Algorithmic Patterns Cheat Sheet - Citadel HackerRank

1. ARRAY & STRING PATTERNS

Prefix Sum Pattern

When to use: Range sum queries, subarray sums

```
# Build prefix sum
prefix = [0]
for num in arr:
    prefix.append(prefix[-1] + num)

# Get sum from i to j
range_sum = prefix[j+1] - prefix[i]

# Example: Find subarrays with sum == k
from collections import defaultdict
count = 0
prefix_sum = 0
sum_count = defaultdict(int)
sum_count[0] = 1

for num in arr:
    prefix_sum += num
    count += sum_count[prefix_sum - k]
    sum_count[prefix_sum] += 1
```

Sliding Window Pattern

When to use: Contiguous subarray/substring problems with constraints

```
# Fixed size window
def fixed_window(arr, k):
    window_sum = sum(arr[:k])
    max_sum = window_sum

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

```
        max_sum = max(max_sum, window_sum)
    return max_sum

# Variable size window
def variable_window(s, k):
    left = 0
    char_count = {}
    max_len = 0

    for right in range(len(s)):
        char_count[s[right]] = char_count.get(s[right], 0) + 1

        # Shrink window if constraint violated
        while len(char_count) > k:
            char_count[s[left]] -= 1
            if char_count[s[left]] == 0:
                del char_count[s[left]]
            left += 1

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

Two Pointer Pattern

When to use: Sorted arrays, palindromes, pair finding

```
# Two sum in sorted array
def two_sum_sorted(arr, target):
    left, right = 0, len(arr) - 1

    while left < right:
        current = arr[left] + arr[right]
        if current == target:
            return [left, right]
        elif current < target:
            left += 1
        else:
            right -= 1
    return [-1, -1]

# Remove duplicates in-place
def remove_duplicates(arr):
    if not arr:
        return 0

    write = 1
```

```
    for read in range(1, len(arr)):
        if arr[read] != arr[read-1]:
            arr[write] = arr[read]
            write += 1
    return write
```

Fast & Slow Pointer (Floyd's Cycle Detection)

When to use: Cycle detection, finding middle

```
# Detect cycle
def has_cycle(head):
    slow = fast = head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False

# Find middle
def find_middle(head):
    slow = fast = head

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

2. HASH TABLE PATTERNS

Frequency Counter

```
from collections import Counter, defaultdict

# Count occurrences
freq = Counter(arr)
# or
freq = defaultdict(int)
for item in arr:
    freq[item] += 1

# Find elements with frequency > k
```

```
result = [key for key, count in freq.items() if count > k]
```

Index Mapping

```
# Two sum using hash
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
    return []
```

Group Anagrams Pattern

```
from collections import defaultdict

def group_anagrams(words):
    groups = defaultdict(list)
    for word in words:
        # Use sorted string as key
        key = ''.join(sorted(word))
        groups[key].append(word)
    return list(groups.values())
```

3. STACK & QUEUE PATTERNS

Monotonic Stack

When to use: Next greater/smaller element, histogram problems

```
# Next greater element
def next_greater(arr):
    result = [-1] * len(arr)
    stack = []  # Store indices

    for i in range(len(arr)):
        while stack and arr[stack[-1]] < arr[i]:
            idx = stack.pop()
            result[idx] = arr[i]
        stack.append(i)
    return result
```

```
# Largest rectangle in histogram
def largest_rectangle(heights):
    stack = []
    max_area = 0

    for i, h in enumerate(heights):
        start = i
        while stack and stack[-1][1] > h:
            idx, height = stack.pop()
            max_area = max(max_area, height * (i - idx))
            start = idx
        stack.append((start, h))

    for i, h in stack:
        max_area = max(max_area, h * (len(heights) - i))
    return max_area
```

Queue with Two Stacks

```
class QueueWithStacks:
    def __init__(self):
        self.s1 = []  # Push stack
        self.s2 = []  # Pop stack

    def enqueue(self, x):
        self.s1.append(x)

    def dequeue(self):
        if not self.s2:
            while self.s1:
                self.s2.append(self.s1.pop())
        return self.s2.pop() if self.s2 else None
```

4. TREE PATTERNS

DFS Traversals

```
# Inorder (Left-Root-Right)
def inorder(root):
    if not root:
        return []
    return inorder(root.left) + [root.val] + inorder(root.right)
```

```
# Preorder (Root-Left-Right)
def preorder(root):
    if not root:
        return []
    return [root.val] + preorder(root.left) + preorder(root.right)

# Postorder (Left-Right-Root)
def postorder(root):
    if not root:
        return []
    return postorder(root.left) + postorder(root.right) + [root.val]
```

BFS (Level Order)

```
from collections import deque

def level_order(root):
    if not root:
        return []

    result = []
    queue = deque([root])

    while queue:
        level = []
        for _ in range(len(queue)):
            node = queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(level)
    return result
```

Path Sum Problems

```
def has_path_sum(root, target):
    if not root:
        return False

    if not root.left and not root.right:
        return root.val == target
```

```
    target -= root.val
    return has_path_sum(root.left, target) or has_path_sum(root.right, target)
```

5. GRAPH PATTERNS

BFS (Shortest Path)

```
from collections import deque

def bfs_shortest_path(graph, start, end):
    queue = deque([(start, 0)])
    visited = {start}

    while queue:
        node, dist = queue.popleft()
        if node == end:
            return dist

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, dist + 1))
    return -1
```

DFS (Connected Components)

```
def count_components(n, edges):
    graph = [[] for _ in range(n)]
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    visited = set()
    count = 0

    def dfs(node):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                dfs(neighbor)

    for i in range(n):
```

```
        if i not in visited:
            dfs(i)
            count += 1
    return count
```

Topological Sort (Kahn's Algorithm)

```
from collections import deque, defaultdict

def topological_sort(n, edges):
    graph = defaultdict(list)
    indegree = [0] * n

    for u, v in edges:
        graph[u].append(v)
        indegree[v] += 1

    queue = deque([i for i in range(n) if indegree[i] == 0])
    result = []

    while queue:
        node = queue.popleft()
        result.append(node)

        for neighbor in graph[node]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)

    return result if len(result) == n else []
```

6. DYNAMIC PROGRAMMING PATTERNS

1D DP

```
# Fibonacci-style
def climb_stairs(n):
    if n <= 2:
        return n

    dp = [0] * (n + 1)
    dp[1], dp[2] = 1, 2
```

```
    for i in range(3, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]

# Space-optimized
def climb_stairs_optimized(n):
    if n <= 2:
        return n

    prev2, prev1 = 1, 2
    for i in range(3, n + 1):
        curr = prev1 + prev2
        prev2, prev1 = prev1, curr
    return prev1
```

2D DP (Grid)

```
# Unique paths
def unique_paths(m, n):
    dp = [[1] * n for _ in range(m)]

    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
    return dp[m-1][n-1]
```

Knapsack Pattern

```
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i-1] <= w:
                dp[i][w] = max(
                    dp[i-1][w],
                    dp[i-1][w - weights[i-1]] + values[i-1]
                )
            else:
                dp[i][w] = dp[i-1][w]
    return dp[n][capacity]
```

7. GREEDY PATTERNS

Interval Scheduling

```
# Merge intervals
def merge_intervals(intervals):
    intervals.sort(key=lambda x: x[0])
    merged = [intervals[0]]

    for start, end in intervals[1:]:
        if start <= merged[-1][1]:
            merged[-1][1] = max(merged[-1][1], end)
        else:
            merged.append([start, end])
    return merged

# Non-overlapping intervals
def erase_overlap(intervals):
    intervals.sort(key=lambda x: x[1])
    end = float('-inf')
    count = 0

    for start, interval_end in intervals:
        if start >= end:
            count += 1
            end = interval_end
    return len(intervals) - count
```

Activity Selection

```
def max_meetings(start, end):
    meetings = sorted(zip(start, end), key=lambda x: x[1])
    count = 1
    last_end = meetings[0][1]

    for s, e in meetings[1:]:
        if s > last_end:
            count += 1
            last_end = e
    return count
```

8. BINARY SEARCH PATTERNS

Standard Binary Search

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Search in Rotated Array

```
def search_rotated(nums, target):
    left, right = 0, len(nums) - 1

    while left <= right:
        mid = left + (right - left) // 2

        if nums[mid] == target:
            return mid

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

Find Peak Element

```
def find_peak(arr):
    left, right = 0, len(arr) - 1
```

```
    while left < right:
        mid = left + (right - left) // 2
        if arr[mid] > arr[mid + 1]:
            right = mid
        else:
            left = mid + 1
    return left
```

9. BIT MANIPULATION

Common Operations

```
# Check if bit is set
def is_bit_set(num, i):
    return (num & (1 << i)) != 0

# Set bit
def set_bit(num, i):
    return num | (1 << i)

# Clear bit
def clear_bit(num, i):
    return num & ~(1 << i)

# Toggle bit
def toggle_bit(num, i):
    return num ^ (1 << i)

# Count set bits
def count_bits(n):
    count = 0
    while n:
        count += n & 1
        n >>= 1
    return count

# Or use built-in
count = bin(n).count('1')
```

XOR Tricks

```
# Find single number (all others appear twice)
```

```
def single_number(nums):
    result = 0
    for num in nums:
        result ^= num
    return result

# Swap without temp
a ^= b
b ^= a
a ^= b
```

QUICK PROBLEM IDENTIFICATION

Pattern	Keywords	Common Problems
Prefix Sum	“range sum”, “subarray sum”	Subarray sum equals K
Sliding Window	“contiguous”, “substring”, “subarray”	Longest substring, max sum subarray
Two Pointer	“sorted array”, “pairs”, “palindrome”	Two sum, container with most water
Fast/Slow Pointer	“cycle”, “middle”, “linked list”	Detect cycle, find middle
Hash Table	“frequency”, “count”, “duplicates”	Two sum, group anagrams
Stack	“next greater”, “valid parentheses”, “histogram”	Valid parentheses, largest rectangle
BFS	“shortest path”, “level order”	Shortest path, level traversal
DFS	“all paths”, “connected components”	Number of islands, path sum
DP	“maximum/minimum”, “count ways”, “optimal”	Coin change, longest substring
Greedy	“intervals”, “scheduling”, “maximum”	Merge intervals, activity selection
Binary Search	“sorted”, “search”, “find peak”	Search insert position, find minimum

CITADEL-SPECIFIC TIPS

1. **Prefix sums + hashing** shows up frequently
2. **Array manipulation** with constraints is common
3. **Stack problems** appear regularly
4. **Buy/sell stock** variations are favorites
5. Edge cases matter more than optimal solution
6. First submission should be correct - no time to debug