
🔷 **WEEK 2 – DAY 10**

Queue, Deque, Sliding Window, Brute-Force Protection

---

1. Queue vs Deque — Interview Foundation

Queue (FIFO)

- Insert at rear, remove from front
- Use cases: task scheduling, BFS, buffering

Deque (Double-Ended Queue)

- Insert/remove from **both ends**
- Enables **sliding window optimization**

Spoken Explanation (What You Say Aloud)

"A **queue** follows FIFO—first in, first out—and is used when order of processing matters, such as task scheduling or BFS.
A **deque** allows insertion and deletion from both ends in constant time, which makes it ideal for optimizing sliding window problems like finding maximum or minimum values efficiently."

Core Differences (Say Briefly)

Feature	Queue	Deque
Insertion	Rear only	Front & Rear
Deletion	Front only	Front & Rear
Typical Use	BFS, scheduling	Sliding window, monotonic structures
Time Ops	O(1)	O(1)

When Sony Expects Deque

- Sliding window maximum / minimum
- Continuous data streams
- Performance-critical real-time systems
- Window-based analytics (logs, metrics)

---

2. Problem: Sliding Window Maximum

Interview Question

Given an array nums and window size k, return the maximum element in every sliding window.

Brute Force (Reject This)

- For every window, scan k elements
- **O(n × k)** ❌

🗣️ Spoken Explanation (What You Say Aloud)

"This problem asks for the maximum in every contiguous subarray of size k.
Brute force would scan every window, which is O(n·k), so it's not optimal.
I use a **monotonic decreasing deque** that stores **indices**, not values.
The front of the deque always represents the maximum of the current window.
As I slide the window, I remove indices that go out of range and discard smaller elements from the back because they can never become the maximum."

🧠 Key Insight (Say This Clearly)

"I maintain a decreasing deque of indices.
The front always holds the maximum for the current window."

📜 Rules of the Deque

1. Remove indices **out of window** from the front
2. Remove **smaller elements** from the back
3. Front of deque = **current window maximum**

🧪 Example Input

nums = [1,3,-1,-3,5,3,6,7]

k = 3

Output = [3,3,5,5,6,7]

🐍 Python Solution (with INPUT)

File: sliding_window_maximum.py

from collections import deque

```
def maxSlidingWindow(nums, k):
    if not nums or k == 0:
        return []

    dq = deque()
    result = []

    for i in range(len(nums)):
        # Remove out-of-window indices
        if dq and dq[0] <= i - k:
            dq.popleft()

        # Maintain decreasing order
        while dq and nums[dq[-1]] < nums[i]:
            dq.pop()

        dq.append(i)

        # Window formed
        if i >= k - 1:
            result.append(nums[dq[0]])

    return result


# ---- INPUT ----
nums = [1, 3, -1, -3, 5, 3, 6, 7]
k = 3

print("Sliding Window Maximum:", maxSlidingWindow(nums, k))
```

🌐 **JavaScript Solution (with INPUT)**

```
// File: slidingWindowMaximum.js
function maxSlidingWindow(nums, k) {
  if (nums.length === 0 || k === 0) return [];

  const dq = [];
  const res = [];

  for (let i = 0; i < nums.length; i++) {
    // Remove out-of-window indices
    if (dq.length && dq[0] <= i - k) dq.shift();

    // Maintain decreasing order
    while (dq.length && nums[dq[dq.length - 1]] < nums[i]) {
      dq.pop();
    }

    dq.push(i);

    if (i >= k - 1) {
      res.push(nums[dq[0]]);
    }
  }
  return res;
}

// ---- INPUT ----
const nums = [1, 3, -1, -3, 5, 3, 6, 7];
const k = 3;
```

```
console.log("Sliding Window Maximum:", maxSlidingWindow(nums, k));
```

⏱️ Complexity

- **Time:** O(n) — each element enters and exits deque once
- **Space:** O(k) — deque holds at most k elements

Sony Follow-Up Questions (Very Common)

1. Why store indices instead of values?
2. What happens if duplicates exist?
3. Can this be solved using heap?
4. Sliding window minimum?
5. Real-time stream adaptation?

1️⃣ Why store indices instead of values?

Spoken

"I store indices so I can easily check whether an element is outside the current window. Values alone don't give position information, which is essential for sliding windows."

2️⃣ What happens if duplicates exist?

Spoken

"Duplicates are handled correctly. The deque keeps indices in decreasing order. If values are equal, the earlier index stays until it goes out of the window."

3️⃣ Can this be solved using a heap?

Spoken

"Yes, using a max heap gives O(n log k), but it's slower. The monotonic deque gives the optimal O(n) solution, which is why it's preferred in performance-critical systems."

4️⃣ How do you find sliding window minimum?

Spoken

"I reverse the logic and maintain a monotonic **increasing deque** instead of decreasing. The front will then give the minimum."

5️⃣ How would you adapt this for a real-time stream?

Spoken

"This approach works naturally for streaming data. As each element arrives, I update the deque and output the max once the window size is reached—no need to store the entire stream."

---

3. Pattern Recognition (Very Important)

If you hear:

- "Subarray of size k"
- "Maximum/minimum in range"
- "Continuous window"

➡️ Think Deque immediately

Below is a **Sony-interview–ready explanation** of **Pattern Recognition for Deque**, with **spoken answers**, **clear reasoning**, **example problem**, **Python + JavaScript code with INPUT**, and **recommended program names**.

3️⃣ Pattern Recognition — VERY IMPORTANT (Sony Favorite)

🗣️ What You Should Say (Out Loud)

"Whenever I hear phrases like *subarray of size k*, *maximum or minimum in a range*, or *continuous window*, I immediately think of a **deque with a sliding window**.
This is because a deque allows me to maintain candidates for max or min efficiently in linear time."

🎯 Keywords → Correct Pattern (Say This Confidently)

If you hear:

- **"Subarray of size k"**
- **"Maximum / minimum in range"**
- **"Continuous window"**
- **"Streaming data with window"**

➡️ Think: Sliding Window + Monotonic Deque

❌ Why NOT Brute Force?

Spoken

“Brute force recomputes the max for every window, leading to O(n·k) time, which doesn’t scale for large inputs.”

✅ Why Deque Works (Core Reason)

Spoken

“With a monotonic deque, each element is added and removed at most once, giving O(n) time complexity.”

🧠 Canonical Example Problem

Sliding Window Maximum

🐍 Python Example (with INPUT)

```
# File: pattern_sliding_window_deque.py

from collections import deque


def slidingWindowMax(nums, k):
    dq = deque()
    result = []


    for i in range(len(nums)):
        # Remove indices outside the window
        if dq and dq[0] <= i - k:
            dq.popleft()


        # Maintain decreasing order
        while dq and nums[dq[-1]] < nums[i]:
            dq.pop()


        dq.append(i)


        if i >= k - 1:
```

```
            result.append(nums[dq[0]])

    return result


# ---- INPUT ----
nums = [4, 2, 12, 3, 8, 7]
k = 3

print("Sliding Window Maximum:", slidingWindowMax(nums, k))
```

Output

```
[12, 12, 12, 8]
```

🌐 **JavaScript Example (with INPUT)**

```
// File: patternSlidingWindowDeque.js
function slidingWindowMax(nums, k) {
    const dq = [];
    const res = [];

    for (let i = 0; i < nums.length; i++) {
        if (dq.length && dq[0] <= i - k) dq.shift();

        while (dq.length && nums[dq[dq.length - 1]] < nums[i]) {
            dq.pop();
        }

        dq.push(i);

        if (i >= k - 1) {
            res.push(nums[dq[0]]);
        }
```

```
    }
    return res;
}

// ---- INPUT ----
const nums = [4, 2, 12, 3, 8, 7];
const k = 3;

console.log("Sliding Window Maximum:", slidingWindowMax(nums, k));
```

🎤 Sony Follow-Up Questions (Answered Clearly)

1️⃣ Why deque instead of stack or heap?

Spoken

“A deque supports removing from both ends in O(1), which is critical for sliding windows. A heap adds log-k overhead.”

2️⃣ When does this pattern fail?

Spoken

“If the window is not contiguous or if we need global sorting, deque is not appropriate.”

3️⃣ Can this handle negative numbers?

Spoken

“Yes. The logic is value-agnostic—it works for positive, negative, or mixed numbers.”

4️⃣ Sliding window max vs prefix sum?

Spoken

“Prefix sum helps with sums, not max or min. Deque is the right structure for range extremes.”

5️⃣ Real-world use case?

Spoken

“Used in real-time analytics, CPU load monitoring, stock price tracking, and network traffic smoothing.”

✅ **Sony One-Line Rule (Memorize This)**

“If the problem says **continuous window + max/min**, the optimal solution is **sliding window with a monotonic deque**.”

---

4. Project: Brute-Force Protection Logic (Security Design)

This directly builds on:

- Login logging (Day 8)
- Failed login counter (Day 9)

Now you **enforce protection**, not just record data.

Goal

Prevent attackers from:

- Credential stuffing
- Password guessing
- User enumeration

Core Strategy (Industry-Standard)

Controls

1. **Rate limiting**
2. **Failure threshold**
3. **Time window**
4. **Temporary lock**
5. **Progressive delay**

High-Level Logic (Say This in Interviews)

“I track failed attempts per user/IP within a sliding time window. If attempts exceed a threshold, I temporarily block further logins and progressively increase delay.”

Sliding Window Idea (Deque Concept Applied)

Each user/IP keeps timestamps of failed attempts.

Attempt timestamps: [t1, t2, t3, t4, t5]

Window: last 10 minutes

If count > 5 → block

This is conceptually **Sliding Window Maximum**, but applied to **security events**.

Pseudocode Logic

```
MAX_ATTEMPTS = 5
WINDOW_SECONDS = 600
BLOCK_TIME = 900

def is_bruteforce(user):
    attempts = get_attempts(user)
    now = current_time()

    # Remove old attempts
    attempts = [t for t in attempts if now - t < WINDOW_SECONDS]

    if len(attempts) >= MAX_ATTEMPTS:
        block_user(user)
        return True

    return False
```

Real Implementation Notes (Sony-Level Answer)

- Use **Redis** for fast expiry
- Use **TTL keys** for windows
- Store counters per:
    - username
    - IP
    - device fingerprint

Redis-Style Design

login_fail:user123 → [timestamps]

TTL: 10 minutes

Advanced Enhancements (Optional but Impressive)

- CAPTCHA after N attempts
- Exponential backoff
- Geo-based risk scoring
- Alert on anomaly detection

Sony Follow-Ups on Brute-Force Protection

1. Why sliding window over fixed counter?
2. How to handle distributed attacks?
3. Redis vs database?
4. How to avoid locking real users?
5. How to prevent timing attacks?

Strong Closing Statement (Memorize This)

“I combine sliding-window rate limiting with progressive lockouts, implemented using Redis for low latency and backed by persistent storage for auditability.”