

Sliding Window

Patterns and Invariants

Move a contiguous interval while its state stays accurate. Choose the update order from the exact validity rule.

WORKED EXAMPLE

A delivery app can examine consecutive hourly net order changes. A moving 3-hour sum needs one entry and one exit per step, not a full recount.

THE LEARNING PATH

- | | | |
|----|-----------------------|---|
| 01 | RECOGNIZE | Look for contiguous ranges and incremental state. |
| 02 | FIX SIZE | Add the new value and remove the outgoing value. |
| 03 | LONGEST VALID | Expand, then shrink until the constraint holds. |
| 04 | SHORTEST VALID | For positive inputs, shrink while the target holds. |
| 05 | VERIFY | Trace boundaries, answer timing, and assumptions. |

The invariant determines the code, not the other way around.

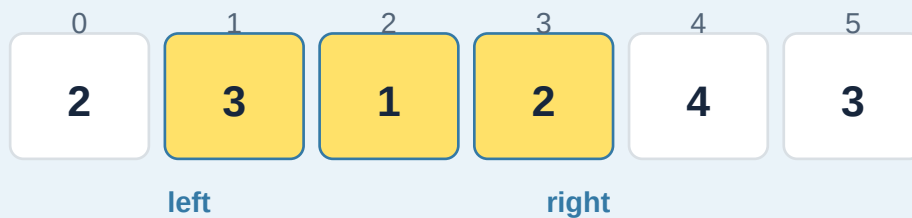
Source topic preserved; this is a larger-type technical review copy.

Recognize a moving interval

A window is a contiguous subarray or substring represented by two boundaries and an updateable state.

THE WINDOW INVARIANT

At every step, the maintained sum, counts, or other state must describe **exactly** the elements in the current inclusive interval **[left, right]**. When right advances, add its contribution. When left advances, remove the departing contribution.



Example window [1,3] contains 3, 1, 2; its sum is 6.

Choose by the validity rule

- **Fixed width:** exactly k values; every move replaces one outgoing value with one incoming value.
- **Longest valid:** expand right, then shrink while invalid; measure only after restoring validity.
- **Shortest satisfying:** for positive sums, shrink while valid and record before removing left.

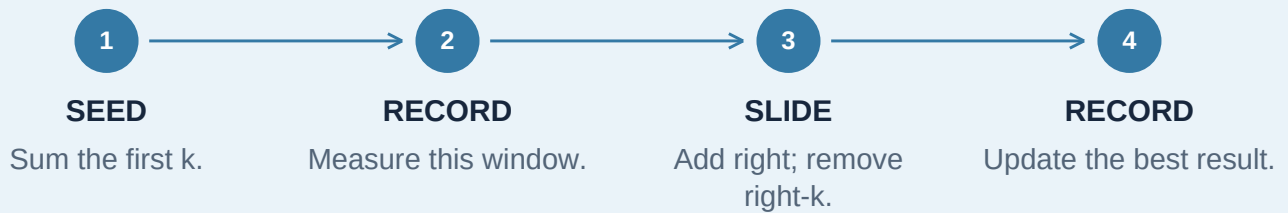
WHEN NOT TO USE THIS TEMPLATE

If negatives can make a sum fall after expansion or rise after shrinking, the positive-only shortest-sum rule loses its monotonicity. Use a different method, such as a prefix/deque approach, when the contract requires signed values.

Fixed-size sums

A fixed-width window is a rolling aggregate over exactly k adjacent items.

UPDATE ONE ENTRY AND ONE EXIT



PYTHON / FIXED WINDOW

```
def max_fixed_sum(a, k):  
    if not 1 <= k <= len(a): raise ValueError(k)  
    current = sum(a[:k]); best = current  
    for right in range(k, len(a)):  
        current += a[right] - a[right-k]  
        best = max(best, current)  
    return best
```

Real-world example: net hourly change

A food-delivery operator compares 3-hour net order changes. With changes $[4, -1, 2, 7, -3]$, the best 3-hour sum is 8 over $[-1, 2, 7]$, so the average change is **8/3**. Signed values are fine here: the width never changes.

COST AND BOUNDARY

Seed $O(k)$, then perform $O(n-k)$ constant-time slides: $O(n)$ time and $O(1)$ extra state. Decide explicitly what $k=0$, $k>n$, and empty input mean before coding.

Expand, then restore validity

The longest-valid family keeps an interval feasible while right advances once and left only moves forward.

AT MOST K DISTINCT VALUES

A map stores the count of each value in $[left, right]$. Add $a[right]$. While the number of distinct keys exceeds k , decrement $a[left]$, delete zero counts, and advance left. Only then can the current length update the answer.

PYTHON / LONGEST VALID

```
from collections import defaultdict
def longest_at_most_k(a, k):
    if k < 0: raise ValueError("k < 0")
    count = defaultdict(int); left = best = 0
    for right, value in enumerate(a):
        count[value] += 1
        while len(count) > k:
            old = a[left]; count[old] -= 1
            if count[old] == 0: del count[old]
            left += 1
        best = max(best, right-left+1)
    return best
```

Practical application: browsing categories

A streaming service asks for the longest consecutive viewing run involving at most two categories. Counts, not a set, retain repeated categories; deleting a category only when its count reaches zero prevents stale state.

WHY $O(N)$?

Both boundaries move right at most n times. A nested while loop does not imply $O(n^2)$ here: all shrink iterations combined are at most n .

Shrink while the target holds

For positive values and a target above zero, removing items cannot increase a sum. This permits a shortest qualifying interval.

WORKED WINDOW TRACE

Target = 7; values = [2,3,1,2,4,3]. Expand until sum reaches 8 at right=3, record length 4, then shrink. At the final two values [4,3], sum is 7 and the best length becomes 2.



The final qualifying window [4,3] has length 2.

PYTHON / POSITIVE VALUES ONLY

```
def min_positive_length(a, target):
    left = total = 0; best = len(a) + 1
    for right, value in enumerate(a):
        total += value
        while total >= target:
            best = min(best, right-left+1)
            total -= a[left]
            left += 1
    return 0 if best > len(a) else best
```

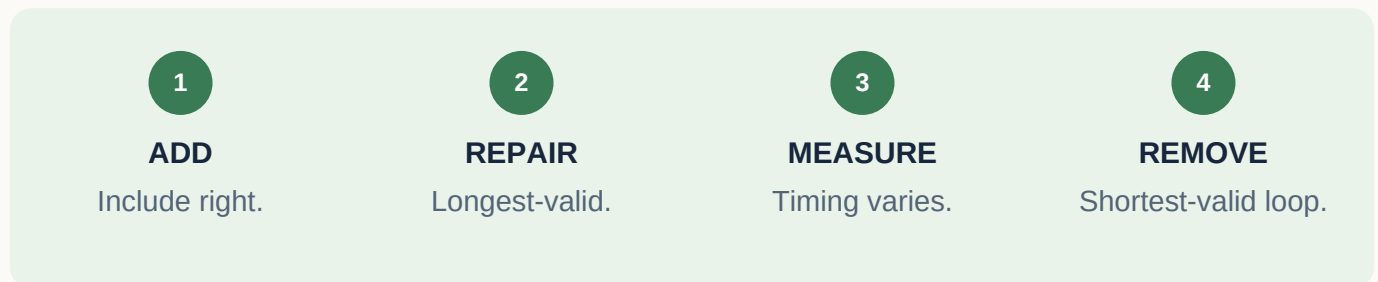
ASSUMPTION IS ESSENTIAL

For [1,-1,5] and target 5, signed values can violate the shrink logic. State that inputs are positive before claiming this O(n) template is correct.

Trace the state, not just pointers

Most window bugs are wrong answers at the exact moment a boundary changes.

CHECKPOINTS, NOT A FIXED ORDER



Debug by invariant

- At each iteration, write [left,right], the current state, validity, and answer on one line.
- Use **while** when several removals may be needed; one **if** can leave the window invalid.
- Remove zero-frequency keys if the constraint counts distinct values.
- For shortest satisfying windows, record **before** removing the left item.

Example trace: target 7

SELECTED STATE TRANSITIONS		
right=3	[2,3,1,2]=8	best=4
right=4	[1,2,4]=7	best=3
right=5	[2,4,3]=9	shrink
right=5	[4,3]=7	best=2

Different rules, different state

The same two-pointer shell can maintain sums, distinct counts, exact multiplicities, or a monotonic deque.

MATCH THE STATE TO THE QUESTION

- **Max Consecutive Ones III:** keep the number of zeroes within a flip budget; shrink if it exceeds k .
- **Fruit Into Baskets:** count fruit types and maintain at most two distinct keys.
- **Permutation in String:** maintain a fixed-width character frequency difference and test exact equality.
- **Minimum Window Substring:** count required characters with multiplicity; minimize only while all requirements are met.
- **Sliding Window Maximum:** keep candidate indices in a decreasing deque; discard expired front indices.

Why the deque is different

A running maximum cannot be updated by subtracting the outgoing maximum. The deque retains only candidates that can still become the maximum, with indices for expiry and decreasing values for dominance.

ENGINEERING TAKEAWAY

A 'sliding window' is not one algorithm. Define the constraint, choose sufficient state, prove how left can move, then choose when to update the answer.

Eight problems in sequence

Practice the rule change, not memorized code. Each rung adds one new state or boundary condition.

SOURCE PRACTICE PROGRESSION

- **1. Maximum Average Subarray I:** fixed-size rolling sum.
- **2. Minimum Size Subarray Sum:** shortest satisfying positive-sum window.
- **3. Longest Substring Without Repeating Characters:** counts and longest-valid update.
- **4. Max Consecutive Ones III:** budgeted violations.
- **5. Fruit Into Baskets:** at most two distinct values.
- **6. Permutation in String:** fixed-size frequency matching.
- **7. Minimum Window Substring:** required multiplicity and shortest cover.
- **8. Sliding Window Maximum:** monotonic deque, not simple rolling sum.

Self-check before implementation

Write the window invariant, name the state, identify the exact invalid/valid condition, and mark when the answer is recorded. Test empty input, repeated values, boundary k, and any assumption about positive values.

TECHNICAL REVIEW GATE

This learning copy preserves the Drive source's subject and examples. A named human reviewer should verify the final lesson before publication.

REFERENCES AND WORKED PRACTICE

`dsa_sliding-window.pdf`

The improved guide ends on the preceding page.

The previous production edition follows in full as an appendix.

It retains its original worked exercises, answers, and references.

Dated standalone edition

https://interviewrunway.com/pdf-guides/practice-2026-10-15/dsa_sliding-window.pdf

Optional short introduction: https://interviewrunway.com/pdf-guides/supplemental-2026-10-15/intro_sliding-window.pdf

Source references

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

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

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

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

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

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

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

<https://leetcode.com/problems/sliding-window-maximum/>



DSA PATTERN REFERENCE

Sliding Window

Recognition cues, invariants, Python templates, and an 8-problem practice ladder

Use sliding window when the answer is a contiguous range and one boundary move can update validity without rescanning the range.

VERSION 1.1

CODE LANGUAGE PYTHON 3

STATUS PRACTICE EDITION

DSA / SLIDING WINDOW

01

RECOGNIZE THE PATTERN

At a glance

CORE SIGNAL

Use sliding window when the answer is a contiguous range and one boundary move can update validity without rescanning the range.

Strong cues

- + The prompt asks about a contiguous subarray or substring.
- + The objective is longest, shortest, count, maximum, or minimum over a range.
- + One incoming and one outgoing value can update the state incrementally.
- + A brute-force solution repeats nearly identical work across overlapping ranges.

When not to force it

- + The chosen elements can be non-contiguous, which usually points to subsequence logic or dynamic programming.
- + Moving a boundary does not change validity monotonically. Negative numbers often break sum-based shrinking.
- + The required state cannot be updated more cheaply than rescanning the entire range.

Choose the variant

Question	If yes	If no
Is the window length exactly k?	Use the fixed-size template.	Continue to a validity-driven window.
Do you need the longest valid range?	Shrink while invalid; record after validity is restored.	For the shortest satisfying range, record while valid and then shrink.

Working invariant

For an inclusive window, the state must describe exactly the values from left through right. Every add and remove operation must preserve that statement.

02.1

REUSABLE TEMPLATE

Fixed-size window

INVARIANT

After eviction, the current window contains at most k items; an answer is recorded only when it contains exactly k .

```
def max_sum_of_k(nums, k):  
    if k <= 0 or k > len(nums):  
        raise ValueError("k must be in [1, len(nums)]")  
  
    left = 0  
    window_sum = 0  
    best = float("-inf")  
  
    for right, value in enumerate(nums):  
        window_sum += value  
  
        if right - left + 1 > k:  
            window_sum -= nums[left]  
            left += 1  
  
        if right - left + 1 == k:  
            best = max(best, window_sum)  
  
    return best
```

Use it for

Maximum or minimum sum/average over exactly k values; frequency matching for an exact-length substring; deque-based extrema over each range of size k .

02.2

REUSABLE TEMPLATE

Longest valid variable window

INVARIANT

After the while-loop, the window satisfies the validity rule.

```
def longest_at_most_k_distinct(values, k):  
    if k < 0:  
        raise ValueError("k cannot be negative")  
  
    left = 0  
    counts = {}  
    best = 0  
  
    for right, value in enumerate(values):  
        counts[value] = counts.get(value, 0) + 1  
  
        while len(counts) > k:  
            outgoing = values[left]  
            counts[outgoing] -= 1  
            if counts[outgoing] == 0:  
                del counts[outgoing]  
            left += 1  
  
        best = max(best, right - left + 1)  
  
    return best
```

Answer placement

Shrink while the window is invalid. Record only after the loop, when the invariant is true again.

02.3

REUSABLE TEMPLATE

Shortest satisfying variable window

INVARIANT

Every candidate is recorded while the sum meets the target; positive inputs make leftward shrinking monotonic.

```
def min_len_at_least(target, nums):  
    if target <= 0 or any(x <= 0 for x in nums):  
        raise ValueError("positive target and values required")  
    left = total = 0  
    best = len(nums) + 1  
    for right, value in enumerate(nums):  
        total += value  
        while total >= target:  
            best = min(best, right - left + 1)  
            total -= nums[left]  
            left += 1  
    return 0 if best > len(nums) else best
```

Answer placement

Record inside the valid shrink loop before removing the left value. The outgoing value may make the next window invalid.

ASSUMPTION

Requires target > 0 and every value > 0. Empty nums returns 0.

03

TRACE THE STATE

Find the shortest subarray with sum at least 7.

EXAMPLE

nums = [2, 3, 1, 2, 4, 3] | Answer: 2, from [4, 3]

R	Added	Window before shrink	Recorded	Window after shrink
0	2	[2] / sum 2	-	[2]
1	3	[2,3] / sum 5	-	[2,3]
2	1	[2,3,1] / sum 6	-	[2,3,1]
3	2	[2,3,1,2] / sum 8	4	[3,1,2] / sum 6
4	4	[3,1,2,4] / sum 10	4, then 3	[2,4] / sum 6
5	3	[2,4,3] / sum 9	3, then 2	[3] / sum 3

What the trace proves

Every recorded range meets the target. Repeated shrinking tests all shorter candidates ending at the same right index. Because the inputs are positive, once a removal breaks the target, more removals cannot restore it.

04

DEBUG AND EXPLAIN

Failure-mode checklist

Symptom	Check first
The answer is off by one.	Use $\text{right} - \text{left} + 1$ for an inclusive window and trace the first full window.
A supposedly valid answer violates the rule.	Record only after the invalid-shrink loop for longest-valid problems.
The state drifts away from the actual range.	Undo every outgoing value before incrementing left.
The algorithm misses a smaller valid range.	Use while, not if, when repeated shrinking can improve or restore the window.
A sum-based solution fails with negative numbers.	The monotonic assumption is false; consider prefix sums with a deque or another method.
The code times out despite two pointers.	Avoid rescanning the window or its whole frequency map after each move.

Complexity proof

Time

+ Usually $O(n)$: right advances at most n times and left advances at most n times.

Space

+ $O(1)$ for a running scalar; $O(u)$ for a map with u distinct values; $O(k)$ for a fixed-window deque.

INTERVIEW PROOF

Nested syntax does not imply $O(n^2)$. Count total pointer moves across the whole run.

05

PRACTICE LADDER

Eight problems, one progression

Solve in order. Before coding, label the variant, state, invariant, shrink rule, and answer position.

#	Problem	Level	Primary focus
1	<u>Maximum Average Subarray I</u>	Easy	Fixed-size sum
2	<u>Minimum Size Subarray Sum</u>	Medium	Shortest satisfying window
3	<u>Longest Substring Without Repeating Characters</u>	Medium	Frequency counts and longest valid window
4	<u>Max Consecutive Ones III</u>	Medium	Budgeted violations
5	<u>Fruit Into Baskets</u>	Medium	At most k distinct values
6	<u>Permutation in String</u>	Medium	Fixed-size frequency matching
7	<u>Minimum Window Substring</u>	Hard	Required frequencies and shortest valid window
8	<u>Sliding Window Maximum</u>	Hard	Monotonic deque state

Re-solve rule

A problem counts as learned only when you can re-solve it from a blank editor after at least 48 hours and explain the invariant before implementation.

06

INTERVIEW-READY RECAP

Five-sentence talk track

- ☐ The objective is over a contiguous range, and brute force repeats work across overlapping ranges.
- ☐ I will track [state] for the inclusive window from left to right.
- ☐ After the shrink loop, my invariant is [validity statement].
- ☐ I update the answer [before/during/after] shrinking because [eligibility reason].
- ☐ Each pointer only moves forward, so total pointer work is $O(n)$.

Source and review note

Original pattern explanations and templates are retained. The added prompts are original InterviewRunway practice; problem titles are optional topic references.

PRACTICE CHECKS

Python templates and expected examples have automated checks. Practice from a blank editor, then compare your trace, tests and explanation with the answer guidance. No named human review is claimed.

Fast recall

REMEMBER

CONTIGUOUS / INCREMENTAL STATE / EXPLICIT INVARIANT / CORRECT ANSWER MOMENT

Eight independent attempts, not eight titles

Use the existing templates after your attempt

Each prompt below is an original exercise linked to one rung of the retained progression. Start from a blank editor, set the timer, and save validity, answer position, trace and tests before opening the review page.

8 min - fixed sum

For [4,-2,3,1] and $k=2$, return the maximum window sum and range.

12 min - shortest sum

For [2,3,1,2,4,3] and target 7, return the minimum length and every candidate considered at the final endpoint.

12 min - unique string

For "abba", return the longest no-repeat length. Show why a last-seen left pointer must never move backward.

14 min - zero budget

For [1,0,0,1,1,0,1] with at most two zero flips, return the longest resulting one-run length.

14 min - two categories

For [1,2,1,3,3], return the longest range containing at most two distinct values.

16 min - exact match

Find an anagram of "aab" in "cabaac", returning one half-open range; preserve repeated-letter counts.

20 min - multiset cover

For text "BAAAC" and requirements A:2,C:1, return the shortest covering substring.

22 min - range maxima

For [2,2,1,3,0] with $k=3$, return all window maxima; define your deque equality rule.

Eight answer checks with distinct reasoning

1 - fixed length

Window sums are 2,1,4; best is 4 at [2,4). Negative values are safe because length determines eviction. Time $O(n)$, scalar state $O(1)$. Invalid $k \leq 0$ or $k > n$ raises `ValueError` in the retained template.

2 - shortest positive sum

Length 2 from [4,3]. At the final endpoint, consider [2,4,3] sum 9 length 3, then [4,3] sum 7 length 2, then stop after removing 4 because [3] sums to 3. Record before removal. The corrected template requires $\text{target} > 0$ and positive values.

3 - uniqueness

Length 2. When second b arrives, move left past its earlier index; the final a has an old occurrence before left and must not move left backward. Use $\text{left} = \max(\text{left}, \text{last}[\text{ch}] + 1)$, then record. A count-based while loop is also valid.

4 - violation budget

Length 5 from [0,5]=[1,0,0,1,1] or [2,7]=[0,1,1,0,1]. Count zeros; shrink while $\text{zeros} > 2$, then record. Ones are not charged against the budget. Time $O(n)$, $O(1)$ scalar state.

5 - distinct categories

Length 3, either [1,2,1] or [1,3,3]. The retained at-most-k template with $k=2$ applies. On adding the first 3, remove 1 then 2 to restore two distinct keys; deleting a count-zero key is required.

6 - exact frequencies

Match [1,4]="aba" or [2,5]="baa". A window of length 3 must contain two a and one b. A membership set is insufficient. Fixed-window counts may be compared over a bounded alphabet; otherwise count full-map comparison costs or maintain a satisfied-count summary.

7 - shortest cover

"AAC" at [2,5), length 3. Cover requires at least two A and one C, so membership alone fails. Track when each required kind reaches or drops below its multiplicity; record while valid, then shrink. Copying the final substring adds $O(L)$ output cost.

8 - monotonic deque

Maxima [2,3,3]. Store indices of nonincreasing values, evict $\text{indices} \leq \text{right} - k$, and pop smaller-or-equal values before appending the new index if keeping the latest equal maximum. Each index enters and leaves at most once: $O(n)$ time, $O(k)$ deque, $O(n-k+1)$ output. Strict smaller popping is also valid if expiration is handled correctly.

Test assumptions, score behavior, and repeat

Expected template edge results

Call	Expected	Why
max_sum_of_k([-3,-1,-2],2)	-3	Fixed length supports negatives
max_sum_of_k([],1)	ValueError	No valid length
longest_at_most_k_distinct([],2)	0	Empty list
longest_at_most_k_distinct([1,1],0)	0	Zero distinct budget
min_len_at_least(5,[])	0	No positive qualifying range
min_len_at_least(0,[1])	ValueError	Positive target is explicit
min_len_at_least(3,[2,-1,2])	ValueError	This implementation excludes negatives

A counterexample to signed-sum shrinking

Target 3, values [1,-1,3]: naive positive-sum shrinking records the whole length3 range, removes 1 and stops at sum2. It misses the final [3] of length1 because another removal would increase the sum. The monotonic premise is false; use a suitable prefix-sum/deque method for the general shortest signed-sum problem.

Behavior	0 - wrong/absent	1 - with help	2 - independent
Answer moment	Records invalid or already-removed range	Moves update after hint	States fixed/longest/shortest eligibility before coding
Removal state	Leaks outgoing count or keeps zero key	Repairs after bad trace	Undoes every departure and deletes empty keys
Assumptions	Forces sum shrinking with negatives	Recognizes failure after prompting	States positivity and target contract before loop
Aggregate cost	Calls nested shrink quadratic or ignores map scans	Adds count after hint	Counts total pointer/deque moves and output

If answer timing scores below 2, replay the target-7 trace and mark each qualifying sum before removal. For removal weakness, rerun [1,2,1,3,3] with counts after both evictions. For assumptions, explain the signed counterexample without running code. Re-solve a changed lowest-scoring rung after 48 hours. These criteria are author-set learning targets, not hiring predictions.

Practice uses a timer and self-review. Sign in with an eligible Library account for related guides; optional coaching and AI services remain separate.

[Sliding Window Problem Ladder](#)

[Prefix Sum](#)

[Monotonic Stack](#)