

Sliding Window Maximum

Find the max of every window of size K in $O(n)$ using Deque ■

● HARD

■ Hook

For every window of size K in an array, find the maximum element. Brute force is $O(nK)$ — way too slow for large inputs. This problem teaches the **Monotonic Deque** — one of the most powerful data structures in competitive programming. It's asked at Google, Amazon, and Microsoft as a senior-level problem.

■ Core Intuition — Monotonic Decreasing Deque

Maintain a deque (double-ended queue) of **indices**. The deque is always ordered such that **the front holds the index of the maximum element** for the current window.

- **Rule 1 — Expire old:** If front index is outside window ($\text{front} \leq \text{right} - K$), pop from front.
- **Rule 2 — Maintain decreasing:** Before adding right, pop from back all indices whose value $\leq \text{arr}[\text{right}]$. They'll never be the max.
- **Rule 3 — Add right:** Push right to back.
- **Rule 4 — Record max:** Once $\text{right} \geq K-1$, $\text{arr}[\text{deque.front()}]$ is the window max.

■ Visual — [1,3,-1,-3,5,3,6,7], $K=3$

```

Array: [1, 3, -1, -3, 5, 3, 6, 7]    K=3
Index:  0  1  2  3  4  5  6  7

r=0: add 0 → deque=[0]          (val=1)   window not full yet
r=1: 3>1 → pop 0; add 1 → deque=[1]      (val=3)
r=2: -1<3 → add 2 → deque=[1,2] (vals=3,-1)   window full!
      front=1 (in range 0..2) → max = arr[1] = 3 ✓
r=3: -3<-1 → add 3 → deque=[1,2,3] (3,-1,-3) window=[1..3]
      front=1 (in range 1..3) → max = arr[1] = 3 ✓
r=4: 5 > all → pop 3,2,1; add 4 → deque=[4]   window=[2..4]
      front=4 (in range 2..4) → max = arr[4] = 5 ✓
r=5: 3<5 → add 5 → deque=[4,5]   window=[3..5]
      front=4 (in range 3..5) → max = arr[4] = 5 ✓
r=6: 6>3,5 → pop 5,4; add 6 → deque=[6]   window=[4..6]
      front=6 → max = arr[6] = 6 ✓
r=7: 7>6 → pop 6; add 7 → deque=[7]   window=[5..7]
      front=7 → max = arr[7] = 7 ✓

★ RESULT = [3, 3, 5, 5, 6, 7]

```

■ Code Implementation (Java)

```

import java.util.ArrayDeque;
import java.util.Deque;

public class SlidingWindowMaximum {

    public int[] maxSlidingWindow(int[] nums, int k) {
        int n = nums.length;
        int[] result = new int[n - k + 1];
        Deque<Integer> dq = new ArrayDeque<>(); // stores INDICES

        for (int right = 0; right < n; right++) {
            // Rule 1: remove indices outside the window
            while (!dq.isEmpty() && dq.peekFirst() <= right - k)
                dq.pollFirst();

            // Rule 2: remove smaller elements from back
            while (!dq.isEmpty() && nums[dq.peekLast()] <= nums[right])
                dq.pollLast();

            // Rule 3: add current index
            dq.offerLast(right);

            // Rule 4: record max once window is full
            if (right >= k - 1)
                result[right - k + 1] = nums[dq.peekFirst()];
        }
        return result;
    }
}

```

■ Time & Space Complexity

■ TIME COMPLEXITY

$O(n)$

■ SPACE COMPLEXITY

$O(K)$

Each element is pushed and popped from the deque at most once $\rightarrow O(2n) = O(n)$. Deque holds at most K elements at any time $\rightarrow O(K)$ space. Output array is $O(n-K+1) \approx O(n)$.

■ Common Mistakes

■ Storing values in deque instead of indices

You need indices to check if the front is outside the window ($\text{front} \leq \text{right} - k$). If you store values, you can't do this check. Always store indices.

■ ■ Wrong comparison: $<$ vs $\leq$

Use `nums[dq.peekLast()] <= nums[right]` (with equals). If you use strict $<$, equal values won't be removed, causing stale duplicates to remain in the deque.

■ Patterns & Connections — Monotonic Structures

The Monotonic Deque pattern generalizes to:

- Sliding Window Minimum (same idea, increasing deque)
- Sum of Subarray Minimums (monotonic stack)
- Largest Rectangle in Histogram (monotonic stack)
- Jump Game VI — DP optimized with monotonic deque
- Any 'range min/max query in sliding window' problem

■ Practice Problems

#	Problem	Difficulty	Pattern
1	Sliding Window Maximum (LC 239)	Hard	Monotonic Deque
2	Sliding Window Minimum	Hard	Monotonic Deque (increasing)
3	Sum of Subarray Minimums (LC 907)	Medium	Monotonic Stack
4	Jump Game VI (LC 1696)	Medium	DP + Monotonic Deque
5	Constrained Subsequence Sum (LC 1425)	Hard	DP + Monotonic Deque

■ Quick Revision

- ✓ Deque stores INDICES (not values)
- ✓ Deque is monotonically DECREASING (front = max)
- ✓ Before adding right: pop back while `nums[back] ≤ nums[right]`
- ✓ Before recording: pop front while `front ≤ right - k`
- ✓ Record `result[right-k+1] = nums[dq.peekFirst()]` when `right ≥ k-1`
- ✓ Time: $O(n)$ — each element pushed/popped at most once
- ✓ Space: $O(K)$ for deque

■ MENTAL MODEL — Remember This Forever

Think of the deque as a 'candidate list' for the maximum. Before a new element joins, any existing candidate weaker than it is eliminated — they can never be maximum while the new element is in the window. The front of the deque is always 'the current champion'. Expired champions are fired from the front. New stronger challengers kick out all weaker ones from the back before entering.