Sliding Window

1. Longest Substring Without Repeating Characters

Pattern: Sliding Window

Problem Statement

> Given a string `s`, find the length of the **longest substring** without repeating characters.

Sample Input & Output

```
Input: "abcabcbb"
Output: 3
Explanation: The answer is "abc", with length 3.
```

```
Input: "bbbbb"
Output: 1
Explanation: All characters are the same; longest valid substring is "b".
```

```
Input: "pwwkew"
Output: 3
Explanation: "wke" is the longest substring without repeats
(not "pwke", which is a subsequence).
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        # STEP 1: Initialize structures
        #   - Use a set to track characters in current window
        #   - Use two pointers (left, right) to define window
        char_set = set()
        left = 0
        max_len = 0

        # STEP 2: Main loop / recursion
        #   - Expand window by moving right pointer
        #   - If duplicate found, shrink from left until unique
        for right in range(len(s)):
            # STEP 3: Update state / bookkeeping
            #   - Why here? Ensures window always has unique chars
            #   - What breaks if not? Duplicates would stay in set
            while s[right] in char_set:
                char_set.remove(s[left])
                left += 1
            char_set.add(s[right])
            max_len = max(max_len, right - left + 1)

        # STEP 4: Return result
        #   - Handles empty string (max_len stays 0)
        return max_len

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.lengthOfLongestSubstring("abcabcbb") == 3

    #  Test 2: Edge case
    assert sol.lengthOfLongestSubstring("bbbbb") == 1

    #  Test 3: Tricky/negative
    assert sol.lengthOfLongestSubstring("pwwkew") == 3
```

```python
# Extra: Empty string
assert sol.lengthOfLongestSubstring("") == 0

print("  All tests passed!")
```

> **How to use**: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `s = "pwwkew"` step by step.
Initial state: `char_set = {}`, `left = 0`, `max_len = 0`.

Step 1: `right = 0` → char `'p'`
- `'p'` not in `char_set` → skip while loop
- Add `'p'` → `char_set = {'p'}`
- `max_len = max(0, 0-0+1) = 1`

State: `left=0`, `right=0`, `set={'p'}`, `max_len=1`

Step 2: `right = 1` → char `'w'`
- `'w'` not in set → skip while
- Add `'w'` → `char_set = {'p','w'}`
- `max_len = max(1, 1-0+1) = 2`

State: `left=0`, `right=1`, `set={'p','w'}`, `max_len=2`

Step 3: `right = 2` → char `'w'`
- `'w'` **is** in set → enter while loop
- Remove `s[left] = 'p'` → `set = {'w'}`, `left = 1`
- Check again: `'w'` still in set → remove `s[1] = 'w'`
→ `set = {}`, `left = 2`
- Now `'w'` not in set → exit while
- Add `'w'` → `set = {'w'}`
- `max_len = max(2, 2-2+1) = 2`

State: `left=2`, `right=2`, `set={'w'}`, `max_len=2`

Step 4: `right = 3` → char `'k'`
- `'k'` not in set → skip while
- Add `'k'` → `set = {'w','k'}`
- `max_len = max(2, 3-2+1) = 2` → still 2

State: `left=2`, `right=3`, `set={'w','k'}`, `max_len=2`

Step 5: `right = 4` → char `'e'`
- `'e'` not in set → skip while
- Add `'e'` → `set = {'w','k','e'}`
- `max_len = max(2, 4-2+1) = 3`

State: `left=2`, `right=4`, `set={'w','k','e'}`, `max_len=3`

Step 6: `right = 5` → char `'w'`
- `'w'` **is** in set → enter while
- Remove `s[2] = 'w'` → `set = {'k','e'}`, `left = 3`
- Now `'w'` not in set → exit while
- Add `'w'` → `set = {'k','e','w'}`
- `max_len = max(3, 5-3+1) = 3` (unchanged)

Final State: `max_len = 3` → returned.

Complexity Analysis

- **Time Complexity**: `O(n)`

  Each character is visited **at most twice** — once by `right` pointer, once by `left` pointer. The inner `while` loop may seem nested, but it's amortized constant per character.

- **Space Complexity**: `O(min(m, n))`

  `m` = size of charset (e.g., ASCII = 128). The set stores at most all unique characters in the string, which is bounded by alphabet size or string length `n`, whichever is smaller.

2. Longest Repeating Character Replacement

Pattern: Sliding Window

Problem Statement

You are given a string `s` and an integer `k`. You can choose **any character** in the string and change it to **any other uppercase English character** at most `k` times.

Return the length of the **longest substring** containing the same letter after performing the above operations.

Sample Input & Output

```
Input: s = "ABAB", k = 2
Output: 4
Explanation: Replace the two 'A's with 'B's or vice versa → "BBBB".
```

```
Input: s = "AABABBA", k = 1
Output: 4
Explanation: Replace one 'B' in "AABABB" → "AAAABB" →
longest valid is "AABA" → "AAAA" (length 4).
```

```
Input: s = "AAAA", k = 0
Output: 4
Explanation: No replacements needed; entire string is already uniform.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def characterReplacement(self, s: str, k: int) -> int:
        # STEP 1: Initialize structures
        #   - freq: tracks count of each char in current window
        #   - max_freq: highest freq of any char in window
        #   - left: start of sliding window
        #   - max_len: result to return
        freq = [0] * 26
        max_freq = 0
        left = 0
        max_len = 0

        # STEP 2: Main loop / recursion
        #   - Expand window by moving right pointer
        #   - Invariant: window is valid if (window_size - max_freq) <= k
        for right in range(len(s)):
            # Update frequency of current character
            idx = ord(s[right]) - ord('A')
            freq[idx] += 1
            max_freq = max(max_freq, freq[idx])

            # STEP 3: Update state / bookkeeping
            #   - If window becomes invalid, shrink from left
            #   - Why? We want largest valid window ending at 'right'
            window_size = right - left + 1
```

```python
            if window_size - max_freq > k:
                left_idx = ord(s[left]) - ord('A')
                freq[left_idx] -= 1
                left += 1

            # Update max length after possible adjustment
            max_len = max(max_len, right - left + 1)

        # STEP 4: Return result
        #   - max_len holds the answer; handles empty string via init
        return max_len

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.characterReplacement("ABAB", 2) == 4, \
        f"Expected 4, got {sol.characterReplacement('ABAB', 2)}"

    #  Test 2: Edge case - no replacements needed
    assert sol.characterReplacement("AAAA", 0) == 4, \
        f"Expected 4, got {sol.characterReplacement('AAAA', 0)}"

    #  Test 3: Tricky/negative - limited replacements
    assert sol.characterReplacement("AABABBA", 1) == 4, \
        f"Expected 4, got {sol.characterReplacement('AABABBA', 1)}"

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `s = "AABABBA"`, `k = 1` step by step.

Initial state:
- `freq = [0]*26` → all zeros
- `max_freq = 0`

- `left = 0`
- `max_len = 0`

Step 1: `right = 0` → char `'A'`
- `idx = 0`
- `freq[0] = 1`
- `max_freq = max(0, 1) = 1`
- `window_size = 1`, `1 - 1 = 0 <= 1` → valid
- `max_len = max(0, 1) = 1`

State: `left=0`, `max_len=1`, `freq[A]=1`

Step 2: `right = 1` → char `'A'`
- `freq[0] = 2`
- `max_freq = 2`
- `window_size = 2`, `2 - 2 = 0 <= 1` → valid
- `max_len = 2`

State: `left=0`, `max_len=2`, `freq[A]=2`

Step 3: `right = 2` → char `'B'`
- `idx = 1`, `freq[1] = 1`
- `max_freq = max(2, 1) = 2`
- `window_size = 3`, `3 - 2 = 1 <= 1` → valid
- `max_len = 3`

State: `left=0`, `max_len=3`, `freq[A]=2, B=1`

Step 4: `right = 3` → char `'A'`
- `freq[0] = 3`
- `max_freq = 3`
- `window_size = 4`, `4 - 3 = 1 <= 1` → valid
- `max_len = 4`

State: `left=0`, `max_len=4`, `freq[A]=3, B=1`

Step 5: `right = 4` → char `'B'`
- `freq[1] = 2`
- `max_freq = max(3, 2) = 3`
- `window_size = 5`, `5 - 3 = 2 > 1` → **invalid!**
- Shrink window:
- `s[left] = 'A'` → `freq[0] = 2`
- `left = 1`
- New window: indices 1–4 → size = 4
- `max_len = max(4, 4) = 4`

State: `left=1`, `max_len=4`, `freq[A]=2, B=2`

Step 6: `right = 5` → char `'B'`
- `freq[1] = 3`
- `max_freq = max(2, 3) = 3`
- `window_size = 5` (1–5), `5 - 3 = 2 > 1` → invalid
- Shrink:
- `s[1] = 'A'` → `freq[0] = 1`
- `left = 2`
- New window: 2–5 → size = 4
- `max_len = 4`

State: `left=2`, `freq[A]=1, B=3`

Step 7: `right = 6` → char `'A'`
- `freq[0] = 2`
- `max_freq = max(3, 2) = 3`
- `window_size = 5` (2–6), `5 - 3 = 2 > 1` → invalid
- Shrink:
- `s[2] = 'B'` → `freq[1] = 2`
- `left = 3`
- New window: 3–6 → size = 4
- `max_len = 4`

Final Output: `4`

Key Insight:
We never reduce `max_freq` when shrinking — it may become stale, but that's okay!
Because we only care about **longer** windows, and a stale `max_freq` only makes the condition stricter (safe).

Complexity Analysis

- **Time Complexity**: `O(n)`

  We traverse the string once with `right` pointer. Each character is visited at most twice (once by `right`, once by `left`). All operations inside loop are `O(1)`.

- **Space Complexity**: `O(1)`

  The frequency array has fixed size 26 (uppercase English letters). No other space scales with input.

3. Minimum Window Substring

Pattern: Sliding Window + Hash Map (Two Pointers with Character Frequency Tracking)

Problem Statement

> Given two strings `s` and `t` of lengths `m` and `n` respectively, return the **minimum substring** of `s` such that every character in `t` (including duplicates) is included in the window.
> If there is no such substring, return the empty string `""`.
> The testcases will be generated such that the answer is **unique**.

Sample Input & Output

```
Input: s = "ADOBECODEBANC", t = "ABC"
Output: "BANC"
Explanation: "BANC" is the smallest substring containing all chars of "ABC".
```

```
Input: s = "a", t = "a"
Output: "a"
Explanation: Single character match - edge case with minimal input.
```

```
Input: s = "a", t = "aa"
Output: ""
Explanation: t has two 'a's but s only has one → impossible.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
from collections import Counter

class Solution:
    def minWindow(self, s: str, t: str) -> str:
        # STEP 1: Initialize structures
        #   - t_count tracks required char frequencies from t
        #   - window_count tracks current window char frequencies
        #   - have = chars in window meeting required count
        #   - need = total unique chars we must satisfy
        if not t or not s:
            return ""

        t_count = Counter(t)
        window_count = {}
        have = 0
        need = len(t_count)
        res = ""
        res_len = float('inf')
        left = 0
```

```python
        # STEP 2: Main loop / recursion
        #   - Expand right pointer to include new char
        #   - Update window_count and check if char requirement met
        for right in range(len(s)):
            char = s[right]
            window_count[char] = (
                window_count.get(char, 0) + 1
            )

            # STEP 3: Update state / bookkeeping
            #   - Only increment 'have' when count exactly matches need
            if (
                char in t_count
                and window_count[char] == t_count[char]
            ):
                have += 1

            # Contract window from left while valid
            while have == need:
                # Update result if current window smaller
                current_length = right - left + 1
                if current_length < res_len:
                    res = s[left:right + 1]
                    res_len = current_length

                # Remove leftmost char and update counts
                left_char = s[left]
                window_count[left_char] -= 1
                if (
                    left_char in t_count
                    and window_count[left_char] < t_count[left_char]
                ):
                    have -= 1
                left += 1

        # STEP 4: Return result
        #   - Return empty string if no valid window found
        if res_len == float('inf'):
            return ""
        return res

# ------------------- INLINE TESTS -------------------
```

```python
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    result1 = sol.minWindow("ADOBECODEBANC", "ABC")
    assert result1 == "BANC", f"Expected 'BANC', got '{result1}'"

    #  Test 2: Edge case
    result2 = sol.minWindow("a", "a")
    assert result2 == "a", f"Expected 'a', got '{result2}'"

    #  Test 3: Tricky/negative
    result3 = sol.minWindow("a", "aa")
    assert result3 == "", f"Expected '', got '{result3}'"

    print("  All tests passed!")
```

> **How to use**: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `s = "ADOBECODEBANC"`, `t = "ABC"` step by step.

Initial State:
- `t_count = {'A':1, 'B':1, 'C':1}`
- `window_count = {}`
- `have = 0`, `need = 3`
- `res = ""`, `res_len = ∞`
- `left = 0`

Step 1: `right = 0` → char `'A'`
- `window_count = {'A':1}`
- `'A'` in `t_count` and count matches → `have = 1`
- `have (1) < need (3)` → skip while loop
- **State**: `left=0`, `have=1`

Step 2: `right = 1` → `'D'`
- `window_count = {'A':1, 'D':1}`
- `'D'` not in `t_count` → `have` unchanged
- **State**: `left=0`, `have=1`

Step 3: `right = 2` → `'O'`
- Add `'O'` → `window_count['O']=1`
- Not in `t` → `have=1`

Step 4: `right = 3` → `'B'`
- `window_count['B']=1`
- `'B'` in `t` and count matches → `have = 2`
- Still < 3 → continue

Step 5: `right = 4` → `'E'`
- Add `'E'` → no effect on `have`

Step 6: `right = 5` → `'C'`
- `window_count['C']=1`
- `'C'` in `t` and count matches → `have = 3`
- Now `have == need` → enter while loop

  Inside while loop (valid window: "ADOBEC")
  - Length $= 6 < \infty$ → update `res = "ADOBEC"`, `res_len = 6`
  - Remove `s[0] = 'A'`:
  - `window_count['A'] = 0`
  - `'A'` in `t` and now `0 < 1` → `have = 2`
  - Exit while loop
  - **State**: `left=1`, `have=2`, `res="ADOBEC"`

Continue expanding `right`…

Eventually, at `right = 12` (`'C'`), window `"ODEBANC"` becomes valid again.

Then we contract:
- Remove `'O'` → still valid
- Remove `'D'` → still valid
- Remove `'E'` → still valid
- Remove `'B'` → `window_count['B']=1` (still ok)
- Remove `'A'` → `window_count['A']=0` → **invalid!**
- Before removal: window = `"BANC"` → length $4 < 6$ → **update `res = "BANC"`**

Final `res = "BANC"`.

Complexity Analysis

- **Time Complexity**: `O(m + n)`

  We traverse `s` with `right` pointer once (`O(m)`), and `left` moves forward at most `m` times total (each char visited at most twice). Building `t_count` is `O(n)`. Total linear.

- **Space Complexity**: `O(k)`

  Where `k` is the number of unique characters in `t` (at most 52 for upper+lower letters). `window_count` and `t_count` store only chars from `t`, so space is bounded by alphabet size, **not** input length. Technically `O(1)` for fixed alphabet, but `O(k)` generally.

4. Find All Anagrams in a String

Pattern: Sliding Window + Hash Map (Fixed-Size Window)

Problem Statement

Given two strings `s` and `p`, return an array of all the start indices of `p`'s anagrams in `s`. You may return the answer in any order.
An **anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Sample Input & Output

```
Input: s = "cbaebabacd", p = "abc"
Output: [0, 6]
Explanation: The substring starting at index 0 ("cba")
and index 6 ("bac") are anagrams of "abc".
```

```
Input: s = "abab", p = "ab"
Output: [0, 1, 2]
Explanation: Every consecutive 2-letter substring is an anagram of "ab".
```

```
Input: s = "a", p = "aa"
Output: []
Explanation: p is longer than s → no anagram possible.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
from collections import Counter

class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        # STEP 1: Initialize structures
        #   - 'need' tracks required char frequencies from p
        #   - 'window' tracks current sliding window frequencies
        need = Counter(p)
        window = Counter()
        res = []
        left = 0
        p_len = len(p)

        # Early exit if p longer than s
        if p_len > len(s):
            return []

        # STEP 2: Main loop - expand window with right pointer
        #   - Add current char to window
        for right, char in enumerate(s):
            window[char] += 1

            # STEP 3: Shrink window if too large
            #   - Maintain fixed size = len(p)
            if right - left + 1 > p_len:
                left_char = s[left]
                window[left_char] -= 1
                if window[left_char] == 0:
                    del window[left_char]
                left += 1
```

```python
            # STEP 4: Check for anagram match
            #   - Only when window is full size
            if right >= p_len - 1 and window == need:
                res.append(left)

        return res

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.findAnagrams("cbaebabacd", "abc") == [0, 6], \
        "Test 1 failed"

    #  Test 2: Edge case - p longer than s
    assert sol.findAnagrams("a", "aa") == [], \
        "Test 2 failed"

    #  Test 3: Tricky - overlapping anagrams
    assert sol.findAnagrams("abab", "ab") == [0, 1, 2], \
        "Test 3 failed"

    print(" All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **two detailed examples** to reinforce understanding.

Example 1: `s = "cbaebabacd"`, `p = "abc"`

Goal: Find all start indices where a 3-letter substring is an anagram of `"abc"`.

Initial Setup:
- `need = {'a':1, 'b':1, 'c':1}`

- `window = {}`
- `left = 0`, `p_len = 3`
- `res = []`

Now step through each `right` (0 to 9):

Step 1: right=0, char='c'
- `window = {'c':1}`
- Window size = 1 3 → no shrink
- `right (0) < 2` → skip match
- **State**: `left=0`, `res=[]`

Step 2: right=1, char='b'
- `window = {'c':1, 'b':1}`
- Size = 2 3 → no shrink
- `right (1) < 2` → skip match
- **State**: `res=[]`

Step 3: right=2, char='a'
- `window = {'c':1, 'b':1, 'a':1}`
- Size = 3 → no shrink
- `right` 2 → check: `window == need` → **True**
- Append `left = 0` → `res = [0]`
- **State**: `res=[0]`

Step 4: right=3, char='e'
- `window = {'c':1, 'b':1, 'a':1, 'e':1}`
- Size = 4 > 3 → shrink: remove `s[0]='c'`
→ `window = {'b':1, 'a':1, 'e':1}`, `left=1`
- Check: `{'b','a','e'} != {'a','b','c'}` → no match
- **State**: `res=[0]`

Step 5: right=4, char='b'
- `window = {'b':2, 'a':1, 'e':1}`
- Size = 4 > 3 → shrink: remove `s[1]='b'`
→ `window = {'b':1, 'a':1, 'e':1}`, `left=2`
- No match
- **State**: `res=[0]`

Step 6: right=5, char='a'
- `window = {'b':1, 'a':2, 'e':1}`
- Shrink: remove `s[2]='a'` → `window = {'b':1, 'a':1, 'e':1}`, `left=3`

- No match
- **State**: `res=[0]`

Step 7: right=6, char='b'
- `window = {'b':2, 'a':1, 'e':1}`
- Shrink: remove `s[3]='e'` → `window = {'b':2, 'a':1}`, `left=4`
- No match
- **State**: `res=[0]`

Step 8: right=7, char='a'
- `window = {'b':2, 'a':2}`
- Shrink: remove `s[4]='b'` → `window = {'b':1, 'a':2}`, `left=5`
- No match
- **State**: `res=[0]`

Step 9: right=8, char='c'
- `window = {'b':1, 'a':2, 'c':1}`
- Shrink: remove `s[5]='a'` → `window = {'b':1, 'a':1, 'c':1}`, `left=6`
- Check: `window == need` → **True**
- Append `left = 6` → `res = [0, 6]`
- **State**: `res=[0,6]`

Step 10: right=9, char='d'
- `window = {'b':1, 'a':1, 'c':1, 'd':1}`
- Shrink: remove `s[6]='b'` → `window = {'a':1, 'c':1, 'd':1}`, `left=7`
- No match
- **Final**: `res = [0, 6]`

Example 2: `s = "a"`, `p = "aa"` (Edge Case)

- `p_len = 2`, `len(s) = 1`
- Early exit: `2 > 1` → return `[]` immediately
- **No loop runs** → efficient!

Complexity Analysis

- **Time Complexity**: `O(n)`

  Single pass over `s` (`n = len(s)`). Each character is added once and removed once. Dictionary comparison (`window == need`) is `O(1)` because the alphabet is bounded (26 lowercase letters).

- **Space Complexity**: `O(1)`

  Both `need` and `window` store at most 26 key-value pairs. Space usage is constant, independent of input size.

5. Sliding Window Maximum

Pattern: Sliding Window

Problem Statement

You are given an array of integers `nums`, there is a sliding window of size `k` which is moving from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

Sample Input & Output

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

```
 1  3  -1  -3 [5  3  6] 7      6
 1  3  -1  -3  5 [3  6  7]     7
```

```
Input: nums = [1], k = 1
Output: [1]
Explanation: Only one window possible.
```

```
Input: nums = [1,-1], k = 1
Output: [1, -1]
Explanation: Window size is 1, so each element is its own max.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
from collections import deque

class Solution:
    def maxSlidingWindow(
        self, nums: List[int], k: int
    ) -> List[int]:
        # STEP 1: Initialize structures
        #   - Use deque to store indices of potential max values
        #   - Maintain decreasing order: front = current max
        dq = deque()
        result = []

        for i in range(len(nums)):
            # STEP 2: Remove indices outside current window
            #   - Window is [i - k + 1, i]
            #   - If front index <= i - k, it's out of bounds
            if dq and dq[0] <= i - k:
                dq.popleft()

            # STEP 3: Maintain decreasing order in deque
            #   - Remove from back while current num >= back val
            #   - Ensures front always holds max for current win
            while dq and nums[dq[-1]] <= nums[i]:
```

```
                dq.pop()

            # STEP 4: Add current index to deque
            dq.append(i)

            # STEP 5: Record max once first window is complete
            #   - First valid window ends at index k - 1
            if i >= k - 1:
                result.append(nums[dq[0]])

        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.maxSlidingWindow(
        [1,3,-1,-3,5,3,6,7], 3
    ) == [3,3,5,5,6,7], "Normal case failed"

    #  Test 2: Edge case - single element
    assert sol.maxSlidingWindow([1], 1) == [1], \
        "Single element failed"

    #  Test 3: Tricky case - window size 1
    assert sol.maxSlidingWindow([1,-1], 1) == [1, -1], \
        "Window size 1 failed"

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `nums = [1,3,-1,-3,5,3,6,7]`, `k = 3`.

Initial state:
`dq = deque()` (empty), `result = []`

---

i = 0 (`num = 1`)
- `dq` empty → skip popleft
- `dq` empty → skip while loop
- Append `0` → `dq = [0]`
- `i=0 < 2` (k-1) → skip append to result
→ **State**: `dq=[0]`, `result=[]`

---

i = 1 (`num = 3`)
- `dq[0]=0 > 1-3=-2` → no popleft
- While: `nums[0]=1 <= 3` → pop `0` → `dq=[]`
- Append `1` → `dq = [1]`
- `i=1 < 2` → skip result
→ **State**: `dq=[1]`, `result=[]`

---

i = 2 (`num = -1`)
- `dq[0]=1 > 2-3=-1` → no popleft
- While: `nums[1]=3 > -1` → don't pop
- Append `2` → `dq = [1,2]`
- `i=2 >= 2` → add `nums[1]=3` to result
→ **State**: `dq=[1,2]`, `result=[3]`

---

i = 3 (`num = -3`)
- `dq[0]=1 <= 3-3=0`? No (1>0) → keep
- While: `nums[2]=-1 > -3` → don't pop
- Append `3` → `dq = [1,2,3]`
- Add `nums[1]=3` → `result=[3,3]`
→ **State**: `dq=[1,2,3]`, `result=[3,3]`

---

i = 4 (`num = 5`)
- `dq[0]=1 <= 4-3=1` → **yes!** → popleft → `dq=[2,3]`
- While:
- `nums[3]=-3 <= 5` → pop → `dq=[2]`
- `nums[2]=-1 <= 5` → pop → `dq=[]`
- Append `4` → `dq=[4]`
- Add `nums[4]=5` → `result=[3,3,5]`
→ **State**: `dq=[4]`, `result=[3,3,5]`

---

i = 5 (`num = 3`)
- `dq[0]=4 > 5-3=2` → keep
- While: `nums[4]=5 > 3` → don't pop
- Append `5` → `dq=[4,5]`
- Add `nums[4]=5` → `result=[3,3,5,5]`
→ **State**: `dq=[4,5]`, `result=[3,3,5,5]`

---

i = 6 (`num = 6`)
- `dq[0]=4 <= 6-3=3`? No (4>3) → keep
- While:
- `nums[5]=3 <= 6` → pop → `dq=[4]`
- `nums[4]=5 <= 6` → pop → `dq=[]`
- Append `6` → `dq=[6]`
- Add `nums[6]=6` → `result=[3,3,5,5,6]`
→ **State**: `dq=[6]`, `result=[3,3,5,5,6]`

---

i = 7 (`num = 7`)
- `dq[0]=6 > 7-3=4` → keep
- While: `nums[6]=6 <= 7` → pop → `dq=[]`
- Append `7` → `dq=[7]`
- Add `nums[7]=7` → `result=[3,3,5,5,6,7]`
→ **Final result**: `[3,3,5,5,6,7]`

Key insight: The deque always keeps indices of elements in **decreasing order**, so the front is always the max of the current window. Out-of-window indices are removed from the front; smaller-or-equal elements are removed from the back before adding the new one.

Complexity Analysis

- **Time Complexity**: `O(n)`

  Each element is pushed and popped from the deque **at most once**. The outer loop runs `n` times, and inner while loop operations are amortized constant time.

- **Space Complexity**: `O(k)`

  The deque stores at most `k` indices (one per window position). The output list is `O(n - k + 1)`, but auxiliary space is dominated by the deque, which is `O(k)`.