Complete Two Pointers Pattern - Deep Study Guide

Table of Contents

Theoretical Foundation

What is Two Pointers Pattern?

Definition: A technique that uses two references (indices/pointers) to traverse data structures, typically reducing algorithmic complexity by eliminating redundant iterations.

Historical Context:

- Rooted in divide-and-conquer paradigms
- Related to Edsger Dijkstra's Dutch National Flag problem (1976)
- Foundation for many modern algorithm optimization techniques

Core Principle: Instead of nested iterations $O(n^2)$, use intelligent pointer movement based on problem constraints to achieve $O(n)$ or $O(n \log n)$ complexity.

Mathematical Analysis

Complexity Proof

Brute Force (Nested Loop):

```
For each element i (n iterations):
    For each element j (n iterations):
        Compare/Process ⟹ O(n²)
```

Two Pointers Optimization:

```
Each pointer moves at most n times total
Left pointer: 0 → n (n moves max)
Right pointer: n → 0 or 0 → n (n moves max)
Total operations: 2n → O(n)
```

Reduction Factor: $n^2/n = n$ (linear improvement)

Space Complexity Analysis

In-place operations: O(1) auxiliary space

- Only storing 2-3 pointer variables
- No additional data structures

Trade-off: Time optimization without space penalty

Five Core Patterns - Deep Dive

Pattern 1: Opposite Direction (Convergent Pointers)

Concept: Two pointers start at opposite ends and move toward each other.

Mathematical Invariant:

- Search space reduces by at least 1 each iteration
- Guaranteed termination when `left >= right`

When to Use:

1. **Sorted Data:** Leverage ordering property
2. **Symmetric Properties:** Palindromes, containers
3. **Binary Decisions:** Either move left OR right (never both randomly)

Decision Logic:

```
At each step:
  If current_sum < target → Increase sum (move left++)
  If current_sum > target → Decrease sum (move right--)
  If found → Process and continue
```

Key Insight: Eliminating impossible solutions without checking them.

Example Approach (Two Sum II):

Approach:
1. Start: left = 0, right = n-1
2. Calculate: sum = arr[left] + arr[right]
3. Decision tree:
   - sum == target → Found (return indices)
   - sum < target → Need larger value → left++
   - sum > target → Need smaller value → right--
4. Why it works: Array is sorted
   - If current sum is too small, arr[left] is too small forever
   - If current sum is too large, arr[right] is too large forever

Complexity:

- Time: O(n) - each pointer moves n times max
- Space: O(1) - only 2 variables

Variations:

- **Two Sum with sorted array**
- **3Sum:** Fix one element, apply two pointers on rest
- **4Sum:** Fix two elements, apply two pointers on rest
- **Container problems:** Choose better option at each step

Pattern 2: Same Direction (Fast & Slow Pointers)

Concept: Both pointers start at beginning, move at different speeds.

Two Sub-patterns:

A. Slow-as-Writer Pattern

Purpose: In-place array modification

Invariant:

- slow = position to write next valid element
- fast = scanning/reading position
- Elements before slow are processed and valid

Approach (Remove Duplicates):

```
Approach:
1. Initialize: slow = 0 (write position)
2. Fast scans array: for fast = 0 to n-1
3. When valid element found:
   - Write to arr[slow]
   - Increment slow++
4. Return slow (new length)

Why slow doesn't increment always:
- Fast finds valid elements
- Slow only moves when writing
- Gap between them = removed elements
```

Applications:

- Remove duplicates
- Remove specific elements
- Partition arrays
- Move zeros to end

B. Cycle Detection (Floyd's Algorithm)

Purpose: Find cycles in sequences

Mathematical Proof:

```
If cycle exists:
- Fast moves 2 steps, Slow moves 1 step
- Relative speed = 1 step per iteration
- Fast will lap slow inside cycle
- Meeting point proves cycle existence

Distance to meet:
- Let cycle length = C
- They meet after at most C iterations once slow enters cycle
```

Approach (Linked List Cycle):

```
Approach:
1. slow = fast = head
2. While fast and fast.next exist:
   - slow moves 1 step
   - fast moves 2 steps
   - if slow == fast → cycle detected
3. If fast reaches null → no cycle
```

Pattern 3: Sliding Window

Concept: Window defined by two pointers expands/contracts based on conditions.

Critical Distinction from Two Pointers:

Aspect	Two Pointers	Sliding Window
Movement	Independent decisions	Window-based movement
Purpose	Find pairs/partition	Find subarrays/substrings
Condition	Global comparison	Window state validation
Window size	N/A	Dynamic or Fixed

Two Types:

A. Fixed Size Window

Invariant: `right - left + 1 = k` (constant)

Approach (Maximum Sum Subarray Size K):

```
Approach:
1. Build initial window [0...k-1]
2. Slide window:
   - Remove element going out (left)
   - Add element coming in (right)
   - Update result
3. Window always maintains size k

Formula: new_sum = old_sum - arr[left] + arr[right+1]
```

B. Variable Size Window

Invariant: Window satisfies condition or is smallest/largest valid window

Approach (Longest Substring Without Repeating Chars):

```
Approach:
1. Expand: Move right, add to window
2. Check: Does window satisfy condition?
3. Contract: While invalid, move left, remove from window
4. Record: Track max/min window size
5. Repeat: Continue expanding

Key insight: Greedy expansion, necessary contraction
```

Window State Management:

- Use HashSet/HashMap for character/element tracking
- Update window properties incrementally
- Check validity after each modification

Complexity Analysis:

- Time: O(n) - each element added once, removed once (2n operations)
- Space: O(k) - k = unique elements in window

Pattern 4: Merge Pattern (Two Array Pointers)

Concept: Two pointers on different arrays, merge based on comparison.

Applications:

1. **Merge sorted arrays**
2. **Intersection of sorted arrays**
3. **Union of sorted arrays**

Approach (Merge Two Sorted Arrays):

```
Approach:
1. Pointers: i = 0 (arr1), j = 0 (arr2)
2. While both valid:
   - Compare arr1[i] vs arr2[j]
   - Take smaller, advance that pointer
3. Copy remaining elements from non-exhausted array

Key optimization: When merging in-place (arr1 has space):
- Start from END to avoid overwriting
- Work backwards: largest elements first
```

Special Case - Squares of Sorted Array:

```
Array can have negatives: [-4, -1, 0, 3, 10]
Squares: [16, 1, 0, 9, 100]

Approach:
1. Realize: Largest squares at either end
2. Two pointers at ends
3. Compare absolute values
4. Fill result array from end
5. Both pointers move inward

Why from end? Largest values at extremes!
```

Pattern 5: Three Pointers (Dutch National Flag)

Historical Background:

- Problem by Edsger Dijkstra (1976)
- Dutch flag: Red-White-Blue sections
- Generalized as 3-way partitioning

Concept: Partition array into three sections using three pointers.

Invariant Maintenance:

```
[0...low-1]     → All 0s (or first category)
[low...mid-1]   → All 1s (or second category)
[mid...high]    → Unknown (to be processed)
[high+1...n-1]  → All 2s (or third category)
```

Approach (Sort Colors - 0,1,2):

```
Approach:
1. Initialize: low=0, mid=0, high=n-1
2. While mid <= high:
  Case arr[mid] == 0:
    - Swap with low
    - Increment both low and mid
    - Why? Element from low is processed (0 or 1)

  Case arr[mid] == 1:
    - Already in correct section
    - Just increment mid

  Case arr[mid] == 2:
    - Swap with high
    - Decrement high only
    - Why? Don't increment mid (unknown element swapped)

3. Single pass: O(n) time
```

Key Insight: Mid pointer processes unknowns, low/high maintain boundaries.

Why mid doesn't increment after swap with high? The element coming from high position is unknown (not yet processed). It needs to be examined in the next iteration.

Advanced Pattern Recognition

Decision Framework

Step 1: Identify Data Structure

```
Is it Array/String? → Continue
Is it Linked List? → Fast-Slow for cycles
Is it Tree/Graph? → Different pattern (not two pointers)
```

Step 2: Check Constraints

```
Is data SORTED?
  YES → Likely Opposite Direction or Merge Pattern
  NO → Can you sort it?
    - If sorting allowed → Sort, then apply pattern
    - If order matters → Sliding Window or Same Direction
```

Step 3: Identify Operation

```
What are you finding?
 - PAIRS/TRIPLETS → Opposite Direction
 - SUBARRAYS → Sliding Window
 - REMOVE/MODIFY → Same Direction
 - MERGE/COMPARE → Merge Pattern
 - PARTITION (3 groups) → Three Pointers
```

Step 4: Check Problem Keywords

```
"in-place" → Same Direction (avoid extra space)
"sorted" → Opposite Direction
"consecutive" → Sliding Window
"without repeating" → Sliding Window with HashSet
"palindrome" → Opposite Direction
"cycle" → Fast-Slow (Floyd's)
"partition" → Three Pointers
```

Complex Problem Analysis

Multi-Pattern Problems:

Some problems require **combining patterns**:

1. **Palindrome Linked List**
   - Fast-Slow to find middle
   - Reverse second half
   - Opposite direction to compare
2. **Container With Most Water**
   - Opposite direction movement
   - Greedy choice: Move pointer with smaller height
   - Why? Larger height might give better area
3. **Trapping Rain Water**
   - Opposite direction
   - Track max_left and max_right
   - Water trapped = min(max_left, max_right) - current_height

---

Problem Categories with Analysis

Category 1: Two Sum Variants

Problems:

1. Two Sum II (Sorted array)
2. 3Sum
3. 3Sum Closest
4. 4Sum
5. Two Sum Less Than K

Progressive Complexity:

Two Sum II (LC 167)

```
Difficulty: Easy
Pattern: Opposite Direction
Key: Sorted array property

Approach:
- Two pointers at ends
- Move based on sum comparison
- O(n) time, O(1) space
```

3Sum (LC 15)

```
Difficulty: Medium
Pattern: Fix one + Opposite Direction
Additional Challenge: Avoid duplicates

Approach:
1. Sort array O(n log n)
2. Fix first element (i)
3. Two pointers on remaining array [i+1...n-1]
4. Skip duplicates:
   - After finding triplet
   - After fixing i

Total: O(n²) time
Why? n iterations × n for two pointers
```

Duplicate Handling Strategy:

After finding valid triplet:
- Skip same values for left pointer
- Skip same values for right pointer

While fixing i:
- if nums[i] == nums[i-1]: skip

4Sum (LC 18)

Difficulty: Medium
Pattern: Fix two + Opposite Direction

Approach:
1. Sort array
2. Fix first element (i)
3. Fix second element (j > i)
4. Two pointers on [j+1...n-1]
5. Skip duplicates at all levels

Total: $O(n^3)$ time

General N-Sum Pattern:

K-Sum problem:
- Fix (K-2) elements
- Apply two pointers on rest
- Time: O(n^(K-1))

Category 2: In-Place Array Modification

Core Concept: Avoid extra space by overwriting in-place.

Remove Duplicates from Sorted Array (LC 26)

Difficulty: Easy
Pattern: Fast-Slow (Slow as Writer)

Approach:
1. slow = 1 (first element always unique)
2. fast scans from index 1
3. When arr[fast] != arr[fast-1]:
   - New unique element found
   - arr[slow] = arr[fast]
   - slow++
4. Return slow (new length)

Why start slow at 1? First element always kept.

Move Zeroes (LC 283)

Difficulty: Easy
Pattern: Fast-Slow with Fill

Approach:
1. Phase 1: Collect non-zeros
   - slow = 0
   - fast finds non-zeros
   - arr[slow] = arr[fast], slow++

2. Phase 2: Fill remaining with zeros
   - From slow to n-1: arr[i] = 0

Alternative Single Pass:
- When non-zero found, swap with slow position
- Maintains relative order

Remove Element (LC 27)

Difficulty: Easy
Pattern: Fast-Slow

Approach:
- Similar to remove duplicates
- slow = write position
- fast scans array
- If arr[fast] != val: write and increment slow
- Return slow

Sort Colors (LC 75)

Difficulty: Medium
Pattern: Three Pointers (Dutch National Flag)

Deep Analysis:
Problem: Sort array of 0s, 1s, 2s in one pass

Naive: Counting sort O(2n) - two passes
Optimal: Dutch National Flag O(n) - one pass

Invariants:
[0..low-1]: all 0s
[low..mid-1]: all 1s
[mid..high]: unknown
[high+1..n-1]: all 2s

Why this works:
- All possible values have designated regions
- Mid processes unknowns systematically
- Swaps maintain invariants

Category 3: Palindrome Problems

Valid Palindrome (LC 125)

Difficulty: Easy
Pattern: Opposite Direction

Approach:
1. Two pointers: left=0, right=n-1
2. Skip non-alphanumeric characters
3. Compare characters (case-insensitive)
4. If mismatch → not palindrome
5. Converge pointers

Complexity: O(n) time, O(1) space

Valid Palindrome II (LC 680)

Difficulty: Easy
Pattern: Opposite Direction with Recursion

Challenge: Can delete at most ONE character

Approach:
1. Normal palindrome check
2. On first mismatch:
   - Try deleting left character
   - Try deleting right character
3. If either results in palindrome ⟶ true
4. Helper function: isPalindrome(left, right) without deletions

Key Insight: At most one mismatch allowed

Palindrome Linked List (LC 234)

Difficulty: Easy
Pattern: Multiple patterns combined

Approach:
1. Find middle using Fast-Slow
   - Slow moves 1 step
   - Fast moves 2 steps
   - When fast ends, slow at middle

2. Reverse second half of list
   - Standard linked list reversal

3. Compare first half with reversed second half
   - Two pointers: head and reversed_head
   - Compare values

Complexity: O(n) time, O(1) space

Category 4: Subarray/Substring Problems

Longest Substring Without Repeating Characters (LC 3)

Difficulty: Medium
Pattern: Sliding Window with HashSet

Deep Analysis:

State Management:
- Window: [left...right]
- HashSet: tracks characters in current window
- maxLength: tracks answer

Approach:
1. Expand: Add right character
2. Conflict Check: Character already in set?
3. Contract: Remove left characters until conflict resolved
4. Update: maxLength = max(maxLength, right - left + 1)

Why HashSet?
- O(1) lookup for duplicates
- O(1) insertion/deletion
- Perfect for "without repeating"

Edge Cases:
- Empty string → 0
- All same characters → 1
- All unique → n

Minimum Size Subarray Sum (LC 209)

Difficulty: Medium
Pattern: Variable Sliding Window

Challenge: Find MINIMUM length subarray with sum ≥ target

Approach:
1. Expand: Add right element to window_sum
2. Contraction: While window_sum ≥ target:
   - Update minimum length
   - Remove left element
   - Increment left
3. Why while not if? Find minimal window

Key Insight: Greedy contraction
- Once sum ≥ target, shrink to find minimum
- Each element added/removed exactly once → O(n)

Subarray Product Less Than K (LC 713)

Difficulty: Medium
Pattern: Sliding Window with Counter

Challenge: Count subarrays with product < k

Key Insight:
For window [left...right]:
- Number of subarrays ending at right = (right - left + 1)
- All these subarrays have product < k

Approach:
1. Maintain window with product < k
2. Expand right
3. Contract left while product ≥ k
4. Add (right - left + 1) to count
5. Why? All subarrays ending at right are valid

Example: [10, 5, 2], k=100
right=0: product=10, count+=1 ([10])
right=1: product=50, count+=2 ([5], [10,5])
right=2: product=100≥k, shrink, product=10, count+=2 ([2], [5,2])
Total: 5 subarrays

Maximum Consecutive Ones III (LC 1004)

Difficulty: Medium
Pattern: Sliding Window with Counter

Problem: Flip at most K zeros to get longest sequence of 1s

Approach:
1. Window tracks zeros count
2. Expand: If 0, increment zero_count
3. Contract: While zero_count > k, shrink from left
4. Update: maxLength at each step

Reformulation: Longest subarray with at most K zeros

Category 5: Container/Water Problems

Container With Most Water (LC 11)

Difficulty: Medium
Pattern: Opposite Direction with Greedy Choice

Problem: Two lines form container, maximize water area

Mathematical Analysis:
Area = min(height[left], height[right]) × (right - left)

Greedy Strategy:
- Start with maximum width (ends)
- Move pointer with SMALLER height
- Why? Larger height is not limiting factor

Proof of Correctness:
- Moving larger height pointer:
  - Width decreases
  - Height can't increase (limited by smaller)
  - Area can only decrease ⟶ suboptimal

- Moving smaller height pointer:
  - Width decreases
  - Height might increase
  - Area might increase ⟶ worth exploring

Complexity: O(n) time, O(1) space

Trapping Rain Water (LC 42)

Difficulty: Hard
Pattern: Opposite Direction with Max Tracking

Deep Analysis:

Key Observation:
Water above position i = min(max_left, max_right) - height[i]

Why?
- Water level limited by shorter boundary
- Can't exceed current height

Approach 1: Two Pass (Easier to understand)
1. Compute max_left[i] for all i
2. Compute max_right[i] for all i
3. Water[i] = min(max_left[i], max_right[i]) - height[i]
Space: O(n)

Approach 2: Two Pointers (Optimal)
Variables: left, right, left_max, right_max

Invariant:
- left_max = max height in [0...left]
- right_max = max height in [right...n-1]

Logic:
if left_max < right_max:
  - Water at left limited by left_max
  - Process left, move left++
else:
  - Water at right limited by right_max
  - Process right, move right--

Why it works:
- Smaller max determines water level
- Process side with smaller max
- Other side guaranteed to be higher

Complexity: O(n) time, O(1) space

Category 6: Linked List Problems

Linked List Cycle (LC 141)

Difficulty: Easy
Pattern: Fast-Slow (Floyd's Cycle Detection)

Mathematical Proof:

Setup:
- Slow moves 1 step
- Fast moves 2 steps
- If cycle exists with length C

Proof:
1. Slow enters cycle at some point
2. Fast is already in cycle (or enters shortly)
3. Gap between them decreases by 1 each iteration
4. They must meet within C iterations

Meeting Point:
Let distance to cycle = D
Distance slow traveled at meeting = D + x
Distance fast traveled = D + nC + x (n laps)
Since fast = 2 × slow:
D + nC + x = 2(D + x)
D + nC = D + 2x
D = 2x - nC

Linked List Cycle II (LC 142)

Difficulty: Medium
Pattern: Floyd's Algorithm Extended

Challenge: Find START of cycle

Mathematical Analysis:

Phase 1: Detect cycle (fast-slow meet)
Phase 2: Find cycle start

Key Insight:
Distance from head to cycle start =
Distance from meeting point to cycle start

Proof:
Let L = distance from head to cycle start
Let M = distance from cycle start to meeting point
Let C = cycle length

Slow traveled: L + M
Fast traveled: L + M + nC (n complete loops)

Since fast = 2 × slow:
L + M + nC = 2(L + M)
L = M + nC - M
L = nC - M = distance from meeting point to cycle start

Algorithm:
1. Detect cycle, find meeting point
2. Reset one pointer to head
3. Move both at same speed
4. Where they meet = cycle start

Remove Nth Node From End (LC 19)

Difficulty: Medium
Pattern: Fast-Slow with Gap

Approach:
1. Create gap of n between fast and slow
2. Move fast n steps ahead
3. Move both together until fast reaches end
4. Slow is now at (n-1)th node from end
5. Remove next node

Why it works:
Gap maintained = n nodes
When fast at end, slow at target position

Optimization Techniques

1. Skip Duplicates Strategy

Problem: 3Sum, 4Sum with duplicate triplets/quadruplets

Technique:

```
After processing element at index i:
while (i < n && arr[i] == arr[i+1]) i++;

After moving left pointer:
while (left < right && arr[left] == arr[left+1]) left++;

After moving right pointer:
while (left < right && arr[right] == arr[right-1]) right--;
```

Why Important?

- Prevents duplicate results
- Still maintains $O(n^2)$ or $O(n^3)$ complexity
- Critical for acceptance in these problems

---

2. Early Termination

3Sum Optimization:

```
if (nums[i] > 0) break;
// If smallest number > 0, no negative sum possible
```

Binary Search Integration:

```
For very large arrays with two pointers:
- Use two pointers for coarse positioning
- Binary search for exact match
- Hybrid approach: O(n log n)
```

---

3. Preprocessing Strategies

Squares of Sorted Array:

```
Insight: Don't actually square first!
- Compare absolute values
- Square only when placing in result
- Saves computation
```

String Problems:

```
Preprocess: Convert to lowercase, remove non-alphanumeric
Then apply two pointers
Trade-off: O(n) extra space but cleaner logic
```

---

4. Invariant Preservation

Key to Correctness:

```
Define invariants clearly:
- What does each pointer represent?
- What is guaranteed about processed regions?
- What remains to be processed?

Verify invariants:
- Initial state
- After each operation
- Termination state
```

Example (Dutch National Flag):

Invariants:
- [0...low-1]: all 0s ✓
- [low...mid-1]: all 1s ✓
- [mid...high]: unknown (being processed)
- [high+1...n-1]: all 2s ✓

Each operation maintains these invariants

Practice Roadmap

Week 1: Foundation (Easy Problems)

Day 1-2: Opposite Direction

1. Two Sum II (LC 167) - Basic pattern
2. Valid Palindrome (LC 125) - Character comparison
3. Reverse String (LC 344) - Simple swap

Day 3-4: Same Direction 4. Remove Duplicates (LC 26) - Slow as writer 5. Move Zeroes (LC 283) - Element collection 6. Remove Element (LC 27) - Filtering

Day 5-6: Merge Pattern 7. Merge Sorted Array (LC 88) - Backward merge 8. Squares of Sorted Array (LC 977) - Compare ends

Day 7: Review

- Solve all 8 problems again without hints
- Document patterns observed

Week 2: Intermediate (Medium Problems)

Day 8-9: Advanced Opposite Direction 9. Container With Most Water (LC 11) - Greedy choice 10. 3Sum (LC 15) - Duplicate handling 11. 3Sum Closest (LC 16) - Track best difference

Day 10-11: Sliding Window 12. Longest Substring Without Repeating (LC 3) - HashSet 13. Minimum Size Subarray Sum (LC 209) - Variable window 14. Subarray Product Less Than K (LC 713) - Counting

Day 12-13: Complex Patterns 15. Sort Colors (LC 75) - Three pointers 16. Partition Labels (LC 763) - Last occurrence tracking 17. Find K Closest Elements (LC 658) - Sliding window on sorted

Day 14: Review & Analysis

- Compare approaches across similar problems
- Identify when each pattern applies

Week 3: Advanced (Medium-Hard)

Day 15-16: Hard Problems 18. Trapping Rain Water (LC 42) - Max tracking 19. Minimum Window Substring (LC 76) - Complex validation 20. Longest Substring with At Most K Distinct (LC 340) - Frequency map

Day 17-18: Linked List Integration 21. Linked List Cycle (LC 141) - Floyd's basic 22. Linked List Cycle II (LC 142) - Find start 23. Palindrome Linked List (LC 234) - Multi-pattern

Day 19-20: Advanced Variations 24. 4Sum (LC 18) - Triple pointer 25. Subarrays with K Different Integers (LC 992) - atMost trick 26. Longest Repeating Character Replacement (LC 424) - Complex window

Day 21: Comprehensive Review

- Solve mixed problems randomly
- Time yourself
- Focus on pattern recognition speed

Extended Practice (50+ Problems)

Opposite Direction (15 problems):

- Two Sum II (LC 167) ⭐ Foundation
- 3Sum (LC 15) ⭐ Classic
- 3Sum Closest (LC 16)
- 4Sum (LC 18)
- Valid Palindrome (LC 125) ⭐ Foundation
- Valid Palindrome II (LC 680)
- Container With Most Water (LC 11) ⭐ Must-do
- Trapping Rain Water (LC 42) ⭐ Hard classic
- Two Sum Less Than K (LC 1099)
- 3Sum Smaller (LC 259)
- Number of Subsequences (LC 1498)
- Boats to Save People (LC 881)
- Reverse String (LC 344)
- Reverse Vowels of String (LC 345)
- Find K Closest Elements (LC 658)

Same Direction (12 problems):

- Remove Duplicates (LC 26) ⭐ Foundation
- Remove Element (LC 27) ⭐ Foundation
- Move Zeroes (LC 283) ⭐ Foundation
- Sort Colors (LC 75) ⭐ Three pointers
- Partition Array (LC 2161)
- Sort Array By Parity (LC 905)
- Backspace String Compare (LC 844)
- Remove Duplicates II (LC 80)
- Segregate Even Odd (custom)
- Partition Equal Subset Sum (variation)
- Quick Select (custom)
- Is Subsequence (LC 392)

Sliding Window (15 problems):

- Longest Substring Without Repeating (LC 3) ⭐ Classic
- Minimum Size Subarray Sum (LC 209) ⭐ Variable window
- Subarray Product Less Than K (LC 713)
- Maximum Consecutive Ones III (LC 1004)
- Longest Repeating Character Replacement (LC 424)
- Permutation in String (LC 567)
- Minimum Window Substring (LC 76) ⭐ Hard classic
- Longest Substring with At Most K Distinct (LC 340)
- Fruit Into Baskets (LC 904)
- Subarrays with K Different Integers (LC 992)
- Max Sum Subarray of Size K (custom)
- Longest Mountain in Array (LC 845)
- Get Equal Substrings Within Budget (LC 1208)
- Grumpy Bookstore Owner (LC 1052)
- Substring Concatenation (LC 30)

Merge Pattern (8 problems):

- Merge Sorted Array (LC 88) ⭐ Foundation
- Squares of Sorted Array (LC 977) ⭐ Compare ends
- Intersection of Two Arrays (LC 349)
- Intersection of Two Arrays II (LC 350)
- Interval List Intersections (LC 986)
- Merge Two Sorted Lists (LC 21)
- Median of Two Sorted Arrays (LC 4) - Hard
- Find Median from Data Stream (LC 295)

Linked List (10 problems):

- Linked List Cycle (LC 141) ⭐ Floyd's basic
- Linked List Cycle II (LC 142) ⭐ Find start
- Remove Nth Node (LC 19)
- Palindrome Linked List (LC 234)
- Middle of Linked List (LC 876)
- Reorder List (LC 143)
- Rotate List (LC 61)
- Swap Nodes in Pairs (LC 24)
- Happy Number (LC 202) - Floyd's on numbers
- Find Duplicate Number (LC 287) - Floyd's on array

Study Notes & Mental Models

Mental Model 1: The "Elimination" Mindset

Concept: Two pointers eliminate invalid solutions without checking them.

Example (Two Sum II):

```
Array: [2, 7, 11, 15], Target: 9
Brute force checks: (2,7), (2,11), (2,15), (7,11), (7,15), (11,15) = 6 comparisons

Two Pointers:
left=0, right=3: 2+15=17 > 9
  Elimination: All pairs with 15 are too large
  Never check (2,15), (7,15), (11,15)
left=0, right=2: 2+11=13 > 9
  Elimination: All pairs with 11 are too large
left=0, right=1: 2+7=9 ✓ Found!
  Total checks: 3
```

Key Insight: Each pointer movement eliminates a whole set of possibilities.

Mental Model 2: The "Window Contract"

For Sliding Window problems, think of window as having a "contract" (condition):

```
Contract: "Sum ≥ target"

When contract violated (sum < target):
  → Expand window (add more elements)

When contract satisfied:
  → Try to minimize (shrink from left)
  → Record current state
```

Analogy: Like an accordion - expand when needed, compress when possible.

Mental Model 3: The "Write vs Read" Model

For Same Direction pattern:

```
slow = Writer (Where to write next)
fast = Reader (What to read next)

Only write when reader finds valid data
Reader always ahead or equal to writer
Gap = invalid/deleted elements
```

Analogy: Like transcribing - you read ahead, but write only what's valid.

Mental Model 4: The "Race and Meeting"

For Floyd's Cycle Detection:

```
Think of two runners on a track:
- Slow runner: 1 lap per hour
- Fast runner: 2 laps per hour

If track is circular (cycle exists):
  - Fast will eventually lap slow
  - Meeting point proves cycle

If track is straight (no cycle):
  - Fast reaches end first
  - Never meets slow
```

Pattern Recognition Flowchart

```
START: Look at problem
   ↓
Is data structure Array/String?
   ↓ YES
   ↓
Check: Is it SORTED or can be sorted?
   ↓
   ├──→ YES (Sorted)
   │       ↓
   │    Finding PAIRS/TRIPLETS?
   │       ↓ YES
   │    Opposite Direction
   │       (Two Sum, 3Sum pattern)
   │
   │    Finding MERGE/INTERSECTION?
   │       ↓ YES
   │    Merge Pattern
   │       (Merge sorted arrays)
   │
   ├──→ NO (Not sorted or order matters)
   │       ↓
   │    Need SUBARRAY/SUBSTRING?
   │       ↓ YES
   │    Sliding Window
   │       (Track window state)
   │
   │    Need IN-PLACE modification?
```

```
    ↓ YES
  Same Direction
    (Fast-Slow as writer)

  Need to PARTITION into 3 groups?
    ↓ YES
  Three Pointers
    (Dutch National Flag)

└──→ Is it LINKED LIST?
    ↓ YES
  Finding CYCLE?
    ↓ YES
  Fast-Slow (Floyd's)
```

Common Pitfalls & Solutions

Pitfall 1: Infinite Loops

```
Problem: Pointers not moving correctly

Solution: Verify these
- Loop condition: while (left < right) not (<=) for opposite direction
- Pointer increment: Must move in at least one branch
- Base case: Handle empty array, single element
```

Pitfall 2: Array Index Out of Bounds

```
Problem: Accessing arr[left] or arr[right] beyond limits

Solution:
- Boundary check: left >= 0 && right < n
- Loop condition prevents: while (left < right)
- For linked lists: Check not null before access
```

Pitfall 3: Duplicate Handling in 3Sum

```
Problem: Getting duplicate triplets

Solution: Skip duplicates at THREE levels
1. After fixing first element
2. After moving left pointer
3. After moving right pointer

// After processing nums[i]
while (i < n-1 && nums[i] == nums[i+1]) i++;
```

Pitfall 4: Window Not Contracting Properly

```
Problem: Using if instead of while for contraction

Wrong:
if (window_invalid) left++;

Correct:
while (window_invalid) left++;

Why? Might need multiple removals to fix window
```

Pitfall 5: Returning Wrong Value Type

```
Problem: Returning index when value needed (or vice versa)

Check problem statement:
- "Return indices" → return [i, j]
- "Return values" → return [arr[i], arr[j]]
- "Return count" → return counter
- "Return boolean" → return true/false
```

Time Complexity Analysis Deep Dive

Why Two Pointers is O(n):

Mathematical Proof:

Let T(n) = total operations

For opposite direction:
- Each pointer moves at most n times
- left: 0 → n (max n moves)
- right: n → 0 (max n moves)
- Total moves: 2n
- Operations per move: O(1)
- T(n) = O(2n) = O(n)

For sliding window:
- right pointer: 0 → n (n iterations)
- left pointer: moves only when window invalid
- Key insight: Each element added once, removed once
- Total operations: 2n (n adds + n removes)
- T(n) = O(2n) = O(n)

Comparison with Brute Force:

```
Brute Force (Nested Loops):
for i in range(n):          # n iterations
    for j in range(i+1, n):  # n-i iterations
        check(i, j)          # O(1)

Total: Σ(i=1 to n) (n-i) = n(n-1)/2 = O(n²)

Two Pointers Optimization:
Reduction: n² → n
Speed-up: n times faster!

Example: n = 10,000
Brute force: ~100,000,000 operations
Two pointers: ~20,000 operations
5000x faster!
```

Space Complexity Patterns

O(1) Space (In-place):

- Opposite Direction
- Same Direction
- Three Pointers
- Merge (when merging in-place)

O(k) Space (Window tracking):

- Sliding Window with HashSet: O(unique elements)
- Sliding Window with HashMap: O(character set size)
- Usually k << n, so effectively O(1) for fixed alphabet

Trade-offs:

```
Problem: Longest Substring Without Repeating Characters

Approach 1: Sliding Window + HashSet
- Time: O(n)
- Space: O(min(n, m)) where m = charset size
- For ASCII: O(128) = O(1)

Approach 2: Track last index in array
- Time: O(n)
- Space: O(m) for index array
- Slightly better constants
```

Advanced Techniques

Technique 1: Binary Search + Two Pointers Hybrid

When to use: Very large sorted arrays, need exact match

```
Approach:
1. Use two pointers for coarse positioning
2. Once close to target, binary search
3. Combines O(n) and O(log n) benefits

Example: Find closest pair sum in sorted array
- Two pointers to get close: O(n)
- Binary search for exact: O(log n)
```

Technique 2: Two Pointers with Greedy Choice

Container With Most Water Analysis:

```
Why moving shorter height is optimal?

Current state: left=L, right=R, area = min(h[L], h[R]) × (R-L)

Assume h[L] < h[R]

Case 1: Move R (larger height)
- New width: (R-1) - L (decreased)
- New height: min(h[L], h[R-1]) ≤ h[L] (still limited)
- New area ≤ current area (both factors decreased/same)
- Result: Suboptimal

Case 2: Move L (smaller height)
- New width: R - (L+1) (decreased)
- New height: min(h[L+1], h[R]) (might increase!)
- New area might increase (height increase can offset width decrease)
- Result: Worth exploring

Conclusion: Always move pointer with smaller height
```

Technique 3: At-Most K Trick

Problem Type: "Exactly K" problems are hard; convert to "At-Most K"

```
Exactly K = (At-Most K) - (At-Most K-1)

Example: Subarrays with K Different Integers
- Direct counting of "exactly K" is complex
- Count "at most K" is straightforward (sliding window)
- Answer = atMost(K) - atMost(K-1)

Function atMost(K):
    window with at most K distinct elements
    use sliding window
    count all valid subarrays
```

Technique 4: Preprocessing for Efficiency

String problems optimization:

```
Approach (Valid Palindrome):

Naive: Check each character, skip non-alphanumeric in loop
while (left < right) {
    while (!isalnum(s[left])) left++;
    while (!isalnum(s[right])) right--;
    // compare
}

Optimized: Preprocess once
1. Clean string: remove non-alphanumeric, convert to lowercase
2. Simple two pointer comparison
3. Trade-off: O(n) space for cleaner logic

When to use each:
- Naive: Space critical, string long
- Optimized: Clarity important, multiple operations on same string
```

Problem-Solving Template

Template 1: Opposite Direction

```
// C++ Approach Structure
left = 0, right = n-1
while (left < right) {
    calculate current_result

    if (found_target) {
        process result
        move both pointers inward
        skip duplicates if needed
    }
    else if (need_larger_value) {
        left++
    }
    else {
        right--
    }
}
```

Template 2: Same Direction (Slow as Writer)

```
// C++ Approach Structure
slow = 0  // write position
for (fast = 0; fast < n; fast++) {
    if (is_valid_element(arr[fast])) {
        arr[slow] = arr[fast]
        slow++
    }
}
return slow  // new length
```

Template 3: Variable Sliding Window

```
// C++ Approach Structure
left = 0
result = initial_value

for (right = 0; right < n; right++) {
    add arr[right] to window

    while (window_invalid) {
        remove arr[left] from window
        left++
    }

    update result with current window
}
return result
```

Template 4: Dutch National Flag