**Two Pointer Technique** is a fundamental and highly efficient strategy for solving a wide range of problems, especially on platforms like LeetCode. It's used to traverse data structures (primarily arrays and linked lists) simultaneously from different positions to solve problems with optimal time and often space complexity.  
The key idea is to use two pointers (which can be indices or actual node references) to track positions and make decisions based on their values.

There are three main variants of the Two Pointer approach:

1. **Opposite Ends:** Pointers start at the beginning and end and move towards each other.
2. **Same Direction (Slow & Fast):** Both pointers start at the beginning, but one moves faster than the other.
3. **Multiple Arrays:** Pointers traverse two different arrays or lists simultaneously.

Let's break down each type with its most important LeetCode problems.

**1. Opposite Ends (Start & End Pointers)**

This is perfect for sorted arrays where we can make decisions based on the sum or values at the pointers.

**Classic Problems:**

**1. Two Sum II - Input Array Is Sorted (LeetCode #167)**

* **Problem:** Given a sorted array, find two numbers that add up to a specific target.
* **Approach:**
  + left pointer at index 0, right pointer at last index.
  + Calculate sum = numbers[left] + numbers[right].
  + If sum == target, return [left+1, right+1].
  + If sum < target, we need a larger sum -> move left forward.
  + If sum > target, we need a smaller sum -> move right backward.
* **Why it works:** The array is sorted, so moving left increases the sum, moving right decreases it.

def twoSum(self, numbers: List[int], target: int) -> List[int]:  
 left, right = 0, len(numbers) - 1  
 while left < right:  
 current\_sum = numbers[left] + numbers[right]  
 if current\_sum == target:  
 return [left + 1, right + 1]  
 elif current\_sum < target:  
 left += 1  
 else:  
 right -= 1  
 return [-1, -1] # Problem states there is exactly one solution

**2. Container With Most Water (LeetCode #11)**

* **Problem:** Find two lines that together with the x-axis form a container that holds the most water.
* **Approach:**
  + left at start, right at end.
  + Calculate area: min(height[left], height[right]) \* (right - left).
  + Track the max area.
  + Move the pointer pointing to the *shorter* wall inward. The rationale is that the area is limited by the shorter wall, so moving the taller wall cannot possibly lead to a larger area.
* **Why it works:** Greedily moving the shorter wall gives us a chance to find a taller one, maximizing our potential area.

def maxArea(self, height: List[int]) -> int:  
 left, right = 0, len(height) - 1  
 max\_water = 0  
 while left < right:  
 # Calculate current area  
 h = min(height[left], height[right])  
 w = right - left  
 max\_water = max(max\_water, h \* w)  
   
 # Move the pointer with the smaller height  
 if height[left] < height[right]:  
 left += 1  
 else:  
 right -= 1  
 return max\_water

**3. 3Sum (LeetCode #15)**

* **Problem:** Find all unique triplets [nums[i], nums[j], nums[k]] such that they add up to 0.
* **Approach:** This combines sorting with the two-pointer technique.
  1. Sort the array.
  2. Iterate i from 0 to n-3.
  3. For each i, set left = i+1 and right = n-1.
  4. Now, solve nums[left] + nums[right] == -nums[i] (i.e., Two Sum II) for the subarray.
  5. Skip duplicates for i, left, and right to ensure unique triplets.

def threeSum(self, nums: List[int]) -> List[List[int]]:  
 nums.sort()  
 n = len(nums)  
 result = []  
   
 for i in range(n - 2):  
 # Skip duplicate for the first number  
 if i > 0 and nums[i] == nums[i-1]:  
 continue  
 left, right = i + 1, n - 1  
 while left < right:  
 total = nums[i] + nums[left] + nums[right]  
 if total < 0:  
 left += 1  
 elif total > 0:  
 right -= 1  
 else:  
 result.append([nums[i], nums[left], nums[right]])  
 # Skip duplicates for left and right pointers  
 while left < right and nums[left] == nums[left+1]:  
 left += 1  
 while left < right and nums[right] == nums[right-1]:  
 right -= 1  
 left += 1  
 right -= 1  
 return result

**2. Same Direction (Slow & Fast Pointers)**

Also known as the "Floyd's Cycle Finding Algorithm" or "Hare and Tortoise" technique. Excellent for linked lists and arrays where you need to find cycles or positions relative to the end.

**Classic Problems:**

**1. Remove Duplicates from Sorted Array (LeetCode #26)**

* **Problem:** Remove duplicates in-place such that each element appears only once. Return the new length.
* **Approach:**
  + slow pointer (or insertIndex) starts at 1. It tracks the position where the next unique element should be placed.
  + fast pointer starts at 1. It traverses the array to find the next unique number.
  + If nums[fast] != nums[fast - 1], it's unique. Copy it to nums[slow] and increment slow.
* **Why it works:** The slow pointer builds the new unique array from the left, while the fast pointer explores ahead.

def removeDuplicates(self, nums: List[int]) -> int:  
 if not nums:  
 return 0  
 insert\_index = 1 # slow pointer  
 for i in range(1, len(nums)): # i is the fast pointer  
 if nums[i] != nums[i - 1]:  
 nums[insert\_index] = nums[i]  
 insert\_index += 1  
 return insert\_index

**2. Linked List Cycle (LeetCode #141)**

* **Problem:** Given a linked list, determine if it has a cycle.
* **Approach:**
  + slow pointer moves one step at a time.
  + fast pointer moves two steps at a time.
  + If there is a cycle, the fast runner will eventually lap the slow runner (fast == slow).
  + If fast reaches null, there is no cycle.
* **Why it works:** The relative speed between the pointers is 1. In a cycle, the faster pointer will always eventually catch up to the slower one.

def hasCycle(self, head: Optional[ListNode]) -> bool:  
 if not head:  
 return False  
 slow = head  
 fast = head.next  
   
 while slow != fast:  
 if fast is None or fast.next is None:  
 return False  
 slow = slow.next  
 fast = fast.next.next  
 return True

**3. Middle of the Linked List (LeetCode #876)**

* **Problem:** Return the middle node of a linked list. If two middles, return the second.
* **Approach:**
  + slow moves 1 step, fast moves 2 steps.
  + When fast reaches the end (fast or fast.next is null), the slow pointer will be at the middle.

def middleNode(self, head: Optional[ListNode]) -> Optional[ListNode]:  
 slow = fast = head  
 while fast and fast.next:  
 slow = slow.next  
 fast = fast.next.next  
 return slow

**3. Multiple Arrays/Lists**

Using a pointer for each sequence to merge or compare them.

**Classic Problem:**

**1. Merge Sorted Array (LeetCode #88)**

* **Problem:** Merge nums2 into nums1 in sorted order. nums1 has enough space at the end.
* **Approach:**
  + Use three pointers, starting from the *ends* of the arrays to avoid overwriting.
  + p1: starts at the last valid element of nums1.
  + p2: starts at the last element of nums2.
  + p: starts at the very last position of nums1 (the allocated space).
  + Compare elements from p1 and p2, put the larger one at p, and move the respective pointers backward.

def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:  
 p1 = m - 1  
 p2 = n - 1  
 p = m + n - 1  
   
 while p1 >= 0 and p2 >= 0:  
 if nums1[p1] > nums2[p2]:  
 nums1[p] = nums1[p1]  
 p1 -= 1  
 else:  
 nums1[p] = nums2[p2]  
 p2 -= 1  
 p -= 1  
   
 # If there are leftover elements in nums2, copy them over  
 # (No need to do this for nums1, as they are already in place)  
 while p2 >= 0:  
 nums1[p] = nums2[p2]  
 p2 -= 1  
 p -= 1

**How to Identify a Two Pointer Problem**

* The input is a **sorted array or linked list**.
* You need to find **a set of elements** that fulfill certain constraints (e.g., pairs, triplets summing to a value).
* Problems asking for **in-place** modifications (e.g., removing duplicates).
* Problems involving **cycles** in sequences.

Mastering these patterns will allow you to efficiently solve a significant portion of array and linked list problems on LeetCode.