Sliding Window with Array/Vector - Complete Mastery Guide

LeetCode, HackerRank, and Codeforces Problems with Solutions

Table of Contents

Chapter 1: Pattern Recognition

1.1 When to Use Sliding Window with Array

Recognition Signals

Keywords:

- "subarray" with **numeric constraints** (sum, product, average)
- "consecutive elements"
- "maximum/minimum sum of size K"
- "at most/at least N elements satisfying condition"
- "binary array" (0s and 1s)
- No need for frequency tracking (just values)

Array vs HashMap Decision

Use Array/Vector When:

- Simple value tracking (sum, count, min, max)
- No frequency map needed
- Working with numeric constraints
- Binary conditions (0/1, even/odd)

Use HashMap When:

- Need character/element frequencies
- Tracking distinct elements
- Pattern matching (anagrams, permutations)

1.2 Core Templates

Template 1: Fixed-Size Window

```
int fixedWindowArray(vector&lt;int&gt;&amp; arr, int k) {
    int n = arr.size();
    if (n &lt; k) return -1;

    // Build initial window
    int window_sum = 0;
    for (int i = 0; i &lt; k; i++) {
        window_sum += arr[i];
    }

    int result = window_sum;

    // Slide window
    for (int i = k; i &lt; n; i++) {
        // Add new element, remove old element
        window_sum = window_sum + arr[i] - arr[i - k];
        result = max(result, window_sum);
    }

    return result;
}
```

Time Complexity: O(n)
Space Complexity: O(1)

Template 2: Variable-Size Window

```
int variableWindowArray(vector&lt;int&gt;&amp; arr, int target) {
    int left = 0;
    int window_sum = 0;
    int max_len = 0;

    for (int right = 0; right &lt; arr.size(); right++) {
        // Expand window
        window_sum += arr[right];

        // Contract window while condition violated
        while (window_sum &gt; target &amp;&amp; left &lt;= right) {
            window_sum -= arr[left];
            left++;
        }
```

```
        // Update result
        max_len = max(max_len, right - left + 1);
    }

    return max_len;
}
```

Time Complexity: O(n)
Space Complexity: O(1)

Chapter 2: Complete Problem Solutions

Problem 1: Maximum Sum Subarray of Size K

Platforms:

- **LeetCode:** Not directly available (similar to 643)
- **HackerRank:** Maximum Subarray Sum
- **Codeforces:** Educational Round problems

Difficulty: Easy

Description: Given array and integer k, find maximum sum of any contiguous subarray of size k.

Examples:

```
Input: arr = [2, 1, 5, 1, 3, 2], k = 3
Output: 9
Explanation: Subarray [5, 1, 3] has maximum sum = 9

Input: arr = [2, 3, 4, 1, 5], k = 2
Output: 7
Explanation: Subarray [3, 4] has maximum sum = 7
```

Solution 1: Brute Force - O(n×k) time, O(1) space

```
int maxSumBruteForce(vector&lt;int&gt;&amp; arr, int k) {
    int n = arr.size();
    if (n &lt; k) return -1;

    int max_sum = INT_MIN;

    // Try every window of size k
    for (int i = 0; i &lt;= n - k; i++) {
        int current_sum = 0;

        // Calculate sum of current window
        for (int j = i; j &lt; i + k; j++) {
```

```
            current_sum += arr[j];
        }

        max_sum = max(max_sum, current_sum);
    }

    return max_sum;
}
```

Why Inefficient:

- Recalculates sum for overlapping elements
- Each window: O(k) operations
- Total: O(n × k)

Solution 2: Sliding Window - O(n) time, O(1) space ⭐

```
int maxSumSlidingWindow(vector&lt;int&gt;&amp; arr, int k) {
    int n = arr.size();
    if (n &lt; k) return -1;

    // Calculate sum of first window
    int window_sum = 0;
    for (int i = 0; i &lt; k; i++) {
        window_sum += arr[i];
    }

    int max_sum = window_sum;

    // Slide window from left to right
    for (int i = k; i &lt; n; i++) {
        // Slide window: add arr[i], remove arr[i-k]
        window_sum = window_sum + arr[i] - arr[i - k];
        max_sum = max(max_sum, window_sum);
    }

    return max_sum;
}
```

Detailed Walkthrough:

```
arr = [2, 1, 5, 1, 3, 2], k = 3

Initial window [2, 1, 5]:
  window_sum = 2 + 1 + 5 = 8
  max_sum = 8

i=3: Slide to [1, 5, 1]
  window_sum = 8 + arr[3] - arr[0] = 8 + 1 - 2 = 7
  max_sum = 8
```

```
i=4: Slide to [5, 1, 3]
  window_sum = 7 + arr[4] - arr[1] = 7 + 3 - 1 = 9
  max_sum = 9

i=5: Slide to [1, 3, 2]
  window_sum = 9 + arr[2] - arr[2] = 9 + 2 - 5 = 6
  max_sum = 9

Answer: 9
```

Problem 2: Minimum Size Subarray Sum (LeetCode 209)

Link: https://leetcode.com/problems/minimum-size-subarray-sum/

Difficulty: Medium

Description: Find minimal length of contiguous subarray with sum greater than or equal to target.

Examples:

```
Input: target = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: [4,3] is minimal subarray with sum 7

Input: target = 4, nums = [1,4,4]
Output: 1
Explanation: [4] has sum 4

Input: target = 11, nums = [1,1,1,1,1,1,1,1]
Output: 0
Explanation: No subarray has sum greater than or equal to 11
```

Solution: Variable-Size Sliding Window - O(n) time, O(1) space

```
class Solution {
public:
    int minSubArrayLen(int target, vector&lt;int&gt;&amp; nums) {
        int n = nums.size();
        int left = 0;
        int current_sum = 0;
        int min_len = INT_MAX;

        for (int right = 0; right &lt; n; right++) {
            // Expand: Add right element
            current_sum += nums[right];

            // Contract: Shrink while sum is still valid
            while (current_sum &gt;= target) {
                // Update minimum length
                min_len = min(min_len, right - left + 1);
```

```
                // Remove left element
                current_sum -= nums[left];
                left++;
            }
        }

        // Return 0 if no valid subarray found
        return (min_len == INT_MAX) ? 0 : min_len;
    }
};
```

Key Insight: Greedy shrinking - as soon as window is valid, try to minimize it.

Detailed Execution:

```
target = 7, nums = [2,3,1,2,4,3]

right=0: nums[0]=2, sum=2, sum &lt; 7, continue
right=1: nums[1]=3, sum=5, sum &lt; 7, continue
right=2: nums[2]=1, sum=6, sum &lt; 7, continue
right=3: nums[3]=2, sum=8, sum &gt;= 7 ✓
  Window [2,3,1,2], len=4, min_len=4
  Shrink: left=1, sum=6 (no longer valid, stop)

right=4: nums[4]=4, sum=10, sum &gt;= 7 ✓
  Window [3,1,2,4], len=4, min_len=4
  Shrink: left=2, sum=7 (still valid)
    Window [1,2,4], len=3, min_len=3
    Shrink: left=3, sum=6 (no longer valid, stop)

right=5: nums[5]=3, sum=9, sum &gt;= 7 ✓
  Window [2,4,3], len=3, min_len=3
  Shrink: left=4, sum=7 (still valid)
    Window [4,3], len=2, min_len=2
    Shrink: left=5, sum=3 (no longer valid, stop)

Answer: 2
```

Why O(n): Each element added once (right pointer) and removed at most once (left pointer).

Problem 3: Max Consecutive Ones III (LeetCode 1004)

Link: https://leetcode.com/problems/max-consecutive-ones-iii/

Difficulty: Medium

Description: Given binary array and integer k, return max consecutive 1s if you can flip at most k zeros.

Examples:

```
Input: nums = [1,1,1,0,0,0,1,1,1,1,0], k = 2
Output: 6
Explanation: Flip nums[5] and nums[10], get [1,1,1,0,0,1,1,1,1,1,1]

Input: nums = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], k = 3
Output: 10
Explanation: Can flip 3 zeros to get 10 consecutive ones
```

Solution: Sliding Window with Zero Count - O(n) time, O(1) space

```
class Solution {
public:
    int longestOnes(vector&lt;int&gt;&amp; nums, int k) {
        int left = 0;
        int zero_count = 0;
        int max_len = 0;

        for (int right = 0; right &lt; nums.size(); right++) {
            // Expand: Count zeros in window
            if (nums[right] == 0) {
                zero_count++;
            }

            // Contract: More than k zeros? Shrink window
            while (zero_count &gt; k) {
                if (nums[left] == 0) {
                    zero_count--;
                }
                left++;
            }

            // Update max length
            max_len = max(max_len, right - left + 1);
        }

        return max_len;
    }
};
```

Alternative Interpretation: Longest subarray with at most k zeros.

Walkthrough:

```
nums = [1,1,1,0,0,0,1,1,1,1,0], k = 2

right=0: nums[0]=1, zeros=0, valid, max_len=1
right=1: nums[1]=1, zeros=0, valid, max_len=2
right=2: nums[2]=1, zeros=0, valid, max_len=3
right=3: nums[3]=0, zeros=1, valid, max_len=4
right=4: nums[4]=0, zeros=2, valid, max_len=5
right=5: nums[5]=0, zeros=3, INVALID (3 &gt; 2)
```

```
  Shrink: left=0, nums[0]=1, zeros=3
  Shrink: left=1, nums[1]=1, zeros=3
  Shrink: left=2, nums[2]=1, zeros=3
  Shrink: left=3, nums[3]=0, zeros=2, VALID
  Window [0,0,0], len=3, max_len=5

right=6: nums[6]=1, zeros=2, valid, max_len=5
right=7: nums[7]=1, zeros=2, valid, max_len=5
right=8: nums[8]=1, zeros=2, valid, max_len=6
right=9: nums[9]=1, zeros=2, valid, max_len=7
right=10: nums[10]=0, zeros=3, INVALID
  Shrink until zeros=2
  ...

Final: max_len=6
```

Problem 4: Grumpy Bookstore Owner (LeetCode 1052)

Link: https://leetcode.com/problems/grumpy-bookstore-owner/

Difficulty: Medium

Description: Bookstore owner can suppress grumpiness for k minutes. Maximize satisfied customers.

Example:

```
Input: customers = [1,0,1,2,1,1,7,5], grumpy = [0,1,0,1,0,1,0,1], k = 3
Output: 16

Explanation:
- Without suppression: 1+1+1+5 = 8 satisfied
- Suppress minutes [3,4,5]: gain 2+1+1 = 4 more
- Total: 8+4 = 12
- Best is suppressing [5,6,7]: 1+7+5 = 13 gained
- Total: 1+1+1+13 = 16
```

Solution: Fixed Window to Find Best Suppression Period - O(n) time, O(1) space

```
class Solution {
public:
    int maxSatisfied(vector&lt;int&gt;&amp; customers, vector&lt;int&gt;&amp; grumpy, int
        int n = customers.size();

        // Calculate base satisfaction (when not grumpy)
        int base_satisfied = 0;
        for (int i = 0; i &lt; n; i++) {
            if (grumpy[i] == 0) {
                base_satisfied += customers[i];
            }
        }
```

```
        // Find best window of size k to suppress grumpiness
        int additional = 0;  // Additional customers if we suppress this window

        // Build initial window
        for (int i = 0; i &lt; k; i++) {
            if (grumpy[i] == 1) {
                additional += customers[i];
            }
        }

        int max_additional = additional;

        // Slide window
        for (int i = k; i &lt; n; i++) {
            // Add new customer if grumpy
            if (grumpy[i] == 1) {
                additional += customers[i];
            }

            // Remove old customer if was grumpy
            if (grumpy[i - k] == 1) {
                additional -= customers[i - k];
            }

            max_additional = max(max_additional, additional);
        }

        return base_satisfied + max_additional;
    }
};
```

Strategy:

1. Calculate baseline (customers satisfied when owner not grumpy)
2. Find window of size k that captures most "lost" customers (grumpy times)
3. Add baseline + best window gain

Problem 5: Maximum Points You Can Obtain from Cards (LeetCode 1423)

Link: https://leetcode.com/problems/maximum-points-you-can-obtain-from-cards/

Difficulty: Medium

Description: Take k cards from either end of array. Maximize sum.

Examples:

```
Input: cardPoints = [1,2,3,4,5,6,1], k = 3
Output: 12
Explanation: Take 3 from right: 6+5+1 = 12

Input: cardPoints = [2,2,2], k = 2
```

```
Output: 4

Input: cardPoints = [9,7,7,9,7,7,9], k = 7
Output: 55
```

Solution 1: Try All Combinations - O(k) time, O(1) space

```
class Solution {
public:
    int maxScore(vector&lt;int&gt;&amp; cardPoints, int k) {
        int n = cardPoints.size();

        // Calculate sum taking all k from left
        int current_sum = 0;
        for (int i = 0; i &lt; k; i++) {
            current_sum += cardPoints[i];
        }

        int max_sum = current_sum;

        // Try taking i from left, k-i from right
        for (int i = 0; i &lt; k; i++) {
            // Remove one from left, add one from right
            current_sum -= cardPoints[k - 1 - i];
            current_sum += cardPoints[n - 1 - i];

            max_sum = max(max_sum, current_sum);
        }

        return max_sum;
    }
};
```

Walkthrough:

```
cardPoints = [1,2,3,4,5,6,1], k = 3

Initial: Take 3 from left [1,2,3], sum = 6

i=0: Remove 3, add 1 → [1,2,1], sum = 6-3+1 = 4
i=1: Remove 2, add 6 → [1,6,1], sum = 4-2+6 = 8
i=2: Remove 1, add 5 → [5,6,1], sum = 8-1+5 = 12

max_sum = 12
```

Solution 2: Inverse Problem (Minimum Subarray) - O(n) time, O(1) space

Key Insight: Maximize k cards from ends = Minimize middle (n-k) cards

```
class Solution {
public:
    int maxScore(vector&lt;int&gt;&amp; cardPoints, int k) {
        int n = cardPoints.size();
        int total_sum = 0;

        // Calculate total sum
        for (int num : cardPoints) {
            total_sum += num;
        }

        // Find minimum sum of subarray of length (n-k)
        int window_size = n - k;

        if (window_size == 0) return total_sum;

        int window_sum = 0;
        for (int i = 0; i &lt; window_size; i++) {
            window_sum += cardPoints[i];
        }

        int min_sum = window_sum;

        for (int i = window_size; i &lt; n; i++) {
            window_sum += cardPoints[i] - cardPoints[i - window_size];
            min_sum = min(min_sum, window_sum);
        }

        return total_sum - min_sum;
    }
};
```

Why This Works:

- Total = k cards from ends + middle (n-k) cards
- Maximize k cards = Minimize middle cards
- Use sliding window to find minimum middle

Problem 6: Subarray Product Less Than K (LeetCode 713)

Link: https://leetcode.com/problems/subarray-product-less-than-k/

Difficulty: Medium

Description: Count subarrays where product of all elements is strictly less than k.

Examples:

```
Input: nums = [10,5,2,6], k = 100
Output: 8
Explanation: [10], [5], [2], [6], [10,5], [5,2], [2,6], [5,2,6]

Input: nums = [1,2,3], k = 0
Output: 0
```

Solution: Sliding Window with Product Tracking - O(n) time, O(1) space

```
class Solution {
public:
    int numSubarrayProductLessThanK(vector&lt;int&gt;&amp; nums, int k) {
        if (k &lt;= 1) return 0;  // Product always greater than or equal to 1

        int left = 0;
        int product = 1;
        int count = 0;

        for (int right = 0; right &lt; nums.size(); right++) {
            // Expand: Multiply by right element
            product *= nums[right];

            // Contract: Shrink while product too large
            while (product &gt;= k) {
                product /= nums[left];
                left++;
            }

            // Count subarrays ending at right
            // [left...right], [left+1...right], ..., [right]
            count += right - left + 1;
        }

        return count;
    }
};
```

Why `right - left + 1`?

When window [left...right] is valid, ALL subarrays ending at right are valid:

- [right] → 1 subarray
- [right-1, right] → if valid
- [right-2, right-1, right] → if valid
- ...
- [left, ..., right] → if valid

Total = `right - left + 1` new subarrays

Example:

```
nums = [10,5,2,6], k = 100

right=0: nums[0]=10, product=10
  Valid (10 &lt; 100)
  count += 0-0+1 = 1  ([10])

right=1: nums[1]=5, product=50
  Valid (50 &lt; 100)
  count += 1-0+1 = 2  ([10,5], [5])
  total = 3

right=2: nums[2]=2, product=100
  Invalid (100 &gt;= 100)
  Shrink: product = 100/10 = 10, left=1
  Valid (10 &lt; 100)
  count += 2-1+1 = 2  ([5,2], [2])
  total = 5

right=3: nums[3]=6, product=60
  Valid (60 &lt; 100)
  count += 3-1+1 = 3  ([5,2,6], [2,6], [6])
  total = 8

Answer: 8
```

Problem 7: Longest Subarray of 1s After Deleting One Element (LeetCode 1493)

Link: https://leetcode.com/problems/longest-subarray-of-1s-after-deleting-one-element/

Difficulty: Medium

Description: Delete exactly one element, find longest subarray of 1s.

Examples:

```
Input: nums = [1,1,0,1]
Output: 3
Explanation: Delete 0, get [1,1,1]

Input: nums = [0,1,1,1,0,1,1,0,1]
Output: 5
Explanation: Delete first 0, get [1,1,1,1,1]

Input: nums = [1,1,1]
Output: 2
Explanation: Must delete one 1, get [1,1]
```

Solution: At Most 1 Zero Allowed - O(n) time, O(1) space

```
class Solution {
public:
    int longestSubarray(vector&lt;int&gt;&amp; nums) {
        int left = 0;
        int zero_count = 0;
        int max_len = 0;

        for (int right = 0; right &lt; nums.size(); right++) {
            // Count zeros
            if (nums[right] == 0) {
                zero_count++;
            }

            // More than 1 zero? Shrink
            while (zero_count &gt; 1) {
                if (nums[left] == 0) {
                    zero_count--;
                }
                left++;
            }

            // Update max (subtract 1 because we must delete one element)
            max_len = max(max_len, right - left + 1 - 1);
        }

        return max_len;
    }
};
```

Key Difference from Max Consecutive Ones III:

- Here we allow at most 1 zero (the element to delete)
- Subtract 1 from window length (must delete one element)

Problem 8: Number of Sub-arrays of Size K and Average ≥ Threshold (LeetCode 1343)

Link: https://leetcode.com/problems/number-of-sub-arrays-of-size-k-and-average-greater-than-or-equal-to-threshold/

Difficulty: Medium

Description: Count subarrays of size k with average greater than or equal to threshold.

Example:

```
Input: arr = [2,2,2,2,5,5,5,8], k = 3, threshold = 4
Output: 3
Explanation: [2,5,5], [5,5,5], [5,5,8] have averages 4, 5, 6 respectively
```

Solution: Fixed Window with Sum Check - O(n) time, O(1) space

```
class Solution {
public:
    int numOfSubarrays(vector&lt;int&gt;&amp; arr, int k, int threshold) {
        int target_sum = k * threshold;  // Avoid division
        int window_sum = 0;
        int count = 0;

        // Build initial window
        for (int i = 0; i &lt; k; i++) {
            window_sum += arr[i];
        }

        if (window_sum &gt;= target_sum) {
            count++;
        }

        // Slide window
        for (int i = k; i &lt; arr.size(); i++) {
            window_sum += arr[i] - arr[i - k];

            if (window_sum &gt;= target_sum) {
                count++;
            }
        }

        return count;
    }
};
```

Optimization Trick: Instead of `window_sum / k greater than or equal to threshold`, use `window_sum greater than or equal to k * threshold` to avoid floating-point.

Problem 9: Maximum Average Subarray II (LeetCode 644) - PREMIUM

Alternative: HackerRank - Maximum Average Subarray

Difficulty: Hard

Description: Find maximum average of contiguous subarray with length greater than or equal to k.

Solution: Binary Search + Sliding Window - O(n log(max-min)) time

```
class Solution {
public:
    double findMaxAverage(vector&lt;int&gt;&amp; nums, int k) {
        double min_val = *min_element(nums.begin(), nums.end());
        double max_val = *max_element(nums.begin(), nums.end());
```

```
        double epsilon = 1e-5;

        // Binary search on answer
        while (max_val - min_val &gt; epsilon) {
            double mid = (min_val + max_val) / 2;

            if (canAchieveAverage(nums, k, mid)) {
                min_val = mid;  // Try larger average
            } else {
                max_val = mid;  // Try smaller average
            }
        }

        return min_val;
    }

private:
    bool canAchieveAverage(vector&lt;int&gt;&amp; nums, int k, double target) {
        // Transform: subtract target from each element
        // Find if there exists subarray of length greater than or equal to k with sum gr

        int n = nums.size();
        vector&lt;double&gt; transformed(n);
        for (int i = 0; i &lt; n; i++) {
            transformed[i] = nums[i] - target;
        }

        double current_sum = 0;
        double prev_sum = 0;
        double min_prev_sum = 0;

        for (int i = 0; i &lt; n; i++) {
            current_sum += transformed[i];

            if (i &gt;= k - 1) {
                // Check if we can achieve sum greater than or equal to 0
                if (current_sum &gt;= min_prev_sum) {
                    return true;
                }

                // Update minimum prefix sum
                prev_sum += transformed[i - k + 1];
                min_prev_sum = min(min_prev_sum, prev_sum);
            }
        }

        return false;
    }
};
```

Problem 10: Diet Plan Performance (LeetCode 1176) - PREMIUM

Similar: Codeforces Problem Subarray Sum

Difficulty: Easy-Medium

Description: Count days where k-day sum is lower, upper, or in-between.

Solution: Sliding Window with Threshold Comparison - O(n) time

```
int dietPlanPerformance(vector&lt;int&gt;&amp; calories, int k, int lower, int upper) {
    int points = 0;
    int window_sum = 0;

    // Build initial window
    for (int i = 0; i &lt; k; i++) {
        window_sum += calories[i];
    }

    // Check first window
    if (window_sum &lt; lower) points--;
    else if (window_sum &gt; upper) points++;

    // Slide window
    for (int i = k; i &lt; calories.size(); i++) {
        window_sum += calories[i] - calories[i - k];

        if (window_sum &lt; lower) points--;
        else if (window_sum &gt; upper) points++;
    }

    return points;
}
```

This comprehensive textbook continues with 5+ more problems, HackerRank and Codeforces problem mappings, complexity comparisons, and practice progression plans.