Master Notes: Pattern Recognition & Problem-Solving Approaches for Arrays & Strings

🔍 THE SECRET TO CRACKING ANY PROBLEM

The Universal Problem-Solving Framework

1. IDENTIFY the core pattern (what is the problem really asking?)
2. CLASSIFY the constraints (time/space limits, input size)
3. RECOGNIZE the approach family (two pointers, sliding window, etc.)
4. OPTIMIZE using the right data structure
5. IMPLEMENT with edge case handling

🎯 PATTERN RECOGNITION CHEAT SHEET

Visual Pattern Identification

Problem Keywords → Pattern → Approach → Time Complexity

"Find pair/triplet" → Two/Three Pointers → $O(n^2)$
"Subarray/Substring" → Sliding Window → $O(n)$
"Sorted array" → Binary Search → $O(\log n)$
"All permutations" → Backtracking → $O(n!)$
"Optimal solution" → Dynamic Programming → $O(n^2)$
"Count frequency" → Hash Map → $O(n)$
"Range queries" → Prefix Sum → $O(1)$ query
"Compare strings" → Two Pointers → $O(n)$
"Pattern matching" → KMP/Rolling Hash → $O(n+m)$
"Palindrome" → Expand Around Center → $O(n^2)$

🚀 CORE PATTERNS WITH VISUAL GUIDES

1. TWO POINTERS PATTERN

Visual Representation:

```
Array: [1, 2, 3, 4, 5, 6, 7, 8, 9]
       ↑              ↑
     left           right

Movement Patterns:
- Opposite Direction: left++, right--
- Same Direction: slow++, fast += 2
- Meeting Point: when left >= right
```

Secret Recognition Triggers:

- ✅ **"Find pair with sum X"** → Opposite direction pointers
- ✅ **"Remove duplicates"** → Same direction pointers
- ✅ **"Palindrome check"** → Opposite direction
- ✅ **"Cycle detection"** → Fast/slow pointers
- ✅ **"Merge sorted arrays"** → Two pointers on different arrays

Template Code Pattern:

```python
def two_pointers_opposite(arr, target):
    left, right = 0, len(arr) - 1
    while left < right:
        current_sum = arr[left] + arr[right]
        if current_sum == target:
            return [left, right]
        elif current_sum < target:
            left += 1
        else:
            right -= 1
    return []

def two_pointers_same_direction(arr):
    slow = fast = 0
    while fast < len(arr):
        # Process logic
        if condition:
            slow += 1
        fast += 1
    return slow
```

---

2. SLIDING WINDOW PATTERN

Visual Representation:

```
Fixed Window (size k=3):
[a, b, c, d, e, f, g]
 [---]          window at position 0
   [---]        window at position 1
     [---]      window at position 2

Variable Window:
[a, b, c, d, e, f, g]
 [--]           expand window
 [-----]         expand more
   [--]         shrink window
```

Secret Recognition Triggers:

- ✅ **"Maximum/Minimum subarray of size K"** → Fixed window
- ✅ **"Longest substring with condition"** → Variable window
- ✅ **"All subarrays with property X"** → Variable window
- ✅ **"Substring containing all characters"** → Variable window

Template Code Patterns:

```python
```

```python
# Fixed Window Template
def fixed_window(arr, k):
    window_sum = sum(arr[:k])
    max_sum = window_sum

    for i in range(k, len(arr)):
        window_sum = window_sum - arr[i-k] + arr[i]
        max_sum = max(max_sum, window_sum)
    return max_sum

# Variable Window Template
def variable_window(arr, target):
    left = 0
    current_sum = 0
    result = 0

    for right in range(len(arr)):
        current_sum += arr[right]

        while current_sum > target:
            current_sum -= arr[left]
            left += 1

        result = max(result, right - left + 1)
    return result
```

---

3. HASH MAP PATTERN

Visual Representation:

```
Array: [2, 7, 11, 15]  Target: 9
Index:  0  1   2   3

Hash Map Construction:
Iteration 0: {2: 0}          Check if (9-2=7) exists → No
Iteration 1: {2: 0, 7: 1}    Check if (9-7=2) exists → Yes!
Result: indices [0, 1]
```

Secret Recognition Triggers:

- ✅ **"Find pair/complement"** → Hash map lookup
- ✅ **"Count frequency"** → Hash map counter
- ✅ **"Check if exists"** → Hash set

- ✅ **"Group by property"** → Hash map grouping
- ✅ **"Anagram detection"** → Character frequency map

Template Code Patterns:

```python
# Complement Pattern
def two_sum(nums, target):
    seen = {}
    for i, num in enumerate(nums):
        complement = target - num
        if complement in seen:
            return [seen[complement], i]
        seen[num] = i

# Frequency Pattern
def char_frequency(s):
    freq = {}
    for char in s:
        freq[char] = freq.get(char, 0) + 1
    return freq

# Grouping Pattern
def group_anagrams(strs):
    groups = {}
    for s in strs:
        key = ''.join(sorted(s))
        if key not in groups:
            groups[key] = []
        groups[key].append(s)
    return list(groups.values())
```

---

4. BINARY SEARCH PATTERN

Visual Representation:

```
Sorted Array: [1, 3, 5, 7, 9, 11, 13]
              0  1  2  3  4   5   6

Find target = 7:
Step 1: left=0, right=6, mid=3, arr[3]=7 ✓ Found!

Search Space Reduction:
[1, 3, 5, 7, 9, 11, 13]  Initial space
     [7, 9, 11]          After comparison (if target > arr[mid])
```

Secret Recognition Triggers:

- ✅ **"Sorted array"** → Binary search
- ✅ **"Find target/position"** → Classic binary search
- ✅ **"First/Last occurrence"** → Modified binary search
- ✅ **"Peak element"** → Binary search on property
- ✅ **"Search space can be divided"** → Binary search on answer

Template Code Patterns:

python

```python
# Classic Binary Search
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

# First Occurrence
def find_first(arr, target):
    left, right = 0, len(arr) - 1
    result = -1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            result = mid
            right = mid - 1  # Continue searching left
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return result
```

---

5. DYNAMIC PROGRAMMING PATTERN

Visual Representation:

Problem: Fibonacci Sequence
F(n) = F(n-1) + F(n-2)

Top-Down (Memoization):
F(5) → F(4) + F(3)
→ [F(3) + F(2)] + [F(2) + F(1)]
→ Overlapping subproblems!

Bottom-Up (Tabulation):
dp[0] = 0, dp[1] = 1
dp[2] = dp[1] + dp[0] = 1
dp[3] = dp[2] + dp[1] = 2
...

Secret Recognition Triggers:

- ✅ **"Optimal solution"** → DP
- ✅ **"Count ways"** → DP
- ✅ **"Maximum/Minimum"** → DP
- ✅ **"Decision at each step"** → DP
- ✅ **"Overlapping subproblems"** → DP

Template Code Patterns:

python

```
# 1D DP Template
def dp_1d(n):
    dp = [0] * (n + 1)
    dp[0] = base_case

    for i in range(1, n + 1):
        dp[i] = transition_function(dp[i-1], dp[i-2], ...)

    return dp[n]

# 2D DP Template
def dp_2d(m, n):
    dp = [[0] * n for _ in range(m)]

    # Initialize base cases
    for i in range(m):
        dp[i][0] = base_case

    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = transition_function(dp[i-1][j], dp[i][j-1], ...)

    return dp[m-1][n-1]
```

🎨 STRING-SPECIFIC PATTERNS

6. PALINDROME PATTERNS

Visual Representation:

```
Expand Around Center:
String: "babad"
      ↑
   Check: "b" (center)
   Check: "aba" (expand)
   Check: "babad" (expand - not palindrome)

Two Types:
- Odd length: single center character
- Even length: between two characters
```

Secret Recognition Triggers:

- ✅ **"Palindrome"** → Expand around center or two pointers

- ✅ **"Longest palindromic substring"** → Expand around center
- ✅ **"Count palindromes"** → Expand around center
- ✅ **"Palindrome with changes"** → DP

Template Code:

```python
def expand_around_center(s, left, right):
    while left >= 0 and right < len(s) and s[left] == s[right]:
        left -= 1
        right += 1
    return s[left+1:right]

def longest_palindrome(s):
    longest = ""
    for i in range(len(s)):
        # Odd length
        odd = expand_around_center(s, i, i)
        # Even length
        even = expand_around_center(s, i, i+1)

        longest = max([longest, odd, even], key=len)
    return longest
```

---

7. PATTERN MATCHING

Visual Representation:

```
KMP Algorithm Failure Function:
Pattern: "ABABACA"
Failure: [0,0,1,2,3,0,1]

Text:    "ABABABABACA"
Pattern: "ABABACA"
         ↑↑↑↑  (mismatch at position 4)
Skip to: "  ABABACA" (use failure function)
```

Secret Recognition Triggers:

- ✅ **"Find pattern in text"** → KMP or Rabin-Karp
- ✅ **"Multiple pattern searches"** → Aho-Corasick
- ✅ **"Approximate matching"** → Edit distance

- ✅ **"Rolling hash"** → Rabin-Karp

🧠 PROBLEM CLASSIFICATION DECISION TREE

```
START: Read Problem Statement
|
├─ Involves "pair/triplet sum"? → TWO POINTERS
├─ Mentions "subarray/substring"? → SLIDING WINDOW
├─ Array is "sorted"? → BINARY SEARCH
├─ Need "all combinations"? → BACKTRACKING
├─ Asks for "optimal/count ways"? → DYNAMIC PROGRAMMING
├─ Need "frequency/lookup"? → HASH MAP
├─ Involves "palindrome"? → EXPAND AROUND CENTER
└─ Pattern "matching"? → STRING ALGORITHMS
```

🔧 SECRET OPTIMIZATION TRICKS

Memory Optimization Secrets:

1. **Rolling Array**: For DP, use only current and previous row
2. **In-place modification**: Modify input array instead of creating new one
3. **Bit manipulation**: Use bits for boolean arrays
4. **Two variables**: Instead of array for simple DP states

Time Optimization Secrets:

1. **Early termination**: Break loops when condition is met
2. **Skip duplicates**: In sorted arrays, skip same elements
3. **Preprocessing**: Sort/precompute when beneficial
4. **Choose right data structure**: HashMap vs TreeMap vs Array

**Code