Today we're going to talk about **Two Pointers** and **Sliding Window**.

Before we dive into actual problems, let's spend a few minutes on the **foundations**. I'll walk you through the core ideas behind each technique and show you the **common patterns** you'll see again and again.

That way, when you face a new problem, you'll be able to quickly recognize which category it belongs to and immediately pick the right approach.

For each technique, here's the plan: first we'll clarify the key concept in plain English, then we'll work through one example problem together to see how the idea plays out in code, and finally I'll share a short list of classic questions so you can strengthen what we cover today on your own.

Two Pointers

Let's start with **Two Pointers**.

What is it?

The idea of "two pointers" is exactly what it sounds like: we place **two pointers** on one sequence—or sometimes on two different sequences— and **move them in a smart way** to cut down unnecessary work and lower time complexity.

We can think of it like this: instead of creating extra data structures, we simply rely on the **relative movement** of two pointers to scan more efficiently.

Many problems that look like they need a double nested loop— `O(n^2)` —can actually be solved in **`O(n)`** if we move the two pointers smartly.

That's why this is one of the most fundamental and useful techniques we'll see on LeetCode.

Common Patterns

When we face a new problem, it helps to first decide **which kind of two-pointer pattern** it belongs to. Here are the classic categories.

Opposite Ends

One pointer starts from the **left**, the other from the **right**, and they move toward each other. At each step, you decide whether to move the left pointer or the right pointer based on a comparison with the target.

The key requirement is that the sequence is **sorted** or at least has some monotonic property so you can decide which side to move safely.

```python
# example: find two numbers in a sorted array whose sum equals target
nums = [2, 7, 11, 15]
target = 9

l, r = 0, len(nums) - 1      # <-- l starts from left, r starts from right
while l < r:
    s = nums[l] + nums[r]
    if s == target:
        print(l, r)          # found the answer
        break
    elif s < target:
        l += 1               # <-- sum too small, move left pointer right
    else:
        r -= 1               # <-- sum too big, move right pointer left

'''
数组:  [ 2   7   11   15 ]
        ^             ^
        l             r
'''
```

Fast–Slow

Both pointers move in the **same direction**, but at **different speeds**.This is useful for comparing positions, detecting cycles, or finding a midpoint.

The classic method to fina the mid point is that fast pointer jumps two steps, slow pointer moves one step.

Typical use cases:

- find the middle node of a linked list,
- detect if a linked list has a cycle (Floyd's cycle detection),
- sliding window problems.

```
# Example: find the middle of a linked list
slow = fast = head
while fast and fast.next:
    slow = slow.next          # slow moves one step
    fast = fast.next.next     # fast moves two steps
# slow will stop at the middle

'''
   head → [1] → [2] → [3] → [4] → [5]
     ^slow
     ^fast
'''
```

Two Sequences

Here we have **two sorted sequences**. We place one pointer in each sequence and scan them together — just like the merge step in merge sort. Each pointer moves independently based on the comparison result.

Typical tasks:

- find the intersection of two sorted arrays,
- merge intervals.

```
# Example: intersection of two sorted arrays
a = [1,2,4,6]
b = [2,4,5]
i = j = 0
while i < len(a) and j < len(b):
    if a[i] == b[j]:
        print(a[i])           # output common element
        i += 1; j += 1
    elif a[i] < b[j]:
        i += 1                # move the smaller side
    else:
        j += 1

'''
   a: [1] [2] [4] [6]
       ^
       i
   b: [2] [4] [5]
       ^
       j
'''
```

Partition

We rearrange elements **in-place** into different regions by using two or three pointers and swapping elements.

Typical tasks:

- the partition step in quicksort,
- the famous Dutch National Flag problem.

```
# Example: Dutch National Flag (sort 0s, 1s, and 2s)
nums = [2,0,2,1,1,0]
low, mid, high = 0, 0, len(nums)-1
while mid <= high:
    if nums[mid] == 0:
        nums[low], nums[mid] = nums[mid], nums[low]
        low += 1; mid += 1   # move 0 to the left
    elif nums[mid] == 1:
        mid += 1             # 1 stays in the middle
    else:
        nums[mid], nums[high] = nums[high], nums[mid]
        high -= 1            # move 2 to the right

'''
   nums: [2] [0] [2] [1] [1] [0]
          ^
          low
          ^
          mid
                        ^
                        high
'''
```

Compress In-Place

Finally, we often need to **filter or compress data in place**. The `fast` pointer scans every element, while the `slow` pointer only writes the "valid" elements. This achieves `O(1)` extra space.

Typical tasks: remove duplicates from a sorted array,

```
# Example: remove duplicates from a sorted array
nums = [1,1,2,2,3]
slow = 0
for fast in range(len(nums)):
    if fast == 0 or nums[fast] != nums[fast-1]:
```

```
        nums[slow] = nums[fast]
        slow += 1             # slow writes valid elements
print(nums[:slow])            # [1,2,3]

'''
   nums: [1] [1] [2] [2] [3]
          ^slow
          ^fast
'''
```

Whenever we meet a new array or string problem, ask ourselves: **"Can I reframe this as a two-pointer problem?"**

If the answer is yes, you can often cut the complexity in half— sometimes from quadratic to linear—by choosing the right pattern above.

Example

Let's look at a classic two-pointer problem.

[167. Two Sum II - Input Array Is Sorted]

Given a **1-indexed** array of integers `numbers` that is already sorted in non-decreasing order, find two numbers such that they add up to a specific `target` number. Let these two numbers be `numbers[index1]` and `numbers[index2]` where `1 <= index1 < index2 <= numbers.length`.

Return *the indices of the two numbers,* `index1` *and* `index2`*,* ***added by one*** *as an integer array* `[index1, index2]` *of length 2.*

The tests are generated such that there is **exactly one solution**. You **may not** use the same element twice.

Your solution must use only constant extra space.

> **Example 1:**
>
> ```
> Input: numbers = [2,7,11,15], target = 9
> Output: [1,2]
> Explanation: The sum of 2 and 7 is 9. Therefore, index1 = 1, index2 = 2. We return [1,
> 2].
> ```

Key Information

- Input: 已排序（非递减）的1-indexed数组 `numbers`。

- Output: 找两个数使得和为 `target`，返回它们的下标（从 1 开始）。
- Rules: 只有唯一解，且同一元素不能重复使用。O(1) 额外空间。

We're given a **1-indexed** array of integers `numbers`, already sorted in **non-decreasing** order. We need to find **two numbers** whose sum equals a given `target`. Return their **indices**—but remember, indices are **1-based** in the final answer.

The problem guarantees:

- exactly **one valid pair**,
- and we cannot reuse the same element twice,
- and we should use only **constant extra space**.

Brute Force

If we ignore the "sorted" hint, the most straightforward idea is: use **two nested loops**, check every pair, and return the first one that sums to `target`.

Here's a quick sketch:

```
def twoSum_bruteforce(numbers, target):
    n = len(numbers)
    for i in range(n):
        for j in range(i + 1, n):
            s = numbers[i] + numbers[j]
            if s == target:
                return [i + 1, j + 1]   # 1-indexed
            if s > target:
                break
    return [-1, -1]
```

This works, but it's still `O(n^2)` time in the worst case.

Smarter Way

The key observation: **the array is sorted**.

That means if the current sum is too small, we can safely move the left pointer to the right to get a bigger sum. If the sum is too large, we move the right pointer left to get a smaller sum.

So:

1. Start with `l` at the **beginning** and `r` at the **end**.
2. Calculate `numbers[l] + numbers[r]`.
    - If it equals `target`, we're done.

- If it's **less** than `target`, move `l` right—need a bigger sum.
- If it's **greater**, move `r` left—need a smaller sum.

3. We keep shrinking the search space until the two pointers meet.

```
def twoSum(numbers, target):
    l, r = 0, len(numbers) - 1
    while l < r:
        s = numbers[l] + numbers[r]
        if s == target:
            return [l + 1, r + 1]   # output is 1-indexed
        elif s < target:
            l += 1                  # sum too small → move left pointer right
        else:
            r -= 1                  # sum too big  → move right pointer left
    return [-1, -1]  # won't happen under the problem's guarantee

'''
numbers: [ 2   7   11   15 ]
           ^            ^
           l            r
'''
```

Complexity

- **Time:** `O(n)` — each pointer only moves forward or backward once, never backtracks.
- **Space:** `O(1)` — just two indices and a few temporary variables.

This is the classic "opposite ends" two-pointer pattern. Whenever you see "find two numbers in a sorted array that add up to a target," this two-pointer approach should immediately come to mind.

Practices

Here's a set of recommand LeetCode problems

- Opposite Ends

  > 125. Valid Palindrome
  >
  > 11. Container With Most Water
  >
  > 42. Trapping Rain Water

- Fast–Slow

  > 141. Linked List Cycle
  >
  > 876. Middle of the Linked List
  >
  > 19. Remove Nth Node From End of List

- Two Sequences

  > 88. Merge Sorted Array
  >
  > 350. Intersection of Two Arrays II
  >
  > 392. Is Subsequence

- Partition

  > 75. Sort Colors
  >
  > 283. Move Zeroes

- Dedup/Compress In-Place

  > 26. Remove Duplicates from Sorted Array
  >
  > 443. String Compression

Sliding Window

Now let's move on to Sliding Window. This concept is actually very intuitive.

What is this?

A sliding window is simply a **continuous range** inside an array or string. We use **two pointers** to mark the left and right boundaries.

Here's the core idea: **expand on the right, shrink on the left**.

- The **right pointer** keeps "absorbing" new elements.
- Whenever the window stops meeting the required condition, we move the **left pointer** forward to "kick out" the extra elements.

From an algorithmic point of view, a sliding window is really just a fast–slow two-pointer pattern.

The key is that the window must maintain a monotonic **maintainable property**.

The key is that the subarray or substring must satisfy some **maintainable property**—
for example a running sum, character frequencies, or the number of distinct elements. Our job is to decide when to expand the right boundary and when to shrink the left boundary.

This technique is perfect when you need to find a **continuous segment** that meets certain criteria.

It turns an `O(n²)` brute force search into a clean **`O(n)`** solution.

Types

When you face a sliding-window question, first classify it. Here are the main types.

Fixed-size Window

The window size is **fixed**. Left and right pointers always stay `k` steps apart. Every time the right pointer moves into a new element, the left pointer moves forward by one as well.

You can update the window's sum or other metric in **`O(1)`**.

```
# Example: Maximum average of any subarray of length k
nums = [1,12,-5,-6,50,3]
k = 4

window_sum = sum(nums[:k])
max_sum = window_sum

for i in range(k, len(nums)):
    window_sum += nums[i] - nums[i - k]  # O(1) update
    max_sum = max(max_sum, window_sum)

print(max_sum / k)   # 12.75

'''
s:  a  b  c  a  b  c  b  b
    ^l
    ^r   …the right pointer expands, and the left pointer contracts only when there is a duplicate
'''
```

Variable-size Window

The window's length is **not fixed**. We grow or shrink it dynamically depending on the condition

The window's length is **not fixed**. We grow or shrink it dynamically depending on the condition.

Single Condition

Maintain just one metric—like a running sum, product, or the count of distinct characters.

Longest / Shortest:

Longest Substring Without Repeating Characters (LeetCode 3).

Minimum Size Subarray Sum (LeetCode 209).

```
# Example: Longest Substring Without Repeating Characters
s = "abcabcbb"
l = 0

seen = set()
max_len = 0
for r, ch in enumerate(s):
    while ch in seen:
        seen.remove(s[l])
        l += 1
    seen.add(ch)
    max_len = max(max_len, r - l + 1)
print(max_len)  # 3 -> "abc"

'''
s:  a  b  c  a  b  c  b  b
    ^l
    ^r
'''
```

Counting

Problems that ask for the number of qualifying subarrays. For instance, *Subarray Product Less Than K* (LeetCode 713).

```
# Example: Subarray Product Less Than K
nums, k = [10,5,2,6], 100
l = 0

prod = 1
count = 0
for r, v in enumerate(nums):
    prod *= v  # each time the right pointer r moves right, update the product of the window
    while prod >= k:  # if product >= k, move the left pointer l to shrink the window
        prod //= nums[l]
        l += 1
```

```
        l += 1
    count += r - l + 1 # all subarrays ending at r and starting between l..r have product < k, number of such subarrays = r - l + 1
print(count)  # 8

'''
   nums: [10] [5] [2] [6]
          ^l
          ^r
   prod = 1
   count = 0
'''
```

Multiple Conditions

When you must track several counts at once. Classic example: *Minimum Window Substring* (LeetCode 76) —you must keep character frequencies for all target letters.

```
# Example: Minimum Window Substring
from collections import Counter

s, t = "ADOBECODEBANC", "ABC"

need = Counter(t)
window = Counter()

l, have, need_count, res = 0, 0, len(need), (0, float('inf'))
for r, ch in enumerate(s): # right pointer r expands and collects characters into window
    window[ch] += 1
    window[ch] += 1
    if ch in need and window[ch] == need[ch]:
        have += 1
    while have == need_count: # when all required chars are satisfied, shrink left pointer l
        # ensure the current window contains at least one A, B, and C, and update result if this window is smaller than previous best
        if r - l < res[1] - res[0]:
            res = (l, r)
        window[s[l]] -= 1
        if s[l] in need and window[s[l]] < need[s[l]]:
            have -= 1
        l += 1
print(s[res[0]:res[1]+1])  # "BANC"

'''
   s:  A  D  O  B  E  C  O  D  E  B  A  N  C
       ^l

       ^r
```

```
   target t: "ABC"
   need   = {A:1, B:1, C:1}
'''
```

Circular / Wrap-around

Sometimes the data itself is **circular**, like a ring buffer or a clock.

Conceptually it's still a sliding window, but we should either **double the array** or use **modulus operations** to handle wrap-around.

```
# Example: find the max sum of any 3 consecutive elements in a circular array
nums = [5, 1, 2, 6]

# Duplicate the array to simulate the circular structure,
# so we can apply a regular fixed-size sliding window.
arr = nums * 2

window_sum = sum(arr[:k])
max_sum = window_sum

# We only need to slide until start + len(nums) - 1,
# because the window must cover exactly len(nums) elements at most.
for i in range(k, len(nums) + k):
    window_sum += arr[i] - arr[i - k]
    max_sum = max(max_sum, window_sum)

print(max_sum)  # 9 -> either [6,5,1] or [1,2,6]

'''
环形数组: [5] [1] [2] [6] [5] [1] [2] [6]
          \_____/
          window of length 3 can wrap around the end to the start
'''
```

Example

[3. Longest Substring Without Repeating Characters]

Given a string `s`, find the length of the **longest substring** without duplicate characters.

> **Example 1:**

> Input: s = "abcabcbb"
> Output: 3
> Explanation: The answer is "abc", with the length of 3.

This is actually a typical sliding window problem. We need to maintain a continuous window and ensure that there are no duplicate characters within the window. If a duplicate occurs, we must "spit out" the characters to the left until the window is valid again.

思路

We maintain a sliding window with **no repeating characters**:

- expand the right pointer,
- whenever a duplicate appears, shrink from the left until the window is valid again,
- track the maximum length.

```
s = "abcabcbb"
l = 0

seen = set()  # set() keeps track of all characters currently in the window
max_len = 0

# right pointer r expands the window; left pointer l shrinks it when duplicates appear
for r, ch in enumerate(s):
    # if current character is already in the window,
    # move the left pointer until the duplicate is removed
    while ch in seen:
        seen.remove(s[l])
        l += 1
    seen.add(ch)
    max_len = max(max_len, r - l + 1)

print(max_len)  # 3

'''
s:  a  b  c  a  b  c  b  b
    ^l
    ^r
'''
```

Time complexity is `O(n)`, since each pointer moves at most once across the string.

Practices

Practices

Here's a set of recommend LeetCode problems

- Fixed-size Window

  > 643. Maximum Average Subarray I
  >
  > 1052. Grumpy Bookstore Owner
  >
  > 239. Sliding Window Maximum

- Variable-size Window

  > - Longest/Shortest：209. Minimum Size Subarray Sum
  > - Counting：992. Subarrays with K Different Integers
  > - Multiple Conditions：76. Minimum Window Substring

- Circular / Wrap-around

  > 918. Maximum Sum Circular Subarray
  >
  > 1191. K-Concatenation Maximum Sum

As we practice, don't just code the solution—identify the window type first. That habit will help you instantly spot which sliding-window pattern to use in real interviews.