashmap) allows **O(1)** lookup to check if a number has been seen before.

💡 Approach (Hash Map / Seen Pattern)

1. Create an empty dictionary `seen`
2. Traverse the array:
   - If the current number is already in `seen` → return `True`
   - Otherwise, store the number in `seen`
3. If the loop finishes without finding duplicates → return `False`

This avoids unnecessary comparisons and exits early when possible.

✅ Python Code

```
def containsDuplicate(nums):
    seen = {}

    for num in nums:
        if num in seen:
            return True
        seen[num] = 1
```

```
    return False
```

🧾 Line-by-Line Explanation

```
def containsDuplicate(nums):
```

`Defines the function that takes a list of integers.`

```
    seen = {}
```

`Creates a dictionary to track numbers already encountered.`

```
    for num in nums:
```

`Iterates through each number in the list.`

```
        if num in seen:
            return True
```

`If the number has been seen before, a duplicate exists → return` `True` `immediately.`

```
        seen[num] = 1
```

`Marks the number as seen.`

```
    return False
```

`If no duplicates are found after the loop, return` `False`.

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- “Have I seen this before?” → **Hash Map / Set**
- Early exit optimization
- Replacing `O(n²)` brute force with `O(n)` hashing

🧩 LeetCode 121 — Best Time to Buy and Sell Stock

📌 Problem Statement

You are given an array `prices` where `prices[i]` represents the stock price on day `i`.

You must:

- Buy **once**
- Sell **once**
- Sell must happen **after** buy

Return the **maximum profit** possible.

If no profit can be made, return `0`.

🧠 Key Observations

1. The selling day must come **after** the buying day.
2. We only care about the **lowest price before today**.
3. Brute force (checking all pairs) is inefficient.
4. The problem can be solved in **one pass**.

💡 Approach (One Pass / Greedy)

1. Track the **minimum price seen so far**
2. For each day:
    - Calculate profit = `current_price - min_price_so_far`
    - Update maximum profit if this profit is higher
3. Update minimum price whenever a lower price is found
4. Return the maximum profit

✅ Python Code

```
def maxProfit(prices):
    min_price = prices[0]
    max_profit = 0

    for price in prices:
        if price < min_price:
            min_price = price
        else:
            profit = price - min_price
            max_profit = max(max_profit, profit)

    return max_profit
```

🧾 Line-by-Line Explanation

```
def maxProfit(prices):
```

Defines the function that takes a list of stock prices.

```
min_price = prices[0]
max_profit = 0
```

Initializes:

- `min_price` as the first day's price
- `max_profit` as `0` (default if no profit possible)

```
for price in prices:
```

Iterates through each day's price.

```
if price < min_price:
    min_price = price
```

Updates the minimum price if a lower price is found.

```
else:
    profit = price - min_price
    max_profit = max(max_profit, profit)
```

Calculates profit for the current day and updates maximum profit if higher.

```
return max_profit
```

Returns the maximum profit found.

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
- **Space Complexity:** `O(1)`

🧠 DSA Pattern Learned

- Tracking minimum value so far
- One-pass greedy approach
- Replacing nested loops with smart state tracking

🧩 LeetCode 53 — Maximum Subarray

📌 Problem Statement

Given an integer array `nums`, find the **contiguous subarray** (containing at least one number) which has the **largest sum**, and return that sum.

🧠 Key Observations

1. The subarray must be **contiguous**.
2. A negative running sum will **reduce** future sums.
3. At every index, we decide:
    - extend the previous subarray
    - OR start a new subarray from the current element
4. This problem can be solved efficiently using **Kadane's Algorithm**.

💡 Approach (Kadane's Algorithm)

1. Initialize:
    - `current_sum` as the first element
    - `max_sum` as the first element
2. Traverse the array from the second element:
    - Update `current_sum` as the maximum of:
        - current element
        - current element + previous `current_sum`
    - Update `max_sum` if `current_sum` is larger
3. Return `max_sum`

✅ Python Code

```
def maxSubArray(nums):
    current_sum = nums[0]
    max_sum = nums[0]

    for i in range(1, len(nums)):
        current_sum = max(nums[i], current_sum + nums[i])
        max_sum = max(max_sum, current_sum)

    return max_sum
```

🗒️ Line-by-Line Explanation

```
def maxSubArray(nums):
```

Defines the function that takes a list of integers.

```
current_sum = nums[0]
max_sum = nums[0]
```

Initializes both the current running sum and the maximum sum with the first element.

```
for i in range(1, len(nums)):
```

Iterates through the array starting from the second element.

```
current_sum = max(nums[i], current_sum + nums[i])
```

Chooses whether to:

- start a new subarray at the current element
- OR extend the existing subarray

```
max_sum = max(max_sum, current_sum)
```

Updates the maximum sum found so far.

```
return max_sum
```

Returns the largest subarray sum.

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
- **Space Complexity:** `O(1)`

🧠 DSA Pattern Learned

- Kadane’s Algorithm
- Dropping negative prefixes
- One-pass greedy optimization
- Handling all-negative arrays correctly

🧩 LeetCode 238 — Product of Array Except Self

📌 Problem Statement

Given an integer array `nums`, return an array `answer` such that:

```
answer[i] = product of all elements of nums except nums[i]
```

Constraints:

- Do **not** use division
- Time complexity must be `O(n)`

🧠 Key Observations

1. For any index `i`, the product can be split into:
   - product of elements to the **left** of `i`
   - product of elements to the **right** of `i`
2. Using division fails when the array contains `0`.
3. Recomputing products for each index would be inefficient.
4. Prefix and suffix products allow reuse of previous computations.

💡 Approach (Prefix & Suffix Product)

1. Initialize an output array `answer` with all elements as `1`.
2. Traverse from left to right:
   - Store the product of all elements **before** index `i`.
3. Traverse from right to left:
   - Multiply each `answer[i]` by the product of all elements **after** index `i`.
4. Return the `answer` array.

✅ Python Code

```
def productExceptSelf(nums):
    n = len(nums)
    answer = [1] * n

    prefix = 1
    for i in range(n):
        answer[i] = prefix
        prefix *= nums[i]

    suffix = 1
    for i in range(n - 1, -1, -1):
        answer[i] *= suffix
        suffix *= nums[i]

    return answer
```

🧾 Line-by-Line Explanation

```
n = len(nums)
answer = [1] * n
```

Creates the output array initialized with `1`.

```
prefix = 1
for i in range(n):
    answer[i] = prefix
    prefix *= nums[i]
```

Stores the product of elements to the **left** of index `i`.

```
suffix = 1
for i in range(n - 1, -1, -1):
    answer[i] *= suffix
    suffix *= nums[i]
```

Multiplies the product of elements to the **right** of index `i`.

```
return answer
```

Returns the final result.

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
- **Space Complexity:** `O(1)` (excluding output array)

🧠 DSA Pattern Learned

- Prefix product
- Suffix product
- Eliminating division safely
- Two-pass array optimization

🧩 LeetCode 125 — Valid Palindrome

📌 Problem Statement

Given a string `s`, return `True` if it is a palindrome, otherwise return `False`.

Rules:

- Ignore non-alphanumeric characters
- Ignore case differences

🧠 Key Observations

1. Characters must be compared from **both ends** of the string.
2. Non-alphanumeric characters should be skipped.
3. Uppercase and lowercase letters are treated as the same.
4. The problem can be solved **in-place** without creating a new string.

💡 Approach (Two Pointers)

1. Initialize two pointers:
    - `l` at the start of the string
    - `r` at the end of the string
2. While `l < r`:
    - If `s[l]` is not alphanumeric → move `l`
    - Else if `s[r]` is not alphanumeric → move `r`
    - Else:
        - Compare `s[l]` and `s[r]` after converting to lowercase
        - If they are not equal → return `False`
        - If equal → move both pointers inward
3. If all valid characters match → return `True`

✅ Python Code (Two Pointers – In Place)

```
def Palindrome(s):
    l = 0
    r = len(s) - 1

    while l < r:
        if not s[l].isalnum():
            l += 1
        elif not s[r].isalnum():
            r -= 1
        else:
            if s[l].lower() == s[r].lower():
                l += 1
                r -= 1
            else:
                return False

    return True
```

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
- **Space Complexity:** `O(1)`

🧠 DSA Pattern Learned

- Two pointers technique
- Skipping invalid characters
- In-place comparison
- Efficient string traversal

🧩 LeetCode 344 – Reverse String

📌 Problem Statement

Given a list of characters `s`, reverse the list **in-place**.

Constraints:

- Do not return anything
- Modify the input list directly
- Use `O(1)` extra space

🧠 Key Observations

1. The input is a **list**, not a string.
2. Lists in Python are **mutable**, so elements can be swapped.
3. Reversing can be done by swapping elements from both ends.
4. No extra array is required.

💡 Approach (Two Pointers)

1. Initialize two pointers:
    - `left` at index `0`
    - `right` at index `len(s) - 1`
2. While `left < right`:
    - Swap `s[left]` and `s[right]`
    - Move `left` forward
    - Move `right` backward
3. Stop when pointers meet or cross

✅ Python Code (In-place)

```
def reverseString(s):
    left = 0
    right = len(s) - 1

    while left < right:
        s[left], s[right] = s[right], s[left]
        left += 1
        right -= 1
```

🧾 Line-by-Line Explanation

```
left = 0
right = len(s) - 1
```

Initialize pointers at the start and end of the list.

```
while left < right:
```

Continue until all characters are swapped.

```
s[left], s[right] = s[right], s[left]
```

Swap the characters in-place.

```
left += 1
right -= 1
```

Move pointers inward.

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
- **Space Complexity:** `O(1)`

🧠 DSA Pattern Learned

- Two pointers
- In-place swapping
- Constant space optimization

🧩 LeetCode 26 – Remove Duplicates from Sorted Array

📌 Problem Statement

Given a **sorted** integer array `nums`, remove the duplicates **in-place** such that each unique element appears only once.

Return the number of unique elements `k`.

The first `k` elements of `nums` should contain the final result.

🧠 Key Observations

1. The array is already **sorted**.
2. Duplicate elements are always **adjacent**.
3. No extra array is allowed.
4. The problem must be solved **in-place**.

💡 Approach (Two Pointers)

1. Use two pointers:
    - `i` → scanning pointer
    - `j` → index of last unique element
2. Initialize `j = 0` (first element is always unique).
3. Traverse the array from index `1`:
    - If `nums[i] != nums[j]`, a new unique element is found.
    - Increment `j` and place `nums[i]` at `nums[j]`.
4. After traversal, the number of unique elements is `j + 1`.

✅ Python Code

```
def removeDuplicates(nums):
    j = 0

    for i in range(1, len(nums)):
        if nums[i] != nums[j]:
            j += 1
            nums[j] = nums[i]
```

```
    return j + 1
```

🧾 Line-by-Line Explanation

```
j = 0
```

`Initializes the index of the first unique element.`

```
for i in range(1, len(nums)):
```

`Scans the array starting from the second element.`

```
if nums[i] != nums[j]:
```

`Checks if a new unique element is found.`

```
j += 1
nums[j] = nums[i]
```

`Moves the unique element forward and updates its position.`

```
return j + 1
```

`Returns the count of unique elements.`

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
- **Space Complexity:** `O(1)`

🧠 DSA Pattern Learned

- Two pointers technique
- In-place array modification
- Leveraging sorted property of arrays

🧩 LeetCode 283 — Move Zeroes

📌 Problem Statement

Given an integer array `nums`, move all `0`s to the **end** of the array while maintaining the **relative order** of the non-zero elements.

Constraints:

- Must be done **in-place**
- No extra array allowed

🧠 Key Observations

1. Order of non-zero elements must remain unchanged.
2. Zeros should naturally shift to the end.
3. The array can be modified directly.

4. Two pointers allow solving this in one pass.

💡 Approach (Two Pointers)

1. Use two pointers:
    - `i` → scans the entire array
    - `j` → position where the next non-zero element should go
2. When `nums[i]` is non-zero:
    - swap `nums[i]` with `nums[j]`
    - increment `j`
3. Continue until the array is fully traversed

✅ Python Code (In-place)

```
def moveZeroes(nums):
    j = 0
    for i in range(len(nums)):
        if nums[i] != 0:
            nums[j], nums[i] = nums[i], nums[j]
            j+=1
    return nums
```

📃 Line-by-Line Explanation

```
j =0
```

Tracks the index where the next non-zero element should be placed.

```
for i in range(len(nums)):
```

Scans each element of the array.

```
if nums[i] != 0:


```

Checks if the current element is non-zero.

```
nums[j], nums[i] = nums[i], nums[j]
```

Swaps the non-zero element to the front.

```
j += 1
```

Moves the non-zero pointer forward.

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
- **Space Complexity:** `O(1)`

🧠 DSA Pattern Learned

- Two pointers technique

- In-place swapping
- Stable rearrangement of array elements

🧩 LeetCode 11 — Container With Most Water

📌 Problem Statement

Given an array `height`, each element represents a vertical line at that index.

Find two lines such that together with the x-axis they form a container that holds the **maximum amount of water**.

🧠 Key Observations

1. Width is determined by the distance between two pointers.
2. Height is limited by the **shorter line**.
3. Moving the taller line cannot increase the area.
4. Only moving the shorter line may lead to a better solution.

💡 Approach (Two Pointers)

1. Initialize two pointers:
   - `left` at the beginning
   - `right` at the end
2. While `left < right`:
   - Calculate area using current pointers
   - Update maximum area
   - Move the pointer with the smaller height inward
3. Return the maximum area found

✅ Python Code

```
defmaxArea(height):
    left =0
    right =len(height) -1
    maxarea =0

    while left < right:
        area = (right - left) *min(height[left], height[right])
        maxarea =max(maxarea, area)

    if height[left] < height[right]:
            left +=1
    else:
            right -=1

    return maxarea
```

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
- **Space Complexity:** `O(1)`

🧠 DSA Pattern Learned

- Two pointers with greedy decision
- Always move the limiting (shorter) wall
- Maximizing area under constraints

🧩 LeetCode 42 — Trapping Rain Water

📌 Problem Statement

Given an array `height` representing an elevation map, compute how much water can be trapped after raining.

Each bar has width `1`.

🧠 Key Observations

1. Water cannot be trapped at the **edges**.
2. Water above an index depends on:
    - maximum height on the left
    - maximum height on the right
3. Water level is decided by the **shorter boundary**.
4. We don’t need exact left and right boundaries for every index — only guarantees.

💡 Approach (Two Pointers)

1. Initialize two pointers:
    - `left` at index `0`
    - `right` at index `n-1`
2. Maintain:
    - `leftMax` → tallest bar seen from the left
    - `rightMax` → tallest bar seen from the right
3. While `left < right`:
    - If `leftMax < rightMax`:
        - water at `left` is decided by `leftMax`
        - add trapped water if any
        - move `left`
    - Else:
        - water at `right` is decided by `rightMax`
        - add trapped water if any
        - move `right`
4. Accumulate total trapped water and return it.

🧾 Line-by-Line Explanation

```
left =0
right =len(height) -1
```

Pointers start at both ends of the array.

```
leftMax =0
rightMax =0
```

Track the tallest bars seen so far from each side.

```
water =0
```

Stores total trapped water.

```
while left < right:
```

Process until pointers meet.

```
if height[left] < height[right]:
```

Left side is the bottleneck — water here depends on `leftMax`.

```
water += leftMax - height[left]
```

Adds trapped water if current bar is lower than `leftMax`.

(Similar mirrored logic for the right side.)

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
- **Space Complexity:** `O(1)`

🧠 DSA Pattern Learned

- Advanced two pointers
- Using **guarantees** instead of exact boundaries
- Accumulation while shrinking window
- Greedy decision making

🧩 LeetCode 977 — Squares of a Sorted Array

📌 Problem Statement

Given a sorted integer array `nums`, return an array of the squares of each number, also sorted in non-decreasing order.

🧠 Observation

- Squaring negative numbers makes them positive.
- The largest square will always come from:
  - the leftmost negative number, or
  - the rightmost positive number.

✅ Approach 1 — Square + Sort (Simple)

💡 Idea

1. Square each element.
2. Sort the resulting array.

🧾 Code

```
def sortedSquares(nums):
    squared = [x * x for x in nums]
return sorted(squared)
```

⏱️ Complexity

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

📌 Notes

- Very readable
- Good for quick solutions
- Does not use the sorted nature of input optimally

✅ Approach 2 — Two Pointers (Optimized)

💡 Idea

1. Use two pointers (`left`, `right`) at both ends.
2. Compare absolute values.
3. Place the larger square at the end of result array.
4. Move inward.

🧾 Code

```
def sortedSquares(nums):
    n = len(nums)
    result = [0] * n

    left = 0
    right = n -1
    pos = n -1

     while left <= right:
        if abs(nums[left]) >abs(nums[right]):
        result[pos] = nums[left] * nums[left]
        left +=1
        else:
        result[pos] = nums[right] * nums[right]
        right -=1
    pos -= 1

     return result
```

⏱️ Complexity

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

📌 Why This Is Better

- Avoids unnecessary sorting
- Uses problem constraints smartly
- Preferred in interviews

🧠 Key Learning

- Knowing **multiple approaches** is powerful.
- Simple ≠ wrong.
- Optimal ≠ always required, but important to know.
- Interviews care about **patterns**, not just correctness.

🧩 LeetCode 27 — Remove Element

📌 Problem Statement

Given an integer array `nums` and an integer `val`, remove all occurrences of `val` in-place.

The order of elements **does not matter**.

Return the number of elements that are **not equal to** `val`.

The first `k` elements of `nums` should contain the result.

🧠 Key Observations

1. Order of elements is **not important**.
2. Extra space is not allowed.
3. We only care about elements **not equal to** `val`.
4. Swapping unwanted elements to the end is allowed.

💡 Approach (Two Pointers — Swap with End)

1. Use two pointers:
   - `l` → scans the array from the beginning
   - `r` → marks the end of the valid array
2. While `l <= r`:
   - If `nums[l]` is not equal to `val`, move `l` forward
   - If `nums[l]` equals `val`, swap it with `nums[r]` and decrement `r`
3. After the loop:
   - Elements from index `0` to `r` are valid
4. Return `r + 1` as the count of valid elements

🧾 Line-by-Line Explanation

```
l = 0
r = len(nums) - 1
```

Initialize two pointers at both ends.

```
while l <= r:
```

Process elements until pointers cross.

```
if nums[l] != val:
    l += 1
```

Keep valid elements and move forward.

```
else:
    nums[l], nums[r] = nums[r], nums[l]
    r -= 1
```

Swap unwanted value to the end and shrink valid range.

```
return r + 1
```

Returns the count of elements not equal to `val`.

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
- **Space Complexity:** `O(1)`

🧠 DSA Pattern Learned

- Two pointers
- In-place array modification
- Swap-with-end optimization
- Handling unordered output constraints

🧩 LeetCode 169 — Majority Element

📌 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**.

It is guaranteed that the majority element always exists.

🧠 Key Observations

1. A majority element appears **more than half** the time.
2. There can be **only one** such element.
3. Counting occurrences will always reveal it.
4. We can stop early once a count exceeds `n // 2`.

💡 Approach (Hash Map / Frequency Count)

1. Use a dictionary to store:
    - key → number
    - value → frequency
2. Traverse the array:
    - update frequency of each number
    - if any frequency becomes greater than `n // 2`, store it as the answer

3. Return the majority element.

✅ Python Code

```
def majorityElement(nums):
    freq = {}

    for num in nums:
        freq[num] = freq.get(num, 0) + 1
        if freq[num] > len(nums) // 2:
            return num
```

⏱️ Time & Space Complexity

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

🧠 Why This Works

- Since the majority element appears more than half the time,
  it must dominate the frequency count.
- Early stopping is safe because no other element can exceed this count later.

🧠 DSA Pattern Learned

- Frequency counting using hash maps
- Early exit optimization
- Guaranteed-answer problems

🧩 LeetCode 189 — Rotate Array

📌 Problem Statement

Given an integer array `nums`, rotate the array to the **right by** `k` **steps**, where `k` is non-negative.

Constraints:

- Rotation must be done **in-place**
- Use **O(1)** extra space

🧠 Key Observations

1. Rotating by `k` steps is equivalent to rotating by `k % n` steps.
2. The last `k` elements move to the front.
3. Extra space is not allowed, so we need an in-place technique.
4. The problem can be solved using **array reversal**.

💡 Approach — In-Place Reversal (Three-Step Trick)

The rotation can be achieved by performing **three reversals**:

1. Reverse the entire array
2. Reverse the first `k` elements

3. Reverse the remaining `n-k` elements

This rearranges the array into the required rotated order.

✅ Python Code (In-Place)

```
def rotate(nums, k):
    n = len(nums)
    k = k % n

    def reverse(l, r):
        while l < r:
            nums[l], nums[r] = nums[r], nums[l]
            l += 1
            r -= 1

    reverse(0, n - 1)
    reverse(0, k - 1)
    reverse(k, n - 1
```

🧾 Line-by-Line Explanation

```
n = len(nums)
```

- Stores the length of the array.
- Needed to manage rotation and indexing.

```
k = k % n
```

- Handles cases where `k` is larger than array length.
- Rotating `n` times gives the same array.

```
def reverse(l, r):
```

- Helper function to reverse a portion of the array.
- `l` is the left index, `r` is the right index.

```
while l < r:
```

- Continues until the two pointers meet.

```
nums[l], nums[r] = nums[r], nums[l]
```

- Swaps the elements at positions `l` and `r`.

```
l += 1
r -=
```

- Moves the pointers inward after each swap.

🔁 Core Logic (Three Reversals)

```
reverse(0, n - 1)
```

- Reverses the entire array.
- Brings the last `k` elements to the front (in reverse order).

```
reverse(0, k - 1)
```

- Reverses the first `k` elements.
- Restores their correct order.

```
reverse(k, n - 1)
```

- Reverses the remaining elements.
- Restores their correct order.

🧪 Example Walkthrough

For:

```
nums = [1,2,3,4,5,6,7]
k = 3
```

1. Reverse all:

```
[7,6,5,4,3,2,1]
```

1. Reverse first `k`:

```
[5,6,7,4,3,2,1]
```

1. Reverse remaining:

```
[5,6,7,1,2,3,4]
```

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
- **Space Complexity:** `O(1)`

🧠 DSA Pattern Learned

- In-place array manipulation
- Reversal technique
- Optimizing space usage
- Using modulo to simplify rotations

🧩 LeetCode 724 — Find Pivot Index

📌 Problem Statement

Given an array of integers `nums`, calculate the **pivot index** of the array.

The pivot index is the index where:

```
sum of elements to the left == sum of elements to the right
```

- The pivot element itself is **not included** in either sum
- If multiple pivot indices exist, return the **leftmost one**
- If no pivot index exists, return `-1`

🧠 Key Observations

1. Recalculating left and right sums for every index is inefficient.
2. The total sum of the array can be computed once.
3. Using a **running (prefix) sum**, we can derive the right sum in `O(1)` time.
4. This avoids nested loops and improves performance.

💡 Optimized Approach (Prefix / Running Sum)

Idea:

- Compute the **total sum** of the array.
- Maintain a variable `left_sum` while traversing the array.
- For index `i`, the right sum can be calculated as:

```
right_sum = total_sum - left_sum - nums[i]
```

- If `left_sum == right_sum`, then `i` is the pivot index.

✅ Python Code (Optimized)

```
def pivotIndex(nums):
    total = sum(nums)
    left_sum = 0

    for i in range(len(nums)):
        if left_sum == total - left_sum - nums[i]:
            return i
        left_sum += nums[i]

    return -1
```

📝 Line-by-Line Explanation

```
total = sum(nums)
```

- Stores the total sum of all elements in the array.

```
left_sum = 0
```

- Tracks the sum of elements to the left of the current index.

```
for i in range(len(nums)):
```

- Iterates through each index of the array.

```
if left_sum == total - left_sum - nums[i]:
```

- Checks if the sum of elements on the left equals the sum on the right.

```
left_sum += nums[i]
```

- Updates the left sum after checking the pivot condition.

```
return -1
```

- Returned when no pivot index is found.

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
- **Space Complexity:** `O(1)`

🧠 DSA Pattern Learned

- Prefix sum / running sum
- Avoiding repeated computation
- Deriving values instead of recalculating
- Optimizing brute-force solutions

🧩 LeetCode 268 – Missing Number

📌 Problem Statement

Given an array `nums` containing `n` distinct numbers taken from the range `[0, n]`, return the **one number that is missing** from the array.

🧠 Key Observations

1. The array contains `n` numbers but the range is from `0` to `n` (total `n + 1` numbers).
2. Exactly **one number is missing**.
3. The sum of numbers from `0` to `n` can be calculated using a formula.
4. Subtracting the actual sum of the array from the expected sum gives the missing number.

💡 Approach – Math (Sum Formula)

Idea:

- Calculate the expected sum of numbers from `0` to `n` using:

  ```
  n * (n + 1) // 2
  ```

- Calculate the actual sum of elements in the array.
- The difference between the two sums is the missing number.

✅ Python Code (My Solution)

```
def missingNumber(nums):
    n = len(nums)
    expected_sum = (n * (n + 1)) // 2
    s = sum(nums)
    if s != expected_sum:
        return expected_sum - s
```

🧾 Line-by-Line Explanation

```
n = len(nums)
```

- Stores the length of the array.

```
expected_sum = (n * (n + 1)) // 2
```

- Calculates the sum of all numbers from `0` to `n`.

```
s = sum(nums)
```

- Calculates the sum of elements present in the array.

```
return expected_sum - s
```

- The difference gives the missing number.

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
- **Space Complexity:** `O(1)`

🧠 DSA Pattern Learned

- Mathematical reasoning
- Using formulas to avoid extra space
- Optimizing from brute-force to optimal
- Handling edge cases like missing `0` or missing `n`

🧩 LeetCode 350 — Intersection of Two Arrays II

📌 Problem Statement

Given two integer arrays `nums1` and `nums2`, return an array of their **intersection**.

Each element in the result should appear **as many times as it appears in both arrays**.

The result can be returned in **any order**.

🧠 Key Observations

1. This is **not** a set intersection — duplicates matter.
2. Frequency of elements must be tracked.
3. Each element can only be used as many times as it appears in **both** arrays.
4. A hash map is the most efficient approach.

💡 Approach — Hash Map (Frequency Counting)

Idea:

1. Build a frequency dictionary from `nums1`.
2. Traverse `nums2`:
    - If the current element exists in the dictionary **and frequency > 0**:
        - add it to result
        - decrement its frequency
3. This ensures duplicates are handled correctly.

✅ Python Code (My Solution)

```
def intersect(nums1, nums2):
    freq = {}
    res = []

    for num in nums1:
        freq[num] = freq.get(num, 0) + 1

    for num in nums2:
        if freq.get(num, 0) > 0:
            res.append(num)
            freq[num] -= 1

    return res
```

📃 Line-by-Line Explanation

```
freq = {}
```

- Dictionary to store frequency of elements from `nums1`.

```
for num in nums1:
    freq[num] = freq.get(num, 0) + 1
```

- Counts how many times each number appears in `nums1`.

```
for num in nums2:
```

- Traverses the second array.

```
if freq.get(num, 0) > 0:
```

- Checks if the element exists and is still available.

```
res.append(num)
freq[num] -= 1
```

- Adds element to result and decrements frequency to avoid overuse.

⏱️ Time & Space Complexity

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

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

🧠 DSA Pattern Learned

- Frequency counting using hash maps
- Handling duplicates correctly
- “Use & consume” pattern
- Efficient intersection logic

🧩 LeetCode 66 — Plus One

📌 Problem Statement

Given a non-negative integer represented as an array of digits, increment the number by one and return the resulting array.

The digits are stored such that the most significant digit is at the beginning of the array.

🧠 Key Observations

1. Addition starts from the **last digit**.
2. If the last digit is less than 9, simply increment and return.
3. If the digit is 9, it becomes 0 and a carry is generated.
4. If all digits are 9, a new digit `1` must be inserted at the beginning.

💡 Approach (Carry Handling)

- Start from the last digit.
- Handle carry by setting trailing 9s to 0.
- Once a non-9 digit is found, increment it.
- If no such digit exists, insert `1` at the front.

✅ Python Code (My Solution)

```
def plusOne(digits):
    n = len(digits)

    if digits[n - 1] < 9:
        digits[n - 1] += 1
        return digits

    i = n - 1
    while i >= 0 and digits[i] == 9:
        digits[i] = 0
        i -= 1

    if i == -1:
        digits.insert(0, 1)
    else:
        digits[i] += 1

    return digits
```

🧾 Line-by-Line Explanation

```
if digits[n - 1] < 9:
```

- `If the last digit is not 9, increment and return immediately.`

```
while i >= 0 and digits[i] == 9:
```

- `Converts trailing 9s to 0 due to carry.`

```
if i == -1:
    digits.insert(0, 1)
```

- `Handles case where all digits were 9 (e.g., 999 → 1000).`

```
else:
    digits[i] += 1
```

- Adds carry to the first non-9 digit.

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
- **Space Complexity:** `O(1)`

🧠 DSA Pattern Learned

- Carry propagation
- Right-to-left array traversal
- Edge case handling
- In-place modification
