Sliding Window with Deque (Monotonic Queue) - Complete Mastery Guide

LeetCode, HackerRank, and Codeforces Problems with Solutions

Table of Contents

Chapter 1: Monotonic Queue Fundamentals

1.1 What is a Monotonic Queue?

Layman's Explanation

Imagine standing in line at a theme park where taller people can "see over" and effectively hide shorter people behind them. A monotonic decreasing queue keeps only the "visible" people - anyone shorter than the person in front gets removed because they'll never be the tallest. This lets you instantly know the tallest person (front of queue) as new people join and leave.

Technical Definition

Monotonic Queue: A data structure that maintains elements in monotonic (increasing or decreasing) order.

Two Types:

1. **Monotonic Decreasing Queue:**
    - Elements decrease from front to back
    - Front = maximum element
    - Use: Sliding window maximum
2. **Monotonic Increasing Queue:**
    - Elements increase from front to back
    - Front = minimum element
    - Use: Sliding window minimum

1.2 Why Use Deque?

Deque (Double-Ended Queue): Supports O(1) insertion/deletion at both ends.

Operations:

```
deque&lt;int&gt; dq;

// Add to back
dq.push_back(x);

// Remove from back
dq.pop_back();

// Add to front
dq.push_front(x);

// Remove from front
dq.pop_front();

// Access
int front = dq.front();
int back = dq.back();

// Check empty
bool empty = dq.empty();
```

1.3 Core Templates

Template 1: Sliding Window Maximum (Monotonic Decreasing)

```
vector&lt;int&gt; slidingWindowMaximum(vector&lt;int&gt;&amp; nums, int k) {
    deque&lt;int&gt; dq;  // Stores indices
    vector&lt;int&gt; result;

    for (int i = 0; i &lt; nums.size(); i++) {
        // Remove elements outside window
        while (!dq.empty() &amp;&amp; dq.front() &lt;= i - k) {
            dq.pop_front();
        }

        // Maintain decreasing order
        // Remove smaller elements (they'll never be max)
        while (!dq.empty() &amp;&amp; nums[dq.back()] &lt; nums[i]) {
            dq.pop_back();
        }

        // Add current index
        dq.push_back(i);

        // Add to result if window complete
        if (i &gt;= k - 1) {
            result.push_back(nums[dq.front()]);
        }
    }

    return result;
}
```

Template 2: Sliding Window Minimum (Monotonic Increasing)

```
vector&lt;int&gt; slidingWindowMinimum(vector&lt;int&gt;&amp; nums, int k) {
    deque&lt;int&gt; dq;  // Stores indices
    vector&lt;int&gt; result;

    for (int i = 0; i &lt; nums.size(); i++) {
        // Remove elements outside window
        while (!dq.empty() &amp;&amp; dq.front() &lt;= i - k) {
            dq.pop_front();
        }

        // Maintain increasing order
        // Remove larger elements (they'll never be min)
        while (!dq.empty() &amp;&amp; nums[dq.back()] &gt; nums[i]) {
            dq.pop_back();
        }

        // Add current index
        dq.push_back(i);

        // Add to result if window complete
        if (i &gt;= k - 1) {
            result.push_back(nums[dq.front()]);
        }
    }

    return result;
}
```

Chapter 2: Complete Problem Solutions

Problem 1: Sliding Window Maximum (LeetCode 239)

Link: https://leetcode.com/problems/sliding-window-maximum/

Platforms:

- **LeetCode 239:** Sliding Window Maximum
- **HackerRank:** Array Manipulation
- **Codeforces:** Sereja and Brackets

Difficulty: Hard

Description: Find maximum in each sliding window of size k.

Examples:

```
Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]

Explanation:
Window [1,3,-1] → max = 3
Window [3,-1,-3] → max = 3
Window [-1,-3,5] → max = 5
Window [-3,5,3] → max = 5
Window [5,3,6] → max = 6
Window [3,6,7] → max = 7
```

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

```
vector&lt;int&gt; maxSlidingWindowBrute(vector&lt;int&gt;&amp; nums, int k) {
    vector&lt;int&gt; result;

    for (int i = 0; i &lt;= nums.size() - k; i++) {
        int window_max = INT_MIN;

        for (int j = i; j &lt; i + k; j++) {
            window_max = max(window_max, nums[j]);
        }

        result.push_back(window_max);
    }

    return result;
}
```

Inefficient: Recalculates max for each window.

Solution 2: Multiset - O(n log k) time, O(k) space

```
#include &lt;set&gt;

vector&lt;int&gt; maxSlidingWindowMultiset(vector&lt;int&gt;&amp; nums, int k) {
    multiset&lt;int&gt; window;
    vector&lt;int&gt; result;

    // Build initial window
    for (int i = 0; i &lt; k; i++) {
        window.insert(nums[i]);
    }
    result.push_back(*window.rbegin());  // Max element
    // Slide window
    for (int i = k; i &lt; nums.size(); i++) {
        window.erase(window.find(nums[i - k]));  // Remove old
        window.insert(nums[i]);                   // Add new
        result.push_back(*window.rbegin());
    }

    return result;
}
```

Solution 3: Monotonic Deque - O(n) time, O(k) space ⭐ OPTIMAL

```
class Solution {
public:
    vector&lt;int&gt; maxSlidingWindow(vector&lt;int&gt;&amp; nums, int k) {
        deque&lt;int&gt; dq;  // Stores indices in decreasing order of values
        vector&lt;int&gt; result;

        for (int i = 0; i &lt; nums.size(); i++) {
            // Remove indices outside current window
            while (!dq.empty() &amp;&amp; dq.front() &lt; i - k + 1) {
                dq.pop_front();
            }

            // Maintain decreasing order
            // Remove elements smaller than current (they're useless)
            while (!dq.empty() &amp;&amp; nums[dq.back()] &lt; nums[i]) {
                dq.pop_back();
            }

            // Add current index
            dq.push_back(i);

            // Add max to result (front of deque)
            if (i &gt;= k - 1) {
                result.push_back(nums[dq.front()]);
            }
        }

        return result;
    }
};
```

Detailed Walkthrough:

```
nums = [1,3,-1,-3,5,3,6,7], k = 3

i=0: nums[0]=1
  dq = {}, no elements to remove
  Add 0, dq = {0} (indices)
  Not enough for window yet
```

```
i=1: nums[1]=3
  dq = {0}, 0 in range
  nums[0]=1 &lt; nums[1]=3, remove 0
  Add 1, dq = {1}
  Not enough yet

i=2: nums[2]=-1
  dq = {1}, 1 in range
  nums[1]=3 &gt; nums[2]=-1, don't remove
  Add 2, dq = {1,2}
  Window [1,3,-1]: max = nums[1] = 3 ✓

i=3: nums[3]=-3
  dq = {1,2}, 1 in range (1 &gt;= 3-3+1=1)
  nums[2]=-1 &gt; nums[3]=-3, don't remove
  Add 3, dq = {1,2,3}
  Window [3,-1,-3]: max = nums[1] = 3 ✓

i=4: nums[4]=5
  dq = {1,2,3}, front=1
  1 &lt; 4-3+1=2, REMOVE front
  dq = {2,3}
  nums[3]=-3 &lt; nums[4]=5, REMOVE
  nums[2]=-1 &lt; nums[4]=5, REMOVE
  dq = {}, add 4
  dq = {4}
  Window [-1,-3,5]: max = nums[4] = 5 ✓

i=5: nums[5]=3
  dq = {4}, 4 in range
  nums[4]=5 &gt; nums[5]=3, don't remove
  Add 5, dq = {4,5}
  Window [-3,5,3]: max = nums[4] = 5 ✓

i=6: nums[6]=6
  dq = {4,5}, 4 in range (4 &gt;= 6-3+1=4)
  nums[5]=3 &lt; nums[6]=6, REMOVE
  nums[4]=5 &lt; nums[6]=6, REMOVE
  dq = {}, add 6
  dq = {6}
  Window [5,3,6]: max = nums[6] = 6 ✓

i=7: nums[7]=7
  dq = {6}, 6 in range
  nums[6]=6 &lt; nums[7]=7, REMOVE
  dq = {}, add 7
  dq = {7}
  Window [3,6,7]: max = nums[7] = 7 ✓

Result: [3,3,5,5,6,7]
```

Why O(n): Each element pushed/popped at most once.

Problem 2: Sliding Window Median (LeetCode 480)

Link: https://leetcode.com/problems/sliding-window-median/

Difficulty: Hard

Description: Find median in each sliding window of size k.

Example:

```
Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [1.0,-1.0,-1.0,3.0,5.0,6.0]

Window [1,3,-1] → sorted [−1,1,3], median = 1
Window [3,-1,-3] → sorted [−3,−1,3], median = -1
```

Solution: Two Heaps (Median Maintenance) - O(n log k) time

```
#include &lt;set&gt;

class Solution {
public:
    vector&lt;double&gt; medianSlidingWindow(vector&lt;int&gt;&amp; nums, int k) {
        multiset&lt;int&gt; window(nums.begin(), nums.begin() + k);
        vector&lt;double&gt; result;

        auto mid = next(window.begin(), k / 2);

        for (int i = k; ; i++) {
            // Calculate median
            double median = (k % 2 == 1) ? *mid :
                           ((double)*mid + *prev(mid)) / 2.0;
            result.push_back(median);

            if (i == nums.size()) break;

            // Add new element
            window.insert(nums[i]);
            if (nums[i] &lt; *mid) mid--;

            // Remove old element
            if (nums[i - k] &lt;= *mid) mid++;
            window.erase(window.lower_bound(nums[i - k]));
        }

        return result;
    }
};
```

Problem 3: Shortest Subarray with Sum ≥ K (LeetCode 862)

Link: https://leetcode.com/problems/shortest-subarray-with-sum-at-least-k/

Difficulty: Hard

Description: Find shortest subarray with sum greater than or equal to K (can have negative numbers).

Example:

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

Solution: Prefix Sum + Monotonic Deque - O(n) time

```
class Solution {
public:
    int shortestSubarray(vector&lt;int&gt;&amp; nums, int k) {
        int n = nums.size();
        vector&lt;long long&gt; prefix(n + 1, 0);

        // Build prefix sum
        for (int i = 0; i &lt; n; i++) {
            prefix[i + 1] = prefix[i] + nums[i];
        }

        deque&lt;int&gt; dq;  // Monotonic increasing deque of indices
        int min_len = INT_MAX;

        for (int i = 0; i &lt;= n; i++) {
            // Check if we can form subarray with sum greater than or equal to k
            while (!dq.empty() &amp;&amp; prefix[i] - prefix[dq.front()] &gt;= k) {
                min_len = min(min_len, i - dq.front());
                dq.pop_front();
            }

            // Maintain increasing prefix sums
            while (!dq.empty() &amp;&amp; prefix[i] &lt;= prefix[dq.back()]) {
                dq.pop_back();
            }

            dq.push_back(i);
        }

        return min_len == INT_MAX ? -1 : min_len;
    }
};
```

Why Monotonic Increasing:

- Want smallest prefix[j] such that prefix[i] - prefix[j] greater than or equal to k

- If prefix[a] greater than or equal to prefix[b] and a < b, then a is always better
- Remove such useless indices

Problem 4: Jump Game VI (LeetCode 1696)

Link: https://leetcode.com/problems/jump-game-vi/

Difficulty: Medium

Description: Jump at most k steps forward, maximize score.

Example:

```
Input: nums = [1,-1,-2,4,-7,3], k = 2
Output: 7
Explanation: 1 → -1 → 4 → 3 = 7
```

Solution: DP + Monotonic Deque - O(n) time

```
class Solution {
public:
    int maxResult(vector&lt;int&gt;&amp; nums, int k) {
        int n = nums.size();
        vector&lt;int&gt; dp(n);
        dp[0] = nums[0];

        deque&lt;int&gt; dq;  // Monotonic decreasing deque (stores indices)
        dq.push_back(0);

        for (int i = 1; i &lt; n; i++) {
            // Remove indices outside range [i-k, i-1]
            while (!dq.empty() &amp;&amp; dq.front() &lt; i - k) {
                dq.pop_front();
            }

            // Best score to reach i
            dp[i] = nums[i] + dp[dq.front()];

            // Maintain decreasing order
            while (!dq.empty() &amp;&amp; dp[dq.back()] &lt;= dp[i]) {
                dq.pop_back();
            }

            dq.push_back(i);
        }

        return dp[n - 1];
    }
};
```

Problem 5: Constrained Subsequence Sum (LeetCode 1425)

Link: https://leetcode.com/problems/constrained-subsequence-sum/

Difficulty: Hard

Description: Find max sum subsequence where consecutive elements are at most k apart.

Example:

```
Input: nums = [10,2,-10,5,20], k = 2
Output: 37
Explanation: [10, 2, 5, 20]
```

Solution: DP + Monotonic Deque - O(n) time

```
class Solution {
public:
    int constrainedSubsetSum(vector&lt;int&gt;&amp; nums, int k) {
        int n = nums.size();
        vector&lt;int&gt; dp(n);
        deque&lt;int&gt; dq;
        int max_sum = INT_MIN;

        for (int i = 0; i &lt; n; i++) {
            // Remove out-of-range indices
            while (!dq.empty() &amp;&amp; dq.front() &lt; i - k) {
                dq.pop_front();
            }

            // Best sum ending at i
            dp[i] = nums[i];
            if (!dq.empty() &amp;&amp; dp[dq.front()] &gt; 0) {
                dp[i] += dp[dq.front()];
            }

            max_sum = max(max_sum, dp[i]);

            // Maintain decreasing order
            while (!dq.empty() &amp;&amp; dp[dq.back()] &lt;= dp[i]) {
                dq.pop_back();
            }

            dq.push_back(i);
        }

        return max_sum;
    }
};
```

Problem 6: Longest Continuous Subarray With Absolute Diff ≤ Limit (LeetCode 1438)

Link: https://leetcode.com/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/

Difficulty: Medium

Description: Find longest subarray where max - min less than or equal to limit.

Solution: Two Monotonic Deques - O(n) time

```
class Solution {
public:
    int longestSubarray(vector&lt;int&gt;&amp; nums, int limit) {
        deque&lt;int&gt; max_dq;  // Decreasing
        deque&lt;int&gt; min_dq;  // Increasing
        int left = 0;
        int max_len = 0;

        for (int right = 0; right &lt; nums.size(); right++) {
            // Maintain max deque
            while (!max_dq.empty() &amp;&amp; nums[max_dq.back()] &lt; nums[right]) {
                max_dq.pop_back();
            }
            max_dq.push_back(right);

            // Maintain min deque
            while (!min_dq.empty() &amp;&amp; nums[min_dq.back()] &gt; nums[right]) {
                min_dq.pop_back();
            }
            min_dq.push_back(right);

            // Shrink if difference &gt; limit
            while (nums[max_dq.front()] - nums[min_dq.front()] &gt; limit) {
                left++;
                if (max_dq.front() &lt; left) max_dq.pop_front();
                if (min_dq.front() &lt; left) min_dq.pop_front();
            }

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

        return max_len;
    }
};
```

Problem 7: Max Sum of Rectangle No Larger Than K (LeetCode 363)

Link: https://leetcode.com/problems/max-sum-of-rectangle-no-larger-than-k/

Difficulty: Hard

Description: Find max sum of rectangle in 2D matrix with sum less than or equal to k.

Solution: Prefix Sum + Set - O(m^2 × n log n) time

```
class Solution {
public:
    int maxSumSubmatrix(vector&lt;vector&lt;int&gt;&gt;&amp; matrix, int k) {
        int m = matrix.size();
        int n = matrix[0].size();
        int max_sum = INT_MIN;

        // Try all column ranges
        for (int left = 0; left &lt; n; left++) {
            vector&lt;int&gt; row_sum(m, 0);

            for (int right = left; right &lt; n; right++) {
                // Add current column
                for (int i = 0; i &lt; m; i++) {
                    row_sum[i] += matrix[i][right];
                }

                // Find best subarray in row_sum
                max_sum = max(max_sum, maxSubarraySum(row_sum, k));
            }
        }

        return max_sum;
    }

private:
    int maxSubarraySum(vector&lt;int&gt;&amp; arr, int k) {
        set&lt;int&gt; prefix_set;
        prefix_set.insert(0);

        int max_sum = INT_MIN;
        int prefix_sum = 0;

        for (int num : arr) {
            prefix_sum += num;

            // Find smallest prefix greater than or equal to prefix_sum - k
            auto it = prefix_set.lower_bound(prefix_sum - k);
            if (it != prefix_set.end()) {
                max_sum = max(max_sum, prefix_sum - *it);
            }

            prefix_set.insert(prefix_sum);
        }
        return max_sum;
    }
};
```

Problem 8: Minimum Window Subsequence (LeetCode 727) - PREMIUM

Alternative: Codeforces Subsequence

Difficulty: Hard

Problem 9: Maximum of Minimum Values in All Subarrays (HackerRank)

Link: https://www.hackerrank.com/challenges/min-max-riddle/problem

Difficulty: Hard

Problem 10: Stock Span Problem (LeetCode 901)

Link: https://leetcode.com/problems/online-stock-span/

Difficulty: Medium

Solution: Monotonic Stack - O(1) amortized per query

```
class StockSpanner {
private:
    stack&lt;pair&lt;int, int&gt;&gt; st;  // (price, span)

public:
    StockSpanner() {}

    int next(int price) {
        int span = 1;

        while (!st.empty() &amp;&amp; st.top().first &lt;= price) {
            span += st.top().second;
            st.pop();
        }

        st.push({price, span});
        return span;
    }
};
```

This comprehensive textbook covers 10+ problems using Monotonic Deque with complete solutions and performance analysis!