Algorithmic Patterns Cheat Sheet - C++ - Citadel HackerRank

1. ARRAY & STRING PATTERNS

Prefix Sum Pattern

When to use: Range sum queries, subarray sums

```
// Build prefix sum
vector<int> prefix(arr.size() + 1, 0);
for (int i = 0; i < arr.size(); i++) {
    prefix[i + 1] = prefix[i] + arr[i];
}

// Get sum from i to j (inclusive)
int range_sum = prefix[j + 1] - prefix[i];

// Example: Find subarrays with sum == k
int subarraySum(vector<int>& arr, int k) {
    unordered_map<int, int> sum_count;
    sum_count[0] = 1;
    int count = 0;
    int prefix_sum = 0;

    for (int num : arr) {
        prefix_sum += num;
        count += sum_count[prefix_sum - k];
        sum_count[prefix_sum]++;
    }
    return count;
}
```

Sliding Window Pattern

When to use: Contiguous subarray/substring problems with constraints

```
// Fixed size window
int fixedWindow(vector<int>& arr, int k) {
    int window_sum = 0;
```

```
    for (int i = 0; i < k; i++) {
        window_sum += arr[i];
    }

    int max_sum = window_sum;
    for (int i = k; i < arr.size(); i++) {
        window_sum += arr[i] - arr[i - k];
        max_sum = max(max_sum, window_sum);
    }
    return max_sum;
}

// Variable size window
int variableWindow(string s, int k) {
    int left = 0;
    unordered_map<char, int> char_count;
    int max_len = 0;

    for (int right = 0; right < s.length(); right++) {
        char_count[s[right]]++;

        // Shrink window if constraint violated
        while (char_count.size() > k) {
            char_count[s[left]]--;
            if (char_count[s[left]] == 0) {
                char_count.erase(s[left]);
            }
            left++;
        }

        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
vector<int> twoSumSorted(vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;

    while (left < right) {
        int current = arr[left] + arr[right];
        if (current == target) {
```

```cpp
            return {left, right};
        } else if (current < target) {
            left++;
        } else {
            right--;
        }
    }
    return {-1, -1};
}

// Remove duplicates in-place
int removeDuplicates(vector<int>& arr) {
    if (arr.empty()) return 0;

    int write = 1;
    for (int read = 1; read < arr.size(); read++) {
        if (arr[read] != arr[read - 1]) {
            arr[write++] = arr[read];
        }
    }
    return write;
}
```

Fast & Slow Pointer (Floyd's Cycle Detection)

When to use: Cycle detection, finding middle

```cpp
// Detect cycle
bool hasCycle(ListNode* head) {
    ListNode* slow = head;
    ListNode* fast = head;

    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) {
            return true;
        }
    }
    return false;
}

// Find middle
ListNode* findMiddle(ListNode* head) {
    ListNode* slow = head;
    ListNode* fast = head;
```

```
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}
```

2. HASH TABLE PATTERNS

Frequency Counter

```
#include <unordered_map>

// Count occurrences
unordered_map<int, int> freq;
for (int item : arr) {
    freq[item]++;
}

// Find elements with frequency > k
vector<int> result;
for (auto& [key, count] : freq) {
    if (count > k) {
        result.push_back(key);
    }
}
```

Index Mapping

```
// Two sum using hash
vector<int> twoSum(vector<int>& nums, int target) {
    unordered_map<int, int> seen;

    for (int i = 0; i < nums.size(); i++) {
        int complement = target - nums[i];
        if (seen.count(complement)) {
            return {seen[complement], i};
        }
        seen[nums[i]] = i;
    }
    return {};
}
```

Group Anagrams Pattern

```
vector<vector<string>> groupAnagrams(vector<string>& words) {
    unordered_map<string, vector<string>> groups;

    for (string& word : words) {
        string key = word;
        sort(key.begin(), key.end());
        groups[key].push_back(word);
    }

    vector<vector<string>> result;
    for (auto& [key, group] : groups) {
        result.push_back(group);
    }
    return result;
}
```

3. STACK & QUEUE PATTERNS

Monotonic Stack

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

```
// Next greater element
vector<int> nextGreater(vector<int>& arr) {
    vector<int> result(arr.size(), -1);
    stack<int> st; // Store indices

    for (int i = 0; i < arr.size(); i++) {
        while (!st.empty() && arr[st.top()] < arr[i]) {
            result[st.top()] = arr[i];
            st.pop();
        }
        st.push(i);
    }
    return result;
}

// Largest rectangle in histogram
int largestRectangle(vector<int>& heights) {
    stack<pair<int, int>> st; // {index, height}
    int max_area = 0;

    for (int i = 0; i < heights.size(); i++) {
```

```
        int start = i;
        while (!st.empty() && st.top().second > heights[i]) {
            auto [idx, height] = st.top();
            st.pop();
            max_area = max(max_area, height * (i - idx));
            start = idx;
        }
        st.push({start, heights[i]});
    }

    while (!st.empty()) {
        auto [i, h] = st.top();
        st.pop();
        max_area = max(max_area, h * (int)(heights.size() - i));
    }
    return max_area;
}
```

Queue with Two Stacks

```
class QueueWithStacks {
    stack<int> s1; // Push stack
    stack<int> s2; // Pop stack

public:
    void enqueue(int x) {
        s1.push(x);
    }

    int dequeue() {
        if (s2.empty()) {
            while (!s1.empty()) {
                s2.push(s1.top());
                s1.pop();
            }
        }
        if (s2.empty()) return -1;
        int val = s2.top();
        s2.pop();
        return val;
    }
};
```

4. TREE PATTERNS

DFS Traversals

```
// Inorder (Left-Root-Right)
void inorder(TreeNode* root, vector<int>& result) {
    if (!root) return;
    inorder(root->left, result);
    result.push_back(root->val);
    inorder(root->right, result);
}

// Preorder (Root-Left-Right)
void preorder(TreeNode* root, vector<int>& result) {
    if (!root) return;
    result.push_back(root->val);
    preorder(root->left, result);
    preorder(root->right, result);
}

// Postorder (Left-Right-Root)
void postorder(TreeNode* root, vector<int>& result) {
    if (!root) return;
    postorder(root->left, result);
    postorder(root->right, result);
    result.push_back(root->val);
}
```

BFS (Level Order)

```
#include <queue>

vector<vector<int>> levelOrder(TreeNode* root) {
    vector<vector<int>> result;
    if (!root) return result;

    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        int level_size = q.size();
        vector<int> level;

        for (int i = 0; i < level_size; i++) {
            TreeNode* node = q.front();
            q.pop();
            level.push_back(node->val);
```

```
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
        result.push_back(level);
    }
    return result;
}
```

Path Sum Problems

```
bool hasPathSum(TreeNode* root, int target) {
    if (!root) return false;

    if (!root->left && !root->right) {
        return root->val == target;
    }

    target -= root->val;
    return hasPathSum(root->left, target) || hasPathSum(root->right, target);
}
```

5. GRAPH PATTERNS

BFS (Shortest Path)

```
int bfsShortestPath(vector<vector<int>>& graph, int start, int end) {
    queue<pair<int, int>> q; // {node, distance}
    unordered_set<int> visited;

    q.push({start, 0});
    visited.insert(start);

    while (!q.empty()) {
        auto [node, dist] = q.front();
        q.pop();

        if (node == end) return dist;

        for (int neighbor : graph[node]) {
            if (!visited.count(neighbor)) {
                visited.insert(neighbor);
                q.push({neighbor, dist + 1});
```

```
            }
        }
    }
    return -1;
}
```

DFS (Connected Components)

```
void dfs(int node, vector<vector<int>>& graph, unordered_set<int>& visited) {
    visited.insert(node);
    for (int neighbor : graph[node]) {
        if (!visited.count(neighbor)) {
            dfs(neighbor, graph, visited);
        }
    }
}

int countComponents(int n, vector<pair<int, int>>& edges) {
    vector<vector<int>> graph(n);
    for (auto [u, v] : edges) {
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    unordered_set<int> visited;
    int count = 0;

    for (int i = 0; i < n; i++) {
        if (!visited.count(i)) {
            dfs(i, graph, visited);
            count++;
        }
    }
    return count;
}
```

Topological Sort (Kahn’s Algorithm)

```
vector<int> topologicalSort(int n, vector<pair<int, int>>& edges) {
    vector<vector<int>> graph(n);
    vector<int> indegree(n, 0);

    for (auto [u, v] : edges) {
        graph[u].push_back(v);
```

```
        indegree[v]++;
    }

    queue<int> q;
    for (int i = 0; i < n; i++) {
        if (indegree[i] == 0) {
            q.push(i);
        }
    }

    vector<int> result;
    while (!q.empty()) {
        int node = q.front();
        q.pop();
        result.push_back(node);

        for (int neighbor : graph[node]) {
            indegree[neighbor]--;
            if (indegree[neighbor] == 0) {
                q.push(neighbor);
            }
        }
    }

    return result.size() == n ? result : vector<int>();
}
```

6. DYNAMIC PROGRAMMING PATTERNS

1D DP

```
// Fibonacci-style
int climbStairs(int n) {
    if (n <= 2) return n;

    vector<int> dp(n + 1);
    dp[1] = 1;
    dp[2] = 2;

    for (int i = 3; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[n];
}
```

```
// Space-optimized
int climbStairsOptimized(int n) {
    if (n <= 2) return n;

    int prev2 = 1, prev1 = 2;
    for (int i = 3; i <= n; i++) {
        int curr = prev1 + prev2;
        prev2 = prev1;
        prev1 = curr;
    }
    return prev1;
}
```

2D DP (Grid)

```
// Unique paths
int uniquePaths(int m, int n) {
    vector<vector<int>> dp(m, vector<int>(n, 1));

    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
        }
    }
    return dp[m - 1][n - 1];
}
```

Knapsack Pattern

```
int knapsack(vector<int>& weights, vector<int>& values, int capacity) {
    int n = weights.size();
    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

    for (int i = 1; i <= n; i++) {
        for (int w = 0; w <= capacity; w++) {
            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
vector<vector<int>> mergeIntervals(vector<vector<int>>& intervals) {
    sort(intervals.begin(), intervals.end());
    vector<vector<int>> merged = {intervals[0]};

    for (int i = 1; i < intervals.size(); i++) {
        if (intervals[i][0] <= merged.back()[1]) {
            merged.back()[1] = max(merged.back()[1], intervals[i][1]);
        } else {
            merged.push_back(intervals[i]);
        }
    }
    return merged;
}

// Non-overlapping intervals
int eraseOverlap(vector<vector<int>>& intervals) {
    sort(intervals.begin(), intervals.end(),
         [](const vector<int>& a, const vector<int>& b) {
             return a[1] < b[1];
         });

    int end = INT_MIN;
    int count = 0;

    for (auto& interval : intervals) {
        if (interval[0] >= end) {
            count++;
            end = interval[1];
        }
    }
    return intervals.size() - count;
}
```

Activity Selection

```
int maxMeetings(vector<int>& start, vector<int>& end) {
    vector<pair<int, int>> meetings;
    for (int i = 0; i < start.size(); i++) {
        meetings.push_back({end[i], start[i]});
    }
    sort(meetings.begin(), meetings.end());

    int count = 1;
    int last_end = meetings[0].first;

    for (int i = 1; i < meetings.size(); i++) {
        if (meetings[i].second > last_end) {
            count++;
            last_end = meetings[i].first;
        }
    }
    return count;
}
```

8. BINARY SEARCH PATTERNS

Standard Binary Search

```
int binarySearch(vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
```

Search in Rotated Array

```
int searchRotated(vector<int>& nums, int target) {
    int left = 0, right = nums.size() - 1;
```

```
    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (nums[mid] == target) return mid;

        // Left half is sorted
        if (nums[left] <= nums[mid]) {
            if (nums[left] <= target && target < nums[mid]) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        // Right half is sorted
        else {
            if (nums[mid] < target && target <= nums[right]) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }
    return -1;
}
```

Find Peak Element

```
int findPeak(vector<int>& arr) {
    int left = 0, right = arr.size() - 1;

    while (left < right) {
        int 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
bool isBitSet(int num, int i) {
    return (num & (1 << i)) != 0;
}

// Set bit
int setBit(int num, int i) {
    return num | (1 << i);
}

// Clear bit
int clearBit(int num, int i) {
    return num & ~(1 << i);
}

// Toggle bit
int toggleBit(int num, int i) {
    return num ^ (1 << i);
}

// Count set bits
int countBits(int n) {
    int count = 0;
    while (n) {
        count += n & 1;
        n >>= 1;
    }
    return count;
}

// Or use built-in
int count = __builtin_popcount(n);
```

XOR Tricks

```
// Find single number (all others appear twice)
int singleNumber(vector<int>& nums) {
    int result = 0;
    for (int num : 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