

Sliding Window with HashMap - Complete Mastery Guide

From Theory to LeetCode Excellence

Table of Contents

1. Pattern Recognition and Theory
2. Core Template and Variations
3. Problem Classification System
4. Complete Problem Solutions (20+ Problems)
5. Difficulty Progression Roadmap
6. Interview Strategies
7. Practice Tracking System

Chapter 1: Pattern Recognition and Theory

1.1 When to Use Sliding Window with HashMap

Recognition Signals

Keywords that indicate this pattern:

- "substring" / "subarray" with specific **character/element constraints**
- "at most K **distinct** characters/elements"
- "exactly K" occurrences
- "contains all characters of"
- "minimum/maximum window"
- "anagram" / "permutation"
- "frequency" constraints

HashMap's Role in Sliding Window

Why HashMap?

- Track **character/element frequencies** within window
- **O(1)** insertion, deletion, lookup (average case)
- Enables **dynamic constraint checking** as window slides

Space Complexity:

- $O(k)$ where k = alphabet size (26 for lowercase, 52 for mixed, 128 for ASCII, 256 for extended ASCII)
- Often considered **$O(1)$** when alphabet is fixed

1.2 Core Template

Template 1: Variable-Size Window with HashMap

```
int slidingWindowHashMap(string s, constraints) {
    unordered_map<char, int> window_freq; // Track frequencies
    int left = 0;
    int result = 0; // Or INT_MAX for minimum
    int valid_count = 0; // Tracks if window satisfies constraints

    for (int right = 0; right < s.length(); right++) {
        // EXPAND: Add right character to window
        char c = s[right];
        window_freq[c]++;

        // Update valid_count based on problem requirements
        if (satisfies_condition(c, window_freq)) {
            valid_count++;
        }

        // CONTRACT: Shrink window while condition violated/satisfied
        while (window_invalid_or_optimal(valid_count, left, right)) {
            // Update result before shrinking (if finding minimum)
            result = min(result, right - left + 1);

            // Remove left character
            char left_char = s[left];
            window_freq[left_char]--;

            if (window_freq[left_char] == 0) {
                window_freq.erase(left_char);
            }

            // Update valid_count
            if (breaks_condition(left_char)) {
                valid_count--;
            }

            left++;
        }

        // UPDATE: Track result after expansion (if finding maximum)
        result = max(result, right - left + 1);
    }
}
```

```
    return result;
}
```

Chapter 2: Problem Classification System

2.1 Problem Categories

Category 1: Fixed Constraints (K distinct elements)

- Longest substring with at most K distinct characters
- Longest substring with exactly K distinct characters

Category 2: Pattern Matching

- Find all anagrams
- Permutation in string
- Minimum window substring

Category 3: Frequency Constraints

- Longest substring without repeating characters
- Longest repeating character replacement
- Subarrays with K different integers

Category 4: Window Optimization

- Minimum window containing all required characters
- Smallest subarray with sum \geq target

Chapter 3: Complete Problem Solutions

Problem 1: Longest Substring Without Repeating Characters (LeetCode 3)

Link: <https://leetcode.com/problems/longest-substring-without-repeating-characters/>

Difficulty: Medium

Pattern: Sliding Window + HashMap (frequency tracking)

Description: Given string `s`, find length of longest substring without repeating characters.

Examples:

```
Input: s = "abcabcbb"
Output: 3 (substring: "abc")
```

```
Input: s = "bbbbbb"
Output: 1 (substring: "b")

Input: s = "pwwkew"
Output: 3 (substring: "wke")
```

Solution 1: Sliding Window with HashSet - $O(n)$ time, $O(\min(n,m))$ space

Approach: Use set to track unique characters in current window.

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        unordered_set<char> char_set; // Track characters in window
        int left = 0;
        int max_len = 0;

        for (int right = 0; right < s.length(); right++) {
            // Shrink window until no duplicates
            while (char_set.count(s[right])) {
                char_set.erase(s[left]);
                left++;
            }

            // Expand window
            char_set.insert(s[right]);

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

        return max_len;
    }
};
```

Complexity Analysis:

- **Time:** $O(n)$ - each character added/removed at most once
- **Space:** $O(\min(n, m))$ where m = alphabet size

Dry Run:

```
s = "abcabcbb"

right=0, left=0: window="a", set={a}, max_len=1
right=1, left=0: window="ab", set={a,b}, max_len=2
right=2, left=0: window="abc", set={a,b,c}, max_len=3
right=3, left=0: 'a' exists → shrink
    left=1: remove 'a', set={b,c}
    window="bca", set={b,c,a}, max_len=3
right=4, left=1: 'b' exists → shrink
```

```
left=2: remove 'b', set={c,a}
left=3: remove 'c', set={a}
window="ab", set={a,b}, max_len=3
...
```

Solution 2: Sliding Window with HashMap (Optimized) - $O(n)$ time, $O(m)$ space

Approach: Store last seen index of each character to skip intermediate positions.

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        unordered_map<char, int> last_seen; // char -> last index
        int left = 0;
        int max_len = 0;

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

            // If character seen in current window, jump left pointer
            if (last_seen.count(c) && last_seen[c] >= left) {
                left = last_seen[c] + 1;
            }

            // Update last seen position
            last_seen[c] = right;

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

        return max_len;
    }
};
```

Why Better:

- Skips intermediate positions instead of shrinking one-by-one
- Fewer operations per iteration
- Same asymptotic complexity but better constant factor

Example:

```
s = "abba"

right=0: c='a', last_seen={a:0}, left=0, max_len=1
right=1: c='b', last_seen={a:0,b:1}, left=0, max_len=2
right=2: c='b', last_seen[b]=1 &gt;= left=0
        → left = 1+1 = 2
        last_seen={a:0,b:2}, max_len=2
```

```
right=3: c='a', last_seen[a]=0 &lt; left=2 (not in current window)
      last_seen={a:3,b:2}, left=2, max_len=2
```

Solution 3: Fixed-Size Array (Fastest for ASCII) - $O(n)$ time, $O(1)$ space

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        vector<int> last_seen(256, -1); // ASCII characters
        int left = 0;
        int max_len = 0;

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

            // Update left pointer
            left = max(left, last_seen[c] + 1);

            // Update last seen
            last_seen[c] = right;

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

        return max_len;
    }
};
```

Advantages:

- Faster than unordered_map (no hashing overhead)
- Better cache locality
- Truly $O(1)$ space (fixed 256 bytes)

Problem 2: Longest Substring with At Most K Distinct Characters (LeetCode 340)

Link: <https://leetcode.com/problems/longest-substring-with-at-most-k-distinct-characters/>

Difficulty: Medium

Description: Find length of longest substring with at most K distinct characters.

Examples:

```
Input: s = "eceba", k = 2
Output: 3 (substring: "ece")
```

Input: s = "aa", k = 1
Output: 2 (substring: "aa")

Solution: Sliding Window with HashMap - O(n) time, O(k) space

```
class Solution {
public:
    int lengthOfLongestSubstringKDistinct(string s, int k) {
        if (k == 0) return 0;

        unordered_map<char, int> freq; // Character frequencies
        int left = 0;
        int max_len = 0;

        for (int right = 0; right < s.length(); right++) {
            // Expand: Add right character
            freq[s[right]]++;

            // Contract: Shrink while more than k distinct
            while (freq.size() > k) {
                freq[s[left]]--;
                if (freq[s[left]] == 0) {
                    freq.erase(s[left]);
                }
                left++;
            }

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

        return max_len;
    }
};
```

Detailed Walkthrough:

```
s = "eceba", k = 2

right=0: s[0]='e', freq={e:1}, size=1, left=0, max_len=1
right=1: s[1]='c', freq={e:1,c:1}, size=2, left=0, max_len=2
right=2: s[2]='e', freq={e:2,c:1}, size=2, left=0, max_len=3
right=3: s[3]='b', freq={e:2,c:1,b:1}, size=3 > k=2
        Shrink: left=0, s[0]='e', freq={e:1,c:1,b:1}, size=3
        Shrink: left=1, s[1]='c', freq={e:1,b:1}, size=2
        max_len=3
right=4: s[4]='a', freq={e:1,b:1,a:1}, size=3 > k=2
        Shrink: left=2, s[2]='e', freq={b:1,a:1}, size=2
        max_len=3
```

Answer: 3

Why This Works:

- `freq.size()` gives count of distinct characters
- Shrinking removes characters until constraint satisfied
- Window always maintains $\leq k$ distinct

Problem 3: Minimum Window Substring (LeetCode 76)

Link: <https://leetcode.com/problems/minimum-window-substring/>

Difficulty: Hard ★

Description: Find minimum window in `s` that contains all characters of `t`.

Examples:

```
Input: s = "ADOBECODEBANC", t = "ABC"
Output: "BANC"
```

```
Input: s = "a", t = "a"
Output: "a"
```

```
Input: s = "a", t = "aa"
Output: "" (not possible)
```

Solution: Two HashMap Approach - $O(n)$ time, $O(m)$ space

```
class Solution {
public:
    string minWindow(string s, string t) {
        if (s.empty() || t.empty()) return "";

        // Step 1: Build required character frequency map
        unordered_map<char, int> required;
        for (char c : t) {
            required[c]++;
        }

        int required_count = required.size(); // Unique chars needed
        int formed_count = 0; // Unique chars satisfied

        // Window frequency map
        unordered_map<char, int> window_freq;

        int left = 0;
        int min_len = INT_MAX;
        int min_left = 0;

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

```

        // Expand: Add character to window
        window_freq[c]++;

        // Check if this character satisfies requirement
        if (required.count(c) &&
            window_freq[c] == required[c]) {
            formed_count++;
        }

        // Contract: Try to shrink window
        while (left &lt;= right && formed_count == required_count) {
            // Update result
            if (right - left + 1 &lt; min_len) {
                min_len = right - left + 1;
                min_left = left;
            }

            // Shrink from left
            char left_char = s[left];
            window_freq[left_char]--;

            // Check if this breaks a requirement
            if (required.count(left_char) &&
                window_freq[left_char] &lt; required[left_char]) {
                formed_count--;
            }

            left++;
        }
    }

    return (min_len == INT_MAX) ? "" : s.substr(min_left, min_len);
}
};

```

Key Insights:

1. Two Maps:

- `required`: What we need (from `t`)
- `window_freq`: What we have (current window)

2. Two Counters:

- `required_count`: How many unique chars needed
- `formed_count`: How many unique chars satisfied

3. Bidirectional Optimization:

- **Expand**: Right pointer always moves forward
- **Contract**: Left pointer shrinks when valid window found

Detailed Example:

```

s = "ADOBECODEBANC", t = "ABC"
required = {A:1, B:1, C:1}, required_count = 3

right=0: 'A', window={A:1}, formed=1
right=1: 'D', window={A:1,D:1}, formed=1
right=2: 'O', window={A:1,D:1,O:1}, formed=1
right=3: 'B', window={A:1,D:1,O:1,B:1}, formed=2
right=4: 'E', window={A:1,D:1,O:1,B:1,E:1}, formed=2
right=5: 'C', window={A:1,D:1,O:1,B:1,E:1,C:1}, formed=3 ✓

Now formed==required, shrink:
  left=0: remove 'A', window={D:1,O:1,B:1,E:1,C:1}, formed=2
  Window invalid, stop shrinking
  Record: "ADOBEC" (len=6)

Continue expanding...
right=6: 'O', window={D:1,O:2,B:1,E:1,C:1}, formed=2
...
right=12: 'C', window={B:1,A:1,N:1,C:1}, formed=3 ✓

Shrink:
  left=9: "BANC" (len=4) - minimum!

Final answer: "BANC"

```

Problem 4: Permutation in String (LeetCode 567)

Link: <https://leetcode.com/problems/permutation-in-string/>

Difficulty: Medium

Description: Check if s2 contains permutation of s1.

Examples:

```

Input: s1 = "ab", s2 = "eidbaooo"
Output: true (s2 contains "ba" which is permutation of "ab")

Input: s1 = "ab", s2 = "eidboaoo"
Output: false

```

Solution 1: Fixed-Size Window with HashMap - O(n) time, O(1) space

```

class Solution {
public:
    bool checkInclusion(string s1, string s2) {
        if (s1.length() > s2.length()) return false;

        // Build s1 frequency map
        unordered_map<char, int> s1_freq;

```

```

        for (char c : s1) {
            s1_freq[c]++;
        }

        unordered_map<char, int> window_freq;
        int window_size = s1.length();

        // Build initial window
        for (int i = 0; i < window_size; i++) {
            window_freq[s2[i]]++;
        }

        // Check initial window
        if (window_freq == s1_freq) return true;

        // Slide window
        for (int i = window_size; i < s2.length(); i++) {
            // Add new character
            window_freq[s2[i]]++;

            // Remove old character
            char old_char = s2[i - window_size];
            window_freq[old_char]--;
            if (window_freq[old_char] == 0) {
                window_freq.erase(old_char);
            }

            // Check if window matches s1
            if (window_freq == s1_freq) return true;
        }

        return false;
    }
};

```

Optimization: Compare maps more efficiently using counter.

Solution 2: Optimized with Single Counter - $O(n)$ time, $O(1)$ space

```

class Solution {
public:
    bool checkInclusion(string s1, string s2) {
        if (s1.length() > s2.length()) return false;

        vector<int> s1_freq(26, 0);
        vector<int> window_freq(26, 0);

        // Build s1 frequency
        for (char c : s1) {
            s1_freq[c - 'a']++;
        }

        int window_size = s1.length();
    }
};

```

```

// Build initial window
for (int i = 0; i < window_size; i++) {
    window_freq[s2[i] - 'a']++;
}

// Count matching characters
int matches = 0;
for (int i = 0; i < 26; i++) {
    if (s1_freq[i] == window_freq[i]) {
        matches++;
    }
}

if (matches == 26) return true;

// Slide window
for (int i = window_size; i < s2.length(); i++) {
    // Add right character
    int right_idx = s2[i] - 'a';
    window_freq[right_idx]++;

    if (window_freq[right_idx] == s1_freq[right_idx]) {
        matches++;
    } else if (window_freq[right_idx] == s1_freq[right_idx] + 1) {
        matches--;
    }

    // Remove left character
    int left_idx = s2[i - window_size] - 'a';
    window_freq[left_idx]--;

    if (window_freq[left_idx] == s1_freq[left_idx]) {
        matches++;
    } else if (window_freq[left_idx] == s1_freq[left_idx] - 1) {
        matches--;
    }

    if (matches == 26) return true;
}

return false;
}
};

```

Why Faster:

- Avoids comparing entire maps
- Tracks match count incrementally
- Array access faster than map

Problem 5: Find All Anagrams in a String (LeetCode 438)

Link: <https://leetcode.com/problems/find-all-anagrams-in-a-string/>

Difficulty: Medium

Description: Find all start indices of p's anagrams in s.

Example:

```
Input: s = "cbaebabacd", p = "abc"
Output: [0, 6]
Explanation:
- s[0:3] = "cba" is anagram of "abc"
- s[6:9] = "bac" is anagram of "abc"
```

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

```
class Solution {
public:
    vector<int> findAnagrams(string s, string p) {
        vector<int> result;
        if (s.length() < p.length()) return result;

        vector<int> p_freq(26, 0);
        vector<int> window_freq(26, 0);

        // Build p frequency
        for (char c : p) {
            p_freq[c - 'a']++;
        }

        int window_size = p.length();

        // Build initial window
        for (int i = 0; i < window_size; i++) {
            window_freq[s[i] - 'a']++;
        }

        // Check initial window
        if (p_freq == window_freq) {
            result.push_back(0);
        }

        // Slide window
        for (int i = window_size; i < s.length(); i++) {
            // Add new character
            window_freq[s[i] - 'a']++;

            // Remove old character
            window_freq[s[i - window_size] - 'a']--;

            // Check if anagram
```

```

        if (window_freq == p_freq) {
            result.push_back(i - window_size + 1);
        }
    }

    return result;
}
};

```

Dry Run:

```

s = "cbaebabacd", p = "abc"
p_freq = [1,1,1,0,...] (a=1, b=1, c=1)

Initial window: "cba"
window_freq = [1,1,1,0,...] == p_freq → add index 0

i=3: Add 'e', remove 'c'
      window = "bae", freq = [1,1,0,0,1,...] ≠ p_freq

i=4: Add 'b', remove 'b'
      window = "aeb", freq = [1,1,0,0,1,...] ≠ p_freq

i=5: Add 'a', remove 'a'
      window = "eba", freq = [1,1,0,0,1,...] ≠ p_freq

i=6: Add 'b', remove 'e'
      window = "bab", freq = [1,2,0,0,0,...] ≠ p_freq

i=7: Add 'a', remove 'b'
      window = "aba", freq = [2,1,0,0,0,...] ≠ p_freq

i=8: Add 'c', remove 'a'
      window = "bac", freq = [1,1,1,0,...] == p_freq → add index 6

Result: [0, 6]

```

Problem 6: Longest Repeating Character Replacement (LeetCode 424)

Link: <https://leetcode.com/problems/longest-repeating-character-replacement/>

Difficulty: Medium

Description: You can replace at most k characters. Find longest substring with same character after replacements.

Example:

```

Input: s = "AABABBA", k = 1
Output: 4
Explanation: Replace one 'A' to get "AABBBBA", substring "BBBB" has length 4

```

Solution: Sliding Window with Frequency Map - $O(n)$ time, $O(1)$ space

```
class Solution {
public:
    int characterReplacement(string s, int k) {
        vector<int> freq(26, 0);
        int left = 0;
        int max_freq = 0; // Most frequent char count in window
        int max_len = 0;

        for (int right = 0; right < s.length(); right++) {
            // Expand: Add right character
            freq[s[right] - 'A']++;

            // Track maximum frequency
            max_freq = max(max_freq, freq[s[right] - 'A']);

            // Window size - max_freq = characters to replace
            // If this exceeds k, shrink window
            int window_size = right - left + 1;
            int chars_to_replace = window_size - max_freq;

            if (chars_to_replace > k) {
                freq[s[left] - 'A']--;
                left++;
            }

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

        return max_len;
    }
};
```

Key Insight:

- **Window valid** if: $\text{window_size} - \text{max_freq} \leq k$
- **max_freq**: Most frequent character count
- **window_size - max_freq**: Characters that need replacing

Example:

s = "AABABBA", k = 1

```
right=0: 'A', freq[A]=1, max_freq=1, valid (1-1=0 ≤ 1)
right=1: 'A', freq[A]=2, max_freq=2, valid (2-2=0 ≤ 1)
right=2: 'B', freq[B]=1, max_freq=2, valid (3-2=1 ≤ 1)
right=3: 'A', freq[A]=3, max_freq=3, valid (4-3=1 ≤ 1)
right=4: 'B', freq[B]=2, max_freq=3, INVALID (5-3=2 > 1)
        Shrink: left=1, freq[A]=2, window="AABBB"
right=5: 'B', freq[B]=3, max_freq=3, valid (5-3=2 > 1)
```

```
Shrink: left=2, freq[A]=1, window="ABBBA"
valid (4-3=1 ≤ 1), max_len=4
right=6: 'A', freq[A]=2, max_freq=3, valid (5-3=2 > 1)
Shrink: left=3, freq[B]=2, window="BBBBA"
valid (4-3=1 ≤ 1), max_len=4
```

Answer: 4

Chapter 4: Complete Problem Set (Organized by Difficulty)

Easy Level (Warm-up)

Problem 7: Contains Duplicate II (LeetCode 219)

Link: <https://leetcode.com/problems/contains-duplicate-ii/>

```
bool containsNearbyDuplicate(vector<int>& nums, int k) {
    unordered_set<int> window;

    for (int i = 0; i < nums.size(); i++) {
        // Maintain window of size k
        if (i > k) {
            window.erase(nums[i - k - 1]);
        }

        if (window.count(nums[i])) {
            return true; // Found duplicate within k distance
        }

        window.insert(nums[i]);
    }

    return false;
}
```

Problem 8: Maximum Average Subarray I (LeetCode 643)

Link: <https://leetcode.com/problems/maximum-average-subarray-i/>

```
double findMaxAverage(vector<int>& nums, int k) {
    double sum = 0;

    // Build initial window
    for (int i = 0; i < k; i++) {
        sum += nums[i];
    }

    double max_sum = sum;
```

```

// Slide window
for (int i = k; i < nums.size(); i++) {
    sum += nums[i] - nums[i - k];
    max_sum = max(max_sum, sum);
}

return max_sum / k;
}

```

Medium Level (Core Practice)

Problem 9: Fruit Into Baskets (LeetCode 904)

Link: <https://leetcode.com/problems/fruit-into-baskets/>

Reframe: Longest subarray with at most 2 distinct elements.

```

int totalFruit(vector<int>& fruits) {
    unordered_map<int, int> basket; // fruit_type -> count
    int left = 0;
    int max_fruits = 0;

    for (int right = 0; right < fruits.size(); right++) {
        basket[fruits[right]]++;

        // More than 2 types? Shrink!
        while (basket.size() > 2) {
            basket[fruits[left]]--;
            if (basket[fruits[left]] == 0) {
                basket.erase(fruits[left]);
            }
            left++;
        }

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

    return max_fruits;
}

```

Problem 10: Subarrays with K Different Integers (LeetCode 992)

Link: <https://leetcode.com/problems/subarrays-with-k-different-integers/>

Difficulty: Hard

Trick: $\text{exactly}(K) = \text{atMost}(K) - \text{atMost}(K-1)$

```

class Solution {
private:
    int atMostK(vector<int>& nums, int k) {
        unordered_map<int, int> freq;
        int left = 0;
        int count = 0;

        for (int right = 0; right < nums.size(); right++) {
            if (freq[nums[right]]++ == 0) k--;

            while (k < 0) {
                if (--freq[nums[left]] == 0) k++;
                left++;
            }

            count += right - left + 1;
        }

        return count;
    }

public:
    int subarraysWithKDistinct(vector<int>& nums, int k) {
        return atMostK(nums, k) - atMostK(nums, k - 1);
    }
};

```

This comprehensive textbook continues with 10+ more problems, all with multiple solutions, detailed explanations, complexity analysis, and practice tracking systems.