



Sliding Window

A focused reference for interview preparation.

Sliding Window — Complete Interview Cheat Sheet

Recognition framework · corrected & annotated code templates · complexity reference, all in one place

FIXED-SIZE WINDOW

VARIABLE-SIZE WINDOW

APPLIED PROBLEMS

HOW TO RECOGNIZE THIS PATTERN

Ask these questions first — they confirm the pattern fits

- Does the problem ask about a contiguous subarray or substring?
- Is the window size fixed, or does it grow/shrink based on a condition?
- Would checking every subarray directly be $O(n^2)$ or worse?
- Can you maintain a running sum/count/frequency as the window slides?
- Does 'shrink the window' have a clear, checkable stopping condition?
- Is the constraint about distinct characters, a sum, or a product threshold?

Goal	Signal	Use	Complexity
Best sum/value over windows of exact size k	Window size given directly	Fixed-size sliding window	$O(n)$
Check if a permutation of s_1 appears in s_2	Fixed length match	Fixed window + character counts	$O(n)$
Longest substring with no repeats	'Without repeating characters'	Variable window + last-seen map	$O(n)$
Longest substring with $\leq k$ distinct chars	Distinct-character constraint	Variable window + count map	$O(n)$
Smallest window containing all of t	'Minimum window substring'	Variable window, shrink while valid	$O(n+m)$
Longest substring after $\leq k$ character swaps	'Character replacement'	Variable window + max frequency tracking	$O(n)$
Count subarrays under a product/sum threshold	'Less than k '	Variable window + running product/sum	$O(n)$

FIXED VS. VARIABLE WINDOW — THE FAST WAY TO TELL THEM APART

If the problem gives you an exact window size k , it's a fixed window — just slide and update a running value. If it instead gives a condition to satisfy ('at most k distinct', 'sum at least k ', 'contains all of t '), it's a variable window — expand the right edge, then shrink the left edge while (or until) the condition holds. One caution: the shrink logic assumes growing the window never helps the constraint — with negative numbers in a sum constraint that breaks, and you need prefix sums or a monotonic deque instead.

FIXED-SIZE WINDOW

Window size k is given up front

1 Maximum Sum Subarray of Size K

$O(n)$ time · $O(1)$ space

Compute the first window's sum directly, then slide: add the element entering on the right, subtract the one leaving on the left. Assumes $\text{len}(\text{nums}) \geq k$.

```
def max_sum_subarray(nums, k):
    window_sum = sum(nums[:k])          # first window, computed once
    best = window_sum
    for i in range(k, len(nums)):
        window_sum += nums[i] - nums[i - k]  # enter right, leave left
        best = max(best, window_sum)
    return best
```

SAY THIS OUT LOUD

"Instead of re-summing k elements per window ($O(n \cdot k)$), each slide is one add and one subtract — $O(1)$ per step, $O(n)$ overall."

2 Permutation in String

$O(26 \cdot n)$ time · $O(26)$ space

A fixed window of s_1 's length slides across s_2 ; the window matches a permutation exactly when its character counts equal s_1 's.

```
from collections import Counter

def check_inclusion(s1, s2):
    need = Counter(s1)
    window = Counter()
    k = len(s1)
    for i, ch in enumerate(s2):
        window[ch] += 1
        if i >= k:
            left_ch = s2[i - k]
            window[left_ch] -= 1
            if window[left_ch] == 0:
                del window[left_ch]      # keep == comparison honest
        if window == need:
            return True
    return False
```

COMMON BUG

Skipping the `del` leaves zero-count entries in the Counter, and `window == need` then never matches — Counter equality compares keys, not just positive counts.

INTERVIEWER FOLLOW-UP

"Can you drop the $O(26)$ dict comparison per step?" — yes: maintain a single 'matches' counter of how many of the 26 letters currently agree, updating it only for the two characters that changed.

VARIABLE-SIZE WINDOW

Grow and shrink the window based on a condition

3 Longest Substring Without Repeating Characters

$O(n)$ time · $O(\min(n, \Sigma))$ space

Expand the right edge; whenever the new character was already seen inside the window, jump the left edge past its previous occurrence — no element-by-element shrinking needed.

```
def length_of_longest_substring(s):
    last_seen = {}          # char -> most recent index
    left = 0
    best = 0
    for right, ch in enumerate(s):
        if ch in last_seen and last_seen[ch] >= left:
            left = last_seen[ch] + 1  # jump past the duplicate
        last_seen[ch] = right
        best = max(best, right - left + 1)
    return best
```

COMMON BUG

The 'last_seen[ch] >= left' check is essential: a stale occurrence BEFORE the current window must not drag left backward — left only ever moves right.

4 Longest Substring with At Most K Distinct Characters

$O(n)$ time · $O(k)$ space

Expand the window; shrink from the left whenever the count of distinct characters exceeds k. The count map's size IS the distinct count — as long as zero-count keys are deleted.

```
from collections import defaultdict

def longest_k_distinct(s, k):
    count = defaultdict(int)
    left = 0
    best = 0
    for right, ch in enumerate(s):
        count[ch] += 1
        while len(count) > k:          # one distinct char too many
            count[s[left]] -= 1
            if count[s[left]] == 0:
                del count[s[left]]     # len(count) must stay truthful
            left += 1
        best = max(best, right - left + 1)
    return best
```

5 Minimum Window Substring

$O(n + m)$ time · $O(m)$ space

Expand until the window contains every character of *t* (missing hits 0); then shrink from the left as far as possible while still valid, recording the smallest such window.

```
from collections import Counter

def min_window(s, t):
    if not t:
        return ''
    need = Counter(t)           # counts still owed; may go negative
    missing = len(t)           # total characters still owed
    left = 0
    best = (float('inf'), 0, 0)
    for right, ch in enumerate(s):
        if need[ch] > 0:
            missing -= 1         # this char was genuinely needed
            need[ch] -= 1
        while missing == 0:      # valid: try to shrink
            if right - left + 1 < best[0]:
                best = (right - left + 1, left, right)
            need[s[left]] += 1
            if need[s[left]] > 0: # just broke validity
                missing += 1
            left += 1
    return '' if best[0] == float('inf') else s[best[1]:best[2] + 1]
```

WHY IT WORKS

Negative counts in *need* are doing real work: they record surplus copies, so releasing a surplus character from the left (count still ≤ 0) doesn't touch *missing* — only releasing a needed one does.

APPLIED PROBLEMS

Classic substring and subarray problems

6 Longest Repeating Character Replacement

$O(n)$ time · $O(26)$ space

A window is fixable with $\leq k$ replacements iff (window length – count of its most frequent character) $\leq k$. When that fails, slide the whole window right by one — it never needs to shrink below the best length found so far.

```
from collections import defaultdict

def character_replacement(s, k):
    count = defaultdict(int)
    left = 0
    max_freq = 0                # max frequency EVER seen in a window
    best = 0
    for right, ch in enumerate(s):
        count[ch] += 1
        max_freq = max(max_freq, count[ch])
        window_len = right - left + 1
        if window_len - max_freq > k: # not fixable: slide, don't grow
            count[s[left]] -= 1
            left += 1
        else:
            best = max(best, window_len)
    return best
```

WHY IT WORKS

max_freq is never decreased when the window slides, and that's deliberately fine: a stale *max_freq* can only keep the window at a size already proven achievable — the answer only improves when a genuinely higher frequency appears.

7 Fruit Into Baskets (at most 2 distinct types)

$O(n)$ time · $O(1)$ space

Special case of 'at most k distinct' with $k = 2$ — expand, then shrink whenever a third distinct type appears. Recognizing the disguise is the whole problem.

```
from collections import defaultdict

def total_fruit(fruits):
    count = defaultdict(int)
    left = 0
    best = 0
    for right, fruit in enumerate(fruits):
        count[fruit] += 1
        while len(count) > 2:           # a third type entered
            count[fruits[left]] -= 1
            if count[fruits[left]] == 0:
                del count[fruits[left]]
            left += 1
        best = max(best, right - left + 1)
    return best
```

SAME SKELETON

Identical skeleton to technique #4 with k hard-coded to 2 — in an interview, say so and reuse the template rather than deriving it from scratch.

8 Subarrays with Product Less Than K

$O(n)$ time · $O(1)$ space

Maintain a running product; shrink from the left while it's $\geq k$. Every subarray ending at right and starting at or after left is then valid, contributing $(\text{right} - \text{left} + 1)$ new subarrays. Assumes positive integers.

```
def num_subarray_product_less_than_k(nums, k):
    if k <= 1:
        return 0           # positive nums: nothing can be < k
    left = 0
    product = 1
    count = 0
    for right, num in enumerate(nums):
        product *= num
        while product >= k:
            product /= nums[left]   # exact: product is divisible by it
            left += 1
        count += right - left + 1   # new subarrays ending at right
    return count
```

COMMON BUG

Counting windows instead of $(\text{right} - \text{left} + 1)$ per step undercounts badly — the increment counts every valid subarray that ENDS at right, which partitions all valid subarrays exactly once.

BREAKS WHEN

Zeros or negatives break both the shrink logic and the exact division — this template requires strictly positive numbers, which the LeetCode constraints guarantee.

QUICK-REFERENCE SUMMARY

One-line reminder for every technique on this sheet

Technique	What it's for
Max Sum Subarray of Size K	Slide the window, add entering value, subtract leaving value
Permutation in String	Fixed window; compare character counts to s1's

Technique	What it's for
Longest Substring Without Repeats	Expand right, jump left past the last duplicate
Longest Substring K Distinct	Expand right, shrink left while distinct count > k
Minimum Window Substring	Expand until valid, then shrink while still valid
Longest Repeating Char Replacement	Slide (never shrink) when window_len - max_freq > k
Fruit Into Baskets	At-most-2-distinct special case of the K-distinct pattern
Subarrays with Product Less Than K	Shrink while product $\geq k$; count via right - left + 1

Verification & Practice

Sliding Window

WHEN NOT TO USE THIS PATTERN

Avoid it when the valid region is non-contiguous or when adding/removing an endpoint cannot update state efficiently.

EDGE-CASE CHECKLIST

- Empty input and zero window
- Window never becomes valid
- Duplicate characters or values

HIGH-VALUE TEST CASES

- longest unique substring of " -> 0
- all same characters -> 1
- minimum window with no solution -> "

AUTHORITATIVE REFERENCES

<https://docs.python.org/3/reference/>

<https://xlinux.nist.gov/dads/>

<https://www.w3.org/WAI/WCAG22/Understanding/contrast-minimum.html>