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

🧩 LeetCode 122 – Best Time to Buy and Sell Stock II

📌 Problem Statement

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

You can complete **multiple transactions**, but:

- You must sell before buying again.

The task is to calculate the **maximum profit** possible.

🧠 Key Observations

1. Profit is only made when the price **increases**.
2. Each upward price movement can be treated as a **separate transaction**.
3. Multiple small profits add up to the same result as one large transaction.
4. Downward price movements should be **ignored**.

💡 Approach (Greedy Profit Accumulation)

- Traverse the array from left to right.
- Compare today’s price with yesterday’s.

- If today's price is higher:
  - Add `(today - yesterday)` to total profit.
- Skip days where the price decreases.

✅ Python Code (My Solution)

```
class Solution:
    def maxProfit(self, prices):
        profit = 0

        for i in range(1, len(prices)):
            if prices[i] > prices[i - 1]:
                profit += prices[i] - prices[i - 1]

        return profit
```

🧾 Line-by-Line Explanation

```
profit = 0
```

- Initializes total profit to zero.

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

- Iterates from the second day to compare with the previous day.

```
if prices[i] > prices[i - 1]:
```

- Checks for an upward price movement.

```
profit += prices[i] - prices[i - 1]
```

- Captures profit from yesterday to today.

```
return profit
```

- Returns the total accumulated profit.

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Greedy strategy
- Adjacent array comparison
- Profit accumulation
- Local vs global optimization

🧩 LeetCode 128 – Longest Consecutive Sequence

📌 Problem Statement

Given an unsorted array of integers `nums`, return the length of the longest sequence of **consecutive integers**.

The algorithm must run in **O(n) time**.

🧠 Key Observations

1. The order of elements does **not** matter.
2. We only care whether a number **exists**.
3. Sorting breaks the O(n) constraint.
4. Each number should be processed **only once**.

💡 Core Idea (Very Important)

We use a **set** for O(1) lookups.

A number is considered the **start of a sequence** if:

- `num - 1` does **not** exist in the set

This guarantees:

- No duplicate counting
- Each sequence is explored exactly once

✅ Python Code (Optimal O(n) Solution)

```
class Solution:
    def longestConsecutive(self, nums):
        num_set = set(nums)
        longest = 0

        for num in num_set:
            # start only if num is the beginning of a sequence
            if num - 1 not in num_set:
                current = num
                count = 1

                while current + 1 in num_set:
                    current += 1
                    count += 1

                longest = max(longest, count)

        return longest
```

🧾 Line-by-Line Explanation

```
num_set = set(nums)
```

- Converts array to a set for O(1) lookups.

```
if num - 1 not in num_set:
```

- Ensures we only start counting from the **beginning** of a sequence.

```
while current + 1 in num_set:
```

- Expands the sequence forward.

```
longest = max(longest, count)
```

- Keeps track of the maximum sequence length found.

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
  - Each number is visited at most once.
- **Space Complexity:** `O(n)`
  - Due to the hash set.

🧠 DSA Pattern Learned

- HashSet for O(1) existence checks
- Sequence start detection
- Avoiding duplicate work
- Replacing sorting with hashing

🧩 LeetCode 3 – Longest Substring Without Repeating Characters

📌 Problem Statement

Given a string `s`, find the length of the longest substring

that contains **no repeating characters**.

🧠 Key Observations

1. Substring must be **contiguous**.
2. Repeating characters make the substring invalid.
3. We must dynamically expand and shrink the window.
4. Checking duplicates must be done in **O(1)** time.

💡 Approach (Sliding Window + HashSet)

- Use two pointers (`l` and `r`) to represent the sliding window.
- Use a `set` to store characters currently inside the window.
- Expand the window by moving `r`.
- If a duplicate character enters:
  - shrink the window from the left
  - remove characters until the window becomes valid again.
- Update the maximum length whenever the window is valid.

✅ Python Code (My Solution)

```
def lengthOfLongestSubstring(self, s):
        char_set = set()
        l = 0
        max_len = 0

        for r in range(len(s)):
            while s[r] in char_set:
                char_set.remove(s[l])
                l += 1

            char_set.add(s[r])
            max_len = max(max_len, r - l + 1)

        return max_len
```

🗒️ Line-by-Line Explanation

```
char_set = set()
```

- Stores unique characters currently in the window.

```
l = 0
```

- Left pointer of the sliding window.

```
for r in range(len(s)):
```

- Right pointer expands the window.

```
while s[r] in char_set:
```

- Window becomes invalid when a duplicate enters.

```
char_set.remove(s[l])
l += 1
```

- Shrinks the window from the left until it becomes valid.

```
char_set.add(s[r])
```

- Adds the current character to the window.

```
max_len = max(max_len, r - l + 1)
```

- Updates the maximum valid window length.

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
  - Each character is added and removed at most once.
- **Space Complexity:** `O(n)`
  - HashSet stores up to `n` characters.

🧠 DSA Pattern Learned

- Sliding Window (variable size)
- Two-pointer technique
- HashSet for O(1) lookup
- Window expansion and contraction

🧩 LeetCode 567 – Permutation in String

📌 Problem Statement

Given two strings `s1` and `s2`, determine if `s2` contains

any permutation of `s1` as a substring.

A permutation means:

- same characters
- same frequencies
- order does NOT matter

🧠 Key Observations

1. Order of characters is irrelevant — **frequency matters**.
2. Any valid permutation must have **length = len(s1)**.
3. We must check **all substrings of fixed size** in `s2`.
4. Rebuilding frequency maps repeatedly is inefficient.

💡 Approach (Fixed-Size Sliding Window)

- Use a **fixed-size sliding window** of length `len(s1)`.
- Maintain:
  - `freq1` → frequency of `s1`
  - `freq2` → frequency of current window in `s2`
- Slide the window one character at a time:
  - remove the left character
  - add the right character
- If `freq1 == freq2` at any point → permutation found.

✅ Python Code (My Solution)

```
class Solution:
    def checkInclusion(self, s1: str, s2: str) -> bool:
        if len(s1) > len(s2):
            return False

        freq1 = {}
        freq2 = {}

        for ch in s1:
            freq1[ch] = freq1.get(ch, 0) + 1

        window = len(s1)
```

```
    for i in range(window):
        freq2[s2[i]] = freq2.get(s2[i], 0) + 1

    if freq1 == freq2:
        return True

    l = 0
    for r in range(window, len(s2)):
        freq2[s2[r]] = freq2.get(s2[r], 0) + 1

        freq2[s2[l]] -= 1
        if freq2[s2[l]] == 0:
            del freq2[s2[l]]
        l += 1

        if freq1 == freq2:
            return True

    return False
```

🧾 Line-by-Line Explanation

```
freq1 = {}
freq2 = {}
```

- Store character frequencies for `s1` and the current window.

```
window = len(s1)
```

- Fixes the sliding window size.

```
freq2[s2[r]] += 1
```

- Adds the incoming character to the window.

```
freq2[s2[l]] -= 1
```

- Removes the outgoing character from the window.

```
if freq2[s2[l]] == 0:
    del freq2[s2[l]]
```

- Keeps the frequency map clean for accurate comparison.

```
if freq1 == freq2:
```

- Checks if current window is a valid permutation.

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
  - Each character is added and removed once.
- **Space Complexity:** `O(1)`

    - At most 26 lowercase letters stored.

🧠 DSA Pattern Learned

- Fixed-size Sliding Window
- Frequency map comparison
- Efficient window updates
- Avoiding brute-force recomputation

🧩 LeetCode 438 – Find All Anagrams in a String

📌 Problem Statement

Given two strings `s` and `p`, find all starting indices of `p`'s anagrams in `s`.

An anagram means:

- same characters
- same frequencies
- order does NOT matter

🧠 Key Observations

1. Any anagram of `p` must have **length = len(p)**.
2. Order is irrelevant – **frequency matters**.
3. We must scan the **entire string** `s`.
4. Rebuilding frequency maps repeatedly is inefficient.

💡 Approach (Fixed-Size Sliding Window)

- Use a sliding window of fixed size `len(p)`.
- Maintain:
    - `freq_p` → frequency map of `p`
    - `freq_w` → frequency map of current window in `s`
- Slide the window one step at a time:
    - add the incoming character
    - remove the outgoing character
- Whenever `freq_w == freq_p`, store the **left index**.

✅ Python Code (My Solution)

```
class Solution:
    def findAnagrams(self, s: str, p: str) -> list[int]:
        if len(p) > len(s):
            return []

        freq_p = {}
        freq_w = {}

        for ch in p:
```

```
            freq_p[ch] = freq_p.get(ch, 0) + 1

        window = len(p)

        for i in range(window):
            freq_w[s[i]] = freq_w.get(s[i], 0) + 1

        res = []
        if freq_w == freq_p:
            res.append(0)

        l = 0
        for r in range(window, len(s)):
            freq_w[s[r]] = freq_w.get(s[r], 0) + 1

            freq_w[s[l]] -= 1
            if freq_w[s[l]] == 0:
                del freq_w[s[l]]
            l += 1

            if freq_w == freq_p:
                res.append(l)

        return res
```

🧾 Line-by-Line Explanation

```
freq_p = {}
freq_w = {}
```

- Store character frequencies for pattern `p` and current window.

```
window = len(p)
```

- Fixes the sliding window size.

```
freq_w[s[r]] += 1
```

- Adds the new character entering the window.

```
freq_w[s[l]] -= 1
```

- Removes the character leaving the window.

```
if freq_w[s[l]] == 0:
    del freq_w[s[l]]
```

- Keeps the frequency map clean for accurate comparison.

```
res.append(l)
```

- Stores the starting index of a valid anagram.

⏱️ Time & Space Complexity

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

- Each character is processed once.
- **Space Complexity:** `O(1)`
  - At most 26 lowercase letters stored.

🧠 DSA Pattern Learned

- Fixed-size Sliding Window
- Frequency map comparison
- Efficient window update (add/remove)
- Collecting multiple valid results

🧩 LeetCode 704 – Binary Search

📌 Problem Statement

Given a sorted array `nums` and an integer `target`,

return the index of `target` if it exists in the array.

Otherwise, return `-1`.

The algorithm must run in `O(log n)` time.

🧠 Key Observations

1. Array is **sorted** → binary search applicable.
2. At each step, half of the search space can be discarded.
3. Binary search works by maintaining a valid search interval.
4. Edge cases (like single-element arrays) must be handled correctly.

💡 Approach (Binary Search – Closed Interval)

- Use two pointers:
  - `low` → start of search space
  - `high` → end of search space
- Treat the search space as a **closed interval** `[low, high]`.
- While there is at least one element left to check:
  - calculate `mid`
  - compare `nums[mid]` with `target`
  - discard the irrelevant half
- If the target is never found, return `-1`.

✅ Python Code (My Solution)

```
class Solution:
    def search(self, nums, target):
        low = 0
        high = len(nums) - 1

        while low <= high:
            mid = (low + high) // 2
```

```
            if nums[mid] == target:
                return mid
            elif nums[mid] < target:
                low = mid + 1
            else:
                high = mid - 1

        return -1
```

🧾 Line-by-Line Explanation

```
low = 0
high = len(nums) - 1
```

- Initializes the search range as a **closed interval**.

```
while low <= high:
```

- Ensures that even a single remaining element is checked.

```
mid = (low + high) // 2
```

- Finds the middle index of the current search space.

```
if nums[mid] == target:
```

- Target found — return index immediately.

```
elif nums[mid] < target:
    low = mid + 1
```

- Discard left half (target must be on the right).

```
else:
    high = mid - 1
```

- Discard right half (target must be on the left).

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Binary Search (Closed Interval)
- Invariant-based thinking
- Correct loop condition (`low <= high`)
- Handling edge cases (single-element arrays)

🧩 LeetCode 35 – Search Insert Position

📌 Problem Statement

Given a sorted array `nums` and an integer `target`,

return the index if the target is found.

If not, return the index where it should be inserted

to maintain the sorted order.

🧠 Key Observations

1. The array is **sorted** → binary search applies.
2. If the target is not present, it still has a **valid position**.
3. Binary search naturally narrows down where the target belongs.
4. The final value of `low` gives the **correct insert position**.

💡 Approach (Binary Search — Closed Interval)

- Use two pointers `low` and `high` representing a **closed interval**.
- Perform standard binary search:
  - if `nums[mid] == target`, return `mid`
  - if `nums[mid] < target`, move right
  - else, move left
- If the loop ends without finding the target:
  - return `low`, which indicates the correct insert index.

✅ Python Code (My Solution)

```
class Solution:
    def searchInsert(self, nums, target):
        low = 0
        high = len(nums) - 1

        while low <= high:
            mid = (low + high) // 2

            if nums[mid] == target:
                return mid
            elif target > nums[mid]:
                low = mid + 1
            else:
                high = mid - 1

        return low
```

🧾 Line-by-Line Explanation

```
low = 0
high = len(nums) - 1
```

- Defines the search space as a **closed interval** `[low, high]`.

```
while low <= high:
```

- Ensures even a single remaining element is checked.

```
mid = (low + high) // 2
```

- Finds the midpoint of the current search space.

```
if nums[mid] == target:
```

- Target found — return its index.

```
elif target > nums[mid]:
    low = mid + 1
```

- Target lies in the right half.

```
else:
    high = mid - 1
```

- Target lies in the left half.

```
return low
```

- When target is not found, `low` gives the correct insert position.

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Binary Search (Closed Interval)
- Insert-position logic using `low`
- Handling “not found” cases cleanly
- Boundary & invariant reasoning

🧩 LeetCode 34 — Find First and Last Position of Element in Sorted Array

📌 Problem Statement

Given a sorted array `nums` and a `target`, return the first and last position of `target` in the array.

If the target does not exist, return `[-1, -1]`.

The solution must run in `O(log n)` time.

🧠 Key Observations

1. The array is **sorted** → binary search applies.
2. The target may appear **multiple times**.
3. A single binary search can find **any one occurrence**, not boundaries.
4. To get exact boundaries, we must perform **two binary searches**.

💡 Approach (Two Binary Searches)

- **First Binary Search**
  - Goal: find the **leftmost (first) occurrence** of `target`.
  - If `nums[mid] == target`, continue searching **left side**.
- **Second Binary Search**
  - Goal: find the **rightmost (last) occurrence** of `target`.
  - If `nums[mid] == target`, continue searching **right side**.
- If the target is never found, return `[-1, -1]`.

✅ Python Code (My Solution)

```
class Solution:
    def searchRange(self, nums, target):
        def findFirst():
            low, high = 0, len(nums) - 1
            first = -1

            while low <= high:
                mid = (low + high) // 2
                if nums[mid] == target:
                    first = mid
                    high = mid - 1
                elif nums[mid] < target:
                    low = mid + 1
                else:
                    high = mid - 1

            return first

        def findLast():
            low, high = 0, len(nums) - 1
            last = -1

            while low <= high:
                mid = (low + high) // 2
                if nums[mid] == target:
                    last = mid
                    low = mid + 1
                elif nums[mid] < target:
                    low = mid + 1
                else:
                    high = mid - 1

            return last

        return [findFirst(), findLast()]
```

🧾 Line-by-Line Explanation

Finding the first occurrence

```
if nums[mid] == target:
    first = mid
```

```
    high = mid - 1
```

- Stores index and continues searching **left**.

Finding the last occurrence

```
if nums[mid] == target:
    last = mid
    low = mid + 1
```

- Stores index and continues searching **right**.

⏱️ Time & Space Complexity

- **Time Complexity:** `O(log n)`
  - Two binary searches.
- **Space Complexity:** `O(1)`

🧠 DSA Pattern Learned

- Binary Search on boundaries
- Leftmost & rightmost occurrence
- Using binary search beyond “find one”
- Precise control of search direction

🧩 LeetCode 151 — Reverse Words in a String

📌 Problem Statement

Given a string `s`, reverse the order of the words.

A word is a sequence of non-space characters.

The output string should contain words separated by **a single space only**, with **no extra spaces**.

🧠 Key Observations

1. Extra spaces (leading, trailing, multiple) must be removed.
2. We don’t need to reverse characters — only **word order**.
3. Python’s `split()` automatically:
   - removes extra spaces
   - gives a clean list of words
4. Once we have words → reverse the list.

💡 Approach (Split + Two Pointers)

- Split the string into words using `split()`.
- Use two pointers:
  - `l` at the start
  - `r` at the end
- Swap words until pointers meet.
- Join the list back using `" ".join()`.

✅ Python Code

```
class Solution:
    def reverseWords(self, s: str) -> str:
        l = 0
        w = s.split()
        r = len(w) - 1

        while l < r:
            w[l], w[r] = w[r], w[l]
            l += 1
            r -= 1

        return " ".join(w)
```

🧾 Line-by-Line Explanation

```
w = s.split()
```

- Removes extra spaces and extracts words.

```
while l < r:
    w[l], w[r] = w[r], w[l]
```

- Reverses word order using two pointers.

```
return " ".join(w)
```

- Joins words with a single space (as required).

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- String preprocessing
- Two-pointer technique
- Clean handling of edge cases
- Leveraging built-in functions smartly

🧩 LeetCode 162 – Find Peak Element

📌 Problem Statement

Given an integer array `nums`, find a **peak element** and return its index.

A peak element is strictly greater than its neighbors.

Assume `nums[-1] = nums[n] = -∞`.

You must solve it in **O(log n)** time.

🧠 Key Observations

1. The array can be visualized as **slopes**.
2. If the slope is rising, a peak exists **on the right**.
3. If the slope is falling, a peak exists **on the left**.
4. Only one comparison is needed:

   `nums[mid]` vs `nums[mid + 1]`

💡 Approach (Binary Search on Slope)

- Initialize two pointers `low` and `high`
- While `low < high`:
  - Compute `mid`
  - Compare `nums[mid]` with `nums[mid + 1]`
  - Move toward the guaranteed peak side
- When loop ends, `low` points to a peak

✅ Python Code

```
class Solution:
    def findPeakElement(self, nums):
        low, high = 0, len(nums) - 1

        while low < high:
            mid = (low + high) // 2

            if nums[mid] < nums[mid + 1]:
                low = mid + 1
            else:
                high = mid

        return low
```

🧾 Why returning `low` works

- The search space always **contains at least one peak**
- `low` and `high` converge at that peak
- No extra checks needed

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Binary search without a target
- Directional decisions
- Peak / slope based problems
- Interview-grade optimization

🧩 LeetCode 367 – Valid Perfect Square

📌 Problem Statement

Given a positive integer `num`, determine whether it is a **perfect square**.

A perfect square is an integer that is the square of another integer

(e.g. 1, 4, 9, 16, 25, …).

⚠ You are **not allowed** to use `sqrt()`.

🧠 Key Observations

1. If `num = x²`, then `x` lies between `1` and `num`.
2. The function `f(x) = x * x` is **monotonically increasing**.
3. Since the range is sorted → **Binary Search applies**.

💡 Approach (Binary Search)

- Set `low = 1`, `high = num`
- Pick `mid`
- Compare `mid * mid` with `num`
  - If equal → perfect square
  - If smaller → search right
  - If larger → search left
- If search ends without match → not a perfect square

✅ Python Code (Final)

```
class Solution:
    def isPerfectSquare(self, num: int) -> bool:
        low, high = 1, num

        while low <= high:
            mid = (low + high) // 2
            square = mid * mid

            if square == num:
                return True
            elif square < num:
                low = mid + 1
            else:
                high = mid - 1

        return False
```

🧾 Line-by-Line Explanation

```
low, high = 1, num
```

- Square root of `num` must lie in this range.

```
mid = (low + high) // 2
square = mid * mid
```

- Check square of mid.

```
if square == num:
    return True
```

- Perfect square found.

```
elif square < num:
    low = mid + 1
```

- Need a larger number.

```
else:
    high = mid - 1
```

- Need a smaller number.

```
return False
```

- No integer square root exists.

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Binary Search on answer space
- Monotonic functions
- Avoiding floating point operations
- Mathematical reasoning with search

🧩 LeetCode 69 – Sqrt(x)

📌 Problem Statement

Given a non-negative integer `x`, return the **integer part** of its square root.

The decimal part must be discarded, and **built-in square root functions are not allowed**.

🧠 Key Observations

1. If `x = n²`, then `√x = n`.
2. If `x` is not a perfect square, we return the **floor value**.
3. The function `f(n) = n²` is **monotonically increasing**.
4. Since the search space is sorted, **binary search can be applied**.

💡 Approach (Binary Search on Answer Space)

- Handle small cases:
  - If `x < 2`, return `x` directly.
- Search for the square root between `1` and `x`.
- Use binary search to find the **largest integer** `mid` such that:

```
mid * mid <= x
```

- Store the last valid `mid` as the answer.

✅ Python Code (My Solution)

```
class Solution:
    def mySqrt(self, x: int) -> int:
        if x < 2:
            return x

        low, high = 1, x
        ans = 0

        while low <= high:
            mid = (low + high) // 2

            if mid * mid <= x:
                ans = mid
                low = mid + 1
            else:
                high = mid - 1

        return ans
```

🧾 Line-by-Line Explanation

```
if x < 2:
    return x
```

- Handles edge cases `x = 0` and `x = 1`.

```
low, high = 1, x
```

- Defines the binary search range for the square root.

```
mid = (low + high) // 2
```

- Computes the midpoint.

```
if mid * mid <= x:
```

- Checks if `mid` is a valid square root candidate.

```
ans = mid
low = mid + 1
```

- Stores valid answer and searches for a larger one.

```
else:
    high = mid - 1
```

- Discards larger values.

```
return ans
```

- Returns the integer square root.

⏱️ Time & Space Complexity

- **Time Complexity:** `O(log x)`
- **Space Complexity:** `O(1)`

🧠 DSA Pattern Learned

- Binary Search on answer space
- Monotonic functions
- Floor value logic
- Handling edge cases cleanly

🧩 LeetCode 33 – Search in Rotated Sorted Array

📌 Problem Statement

You are given a sorted array `nums` that has been rotated at an unknown pivot.

All values are **distinct**.

Given a `target`, return its index if it exists in the array.

Otherwise, return `-1`.

The algorithm must run in `O(log n)` time.

🧠 Key Observations

1. The array is **not fully sorted**, but **one half is always sorted**.
2. A normal binary search does not work directly due to rotation.
3. At each step, we must:
    - detect the sorted half
    - check if the target lies in that half
4. Only one half can be safely discarded each iteration.

💡 Approach (Binary Search on Rotated Array)

- Use binary search with pointers `low` and `high`.
- At each iteration:
    1. Compute `mid`
    2. If `nums[mid] == target`, return `mid`
    3. Determine which half is sorted
    4. Decide which half to discard based on target range
- Continue until the search space is exhausted.

🔑 Sorted-Half Detection Logic

Case 1: Left half is sorted

```
nums[low] <= nums[mid]
```

- If:

```
nums[low] <= target < nums[mid]
```

→ target lies in the left half

→ move `high = mid - 1`

- Else:

  → discard left half

  → move `low = mid + 1`

Case 2: Right half is sorted

```
nums[low] > nums[mid]
```

- If:

```
nums[mid] < target <= nums[high]
```

→ target lies in the right half

→ move `low = mid + 1`

- Else:

  → discard right half

  → move `high = mid - 1`

✅ Python Code (My Solution)

```
class Solution:
    def search(self, nums, target):
        low, high = 0, len(nums) - 1

        while low <= high:
            mid = (low + high) // 2

            if nums[mid] == target:
                return mid

            # Left half is sorted
            if nums[low] <= nums[mid]:
                if nums[low] <= target < nums[mid]:
                    high = mid - 1
                else:
                    low = mid + 1
            # Right half is sorted
            else:
                if nums[mid] < target <= nums[high]:
                    low = mid + 1
                else:
```

```
                    high = mid - 1

        return -1
```

🗒️ Line-by-Line Explanation

```
if nums[mid] == target:
```

- Immediate success case.

```
if nums[low] <= nums[mid]:
```

- Detects that the **left half is sorted**.

```
if nums[low] <= target < nums[mid]:
```

- Checks whether the target lies inside the sorted left half.

```
low = mid + 1 / high = mid - 1
```

- Discards the half where the target cannot exist.

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Binary Search on rotated arrays
- Sorted-half detection
- Conditional space elimination
- Combining binary search with invariants

🧩 LeetCode 1295 — Find Numbers with Even Number of Digits

📌 Problem Statement

Given an array of integers `nums`, count how many numbers

contain an **even number of digits**.

🧠 Key Observations

1. Every number must be checked → **linear traversal required**
2. Digit count determines validity
3. Python provides an easy way to count digits using strings
4. Even digit count → number qualifies

💡 Approach (Linear Search + String Conversion)

- Traverse each number in the array

- Convert the number to a string
- Use `len()` to count digits
- If digit count is even, increment the counter

This approach is:

- very readable
- concise
- less error-prone

✅ Python Code (My Solution)

```
class Solution:
    def findNumbers(self, nums):
        count = 0

        for num in nums:
            if len(str(num)) % 2 == 0:
                count += 1

        return count
```

🧾 Line-by-Line Explanation

```
for num in nums:
```

- Traverses each element (linear search).

```
len(str(num))
```

- Converts number to string and counts digits.

```
% 2 == 0
```

- Checks if digit count is even.

```
count += 1
```

- Increments result counter.

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
- **Space Complexity:** `O(1)`
  - Temporary string creation does not scale with input size.

🧠 DSA Pattern Learned

- Linear Search
- Element-wise validation
- Smart use of built-in functions
- Writing clean and readable code

🧩 LeetCode 20 – Valid Parentheses

📌 Problem Statement

Given a string `s` consisting of parentheses characters `()[]{}`, determine whether the string is **valid**.

A valid string must satisfy:

- Every opening bracket has a corresponding closing bracket
- Brackets are closed in the correct **order**

🧠 Key Observations

1. Brackets must be matched in **reverse order of appearance**
2. The most recent opening bracket must close first
3. This behavior matches **LIFO (Last In, First Out)**

👉 Hence, a **stack** is the ideal data structure.

💡 Approach (Stack-Based Validation)

- Use a stack to store **opening brackets**
- Traverse the string character by character
- If the character is:
  - an **opening bracket** → push to stack
  - a **closing bracket** → check against the stack's top
- If mismatch or stack is empty → invalid
- At the end, stack must be empty

🔑 Matching Logic

Use a dictionary to map:

```
closing bracket → opening bracket
```

This allows quick validation of correct pairs.

✅ Python Code (My Solution)

```
class Solution:
    def isValid(self, s: str) -> bool:
        stack = []
        mapping = {
            ')': '(',
            '}': '{',
            ']': '['
        }

        for ch in s:
            if ch in mapping:
                if not stack or stack[-1] != mapping[ch]:
                    return False
                stack.pop()
            else:
```

```
                stack.append(ch)

        return not stack
```

🗒️ Line-by-Line Explanation

```
stack = []
```

- Stores unmatched opening brackets.

```
mapping = {')':'(', '}':'{', ']':'['}
```

- Defines valid bracket pairs.

```
if ch in mapping:
```

- Identifies a closing bracket.

```
if not stack or stack[-1] != mapping[ch]:
```

- Invalid if:
  - no opening bracket exists
  - or wrong bracket order

```
stack.pop()
```

- Removes matched opening bracket.

```
return not stack
```

- Valid only if no unmatched brackets remain.

⏱️ Time & Space Complexity

- **Time Complexity:** `O(n)`
- **Space Complexity:** `O(n)` (stack in worst case)

🧠 DSA Pattern Learned

- Stack (LIFO)
- Matching pairs
- Order validation
- Handling nested structures

🧩 LeetCode 1047 – Remove All Adjacent Duplicates in String

📌 Problem Statement

Given a string `s`, remove all **adjacent duplicate characters** repeatedly

until no such duplicates exist.

Return the final string.

🧠 Key Observations

1. Only **adjacent** duplicates matter.
2. When a duplicate pair is removed, new adjacent duplicates may form.
3. We must always compare the **current character** with the **previous one**.
4. This behavior matches **Last In, First Out (LIFO)** → Stack.

💡 Approach (Stack-Based Removal)

- Use a stack to process characters one by one.
- For each character:
  - If stack is not empty **and** top of stack equals current character → pop
  - Otherwise → push current character
- At the end, the stack contains the final string without adjacent duplicates.

✅ Python Code (My Solution)

```
class Solution:
    def removeDuplicates(self, s: str) -> str:
        stack = []

        for ch in s:
            if stack and stack[-1] == ch:
                stack.pop()
            else:
                stack.append(ch)

        return "".join(stack)
```

🧾 Line-by-Line Explanation

```
stack = []
```

- Stack stores characters after removing duplicates so far.

```
for ch in s:
```

- Traverse each character in the string.

```
if stack and stack[-1] == ch:
```

- Checks if the current character forms an adjacent duplicate.

```
stack.pop()
```

- Removes the duplicate pair.

```
else:
    stack.append(ch)
```

- Keeps the character if no duplicate exists.

```
return "".join(stack)
```

- Converts stack back to the final string.

⏱️ Time & Space Complexity

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

  (each character is pushed and popped at most once)

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

  (stack in worst case)

🧠 DSA Pattern Learned

- Stack (LIFO)
- Adjacent comparison
- Repeated removal using stack
- In-place-like behavior with auxiliary stack

🧩 LeetCode 155 – Min Stack

📌 Problem Statement

Design a stack that supports the following operations **in constant time O(1)**:

- `push(val)` → Push element `val` onto the stack
- `pop()` → Remove the top element
- `top()` → Get the top element
- `getMin()` → Retrieve the **minimum element** in the stack

🧠 Key Observation

A normal stack can easily:

- push
- pop
- return top

But finding the **minimum element** requires scanning the entire stack, which is `O(n)` ❌.

👉 To achieve **O(1)** for `getMin()`, we need **extra help**.

💡 Core Idea (Two Stack Technique)

Use **two stacks**:

1️⃣ Main Stack

- Stores all the values normally.

2️⃣ Min Stack

- Stores the **minimum value so far** at each level of the stack.

Both stacks:

- Always grow together

- Always shrink together

🔁 How Operations Work

🔹 Push Operation

When pushing a value `x`:

- Push `x` into the main stack
- Push `min(x, current_min)` into the min stack

This way, the min stack always knows the minimum **up to that point**.

🔹 Pop Operation

- Pop the top element from **both stacks**

This keeps them perfectly synchronized.

🔹 Top Operation

- Return the top of the main stack

🔹 Get Minimum Operation

- Return the top of the min stack
- This is always the current minimum

📊 Example Walkthrough

Operations:

```
push(5)
push(3)
push(7)
```

Stacks become:

```
Main Stack: [5, 3, 7]
Min Stack : [5, 3, 3]
```

Now:

- `top()` → 7
- `getMin()` → 3 ✅

✅ Python Code

```
class MinStack:

    def __init__(self):
        self.stack = []
        self.min_stack = []

    def push(self, val: int) -> None:
        self.stack.append(val)
        if not self.min_stack:
            self.min_stack.append(val)
        else:
            self.min_stack.append(min(val, self.min_stack[-1]))
```

```
    def pop(self) -> None:
        self.stack.pop()
        self.min_stack.pop()

    def top(self) -> int:
        return self.stack[-1]

    def getMin(self) -> int:
        return self.min_stack[-1]
```

⏱️ Time & Space Complexity

Operation	Time
push	O(1)
pop	O(1)
top	O(1)
getMin	O(1)

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

🧠 DSA Pattern Learned

- Stack design problems
- Auxiliary stack usage
- Maintaining historical minimum
- Synchronization between data structures

🧩 LeetCode 496 – Next Greater Element I

📌 Problem Statement

You are given two arrays `nums1` and `nums2`, where `nums1` is a subset of `nums2`.

For each element in `nums1`, find the **next greater element** to its right in `nums2`.

If no such element exists, return `-1` for that element.

🧠 Key Observations

1. Brute force would scan to the right for every element → **$O(n^2)$**.
2. Each element in `nums2` needs to know its next greater element **only once**.
3. Elements may need to **wait** until a greater element appears.
4. Stack is ideal for tracking elements waiting for a greater value.

💡 Approach (Monotonic Decreasing Stack)

1. Traverse `nums2` from left to right.
2. Use a stack to store elements that are **waiting** for their next greater element.
3. While the current element is greater than the stack top:
    - Pop the stack
    - Mark the current element as the next greater of the popped value
4. Push the current element into the stack.

5. Remaining elements in stack have no greater element → assign `1`.
6. Use a dictionary for quick lookup when processing `nums1`.

✅ Python Code

```
class Solution:
    def nextGreaterElement(self, nums1, nums2):
        stack = []
        next_greater = {}

        for num in nums2:
            while stack and num > stack[-1]:
                prev = stack.pop()
                next_greater[prev] = num
            stack.append(num)

        while stack:
            next_greater[stack.pop()] = -1

        return [next_greater[num] for num in nums1]
```

📃 Line-by-Line Explanation

```
stack = []
next_greater = {}
```

Stack stores elements waiting for a greater element.

Dictionary stores final mappings.

```
for num in nums2:
```

Traverse the main array once.

```
while stack and num > stack[-1]:
```

If current number is greater, it resolves waiting elements.

```
next_greater[prev] = num
```

Stores the next greater element.

```
return [next_greater[num] for num in nums1]
```

Direct lookup for `nums1`.

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Monotonic decreasing stack
- Next Greater Element pattern

- Stack + HashMap combination
- Preprocessing for efficient queries

🧩 LeetCode 739 – Daily Temperatures

📌 Problem Statement

You are given an array `temperatures`, where each element represents the temperature of a day.

For each day, return how many days you have to wait until a **warmer temperature** occurs.

If no such day exists, return `0`.

🧠 Key Observations

1. For each day, we can only look **to the right**.
2. Brute force scanning causes repeated work → **O(n²)**.
3. Some days must **wait** for a warmer day.
4. Once a warmer day appears, multiple previous days may get resolved.

💡 Approach (Monotonic Decreasing Stack)

1. Use a result array initialized with `0`.
2. Use a stack to store **indices** of days waiting for a warmer temperature.
3. Traverse temperatures from left to right.
4. While current temperature is greater than temperature at stack top:
    - Pop the index
    - Calculate days waited using index difference
5. Push the current index into the stack.
6. Unresolved days automatically remain `0`.

✅ Python Code

```
class Solution:
    def dailyTemperatures(self, temperatures):
        n = len(temperatures)
        res = [0] * n
        stack = []

        for i in range(n):
            while stack and temperatures[i] > temperatures[stack[-1]]:
                prev_index = stack.pop()
                res[prev_index] = i - prev_index
            stack.append(i)

        return res
```

🧾 Line-by-Line Explanation

```
res = [0] * n
```

Initializes result array with default `0`.

```
stack = []
```

Stores indices of days waiting for warmer temperatures.

```
while stack and temperatures[i] > temperatures[stack[-1]]:
```

Checks if current day resolves waiting days.

```
res[prev_index] = i - prev_index
```

Computes how many days were waited.

```
stack.append(i)
```

Marks current day as waiting.

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Monotonic stack using indices
- Next Greater Element concept
- Index-based distance calculation
- Efficient one-pass traversal

🧩 LeetCode 844 – Backspace String Compare

📌 Problem Statement

Given two strings `s` and `t`, where `#` represents a backspace,

determine whether the two strings are equal after processing all backspaces.

🧠 Key Observations

1. Backspace removes the **most recent character**.
2. This behavior matches **LIFO (Last In, First Out)**.
3. Stack is ideal for simulating text editor behavior.
4. Both strings can be processed independently and compared.

💡 Approach (Stack Simulation)

1. Traverse the string character by character.
2. If character is `#`:
    - Remove last character from stack (if exists).
3. Else:
    - Push character into stack.
4. After processing both strings, compare the final stacks.

✅ Python Code

```
class Solution:
    def backspaceCompare(self, s: str, t: str) -> bool:
        def process(st):
            stack = []
            for ch in st:
                if ch == '#':
                    if stack:
                        stack.pop()
                else:
                    stack.append(ch)
            return stack

        return process(s) == process(t)
```

🧾 Line-by-Line Explanation

```
stack = []
```

Stores characters after applying backspaces.

```
if ch == '#':
```

Represents a backspace operation.

```
stack.pop()
```

Removes the most recent character.

```
return process(s) == process(t)
```

Both processed strings must match.

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Stack for undo / backspace simulation
- Independent processing + comparison
- LIFO behavior in strings

🧩 LeetCode 206 – Reverse Linked List

📌 Problem Statement

Given the `head` of a singly linked list, reverse the list and return the new head.

You must reverse the **links (next pointers)**, not the values.

🧠 Key Observations

1. Reversing a linked list means **reversing pointers**, not data.
2. If we change `curr.next` without saving it, we lose the rest of the list.
3. We need three pointers to do this safely:
    - `prev`
    - `curr`
    - `next`
4. The original head becomes the **tail** after reversal.

💡 Approach (Iterative Pointer Reversal)

1. Initialize:
    - `prev = None`
    - `curr = head`
2. Traverse the list while `curr` is not `None`.
3. In each iteration:
    - Store the next node
    - Reverse the current node's pointer
    - Move `prev` and `curr` forward
4. After the loop ends, `prev` will be the new head.

✅ Python Code

```
class Solution:
    def reverseList(self, head):
        prev = None
        curr = head

        while curr:
            next_node = curr.next
            curr.next = prev
            prev = curr
            curr = next_node

        return prev
```

🧾 Line-by-Line Explanation

```
prev = None
```

Initializes `prev` to `None` because the new tail must point to `None`.

```
curr = head
```

Starts traversal from the head of the list.

```
next_node = curr.next
```

Stores the next node so we don't lose the remaining list.

```
curr.next = prev
```

Reverses the pointer direction.

```
prev = curr
curr = next_node
```

Moves both pointers forward.

```
return prev
```

Returns the new head of the reversed list.

⏱️ Time & Space Complexity

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

  (each node is visited once)

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

  (reversal is done in-place)

🧠 DSA Pattern Learned

- Pointer manipulation in linked lists
- Three-pointer technique (`prev`, `curr`, `next`)
- Safe traversal while modifying links
- Foundation for many linked list problems

🧩 LeetCode 876 – Middle of the Linked List

📌 Problem Statement

Given the `head` of a singly linked list, return the **middle node** of the linked list.

If there are **two middle nodes**, return the **second middle** node.

🧠 Key Observations

1. We cannot access elements by index in a linked list.
2. Counting nodes requires either:
   - two passes, or
   - extra storage.
3. The problem can be solved in **one pass** using pointers.
4. Using two pointers moving at different speeds helps locate the middle efficiently.

💡 Approach (Fast & Slow Pointer)

1. Initialize two pointers:
   - `slow` → moves one step at a time
   - `fast` → moves two steps at a time
2. Traverse the list while:
   - `fast` is not `None`

- and `fast.next` is not `None`

3. When `fast` reaches the end of the list:
   - `slow` will be at the middle
4. This method naturally returns the **second middle** for even-length lists.

✅ Python Code

```
class Solution:
    def middleNode(self, head):
        slow = head
        fast = head

        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        return slow
```

🧾 Line-by-Line Explanation

```
slow = head
fast = head
```

Both pointers start from the head of the list.

```
while fast and fast.next:
```

Ensures:

- safe movement of `fast.next.next`
- correct stopping point for even-length lists

```
slow = slow.next
```

Moves one step forward.

```
fast = fast.next.next
```

Moves two steps forward.

```
return slow
```

When the loop ends:

- `fast` has reached the end
- `slow` is at the middle node

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Fast & Slow pointer technique
- One-pass linked list traversal
- Middle node detection
- Foundation for cycle detection and palindrome checks

🔑 One-Line Takeaway

> When one pointer moves twice as fast as another, the slower one reaches the middle when the faster one reaches the end.

🧩 LeetCode 141 – Linked List Cycle

📌 Problem Statement

Given the `head` of a singly linked list, determine whether the linked list contains a **cycle**.
A cycle exists if a node's `next` pointer points to a **previous node**, causing traversal to loop indefinitely.

Return `True` if a cycle exists, otherwise return `False`.

🧠 Key Observations

1. In a normal linked list, traversal eventually reaches `None`.
2. In a cyclic linked list, traversal **never reaches** `None`.
3. A brute-force solution can store visited nodes, but uses extra space.
4. The problem can be solved efficiently using **two pointers** moving at different speeds.

💡 Approach (Fast & Slow Pointer / Floyd's Algorithm)

1. Use two pointers:
   - `slow` → moves one step at a time
   - `fast` → moves two steps at a time
2. Traverse the list while both `fast` and `fast.next` exist.
3. If the list has a cycle:
   - `fast` will eventually **catch up to** `slow`.
4. If `fast` reaches `None`, the list has **no cycle**.

✅ Python Code

```
class Solution:
    def hasCycle(self, head):
        slow = head
        fast = head

        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

            if slow == fast:
                return True
```

```
        return False
```

🧾 Line-by-Line Explanation

```
slow = head
fast = head
```

Initializes both pointers at the start of the list.

```
while fast and fast.next:
```

Ensures:

- safe access to `fast.next.next`
- correct termination if no cycle exists

```
slow = slow.next
```

Moves slow pointer by one step.

```
fast = fast.next.next
```

Moves fast pointer by two steps.

```
if slow == fast:
    return True
```

If both pointers meet, a cycle is guaranteed.

```
return False
```

If traversal ends, no cycle exists.

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Fast & Slow pointer technique
- Cycle detection without extra space
- Pointer safety using `fast` and `fast.next`
- Difference between `while temp` and `while temp.next`

🔑 One-Line Takeaway

> If two pointers move at different speeds in a linked list and meet, the list contains a cycle.

🧩 LeetCode 234 – Palindrome Linked List

📌 Problem Statement

Given the `head` of a singly linked list, determine whether the linked list is a **palindrome**.

A palindrome means the sequence of values reads the same forward and backward.

🧠 Key Observations

1. In arrays, palindrome can be checked using two pointers from both ends.
2. Linked lists cannot be traversed backward.
3. We must simulate backward traversal without using extra space.
4. This can be achieved by reversing **only the second half** of the list.
5. The problem combines:
    - finding the middle of a linked list
    - reversing a linked list

💡 Approach (Fast & Slow Pointer + Reversal)

1. Use **fast and slow pointers** to find the middle of the linked list.
2. If the list length is odd, the middle element does not affect palindrome checking.
3. Reverse the **second half** of the linked list.
4. Compare:
    - first half
    - reversed second half
5. If all corresponding values match, the list is a palindrome.

✅ Python Code

```
class Solution:
    def isPalindrome(self, head):
        if not head or not head.next:
            return True

        # Step 1: Find the middle
        slow = head
        fast = head

        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        # Step 2: Reverse second half
        prev = None
        curr = slow
        while curr:
            next_node = curr.next
            curr.next = prev
            prev = curr
            curr = next_node

        # Step 3: Compare both halves
        first = head
        second = prev

        while second:
```

```
            if first.val != second.val:
                return False
            first = first.next
            second = second.next

        return True
```

🧾 Line-by-Line Explanation

```
if not head or not head.next:
    return True
```

Handles empty and single-node lists (always palindromes).

```
slow = head
fast = head
```

Initializes two pointers to find the middle.

```
while fast and fast.next:
```

Ensures safe traversal and correct middle detection.

```
prev = None
curr = slow
```

Prepares to reverse the second half of the list.

```
next_node = curr.next
curr.next = prev
```

Stores the next node and reverses the pointer.

```
first = head
second = prev
```

Sets pointers to compare both halves.

```
while second:
```

Compares only till the end of the second half.

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Fast & Slow pointer technique
- Linked list reversal
- In-place palindrome check
- Combining multiple linked list patterns

🔑 One-Line Takeaway

> To check palindrome in a linked list, reverse the second half and compare it with the first half.

🧩 LeetCode 21 – Merge Two Sorted Lists

📌 Problem Statement

You are given the heads of two **sorted** singly linked lists `l1` and `l2`.

Merge the two lists into a single sorted linked list and return its head.

🧠 Key Observations

1. Both linked lists are already sorted.
2. We only need to compare the **current nodes** of both lists.
3. No new nodes are created – existing nodes are reused.
4. A dummy node helps avoid edge cases when building the result list.

💡 Approach (Two Pointers + Dummy Node)

1. Create a dummy node to act as the starting point of the merged list.
2. Use a `current` pointer to build the merged list.
3. Compare the current nodes of `l1` and `l2`:
    - Attach the smaller node to `current`.
    - Move the pointer of the list from which the node was taken.
4. Move the `current` pointer forward.
5. When one list becomes empty, attach the remaining part of the other list.
6. Return `dummy.next` as the head of the merged list.

✅ Python Code

```
class Solution:
    def mergeTwoLists(self, l1, l2):
        dummy = ListNode(0)
        current = dummy

        while l1 and l2:
            if l1.val <= l2.val:
                current.next = l1
                l1 = l1.next
            else:
                current.next = l2
                l2 = l2.next

            current = current.next

        if l1:
            current.next = l1
        else:
            current.next = l2
```

```
        return dummy.next
```

🧾 Line-by-Line Explanation

```
dummy = ListNode(0)
current = dummy
```

Creates a dummy node to simplify edge cases.

`current` is used to build the merged list.

```
while l1 and l2:
```

Runs while both lists still have nodes to compare.

```
if l1.val <= l2.val:
```

Chooses the smaller value between the two current nodes.

```
current.next = l1
l1 = l1.next
```

Attaches the chosen node and moves the corresponding list pointer.

```
current = current.next
```

Moves the result pointer forward.

```
if l1:
    current.next = l1
else:
    current.next = l2
```

Attaches the remaining nodes once one list is exhausted.

```
return dummy.next
```

Returns the head of the merged linked list (skipping the dummy node).

⏱️ Time & Space Complexity

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

  (`n` and `m` are the lengths of the two lists)
- **Space Complexity:** `O(1)`

  (in-place merge, no extra data structures)

🧠 DSA Pattern Learned

- Two-pointer technique on linked lists
- Dummy node usage
- In-place list merging
- Foundation for merge sort and advanced LL problems

🔑 One-Line Takeaway

> Merging two sorted linked lists is just repeatedly picking the smaller head and moving forward.

🧩 LeetCode 203 – Remove Linked List Elements

📌 Problem Statement

Given the `head` of a singly linked list and an integer `val`, remove all nodes whose value equals `val`.

Return the head of the modified linked list.

🧠 Key Observations

1. The value to remove can appear **anywhere** (including the head).
2. Deleting nodes in a singly linked list requires access to the **previous node**.
3. A **dummy node** simplifies handling cases where the head itself must be removed.
4. Only pointer reassignment is needed—no new nodes.

💡 Approach (Dummy Node + One Pass)

1. Create a dummy node pointing to `head`.
2. Use a pointer `curr` starting from the dummy node.
3. Traverse the list:
   - If `curr.next.val == val`, skip that node.
   - Otherwise, move `curr` forward.
4. Return `dummy.next`.

✅ Python Code

```
class Solution:
    def removeElements(self, head, val):
        dummy = ListNode(0)
        dummy.next = head

        curr = dummy
        while curr.next:
            if curr.next.val == val:
                curr.next = curr.next.next
            else:
                curr = curr.next

        return dummy.next
```

🧾 Line-by-Line Explanation

```
dummy = ListNode(0)
dummy.next = head
```

```

```

Creates a dummy node to handle deletion at the head.

```
curr = dummy
```

Starts traversal from the dummy node.

```
while curr.next:
```

Ensures there is a node ahead to check/remove.

```
if curr.next.val == val:
    curr.next = curr.next.next
```

Skips the node whose value matches `val`.

```
else:
    curr = curr.next
```

Moves forward when no deletion happens.

```
return dummy.next
```

Returns the updated head.

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Dummy node usage
- Safe node deletion in singly linked lists
- Single-pass traversal
- Handling head deletions cleanly

🧩 LeetCode 19 – Remove Nth Node From End of List

📌 Problem Statement

Given the head of a linked list, remove the **n-th node from the end** of the list and return the head.

🧠 Key Observations

1. We don’t know the length beforehand.
2. Traversing twice is possible but inefficient.
3. Using **two pointers** allows us to do it in **one pass**.

4. A **dummy node** simplifies edge cases (like removing head).

💡 Approach (Two Pointers)

1. Create a dummy node pointing to `head`.
2. Keep two pointers: `fast` and `slow` at dummy.
3. Move `fast` ahead by `n` steps.
4. Move both `fast` and `slow` until `fast` reaches the end.
5. `slow.next` is the node to remove.
6. Skip it using pointer reassignment.

✅ Python Code

```
class Solution:
    def removeNthFromEnd(self, head, n):
        dummy = ListNode(0)
        dummy.next = head

        fast = slow = dummy

        for _ in range(n):
            fast = fast.next

        while fast.next:
            fast = fast.next
            slow = slow.next

        slow.next = slow.next.next
        return dummy.next
```

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Two-pointer gap technique
- Dummy node for safe deletion
- One-pass linked list traversal

🧩 LeetCode 83 – Remove Duplicates from Sorted List

📌 Problem Statement

Given the head of a **sorted** linked list, remove all duplicates so each element appears only once.

🧠 Key Observations

1. List is already **sorted** → duplicates are adjacent.
2. Only need **one pointer**.
3. Compare current node with its next node.

💡 Approach (Single Pointer)

1. Start from head.
2. While next node exists:
    - If `curr.val == curr.next.val` → remove next node.
    - Else → move forward.
3. Return head.

✅ Python Code

```
class Solution:
    def deleteDuplicates(self, head):
        curr = head
        while curr and curr.next:
            if curr.val == curr.next.val:
                curr.next = curr.next.next
            else:
                curr = curr.next
        return head
```

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Sorted list advantage
- In-place node deletion
- Pointer comparison logic

🧩 LeetCode 160 – Intersection of Two Linked Lists

📌 Problem Statement

Given heads of two singly linked lists, return the node where they intersect, or `None` if they don’t.

🧠 Key Observations

1. Lists may have different lengths.
2. Intersection means **same memory address**, not value.
3. Two-pointer switching aligns traversal lengths automatically.

💡 Approach (Pointer Switching)

1. Use two pointers: `p1` and `p2`.
2. Traverse both lists.
3. When a pointer reaches end, switch it to the other list’s head.
4. If intersection exists → pointers meet.

5. If not → both become `None`.

✅ Python Code

```
class Solution(object):
    def getIntersectionNode(self, headA, headB):
        """
        :type head1, head1: ListNode
        :rtype: ListNode
        """
        first = headA
        second = headB
        while first != second :
            if first :
                first = first.next
            else :
                first = headB

            if second :
                second = second.next
            else :
                second = headA

        return first
```

⏱️ Time & Space Complexity

- **Time:** `O(n + m)`
- **Space:** `O(1)`

🧠 DSA Pattern Learned

- Pointer switching trick
- Length alignment without counting
- Reference-based comparison

🧩 LeetCode 150 – Evaluate Reverse Polish Notation

📌 Problem Statement

Given an array of tokens representing a **Reverse Polish Notation (RPN)** expression, evaluate the expression and return the result.

👉 In RPN:

- Operators come **after** operands
- Example: `"2 1 +"` → `2 + 1`

🧠 Key Observations

1. Numbers appear before operators.
2. Operators always act on the **last two operands**.
3. This is a **perfect use-case for a stack**.
4. Division must **truncate toward zero** (important edge case).

💡 Approach (Stack-Based Evaluation)

1. Initialize an empty stack.
2. Traverse tokens one by one:
    - If token is a number → push to stack.
    - If token is an operator:
        - Pop two elements from stack.
        - Apply the operation.
        - Push result back to stack.
3. Final stack element is the answer.

✅ Python Code

```
import math

class Solution:
    def evalRPN(self, tokens):
        stack = []

        for ch in tokens:
            if ch == '+':
                stack.append(stack.pop() + stack.pop())

            elif ch == '-':
                right = stack.pop()
                left = stack.pop()
                stack.append(left - right)

            elif ch == '*':
                stack.append(stack.pop() * stack.pop())

            elif ch == '/':
                right = stack.pop()
                left = stack.pop()
                stack.append(math.trunc(left / right))

            else:
                stack.append(int(ch))

        return stack.pop()
```

🧾 Line-by-Line Explanation

```
stack = []
```

Stack stores operands temporarily.

```
if ch == '+':
    stack.append(stack.pop() + stack.pop())
```

Addition is order-independent.

```
right = stack.pop()
left = stack.pop()
stack.append(left - right)
```

Subtraction **needs order** → left first, right second.

```
stack.append(math.trunc(left / right))
```

Division must **truncate toward zero**, not floor divide.

```
return stack.pop()
```

Final evaluated result remains on stack.

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Stack-based expression evaluation
- Reverse Polish Notation parsing
- Operator precedence via stack
- Handling tricky division edge cases

🧩 LeetCode 1544 – Make The String Great

📌 Problem Statement

Given a string `s` consisting of lowercase and uppercase English letters, the string is considered **bad** if it contains two **adjacent characters** such that:

- They are the **same letter**
- One is lowercase and the other is uppercase

You can repeatedly remove such adjacent bad pairs until the string becomes **good**.

Return the final string.

🧠 Key Observations

1. Only **adjacent characters** matter.
2. Removing a pair can create **new bad pairs**.
3. We must be able to:
    - Look at the **previous character**
    - Remove it if a bad pair forms
4. This behavior naturally fits a **stack**.

💡 Approach (Stack – Explicit Case Handling)

We traverse the string and use a stack to keep track of valid characters.

At each character:

- If the current character and the top of the stack form a **bad pair**, we remove the top.
- Otherwise, we push the current character onto the stack.

✅ Python Code (My Solution)

```
def makeGood(self, s):
    """
    :type s: str
    :rtype: str
    """
    stack = []

    for ch in s:
        if stack and ch.isupper() and stack[-1] == ch.lower():
            stack.pop()
        elif stack and stack[-1].isupper() and ch == stack[-1].lower():
            stack.pop()
        else:
            stack.append(ch)

    if stack:
        return "".join(stack)
    else:
        return ""
```

🗒️ Line-by-Line Explanation

```
stack = []
```

Stack stores characters processed so far.

```
for ch in s:
```

Iterate through each character in the string.

🔹 Case 1: Current char is uppercase, stack top is same letter lowercase

```
if stack and ch.isupper() and stack[-1] == ch.lower():
    stack.pop()
```

Example: `"aA"`

→ bad pair → remove both

🔹 Case 2: Stack top is uppercase, current char is same letter lowercase

```
elif stack and stack[-1].isupper() and ch == stack[-1].lower():
    stack.pop()
```

Example: `"Aa"`

→ bad pair → remove both

🔹 Otherwise: Safe character

```
else:
    stack.append(ch)
```

No bad pair → keep the character.

🔹 Final result

```
if stack:
    return "".join(stack)
else:
    return ""
```

Join remaining characters to form the good string.

⏱️ Time & Space Complexity

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

🧠 DSA Pattern Learned

- Stack for adjacent cancellation
- Explicit condition handling
- Case comparison using `.isupper()` and `.lower()`
- Undoing operations using stack pop

🧩 LeetCode 2 — Add Two Numbers

📌 Problem Statement

You are given **two non-empty linked lists** representing two non-negative integers.

- The digits are stored in **reverse order**
- Each node contains a **single digit**
- Add the two numbers and return the **sum as a linked list**

🔢 Example

```
Input:
l1 = 2 → 4 → 3   (342)
l2 = 5 → 6 → 4   (465)

Output:
7 → 0 → 8        (807)
```

🧠 Key Observations

1. Digits are stored in **reverse order**
2. We add digits **node by node**
3. Carry must be handled (just like normal addition)
4. Lists can be of **different lengths**
5. Even after both lists end, **carry may still exist**

💡 Approach (Linked List + Carry)

1. Use two pointers:
    - `p1` → first linked list
    - `p2` → second linked list
2. Maintain a `carry` variable
3. Use a **dummy node** to simplify result list creation
4. Loop while:
    - `p1` exists **OR**
    - `p2` exists **OR**
    - `carry` exists
5. At each step:
    - Add values + carry
    - Store `sum % 10` in new node
    - Update `carry = sum // 10`

✅ Python Code

```
class Solution:
    def addTwoNumbers(self, l1, l2):
        dummy = ListNode(0)
        curr = dummy
        carry = 0

        p1, p2 = l1, l2

        while p1 or p2 or carry:
            x = p1.val if p1 else 0
            y = p2.val if p2 else 0

            total = x + y + carry
            carry = total // 10
            digit = total % 10

            curr.next = ListNode(digit)
            curr = curr.next

            if p1:
                p1 = p1.next
            if p2:
                p2 = p2.next

        return dummy.next
```

🧾 Line-by-Line Explanation

```
dummy = ListNode(0)
curr = dummy
carry = 0
```

- `dummy` helps avoid edge cases for head
- `curr` builds the result list
- `carry` stores overflow

```
while p1 or p2 or carry:
```

Loop continues as long as **any digit or carry remains**

```
x = p1.val if p1 else 0
y = p2.val if p2 else 0
```

Safely extract values (0 if list ended)

```
total = x + y + carry
carry = total // 10
digit = total % 10
```

- `digit` → goes into current node
- `carry` → forwarded to next addition

```
curr.next = ListNode(digit)
curr = curr.next
```

Create and move forward in result list

⏱️ Time & Space Complexity

- **Time Complexity:** `O(max(n, m))`
- **Space Complexity:** `O(max(n, m))`

🧠 DSA Patterns Learned

- Linked List traversal
- Dummy node technique
- Carry handling
- Multiple pointer coordination
- Simulating arithmetic with linked lists

🧩 LeetCode 92 – Reverse Linked List II

📌 Problem Statement

You are given the head of a singly linked list and two integers `left` and `right`.

- Positions are **1-indexed**
- Reverse the nodes from position `left` to position `right`
- The rest of the linked list must remain **unchanged**
- Return the modified linked list

🔢 Example

```
Input:
1 → 2 → 3 → 4 → 5
left = 2
right = 4

Output:
1 → 4 → 3 → 2 → 5
```

🧠 Key Observations

1. Only a **portion** of the linked list needs to be reversed
2. The head may change when `left = 1`
3. Pointer manipulation is more important than traversal
4. Using a **dummy node** simplifies edge cases
5. Reversal can be done **in-place** without extra memory

💡 Approach (Dummy Node + Head Insertion)

1. Create a **dummy node** pointing to `head`
2. Move a pointer `prev` to the node **just before** position `left`
3. Set `curr` to the first node of the sublist to be reversed
4. Repeat `(right - left)` times:
    - Take the node after `curr`
    - Remove it from its position
    - Insert it immediately after `prev`
5. Return `dummy.next` as the new head

✅ Python Code

```python
class Solution:
    def reverseBetween(self, head: ListNode, left: int, right: int) -> ListNode:
        if not head or left == right:
            return head

        dummy = ListNode(0)
        dummy.next = head
        prev = dummy

        for _ in range(left - 1):
            prev = prev.next

        curr = prev.next

        for _ in range(right - left):
            temp = curr.next
            curr.next = temp.next
            temp.next = prev.next
            prev.next = temp

        return dummy.next
```

🧾 Line-by-Line Explanation

```
dummy = ListNode(0)
dummy.next = head
```

- `dummy` handles edge cases when reversal starts at head

```
prev = dummy
for _ in range(left - 1):
    prev = prev.next
```

- Moves `prev` to the node **before** the reversal segment

```
curr = prev.next
```

- `curr` points to the first node that will be reversed

```
for _ in range(right - left):
```

- Loop runs exactly the number of nodes that need repositioning

---

```
temp = curr.next
curr.next = temp.next
```

- Detaches the node after `curr` from the list

```
temp.next = prev.next
prev.next = temp
```

- Inserts the detached node at the **front of the sublist**

```
return dummy.next
```

- Returns the updated head of the linked list

⏱️ Time & Space Complexity

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

🧠 DSA Patterns Learned

- Partial linked list reversal
- Dummy node technique
- Head insertion method
- In-place pointer manipulation
- Boundary-safe linked list operations
