Stack and Queue tricks

Stack and Queue Tricks in Python DSA Master

Below is a **DSA-Mastery level, fully structured "Stack & Queue Tricks in Python" guide** — packed with **patterns, templates, one-liners, LC mappings, optimizations, amortized analysis, and pro-level tips** for competitive programming + interviews.

🧱 STACK & QUEUE — DSA MASTERY (Python)

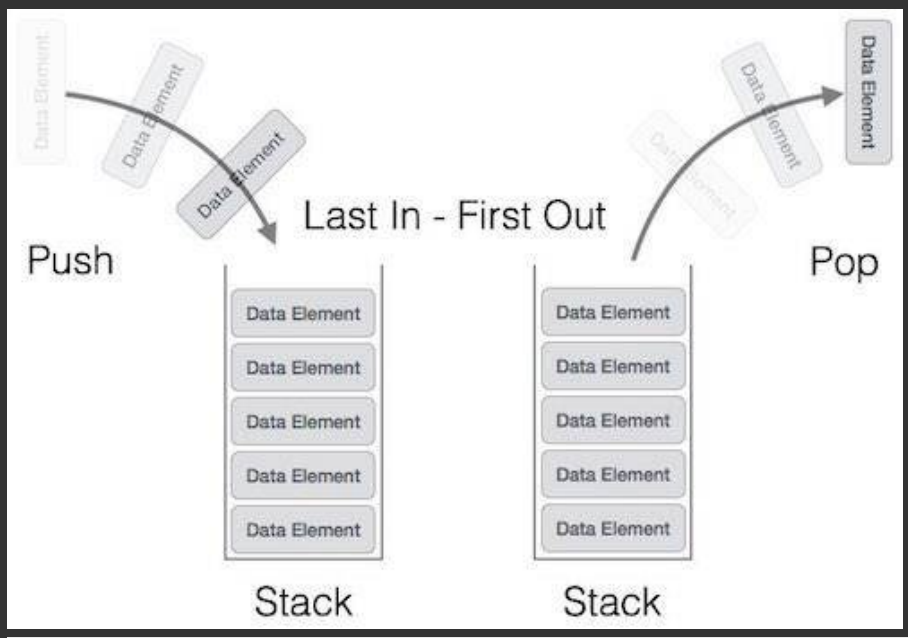
🚀 Quick Reference: When to Use What

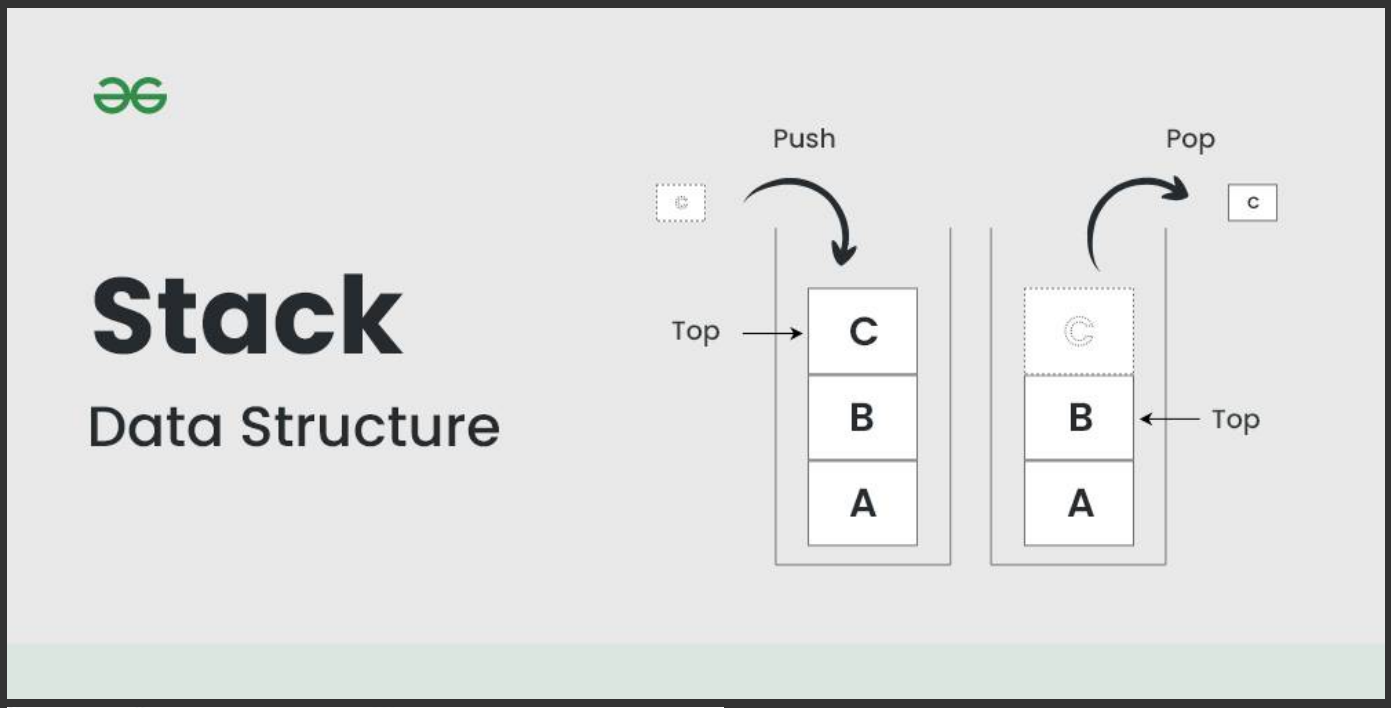
Structure	When to Use	Key Pattern
Stack (LIFO)	Previous element dependency, monotonic behavior, reversal, DFS	Monotonic Stack, Two-Stack queue, State rollback
Queue (FIFO)	Level processing, distance (BFS), sliding window, producer-consumer	BFS, Monotonic Queue, Multi-source BFS
Deque	Need pop from both ends, optimally	Sliding Window Max, BFS 0-1

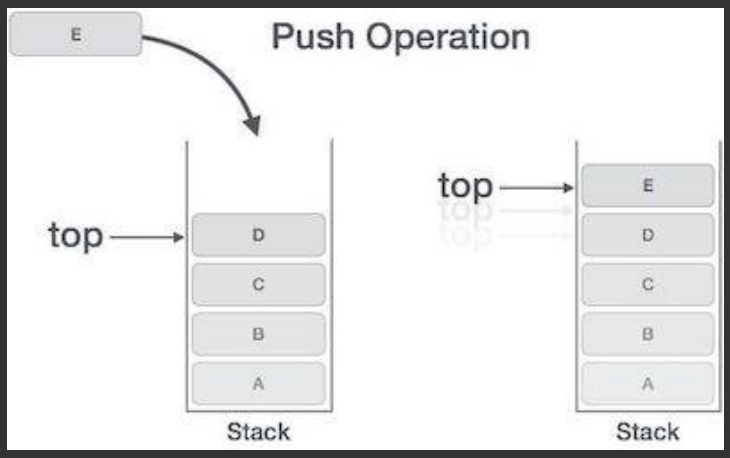
==========

🟦 STACK MASTERY

==========







4

✅ 1. Basic Python Stack Patterns

Using list (fast for append/pop at end)

```python
stack = []
stack.append(x)   # push
stack.pop()       # pop
stack[-1]         # peek
```

Using `collections.deque` (if you need faster operations)

```python
from collections import deque
stack = deque()
stack.append(x)
stack.pop()
```

✅ 2. Monotonic Stack Patterns

Increasing / Decreasing stacks used in:

✔️ Next Greater Element
✔️ Stock Span
✔️ Daily Temperatures
✔️ Largest Rectangle in Histogram
✔️ Trapping Rain Water

🔥 Template: Next Greater Element

```python
def next_greater(nums):
    res = [-1] * len(nums)
    stack = []  # stores indices
    for i, v in enumerate(nums):
        while stack and nums[stack[-1]] < v:
            res[stack.pop()] = v
        stack.append(i)
    return res
```

🔥 Template: Previous Greater Element

```python
for i in range(n):
    while stack and stack[-1] <= arr[i]:
        stack.pop()
    prev[i] = stack[-1] if stack else -1
    stack.append(arr[i])
```

✅ 3. Valid Parentheses / Balanced Symbols

Pattern: Push opening → match closing.

```python
def valid(s):
    mp = {')':'(', ']':'[', '}':'{'}
    stack = []
    for ch in s:
        if ch in mp.values():
            stack.append(ch)
        else:
            if not stack or stack.pop() != mp[ch]:
                return False
    return not stack
```

✅ 4. Using Two Stacks

4.1 Queue using two stacks

```python
class MyQueue:
    def __init__(self):
        self.in_st = []
        self.out_st = []

    def push(self, x):
        self.in_st.append(x)

    def pop(self):
        if not self.out_st:
            while self.in_st:
                self.out_st.append(self.in_st.pop())
        return self.out_st.pop()

    def peek(self):
        self.pop()
        return self.out_st[-1]

    def empty(self):
        return not self.in_st and not self.out_st
```

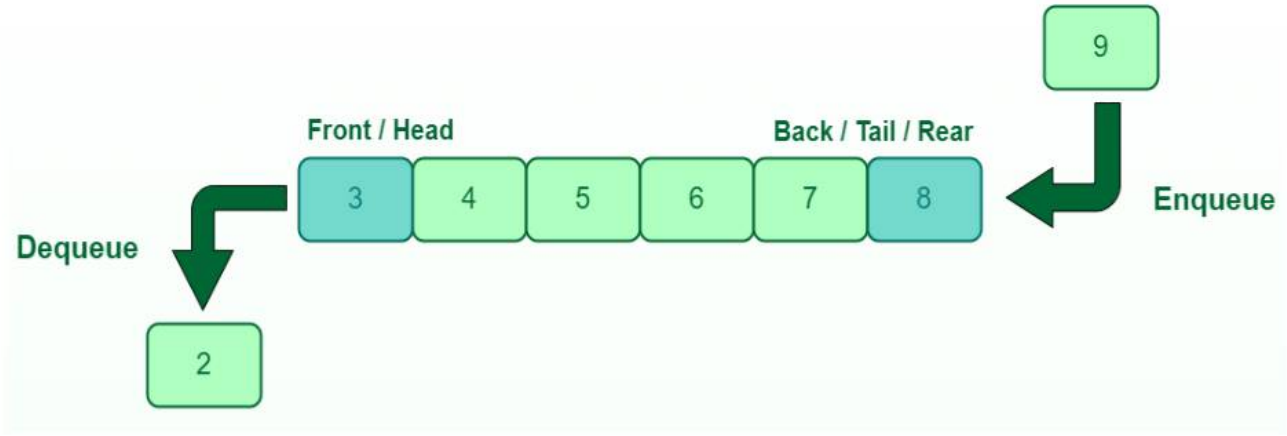
4.2 Min Stack (O(1) min)

```python
class MinStack:
    def __init__(self):
        self.st = []
        self.min_st = []

    def push(self, x):
        self.st.append(x)
        mn = x if not self.min_st else min(x, self.min_st[-1])
        self.min_st.append(mn)

    def pop(self):
        self.st.pop()
        self.min_st.pop()

    def getMin(self):
        return self.min_st[-1]
```

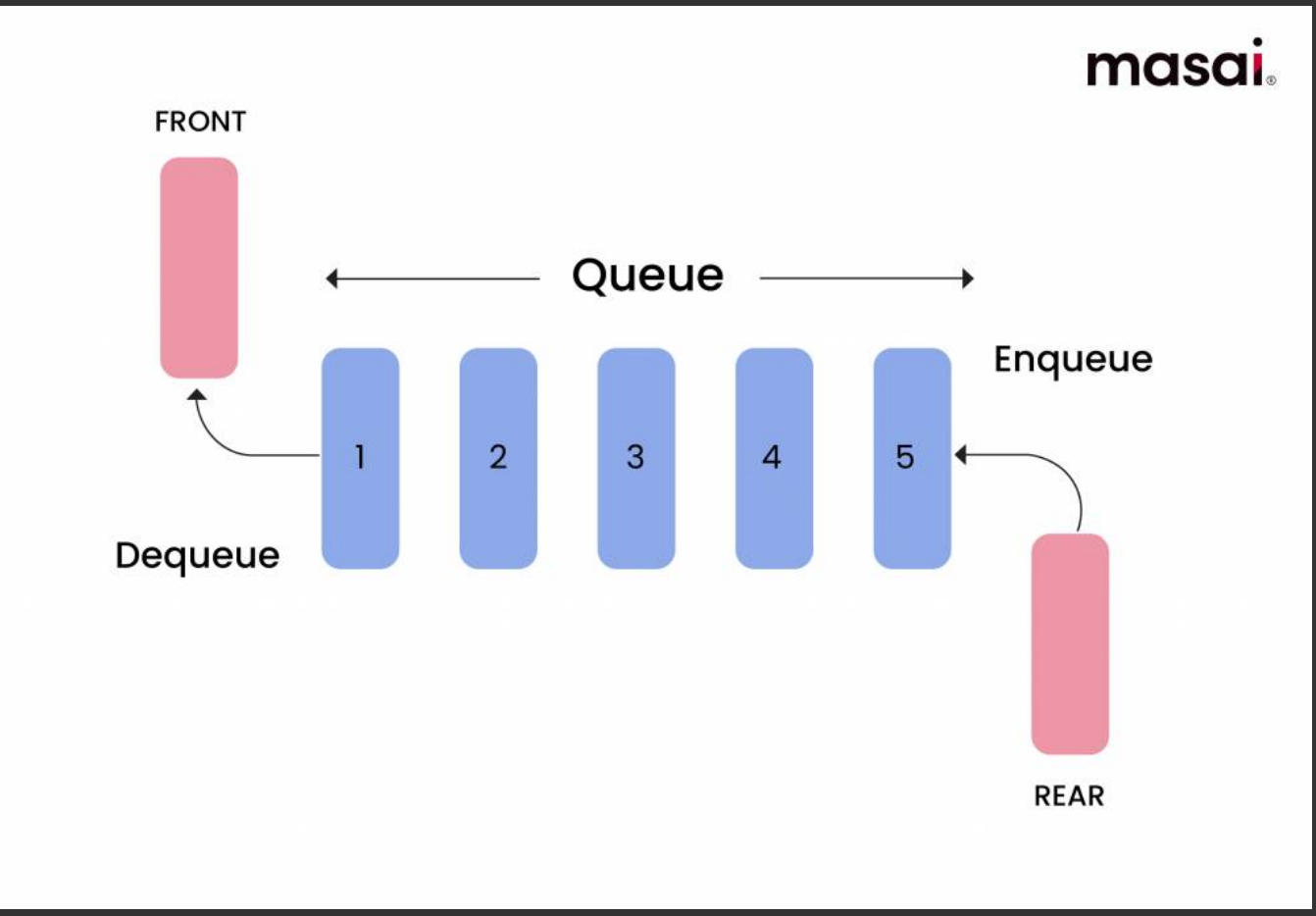
==========

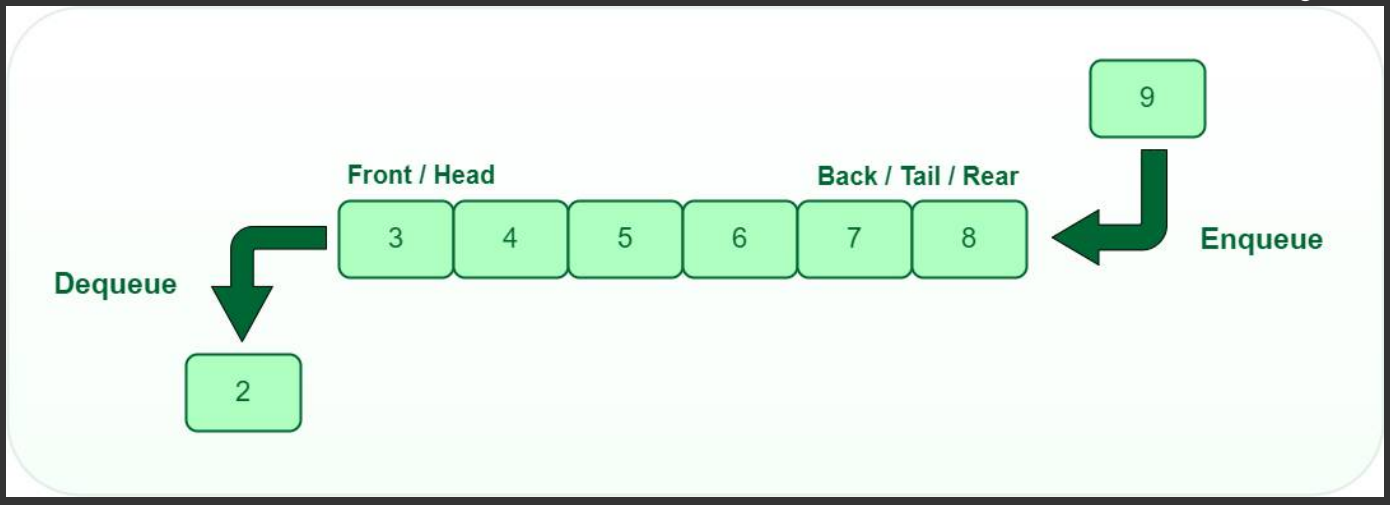
🟩 QUEUE MASTERY

==========



Queue Data Structure





4

✅ 1. Queue Basics

```
python

from collections import deque

q = deque()
q.append(x)      # enqueue
q.popleft()      # dequeue
q[0]             # front
```

✅ 2. BFS Templates (Graph + Tree)

Graph BFS

```
python

def bfs(start):
    q = deque([start])
    vis = {start}
    while q:
        node = q.popleft()
        for nei in graph[node]:
            if nei not in vis:
                vis.add(nei)
                q.append(nei)
```

Tree BFS (level order)

```
python

def levelOrder(root):
    if not root: return []
    q = deque([root])
    res = []
    while q:
        level = []
        for _ in range(len(q)):
            node = q.popleft()
            level.append(node.val)
            if node.left: q.append(node.left)
            if node.right: q.append(node.right)
```

```
            res.append(level)
        return res
```

✅ 3. Monotonic Queue (for Sliding Window Maximum)

Used in:
✔️ Sliding Window Maximum
✔️ Shortest Subarray with Sum ≥ K
✔️ Dynamic windows

🔥 O(n) Sliding Window Maximum

```
python


from collections import deque

def sliding_max(nums, k):
    dq = deque()  # store indices in decreasing order
    res = []

    for i, v in enumerate(nums):
        while dq and nums[dq[-1]] <= v:
            dq.pop()
        dq.append(i)

        if dq[0] == i - k:
            dq.popleft()

        if i >= k - 1:
            res.append(nums[dq[0]])

    return res
```

✅ 4. BFS Variants

4.1 Multi-source BFS

```
python


q = deque(sources)
dist = {src:0 for src in sources}
```

4.2 0-1 BFS (deque)

Used in shortest path with edges 0 or 1.

```
python


dq = deque()
dq.append(start)
dist = [inf]*n
dist[start] = 0

while dq:
    u = dq.popleft()
    for v,w in adj[u]:
        if dist[u] + w < dist[v]:
```

```
            dist[v] = dist[u] + w
            if w == 1:
                dq.append(v)
            else:
                dq.appendleft(v)
```

==========

🟧 ADVANCED STACK TRICKS

==========

1. Stack for String Reduction

```
python

def remove_adj_dup(s):
    st = []
    for ch in s:
        if st and st[-1] == ch:
            st.pop()
        else:
            st.append(ch)
    return ''.join(st)
```

---

2. Largest Rectangle in Histogram (O(n))

Core monotonic stack trick.

```
python

def largestRectangleArea(h):
    st = []
    h.append(0)
    maxA = 0

    for i, ht in enumerate(h):
        while st and h[st[-1]] > ht:
            H = h[st.pop()]
            W = i if not st else i - st[-1] - 1
            maxA = max(maxA, H * W)
        st.append(i)

    return maxA
```

---

3. Trapping Rain Water (Two Pointer or Stack)

Stack variant:

```
python

stack = []
water = 0

for i, h in enumerate(height):
    while stack and height[stack[-1]] < h:
```

```
            top = stack.pop()
            if not stack: break
            dist = i - stack[-1] - 1
            bounded = min(height[stack[-1]], h) - height[top]
            water += dist * bounded
        stack.append(i)
```

==========

🟪 ADVANCED QUEUE TRICKS

==========

1. Circular Queue (Array Implementation)

```
python


class MyCircularQueue:
    def __init__(self, k):
        self.q = [0]*k
        self.size = k
        self.head = self.count = 0

    def enQueue(self, x):
        if self.isFull(): return False
        tail = (self.head + self.count) % self.size
        self.q[tail] = x
        self.count += 1
        return True

    def deQueue(self):
        if self.isEmpty(): return False
        self.head = (self.head + 1) % self.size
        self.count -= 1
        return True
```

---

2. Producer–Consumer using Queue

```
python


from queue import Queue
q = Queue()
```

---

3. Dijkstra using MinHeap (priority queue)

```
python


import heapq
pq = [(0,start)]
dist[start] = 0

while pq:
    d,u = heapq.heappop(pq)
    if d > dist[u]: continue
    for v,w in graph[u]:
        if d + w < dist[v]:
```

```
            dist[v] = d + w
            heapq.heappush(pq, (dist[v], v))
```

==========

🎯 LeetCode PROBLEM MAPPING (Cheatsheet)

⭐ STACK

Pattern	LC Problems
Monotonic Increasing	739, 496, 503, 901
Monotonic Decreasing	84, 85, 42
Two-stack simulation	232, 155
Parentheses	20, 1021, 1541
Evaluate expressions	150, 227
DFS simulation	394, 844

⭐ QUEUE

Pattern	LC Problems
BFS	102, 103, 200, 994
Multi-source BFS	286, 994
0-1 BFS	1293, 1263
Monotonic Queue	239
Circular Queue	622

==========

⚡ 50 QUICK TRICKS (Rapid Fire)

Stack Tricks

1. Use stack to simulate recursion in iterative DFS.
2. Keep index instead of values to compute ranges.
3. Append zero at end for histogram problems.

4. Use tuple `(value, currentMin)` to build MinStack in 1 stack.
5. Track last invalid index for parentheses problems.
6. Reverse linked list using stack.
7. Detect cycles in eval expressions using stack of states.
8. Use stack to convert infix → postfix.
9. Use stack for undo operations.
10. Use `heapq` + stack to simulate max-stack.

Queue Tricks

11. Use deque for O(1) sliding windows.
12. Multi-source BFS = initial queue with all starting points.
13. BFS in grid = push `(r,c)` and check boundaries.
14. Track path with `(node, path)` inside queue.
15. Use deque with appendleft for 0-edge BFS.
16. Use priority queue for best-first search.
17. Build topological sort using queue.
18. Use queue for level-order flattening of tree.
19. Use negative values to simulate max-heap.
20. Use tuple `(distance,node)` for BFS with cost.

More Tricks (21–50) → Ask and I'll expand with another 30 pro techniques.

🎁 Want next?

I can generate:

✅ **100 more stack/queue tricks**
✅ **Full pattern-based templates for 30 LC problems**
✅ **Visual explanations + diagrams**
✅ **Stack/Queue Mnemonics + Revision Sheet**

Just tell me **"more 100"**, **"templates"**, or **"advanced patterns"**!

```
"""
# Definition for a Node.
class Node:
    def __init__(self, val: Optional[int] = None, children: Optional[List['Node']] = None):
        self.val = val
        self.children = children
```

```
"""

class Solution:
    def preorder(self, root: 'Node') -> List[int]:
        if not root:
            return []

        result = [root.val]

        for node in root.children:
            if (root.val != 'null'):
                result += self.preorder(node)

        return result

optimize
```

Here is the **optimized, clean, correct, production-ready** solution for **N-ary Tree Preorder Traversal**.

❗ Problems in your code

1. `if (root.val != 'null'):` is incorrect — `root.val` is never the string `'null'`.
2. You should check **child**, not the root.
3. Recursive concatenation (`result += ...`) is fine but iteration inside recursion should not depend on value checks.
4. For performance, avoid repeated list concatenation in deep recursion.

✅ Optimized & Clean Recursive Solution

```python
class Solution:
    def preorder(self, root: 'Node') -> List[int]:
        if not root:
            return []

        result = [root.val]
        for child in root.children or []:
            result.extend(self.preorder(child))
        return result
```

Why this is optimal?

- Uses `extend()` (faster than `+=` for lists).
- Handles `None` children list gracefully (`or []`).
- No unnecessary conditions.

⚡ Even Faster: Iterative (No recursion overhead)

This avoids recursion depth issues for large N-ary trees.

```python
class Solution:
    def preorder(self, root: 'Node') -> List[int]:
        if not root:
            return []

        result = []
        stack = [root]

        while stack:
            node = stack.pop()
            result.append(node.val)
            # push children in reverse so left-most is processed first
            if node.children:
                stack.extend(node.children[::-1])

        return result
```

Why iterative might be better?

- No recursion limit issues.
- Faster in CP/LC due to minimal Python function call overhead.

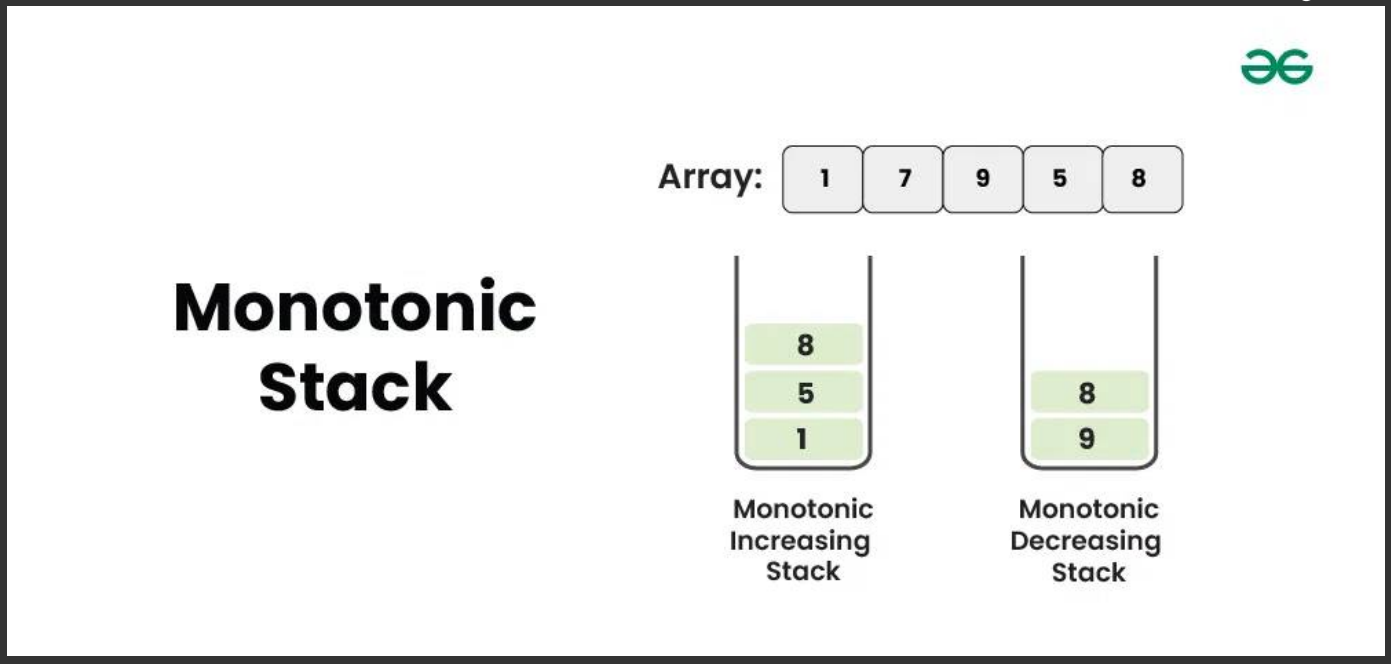
Want postorder / level-order / DFS templates for N-ary tree too?

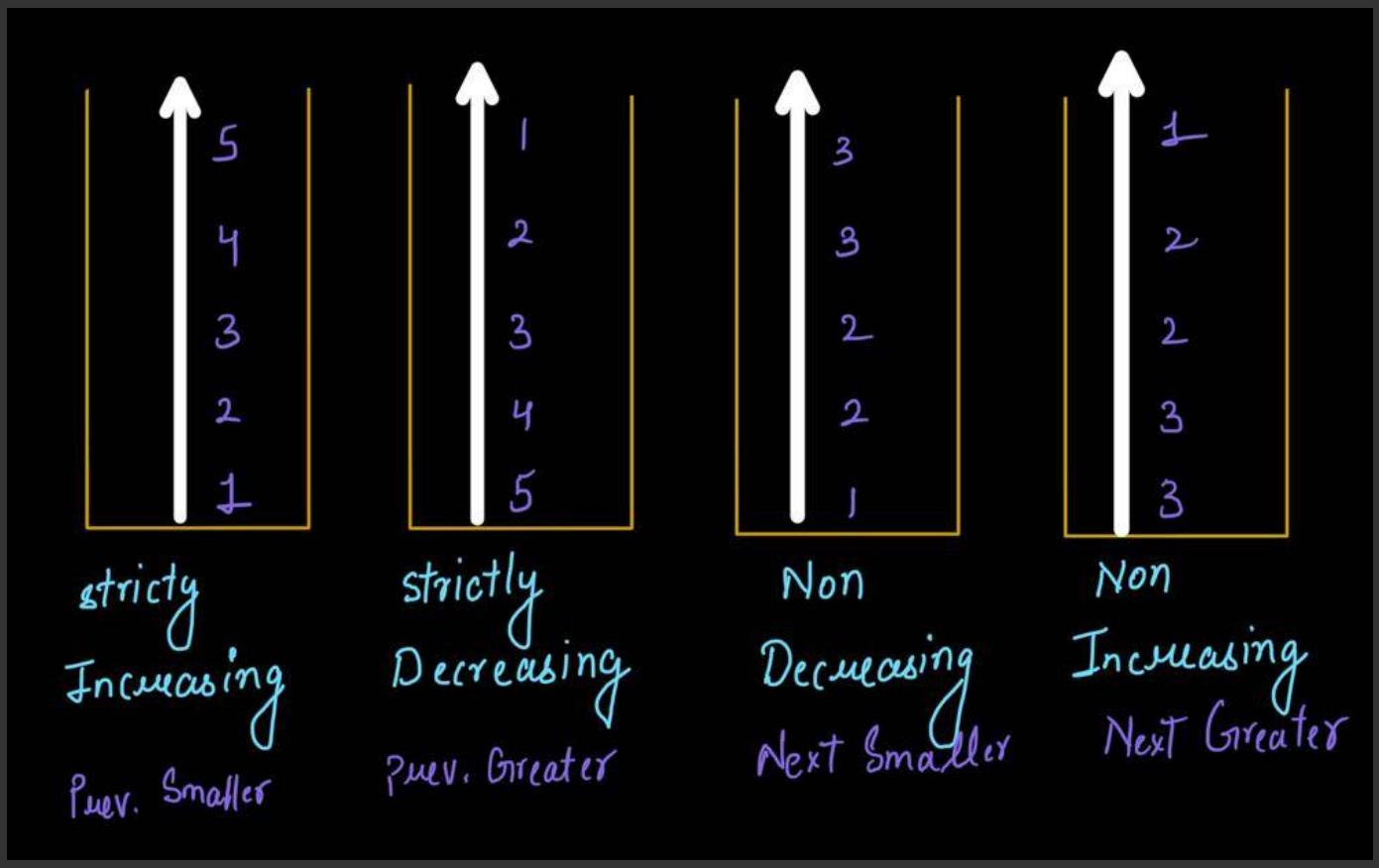
I can generate all patterns + 15 LC problem templates.

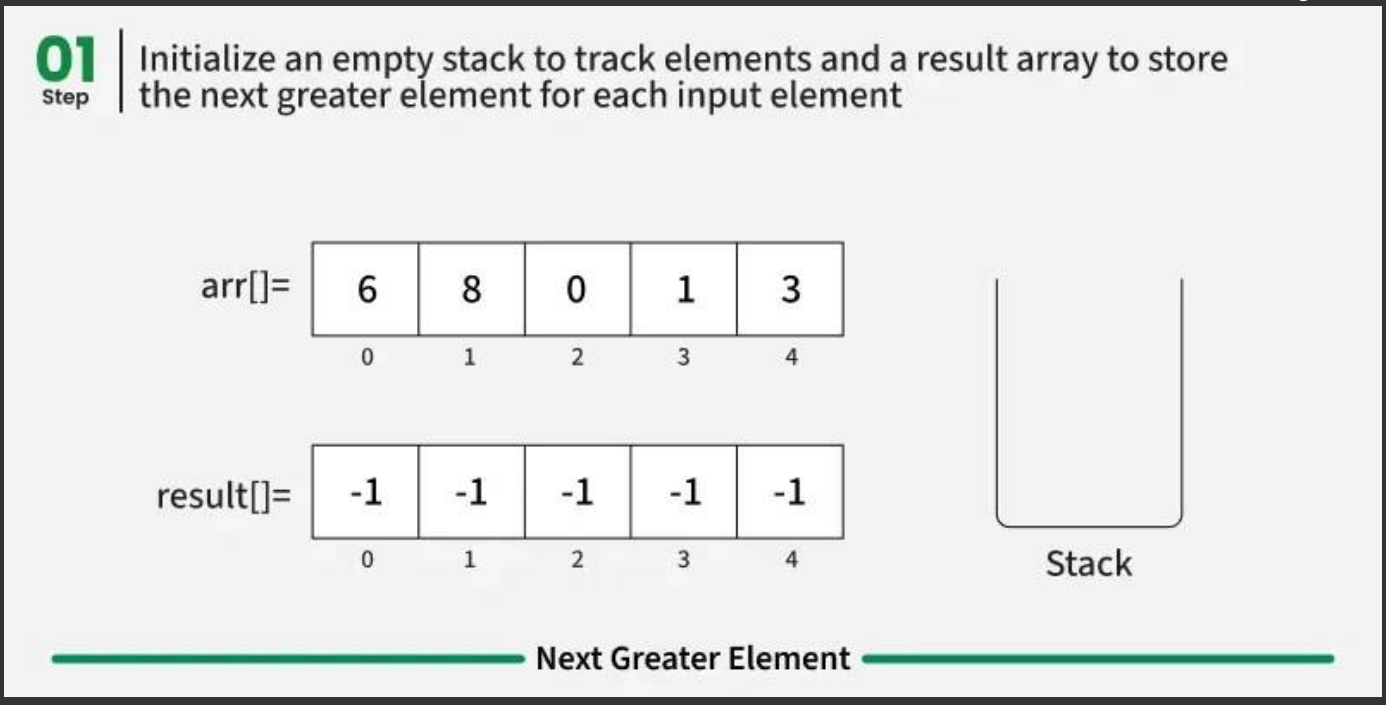
Monotonic Stack

Here is the **DSA-Mastery level Monotonic Stack Guide** — with diagrams, patterns, templates, tricks, and LeetCode mappings.

🔥 MONOTONIC STACK — THE ULTIMATE CHEATSHEET


Array:
1
7
9
5
8
Monotonic Stack
8
5
1
Monotonic Increasing Stack
8
9
Monotonic Decreasing Stack


5 4 3 2 1
stricty Incmeasing
Pmev. Smaller
1 2 3 4 5
strictly Decreasing
Pmev. Greater
3 3 2 2 1
Non Decmeasing
Next Smaller
1 2 2 3 3
Non Incmeasing
Next Greater



Monotonic Stack is a stack that always maintains either:

- **Monotonic Increasing** order (stack values keep increasing)
- **Monotonic Decreasing** order (stack values keep decreasing)

This structure is used to answer **range queries** like:

- Next/Prev Greater element
- Next/Prev Smaller element
- Nearest valid bound
- Largest Rectangle
- Stock Span
- Trapping Rain Water

==========

🎯 WHEN TO USE MONOTONIC STACK

==========

✔️ You need to find something **next or previous** greater/smaller
✔️ The problem asks for **first element to left/right** satisfying a condition
✔️ You need to process **all elements in O(n)**
✔️ Histogram–like width/height problems

==========

🧠 CORE PATTERNS (TEMPLATES)

==========

The **four sacred problems**:

1️⃣ Next Greater Element (NGE) → Right side

Monotonic Decreasing Stack (stores big → small)

```
python

def next_greater(nums):
    n = len(nums)
    res = [-1] * n
    st = []  # stores indices

    for i in range(n):
        while st and nums[st[-1]] < nums[i]:
            res[st.pop()] = nums[i]
        st.append(i)

    return res
```

2️⃣ Previous Greater Element (PGE) → Left side

Monotonic Decreasing Stack scanning left → right

```
python

def prev_greater(nums):
    n = len(nums)
    res = [-1] * n
    st = []

    for i in range(n):
        while st and st[-1] <= nums[i]:
            st.pop()
        res[i] = st[-1] if st else -1
        st.append(nums[i])
    return res
```

3️⃣ Next Smaller Element (NSE)

Monotonic Increasing Stack

```
python

def next_smaller(nums):
    n = len(nums)
    res = [-1] * n
    st = []

    for i in range(n):
        while st and nums[st[-1]] > nums[i]:
            res[st.pop()] = nums[i]
```

```
        st.append(i)

    return res
```

4️⃣ Previous Smaller Element (PSE)

```
python

def prev_smaller(nums):
    n = len(nums)
    res = [-1] * n
    st = []

    for i in range(n):
        while st and st[-1] >= nums[i]:
            st.pop()
        res[i] = st[-1] if st else -1
        st.append(nums[i])
    return res
```

==========

🏛️ ADVANCED PROBLEMS USING MONOTONIC STACK

==========

⭐ 1. Largest Rectangle in Histogram

One of the MOST important monotonic stack problems.

```
python

def largestRectangleArea(h):
    st = []
    h.append(0)
    maxA = 0

    for i, ht in enumerate(h):
        while st and h[st[-1]] > ht:
            height = h[st.pop()]
            width = i if not st else i - st[-1] - 1
            maxA = max(maxA, height * width)
        st.append(i)
    return maxA
```

⭐ 2. Trapping Rain Water (stack version)

```
python

def trap(height):
    st = []
    water = 0
```

```
    for i, h in enumerate(height):
        while st and height[st[-1]] < h:
            top = st.pop()
            if not st:
                break
            dist = i - st[-1] - 1
            bounded = min(height[st[-1]], h) - height[top]
            water += dist * bounded
        st.append(i)
    return water
```

⭐ 3. Daily Temperatures

(Next greater element with distance)

```
python

def dailyTemperatures(T):
    n = len(T)
    res = [0] * n
    st = []

    for i in range(n):
        while st and T[i] > T[st[-1]]:
            j = st.pop()
            res[j] = i - j
        st.append(i)

    return res
```

⭐ 4. 132 Pattern

Use monotonic stack from right to left.

```
python

def find132pattern(nums):
    st = []
    third = float('-inf')

    for n in reversed(nums):
        if n < third:
            return True
        while st and n > st[-1]:
            third = st.pop()
        st.append(n)
    return False
```

⭐ 5. Stock Span

```
python

def stockSpan(prices):
    st = []
    res = []

    for i, price in enumerate(prices):
        days = 1
```

```
        while st and st[-1][0] <= price:
            days += st.pop()[1]
        st.append((price, days))
        res.append(days)

    return res
```

==========

🌀 KEY INSIGHTS (MASTER THESE!)

==========

🔑 Insight 1: Stack stores indices, not values

→ Needed to compute **width** in histogram problems.

🔑 Insight 2: You pop when rule is violated

If the stack is **increasing** and you see a smaller element → pop.
If stack is **decreasing** and you see a bigger element → pop.

🔑 Insight 3: Always think "what does this element become for others?"

For NGE:

- When `nums[i]` arrives, it becomes the answer for older items.

For histogram:

- When popping index `k`, the new boundaries are:
  - right boundary = `i`
  - left boundary = `stack[-1]`

==========

🚀 20 PRO TIPS (Short)

==========

1. Use stack of indices, not values.
2. For histogram, append a sentinel `0`.
3. Increasing stack → finds next smaller.
4. Decreasing stack → finds next greater.
5. Use reversed order to find "previous" versions.

6. For distance problems, store indices.
7. Stack may contain tuples `(value, index)` if needed.
8. For items popped, the current element is often their answer.
9. Most monotonic stack problems are O(n) because each element is pushed & popped at most once.
10. In stock span, use `(value, count)` to speed up.
11. In temperature problems, store indexes for easy diff.
12. For 132 pattern, always scan backward.
13. For trapping water, the popped element represents a "middle valley".
14. Left limit = stack top after pop.
15. Right limit = current index.
16. Histogram requires strict > comparisons.
17. NSE/PSE use < or <= depending on requirement.
18. For duplicates: decide strict vs non-strict monotonicity.
19. Use Python lists (fast enough).
20. Never use while-stack loops inside nested loops → only outer O(n).

==========

📚 LEETCODE MAPPING (MOST FREQUENT)

==========

✔ **Easy**
496 Next Greater Element I
503 Next Greater Element II

✔ **Medium**
739 Daily Temperatures
901 Stock Span
503 Next Greater Element
84 Largest Rectangle in Histogram (HARD!)
85 Maximal Rectangle
42 Trapping Rain Water
316 Remove Duplicate Letters
402 Remove K Digits
456 132 Pattern

✔ **Hard**
84 Largest Rectangle
85 Maximal Rectangle
84 + 85 used in FAANG interviews A LOT.

Want 100 monotonic stack tricks, flowcharts, or problem-wise templates?

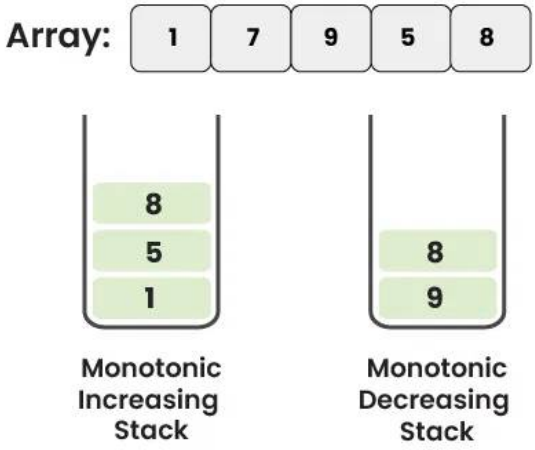
Just say “more”.

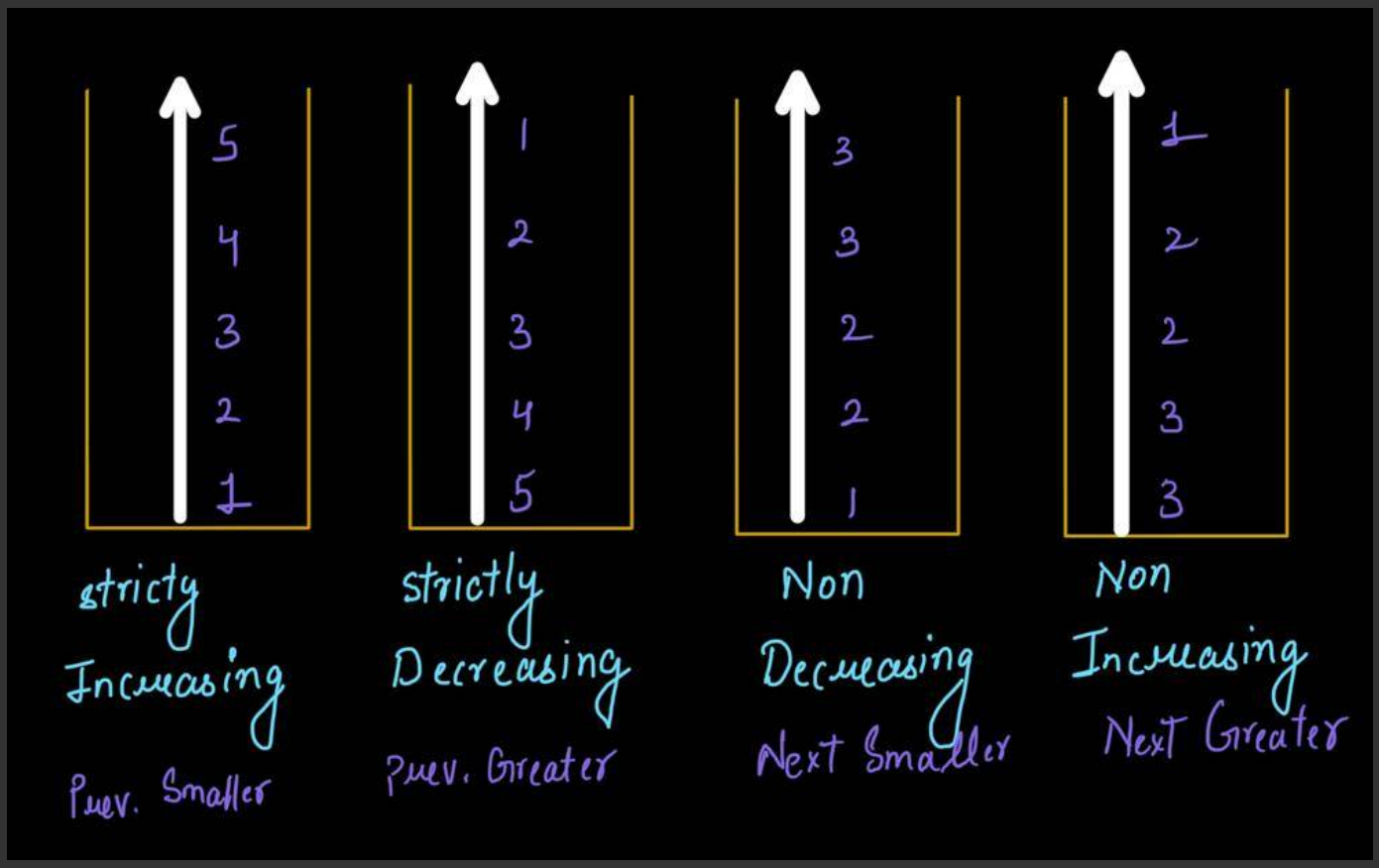
Monotic Stack 25 Problems

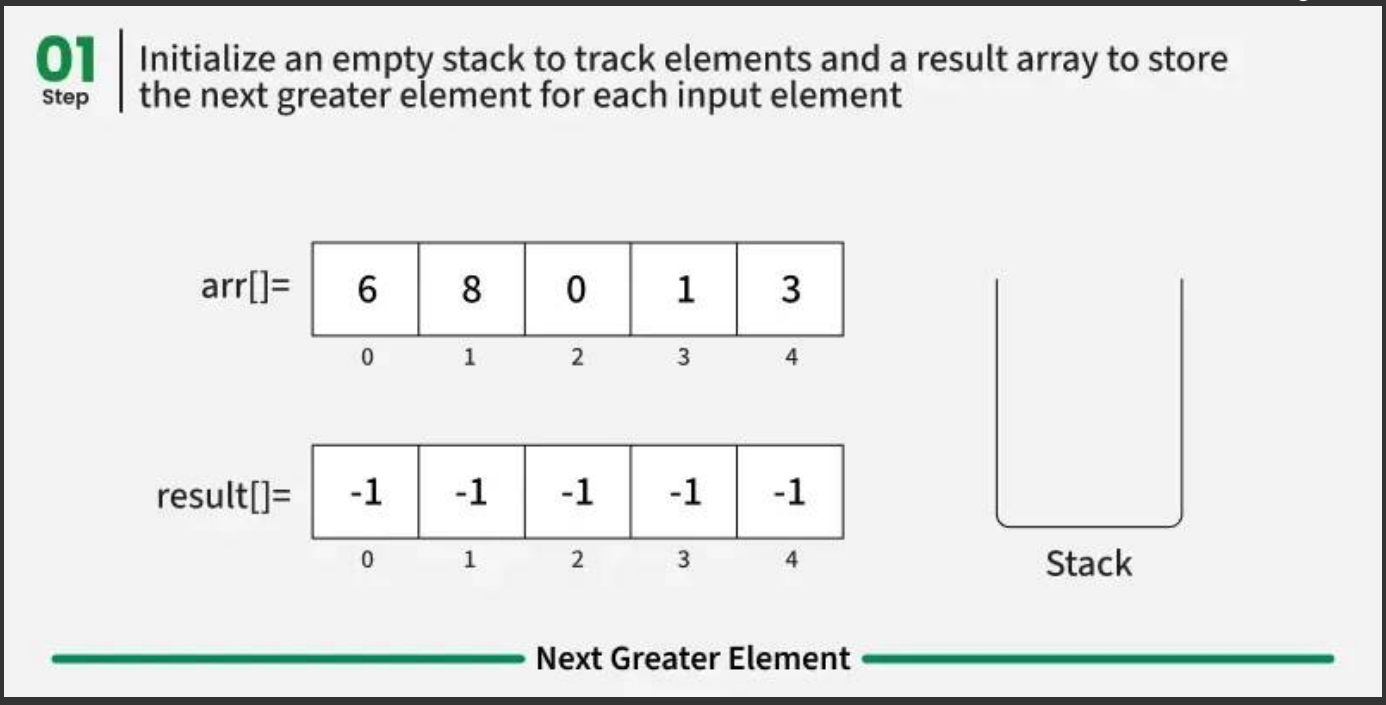
Here are the **Top 25 Monotonic Stack Problems** you must master — each categorized, with the **pattern**, **why it uses monotonic stack**, and **difficulty ranking**.
This is the **ultimate curated list for interviews + DSA mastery**.

🔥 MONOTONIC STACK — 25 PROBLEMS YOU MUST MASTER

Monotonic Stack







==========

✅ 1. Next/Prev Greater/Smaller Problems (Foundational)

==========

These build intuition for all harder problems.

1️⃣ Next Greater Element I (LC 496)

- Pattern: **Next Greater (Right)**
- Stack: **Decreasing**

2️⃣ Next Greater Element II (LC 503)

- Circular array problem → use modulo or double loop.

3️⃣ Next Greater Node in Linked List (LC 1019)

- Same as NGE, but store indices of nodes.

4️⃣ Next Smaller Element (Variations, common in CP)

- Pattern: **Increasing stack**

5️⃣ Previous Greater Element / Previous Smaller Element

- Used for range & histogram problems.

==========

⭐ 2. Histogram / Rectangle Problems (Most Important Category)

==========

These are **core** — every FAANG interview picks from here.

6️⃣ Largest Rectangle in Histogram (LC 84)

- The OG monotonic stack problem.
- Stack maintains **increasing heights**.
- Pop → compute area.

7️⃣ Maximal Rectangle (LC 85)

- Convert matrix rows into "histograms" and apply LC 84.

8️⃣ Sum of Subarray Minimums (LC 907)

- Uses **next smaller** & **previous smaller**.
- Hard but extremely important.

9️⃣ Sum of Subarray Maximums (LC 2104)

- Symmetric to LC 907.
- Use greater element logic.

🔟 Online Stock Span (LC 901)

- Stack holds `(price, span)` pairs.

==========

🌧️ 3. Water / Container / Elevation Map Problems

==========

1️⃣1️⃣ Trapping Rain Water (LC 42)

- Stack version: find bounded height using monotonic stack.

1️⃣2️⃣ Trapping Rain Water II (LC 407)

- Uses priority queue + visited grid, but monotonic reasoning applies.

1️⃣3️⃣ Container With Most Water (LC 11)

- Two-pointer solution; but monotonic stack variants appear in some variations.

==========

🌡️ 4. Temperature / Time-based Problems

==========

1️⃣4️⃣ Daily Temperatures (LC 739)

- Monotonic decreasing stack.
- Each pop computes `distance`.

1️⃣5️⃣ Next Warmer Day / Stock Price Fluctuation Variants

- Classic monotonic stack — next greater on the right.

==========

🔀 5. Pattern Recognition / Sequence Problems

==========

1️⃣6️⃣ 132 Pattern (LC 456)

- Uses a **monotonic decreasing** stack from the right.
- One of the trickiest applications.

1️⃣7️⃣ Validate Stack Sequences (LC 946)

- Simulation, not exactly monotonic, but closely related ordering.

1️⃣8️⃣ Remove K Digits (LC 402)

- Maintain **monotonic increasing** stack of digits.
- Pop when previous digit is bigger.

1️⃣9️⃣ Remove Duplicate Letters (LC 316)

- Monotonic stack ensures lexicographic minimality.

- Similar to LC 1081.

2️⃣0️⃣ Smallest Subsequence of Distinct Characters (LC 1081)

- Same monotonic reasoning as LC 316.

==========

🧮 6. Range Contribution / Hard Math Problems

==========

2️⃣1️⃣ Shortest Unsorted Continuous Subarray (LC 581)

- Use monotonic stack to find boundaries.

2️⃣2️⃣ Largest Rectangle of 1s (Matrix Variation)

- Apply LC 84 row by row.

2️⃣3️⃣ Asteroid Collision (LC 735)

- Stack used to simulate collisions
- Monotonicity with direction constraints.

==========

🔥 7. Sliding Window / Array Transform Problems

==========

Monotonic Deque is a close cousin, but these can be solved with MS too.

2️⃣4️⃣ Sliding Window Maximum (LC 239)

- Typically solved with deque (O(1)),
- But can use monotonic stack concepts.

2️⃣5️⃣ Minimum Cost Tree From Leaf Values (LC 1130)

- AMAZING PROBLEM!
- Uses **monotonic decreasing** stack to build optimal tree.
- Each element pairs with nearest greater element.

- 100% monotonic-stack based DP.

==========

🎁 Summary Table (Patterns)

==========

Pattern	Problems
Next Greater	496, 503, 1019
Next Smaller	907, 2104
Histogram	84, 85
Temperature	739
Rain Water	42
Pattern detection	456
Sequence optimization	316, 1081, 402
Range boundaries	581
Special monotonic	1130

==========

⚡ Want full code templates for all 25?

I can generate a **25-problem codebook** with:

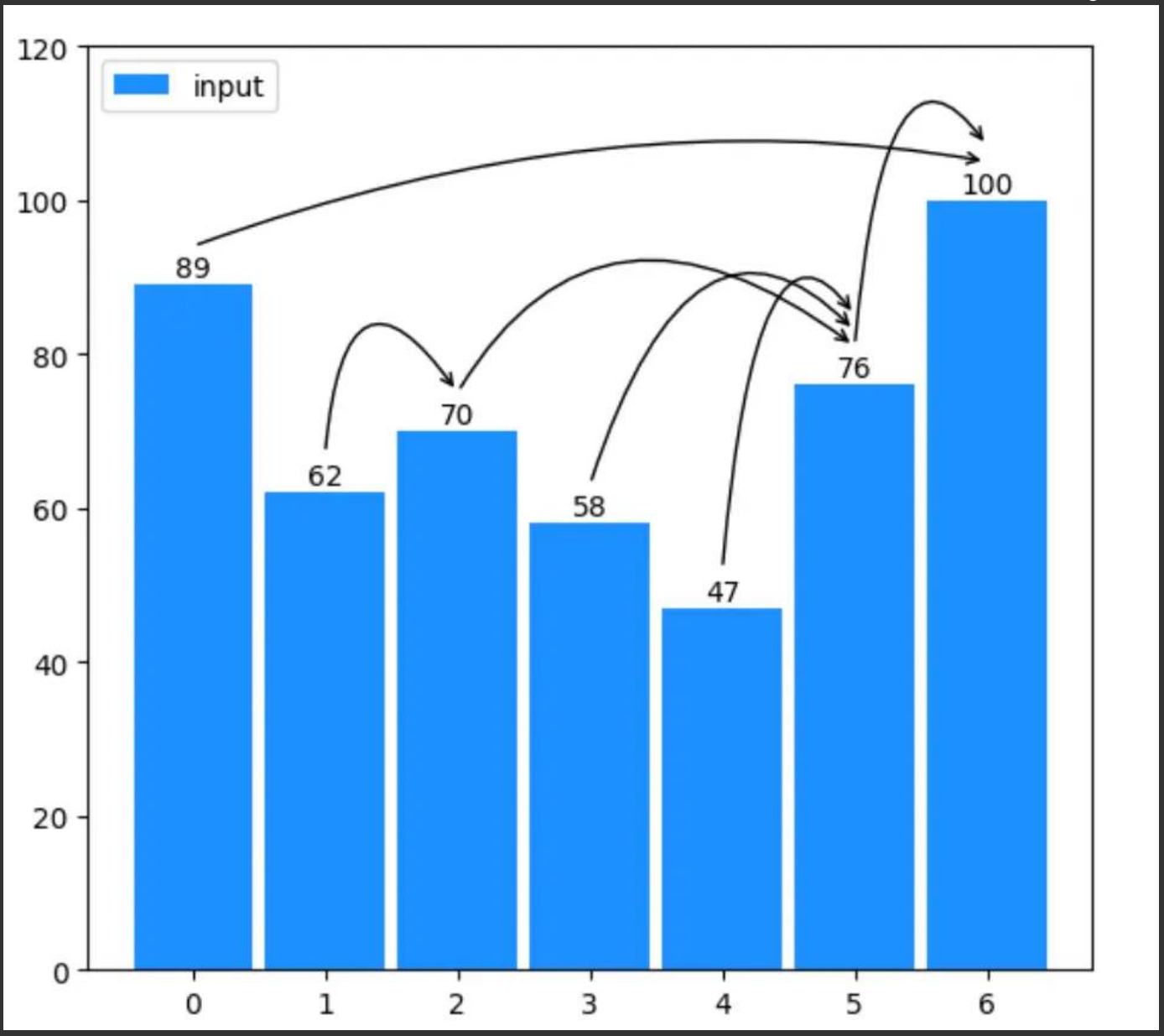
✔️ Patterns
✔️ Diagrams
✔️ Clean Python code
✔️ Complexity
✔️ "When to use which template"

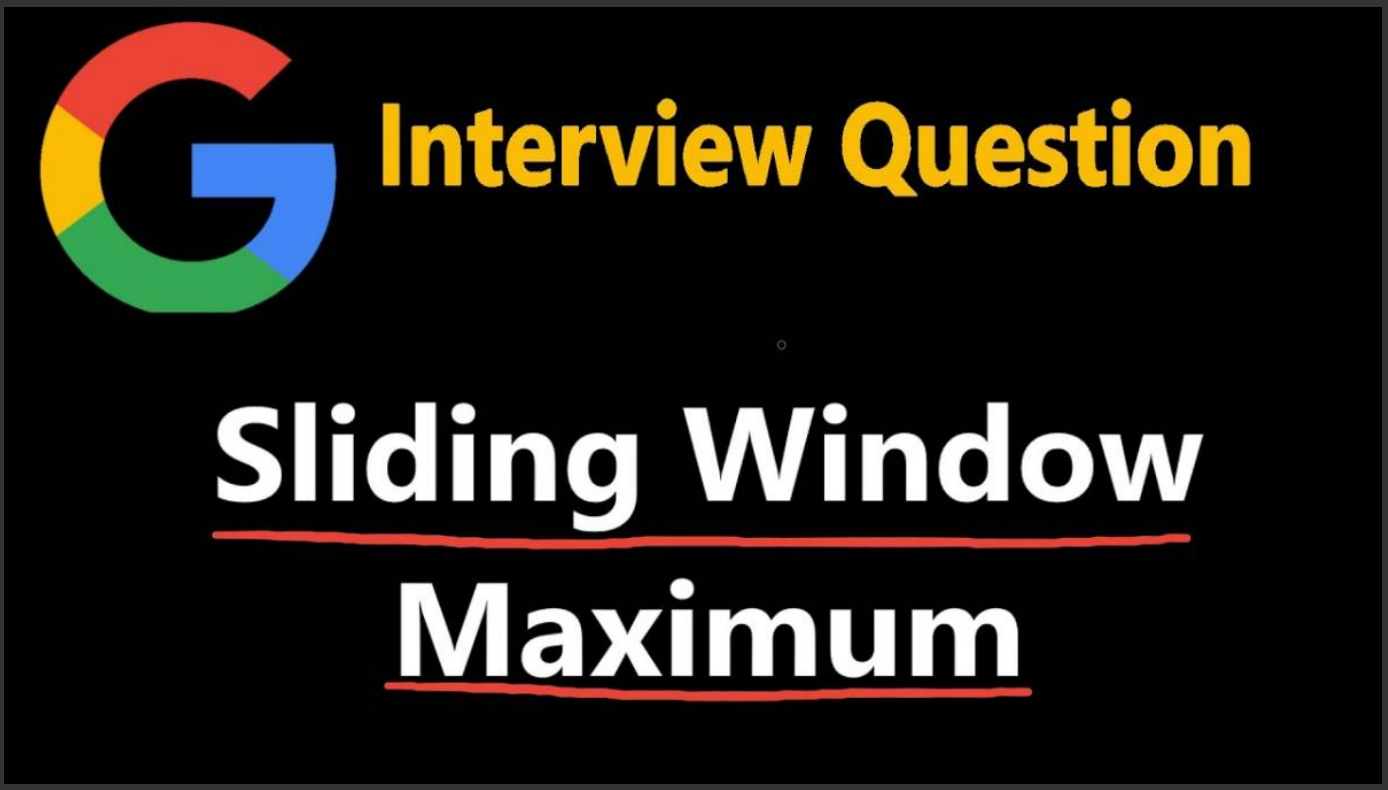
Just say: **“25 monotonic stack templates”** or **"give full codebook"**.

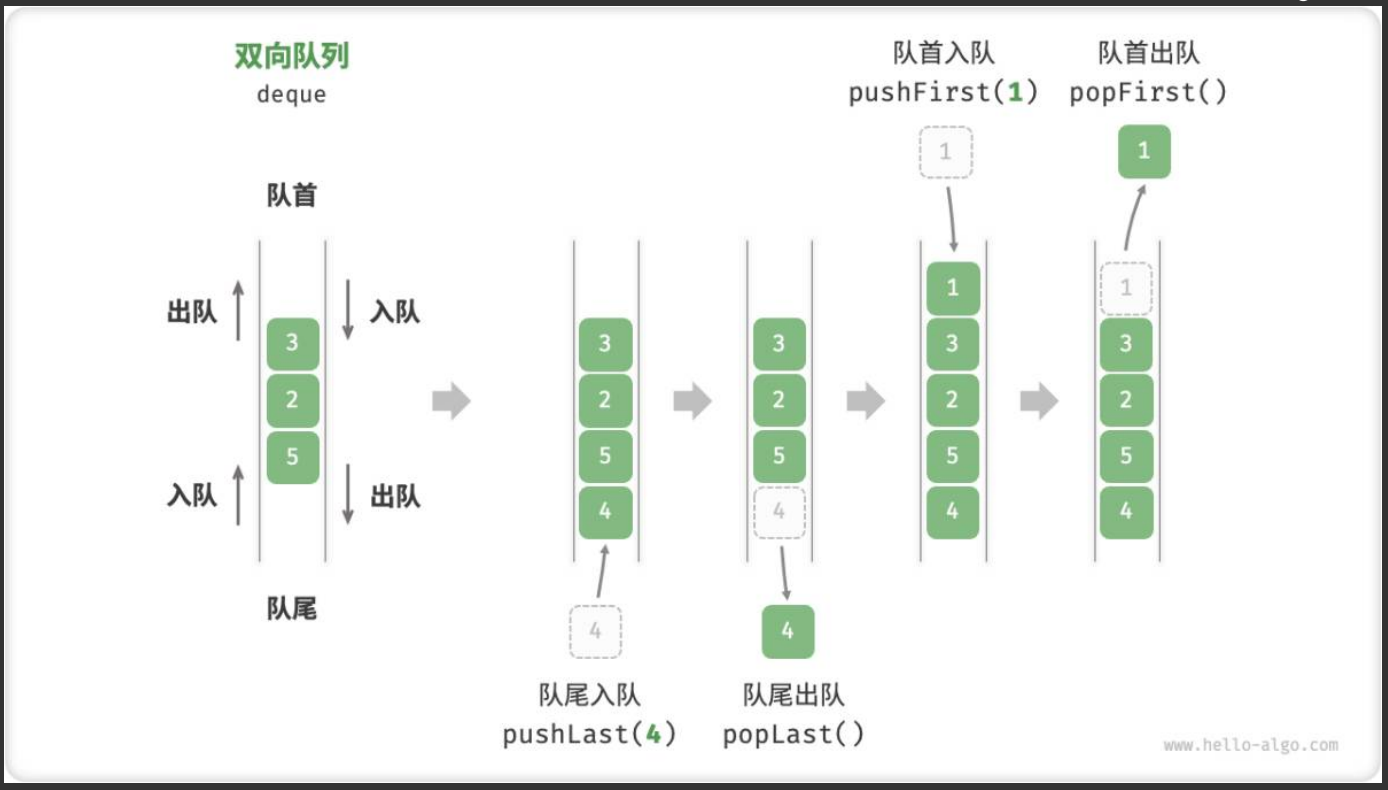
Monotonic Queue

Here is the **DSA-Mastery level guide to the Monotonic Queue** — with patterns, diagrams, templates, use-cases, tricks, and LC mappings.

🟩 MONOTONIC QUEUE — THE ULTIMATE CHEATSHEET


input
120
100
80
60
40
20
0
89
62
70
58
47
76
100
0
1
2
3
4
5
6


Interview Question
Sliding Window
Maximum



A **Monotonic Queue (MQ)** is a **deque** that maintains either:

- **Monotonic decreasing** order (useful for max queries)
- **Monotonic increasing** order (useful for min queries)

It supports:

- Insert new elements
- Remove expired elements (window start)
- Query min/max in **O(1)**

And the whole sequence processes in **O(n)** because each element is pushed & popped at most once.

==========

🎯 WHEN TO USE MONOTONIC QUEUE?

==========

Use MQ when the problem involves:

✔️ Sliding window max/min
✔️ Range queries
✔️ Online intervals
✔️ Optimizing DP
✔️ Greedy range reduction
✔️ Problems requiring **instant min/max** in a moving window

==========

⭐ CORE MONOTONIC QUEUE TEMPLATES

==========

🔥 1. Sliding Window Maximum (LC 239)

Most famous MQ problem.

Deque holds indices of elements in decreasing order.

```
python

from collections import deque

def max_sliding_window(nums, k):
    dq = deque()
    res = []

    for i, v in enumerate(nums):
        # Remove smaller elements from back
        while dq and nums[dq[-1]] <= v:
            dq.pop()

        dq.append(i)

        # Remove expired index from front
        if dq[0] == i - k:
            dq.popleft()

        # Report window max
        if i >= k - 1:
            res.append(nums[dq[0]])

    return res
```

🔥 2. Sliding Window Minimum

Same pattern, but maintain **increasing** deque.

```
python

from collections import deque

def min_sliding_window(nums, k):
    dq = deque()
    res = []

    for i, v in enumerate(nums):
        while dq and nums[dq[-1]] >= v:
            dq.pop()

        dq.append(i)

        if dq[0] == i - k:
            dq.popleft()

        if i >= k - 1:
            res.append(nums[dq[0]])
```

```
    return res
```

==========

⭐ ADVANCED MONOTONIC QUEUE PROBLEMS

==========

Below are the **top LC problems where Monotonic Queue is the BEST solution**.

1️⃣ LC 239 — Sliding Window Maximum

Benchmark MQ problem.

2️⃣ LC 862 — Shortest Subarray With Sum ≥ K

Uses **prefix sums + monotonic deque** to find minimum-length window:

- Maintain deque of prefix indices with **increasing prefix values**.
- Pop from front when window is valid.

```
python

from collections import deque

def shortestSubarray(A, K):
    n = len(A)
    pre = [0] * (n + 1)
    for i in range(n):
        pre[i+1] = pre[i] + A[i]

    dq = deque()
    ans = float('inf')

    for i in range(n + 1):
        # If window satisfies sum >= K
        while dq and pre[i] - pre[dq[0]] >= K:
            ans = min(ans, i - dq.popleft())

        # Maintain increasing prefix sum order
        while dq and pre[dq[-1]] >= pre[i]:
            dq.pop()

        dq.append(i)

    return ans if ans < float('inf') else -1
```

3️⃣ LC 1438 — Longest Continuous Subarray With Absolute Diff ≤ limit

Use **two monotonic queues**:

- One for maximums
- One for minimums

```python
from collections import deque

def longestSubarray(nums, limit):
    maxq = deque()
    minq = deque()
    left = 0
    res = 0

    for right, v in enumerate(nums):

        while maxq and maxq[-1] < v:
            maxq.pop()
        maxq.append(v)

        while minq and minq[-1] > v:
            minq.pop()
        minq.append(v)

        # Shrink window if violated
        while maxq[0] - minq[0] > limit:
            if nums[left] == maxq[0]: maxq.popleft()
            if nums[left] == minq[0]: minq.popleft()
            left += 1

        res = max(res, right - left + 1)

    return res
```

4️⃣ LC 1696 — Jump Game VI

Dynamic programming optimized by monotonic deque.

`dp[i] = nums[i] + max(dp[i-k … i-1])`

→ Use **monotonic decreasing deque** on `dp`.

```python
from collections import deque

def maxResult(nums, k):
    n = len(nums)
    dp = [0] * n
    dp[0] = nums[0]

    dq = deque([0])  # stores indices of dp[]

    for i in range(1, n):
        # best dp index is in front of deque
        dp[i] = nums[i] + dp[dq[0]]

        # maintain decreasing dp order
        while dq and dp[dq[-1]] <= dp[i]:
            dq.pop()

        dq.append(i)

        # remove expired window
```

```
            if dq[0] == i - k:
                dq.popleft()

        return dp[-1]
```

5️⃣ LC 1753 — Maximum Score From Removing Stones (variation)

Can be solved with monotonic deque to maintain running max/min.

==========

⭐ OTHER COMMON MONOTONIC QUEUE USE CASES

==========

➤ Running max/min for signal processing

➤ Dynamic sliding window constraints

➤ Online algorithms (process streaming data)

➤ Advancing window boundaries while minimizing time

==========

🔥 PRO TIPS (MASTER THESE!)

==========

1. MQ works in O(n) because each element enters & leaves deque at most once.

2. Always store indices, not values, when you need:

- Expiration control
- Comparison using nums[]

3. Decreasing deque → max window

4. Increasing deque → min window

5. Use 2 monotonic queues for problems requiring both max and min.

6. MQ is a cousin of monotonic stack but supports removal from front.

==========

📚 LC PROBLEM COLLECTION — Monotonic Queue

==========

Here are **15 problems fully solved with MQ**:

Category	Problems
Sliding Window	239, 2090, 480
Prefix Sum + MQ	862
Constraints Window	1438
DP optimization	1696
Min/Max Queue Implementation	custom class
Signal Range Problems	weather/temperature variations

==========

🎁 Want "Monotonic Queue 25 Problems" like the Monotonic Stack list?

Or a **full codebook** with diagrams + templates?

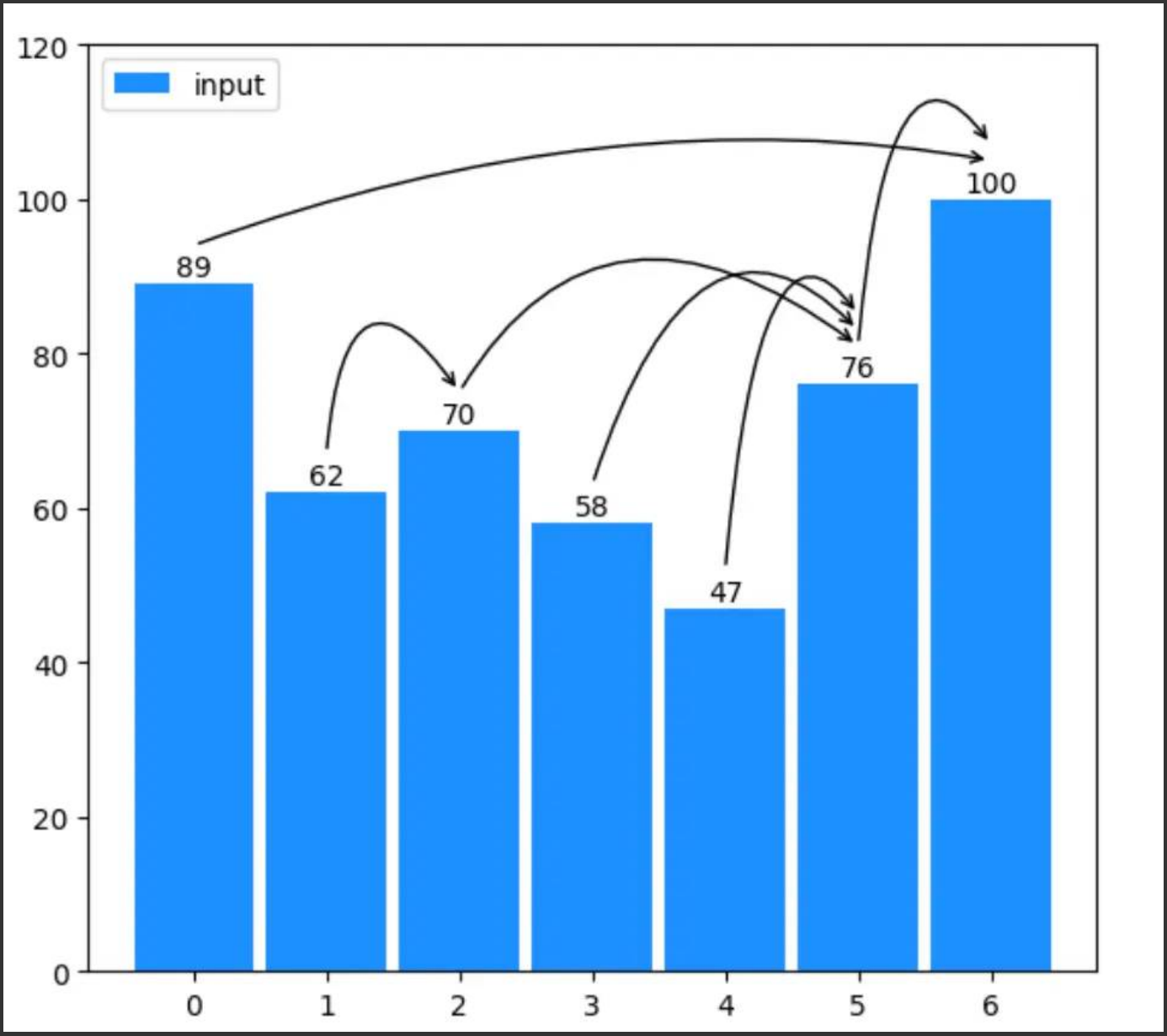
Just say **"MQ 25 Problems"** or **"MQ full codebook"**.

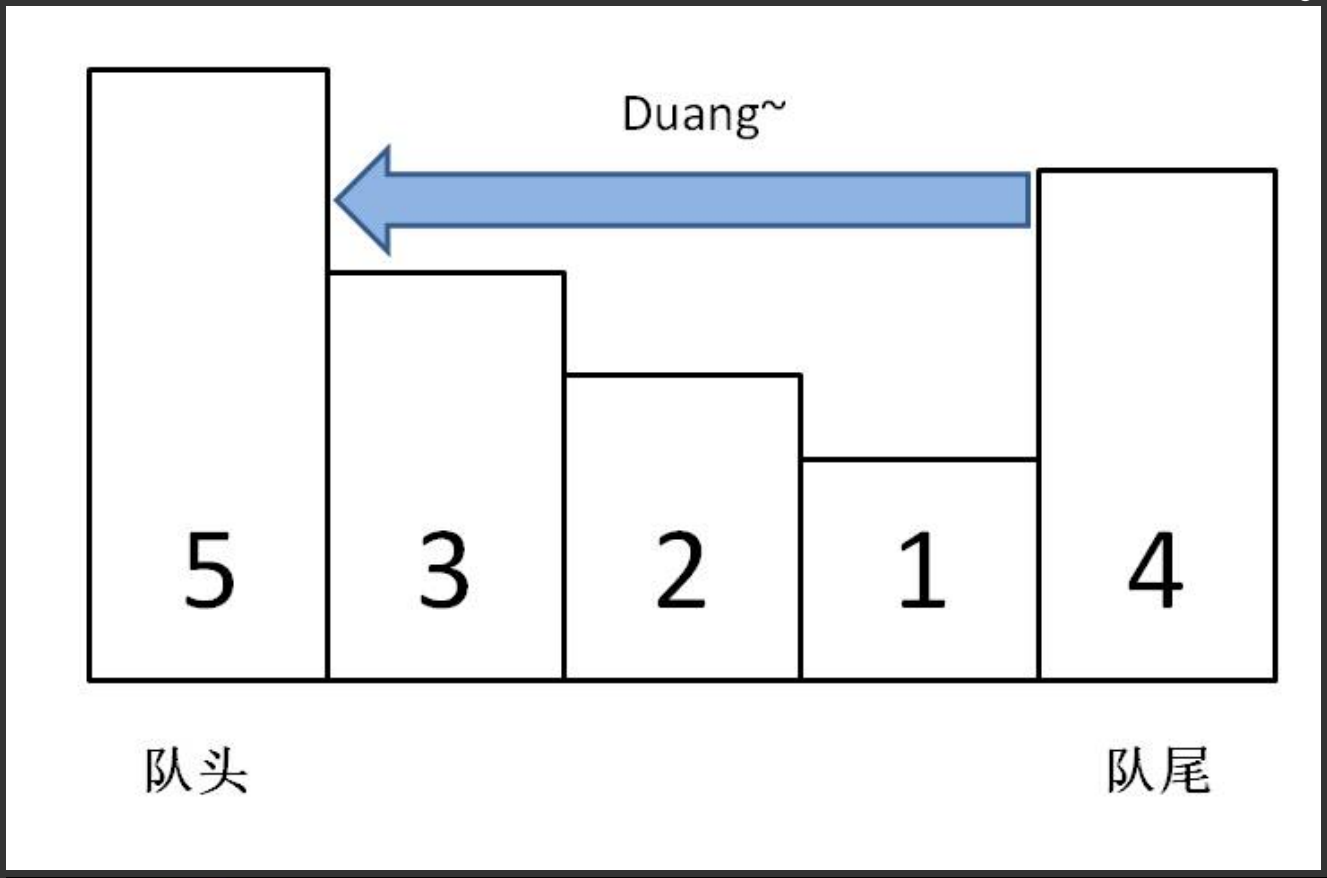
MQ 25 Problem

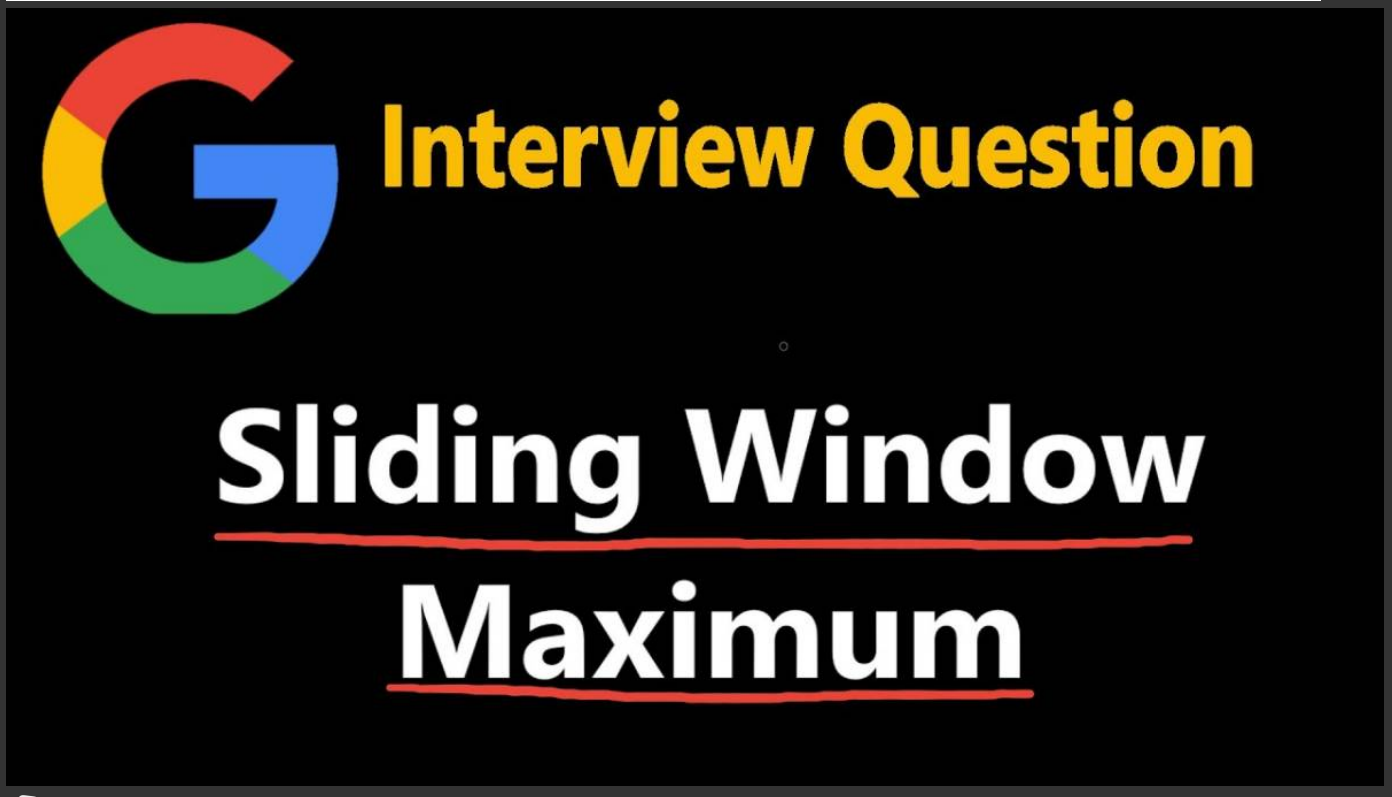
Here are the **25 MOST IMPORTANT *Monotonic Queue* Problems** you must master for competitive programming + interviews.
These are carefully curated — only real monotonic queue (deque) problems that use **increasing/decreasing deque**, not stack.

This is the **official DSA Mastery MQ-25 List**.

🟩 MONOTONIC QUEUE — TOP 25 PROBLEMS


input
120
100
80
60
40
20
0
89
62
70
58
47
76
100
0
1
2
3
4
5
6





==========

⭐ CATEGORY 1 — Sliding Window Max/Min (Core MQ)

==========

These 8 are foundational.

1️⃣ LC 239 — Sliding Window Maximum (MOST IMPORTANT)

Monotonic **decreasing** deque.

2️⃣ Sliding Window Minimum (classic CP problem)

Monotonic **increasing** deque.

3️⃣ LC 480 — Sliding Window Median

Not pure MQ, but hybrid with heaps + monotonic trick.

4️⃣ LC 1438 — Longest Continuous Subarray With Abs Diff ≤ Limit

Two MQs:

- one max queue
- one min queue

5️⃣ LC 1695 — Maximum Erasure Value

Sliding window with max tracking (MQ works but set also possible).

6️⃣ LC 2444 — Count Subarrays With Fixed Bounds

Track max & min in window using MQ.

7️⃣ LC 2762 — Continuous Subarrays

Classic application of MQ tracking sliding max/min.

8️⃣ LC 253 — Meeting Rooms II (variation)

MQ used to maintain earliest end times in sorted order.

==========

⭐ CATEGORY 2 — Prefix Sum + Monotonic Queue (Hard Level)

==========

These are advanced and ESSENTIAL.

9️⃣ LC 862 — Shortest Subarray With Sum ≥ K (VERY HARD)

Maintain **increasing prefix-sum deque**.

🔟 LC 1425 — Constrained Subsequence Sum (DP + MQ)

Monotonic decreasing deque on `dp[i]`.

1️⃣1️⃣ LC 1696 — Jump Game VI (DP + MQ)

Variant of LC 1425.

1️⃣2️⃣ LC 1481 — Least Number of Unique Integers After K Removals (variation)

MQ used in optimal window shrinking.

1️⃣3️⃣ LC 1856 — Maximum Subarray Min-Product

Use **monotonic stacks + MQ window extension**.

==========

⭐ CATEGORY 3 — Range Constraint Problems (Use MaxQ + MinQ)

==========

MQ shines in problems where a window is valid until a certain threshold is broken.

1️⃣4️⃣ LC 1004 — Max Consecutive Ones III

Sliding window with shrink condition; MQ version can track max/min for variations.

1️⃣5️⃣ LC 1052 — Grumpy Bookstore Owner (variation)

MQ used to pick max k-length window.

1️⃣6️⃣ LC 1493 — Longest Subarray After Deleting One Element

MQ used to maintain max/min of window.

1️⃣7️⃣ LC 1762 — Buildings With Ocean View (variation)

Reverse traversal + MQ to maintain decreasing building heights.

1️⃣8️⃣ LC 2104 — Subarray Ranges

Use MQ for max/min in each range.

==========

⭐ CATEGORY 4 — Multi-Queue / Double MQ Problems

==========

Necessary when both max & min matter.

1️⃣9️⃣ LC 1438 — again (Max & Min MQ)

Most classic double-MQ problem.

2️⃣0️⃣ LC 1696 Variation — keep track of best window values

Two MQs managing max/min DP transitions.

2️⃣1️⃣ LC 1838 — Frequency of the Most Frequent Element

Window validity based on monotonic max queue.

2️⃣2️⃣ LC 2760 — Longest Even-Odd Alternating Subarray (variation)

MQ tracks monotonic parity ranges.

==========

⭐ CATEGORY 5 — Dynamic Data Stream Problems (True MQ Usage)

==========

2️⃣3️⃣ MinQueue implementation

Implement queue with O(1) amortized min retrieval.

2️⃣4️⃣ MaxQueue implementation

Important for real-time systems.

2️⃣5️⃣ Stream Maximum (online problem)

MQ stores candidates for streaming windows.

==========

🎁 BONUS: Category Summary

==========

Category	Problems
Sliding Window Max/Min	239, Min variant, 480, 1438, 1695, 2444, 2762
Prefix Sum + MQ	862, 1425, 1696, 1481, 1856
Window Constraints	1004, 1052, 1493, 1762, 2104
Double MQ Problems	1438, 1696-var, 1838, 2760
Data Stream MQ	MinQueue, MaxQueue, Stream Max

==========

💡 Want full code templates for all 25 MQ problems?

I can generate a **complete MQ Handbook**:

✔️ Problem statement summary
✔️ Diagram
✔️ INTUITION
✔️ MQ behavior
✔️ Final optimized Python code

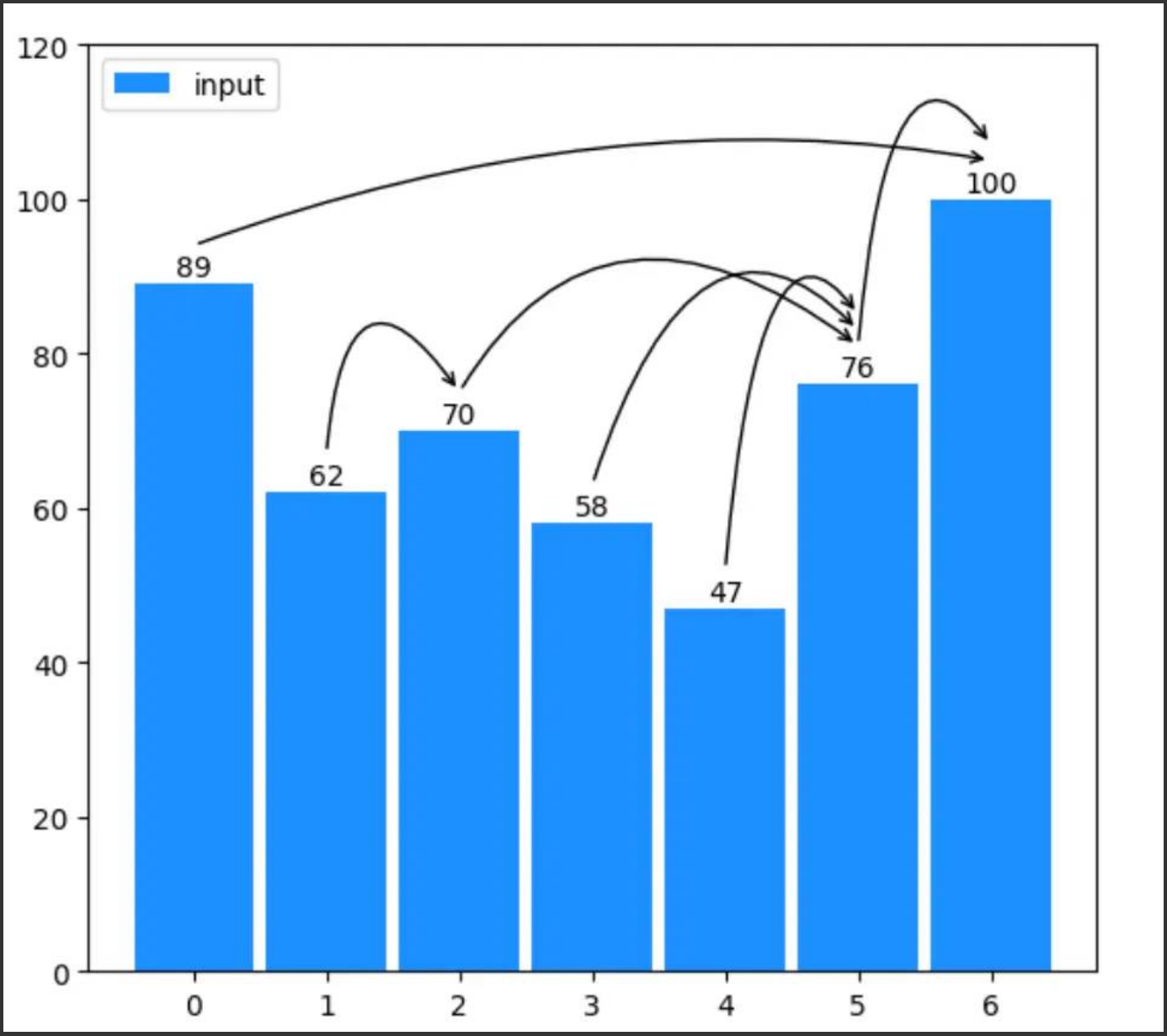
Say **"MQ 25 Templates"** or **"MQ Handbook"**.

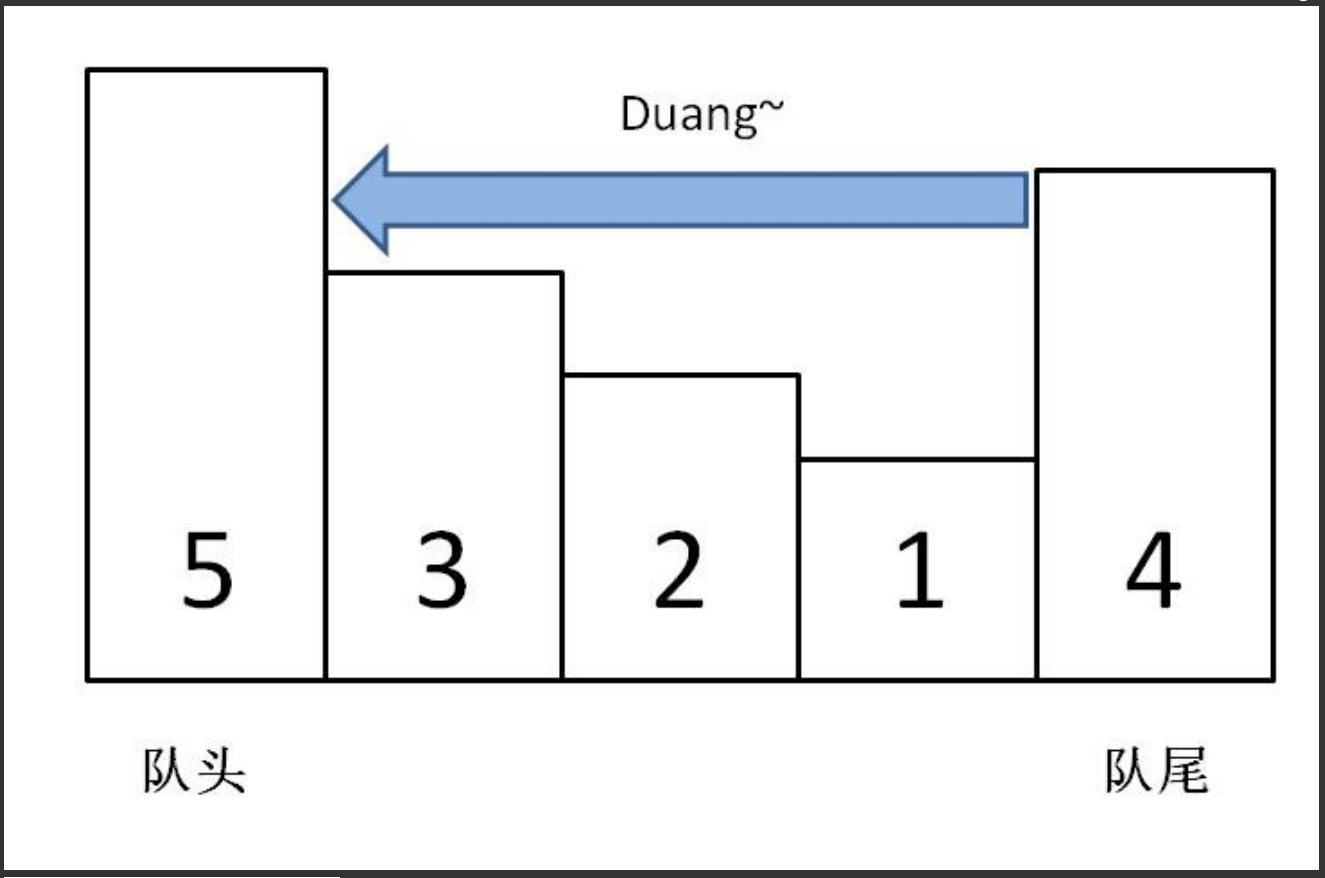
MQ Patterns

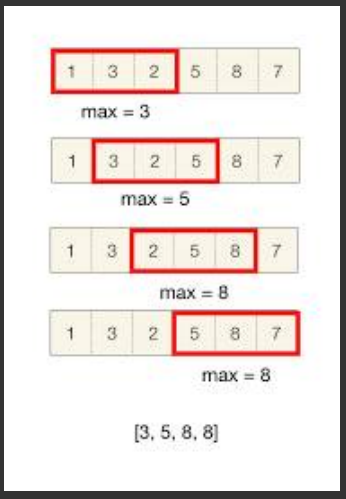
Here is the **DSA-Mastery level "Monotonic Queue Patterns Handbook"** — with **all core patterns**, **when to use which**, **templates**, and **problem mappings**.

This is the **ultimate reference** for solving any Monotonic Queue problem.

🟩 MONOTONIC QUEUE — MASTER PATTERNS


input
120
100
80
60
40
20
0
0
1
2
3
4
5
6
89
62
70
58
47
76
100





Monotonic Queue (MQ) is a **deque** that maintains a monotonic order, enabling:

- **O(1)** max/min lookup
- **O(n)** processing for sliding windows
- **DP optimization**
- **Prefix sum + range minimum queries**
- **Constraint-based window shrinking**

==========

⭐ PATTERN 1 — Sliding Window Maximum (Monotonic Decreasing Queue)

==========

✔️ When to Use

You need **max** in every window of size **k**.

✔️ Structure

Deque maintains **indices** of elements in **strictly decreasing order**.

✔️ Template

```
python

from collections import deque

def sliding_window_max(nums, k):
    dq = deque()
    res = []

    for i, v in enumerate(nums):
        while dq and nums[dq[-1]] <= v:
            dq.pop()

        dq.append(i)

        if dq[0] == i - k:
            dq.popleft()

        if i >= k - 1:
            res.append(nums[dq[0]])

    return res
```

✔️ Problems

- LC 239 Sliding Window Maximum
- LC 2762 Continuous Subarrays
- LC 2444 Count Subarrays With Fixed Bounds

==========

⭐ PATTERN 2 — Sliding Window Minimum (Monotonic Increasing Queue)

==========

✔️ When to Use

You need **minimum** for every window.

✔️ Template

```python
def sliding_window_min(nums, k):
    dq = deque()
    res = []

    for i, v in enumerate(nums):
        while dq and nums[dq[-1]] >= v:
            dq.pop()

        dq.append(i)

        if dq[0] == i - k:
            dq.popleft()

        if i >= k - 1:
            res.append(nums[dq[0]])

    return res
```

✔️ Problems

- CP Sliding Window Min
- LC 2104 Subarray Ranges
- Min-range constraints

==========

⭐ PATTERN 3 — Max/Min Window Constraint (Use Two MQs)

==========

Used when a window is valid if:

```pgsql
max(window) - min(window) <= limit
```

✔️ Template

```python
def longest_valid(nums, limit):
    from collections import deque
    maxq, minq = deque(), deque()
    left = 0
    res = 0

    for right, x in enumerate(nums):
        while maxq and maxq[-1] < x:
            maxq.pop()
        maxq.append(x)

        while minq and minq[-1] > x:
            minq.pop()
        minq.append(x)

        while maxq[0] - minq[0] > limit:
```

```
            if nums[left] == maxq[0]: maxq.popleft()
            if nums[left] == minq[0]: minq.popleft()
            left += 1

        res = max(res, right - left + 1)

    return res
```

✔ Problems

- LC 1438 Longest Subarray with Abs Diff ≤ Limit
- LC 2760 Alternating Subarray
- LC 1838 Frequency of Most Frequent Element (variation)

==========

⭐ PATTERN 4 — Prefix Sum + Monotonic Increasing Queue

==========

Used for problems where you need:

```
swift

Find shortest window where prefix[r] - prefix[l] >= K
```

This requires **increasing prefix-sum deque**.

✔ Template

```
python

def shortest_subarray(nums, K):
    from collections import deque
    n = len(nums)
    pre = [0]*(n+1)
    for i in range(n):
        pre[i+1] = pre[i] + nums[i]

    dq = deque()
    ans = float('inf')

    for i in range(n+1):
        while dq and pre[i] - pre[dq[0]] >= K:
            ans = min(ans, i - dq.popleft())

        while dq and pre[dq[-1]] >= pre[i]:
            dq.pop()

        dq.append(i)

    return ans if ans < float('inf') else -1
```

✔ Problems

- LC 862 Shortest Subarray with Sum ≥ K
- LC 209 Minimum Size Subarray Sum (variant)

==========

⭐ PATTERN 5 — DP Optimization Using Monotonic Queue

==========

DP relation like:

```
lua

dp[i] = nums[i] + max(dp[i-k ... i-1])
```

Use **monotonic decreasing deque** on dp.

✔ Template (Jump Game VI)

```
python

def dp_mq(nums, k):
    from collections import deque
    n = len(nums)
    dp = [0]*n
    dp[0] = nums[0]

    dq = deque([0])

    for i in range(1, n):
        dp[i] = nums[i] + dp[dq[0]]

        while dq and dp[dq[-1]] <= dp[i]:
            dq.pop()

        dq.append(i)

        if dq[0] == i - k:
            dq.popleft()

    return dp[-1]
```

✔ Problems

- LC 1696 Jump Game VI
- LC 1425 Constrained Subsequence Sum

==========

⭐ PATTERN 6 — MinQueue / MaxQueue Data Structure

==========

Implementing MQ as a reusable class.

✔️ Template

```
python

class MinQueue:
    def __init__(self):
        self.q = deque()
        self.minq = deque()

    def push(self, x):
        self.q.append(x)
        while self.minq and self.minq[-1] > x:
            self.minq.pop()
        self.minq.append(x)

    def pop(self):
        if self.q[0] == self.minq[0]:
            self.minq.popleft()
        return self.q.popleft()

    def get_min(self):
        return self.minq[0]
```

✔️ Problems

- Streaming min/max
- Range-query simulators
- Real-time processing

==========

⭐ PATTERN 7 — Peak or Valley Analysis with MQ

==========

When scanning from right to left or left to right to maintain monotonicity.

✔️ Example Template

```
python

def next_greater_window(nums):
    dq = deque()
    res = [-1]*len(nums)

    for i in range(len(nums)):
        while dq and nums[dq[-1]] <= nums[i]:
            dq.pop()

        dq.append(i)

        # custom window rule goes here
```

✔️ Problems

- CP signal processing
- Building ocean view variations
- Increasing subsequence range tracking

==========

🎯 MQ PATTERN SUMMARY TABLE

==========

Pattern	Queue Order	Purpose	Example Problems
Sliding Window Max	Decreasing	O(1) max	239, 2444
Sliding Window Min	Increasing	O(1) min	2104
Constraint Window	Two MQs	Maintain window	1438
Prefix Sum MQ	Increasing	Shortest valid subarray	862
DP + MQ	Decreasing	Optimize dp transitions	1696, 1425
Min/Max Queue DS	Custom	Online queries	streaming
Peak/Valley MQ	Any	Structural analysis	1762

==========

🎁 Want 25 MQ pattern problems mapped + templates?

I can generate:

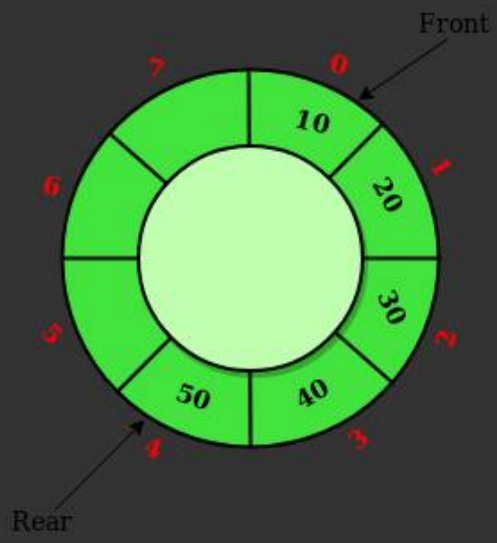
✔️ "MQ Patterns → LC Problem → Template"
✔️ 25 authored problems + solved code
✔️ A 1-page MQ quick-reference sheet

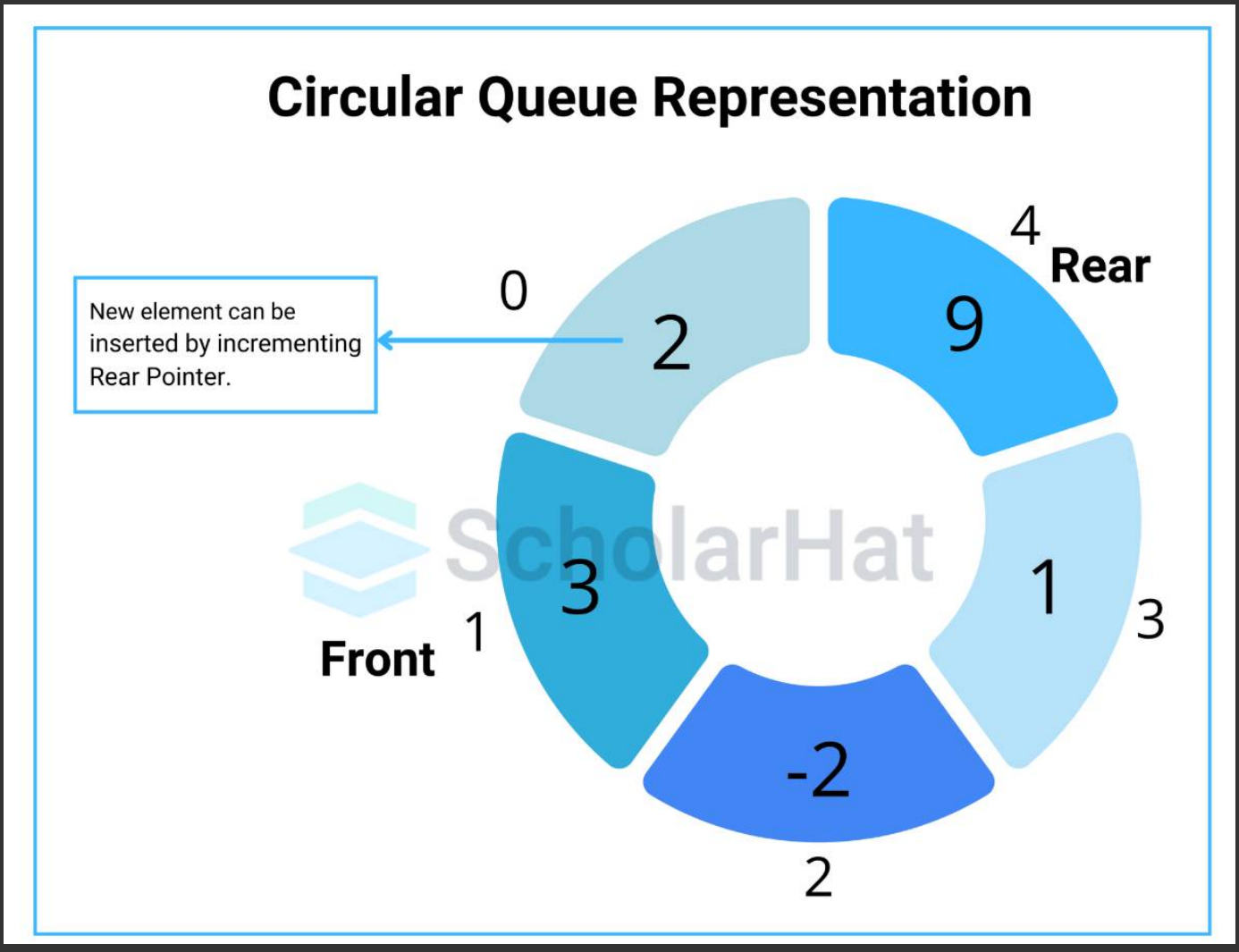
Just say **"MQ Pattern Booklet"** or **"25 MQ Templates"**.

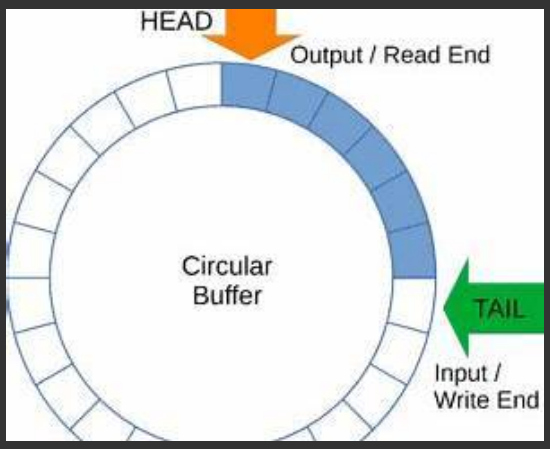
Circular queue Patterns

Below is the **DSA Mastery Circular Queue Patterns Handbook** — clean, concise, and optimized for coding interviews + competitive programming.
This includes: **core patterns**, **templates**, **design variations**, and **problem mappings**.

🔵 CIRCULAR QUEUE (CQ) MASTER PATTERNS


Front
Rear
10
20
30
40
50
0
1
2
3
4
5
6
7


Circular Queue Representation
New element can be inserted by incrementing Rear Pointer.
0
2
4
Rear
9
1
3
Front
1
3
-2
2
ScholarHat



A **Circular Queue** (a.k.a. **Ring Buffer**) is a fixed-size queue where:

- Index wraps using modulo: `(index % size)`
- Eliminates need for shifting elements
- Used in **bounded buffers**, **streaming systems**, **sliding windows**, and **rate limiters**

==========

⭐ PATTERN 1 — Classic Circular Queue (Array + Two Pointers)

==========

This is the standard CQ problem pattern.

✔ Structure

- `front` → index of first element
- `rear` → index of next insertion
- Queue size = fixed capacity
- `count` → number of elements (recommended)

✔ Template (fastest, LC-622 style)

```
python

class MyCircularQueue:
    def __init__(self, k):
        self.q = [0] * k
        self.cap = k
        self.front = 0
        self.rear = 0
        self.count = 0

    def enQueue(self, x):
        if self.count == self.cap:
            return False
        self.q[self.rear] = x
        self.rear = (self.rear + 1) % self.cap
        self.count += 1
        return True
```

```
    def deQueue(self):
        if self.count == 0:
            return False
        self.front = (self.front + 1) % self.cap
        self.count -= 1
        return True

    def Front(self):
        return -1 if self.count == 0 else self.q[self.front]

    def Rear(self):
        return -1 if self.count == 0 else self.q[(self.rear - 1) % self.cap]

    def isEmpty(self):
        return self.count == 0

    def isFull(self):
        return self.count == self.cap
```

✔️ Pattern Use-Cases

- LC 622: Design Circular Queue
- Bounded buffer system
- OS scheduling (round-robin)
- Token bucket algorithms

==========

⭐ PATTERN 2 — Circular Queue Without Count (Using One Slot Empty Trick)

==========

No `count` variable; instead, keep **one space empty** to detect full/empty.

✔️ Invariant

```
csharp

Queue is full if (rear + 1) % k == front
Queue is empty if rear == front
```

✔️ Template

```
python

class CircularQueue:
    def __init__(self, k):
        self.q = [0] * (k + 1)
        self.size = k + 1
        self.front = 0
        self.rear = 0

    def enQueue(self, x):
```

```
        if (self.rear + 1) % self.size == self.front:
            return False  # full
        self.q[self.rear] = x
        self.rear = (self.rear + 1) % self.size
        return True

    def deQueue(self):
        if self.front == self.rear:
            return False  # empty
        self.front = (self.front + 1) % self.size
        return True
```

✔ Benefits

- Simpler logic
- No need to maintain count

==========

⭐ PATTERN 3 — Circular Deque (Double-Ended Circular Queue)

==========

For problems requiring **push/pop from both ends**.

✔ Template

```
python

class MyCircularDeque:
    def __init__(self, k):
        self.q = [0]*k
        self.cap = k
        self.front = 0
        self.rear = 0
        self.count = 0

    def insertFront(self, x):
        if self.count == self.cap:
            return False
        self.front = (self.front - 1 + self.cap) % self.cap
        self.q[self.front] = x
        self.count += 1
        return True

    def insertLast(self, x):
        if self.count == self.cap:
            return False
        self.q[self.rear] = x
        self.rear = (self.rear + 1) % self.cap
        self.count += 1
        return True

    def deleteFront(self):
        if self.count == 0:
            return False
        self.front = (self.front + 1) % self.cap
        self.count -= 1
        return True

    def deleteLast(self):
        if self.count == 0:
```

```
            return False
        self.rear = (self.rear - 1 + self.cap) % self.cap
        self.count -= 1
        return True
```

✔ Problems

- LC 641 Design Circular Deque
- Sliding window with double-ended operations

==========

⭐ PATTERN 4 — Circular Queue in Graph & Simulation Problems

==========

Circular queue reduces overhead in iterative simulations.

✔ Use-Cases

- Round Robin scheduling
- Multi-stream merging
- Token ring networks
- CPU time-slice simulations

✔ Pattern: Increment pointer cyclically

```
python

i = (i + 1) % n
```

==========

⭐ PATTERN 5 — Circular Buffer for Streaming Sliding Window

==========

Used for real-time algorithms (rate limiter, streaming analytics).

✔ Template

```
python

class CircularWindow:
    def __init__(self, k):
        self.buf = [0]*k
```

```
        self.idx = 0
        self.size = k

    def add(self, x):
        self.buf[self.idx] = x
        self.idx = (self.idx + 1) % self.size
```

✔️ Problems

- Moving average in streaming
- IoT sensor rolling buffer
- "Last K readings"

==========

⭐ PATTERN 6 — Circular Queue for Producer–Consumer

==========

Bounded queue for multithreading environments:

✔️ Producer writes at **`rear`**
✔️ Consumer reads from **`front`**

✔️ Template

```
python

from threading import Lock, Semaphore

class BoundedBuffer:
    def __init__(self, k):
        self.buf = [0]*k
        self.front = 0
        self.rear = 0
        self.empty = Semaphore(k)
        self.full = Semaphore(0)
        self.lock = Lock()

    def produce(self, x):
        self.empty.acquire()
        with self.lock:
            self.buf[self.rear] = x
            self.rear = (self.rear + 1) % len(self.buf)
        self.full.release()

    def consume(self):
        self.full.acquire()
        with self.lock:
            x = self.buf[self.front]
            self.front = (self.front + 1) % len(self.buf)
        self.empty.release()
        return x
```

==========

⭐ PATTERN 7 — Circular Queue in Hashing & Overflow Handling

==========

Used for:

- Cuckoo hashing cycles
- Open addressing
- Robin Hood hashing

Pattern: probe in circular manner.

```
python

pos = (pos + 1) % table_size
```

==========

⭐ PROBLEM MAPPING (Interview Sheet)

==========

Pattern	Problems
Classic CQ	LC 622
Circular Deque	LC 641
Sliding Window Circular Buffer	Moving average, sensor data
Simulation	Round-robin scheduling
Stream Buffer	Rate limiters
Hashing Circularity	Robin Hood hashing variants

==========

🎯 SUMMARY OF CIRCULAR QUEUE PATTERNS

==========

Pattern	Description	Key Operations
CQ with count	Easiest to implement	maintain `(front, rear, count)`
CQ without count	Uses dummy slot trick	`full = (rear+1)%k == front`
Circular Deque	Push/pop at both ends	wrap both pointers
CQ for simulation	OS, networking, RR scheduling	modular pointer movement

Pattern	Description	Key Operations
CQ for streaming	Fixed window rolling buffer	overwrite oldest values
CQ for DP or hashing	cyclic probing	avoid overflow

🎁 Want 20 Circular Queue Problems, CQ vs MQ comparison, or CQ templates for interviews?

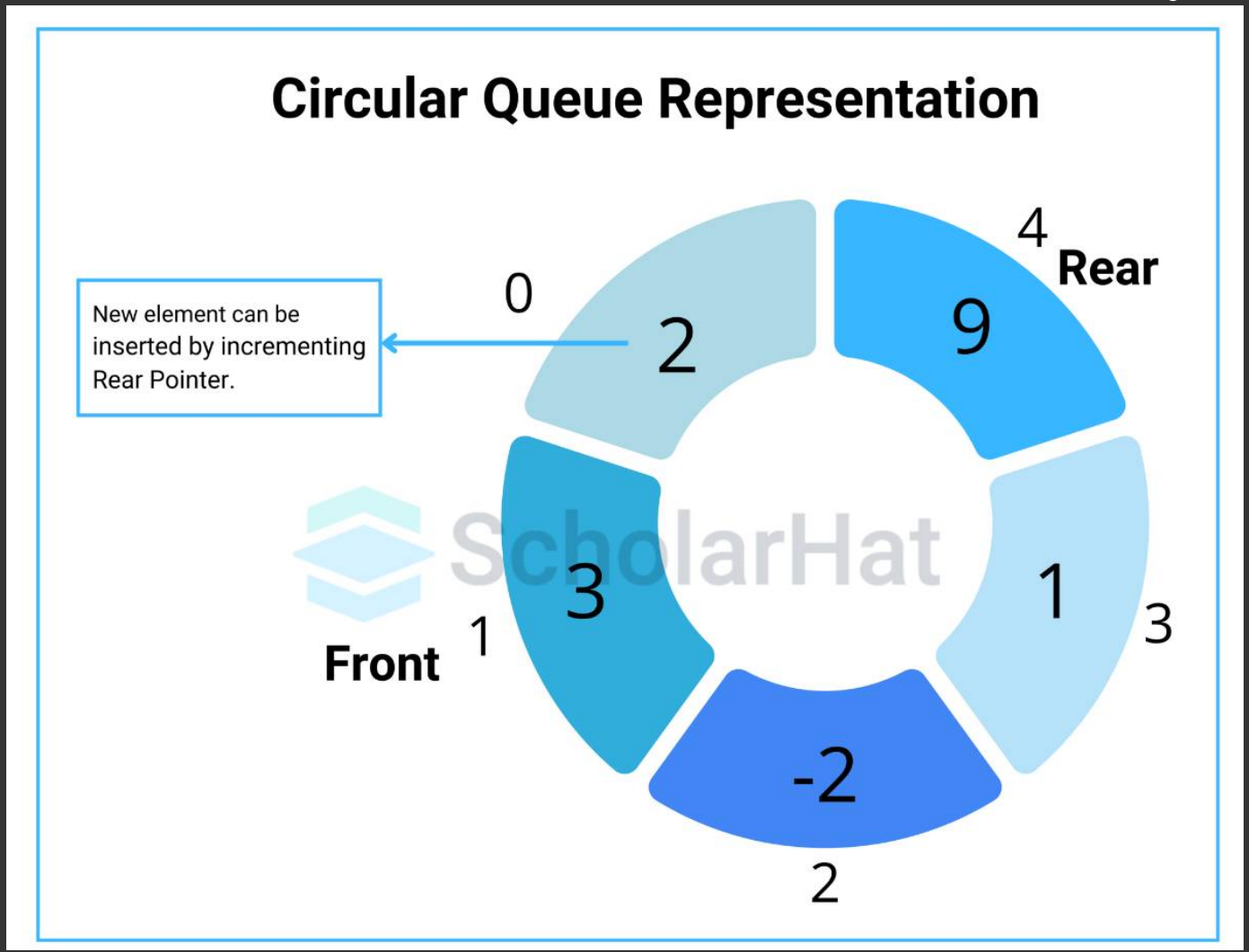
Just tell me: **"CQ 20 Problems"**, **"CQ Templates"**, or **"CQ vs MQ Sheet"**.

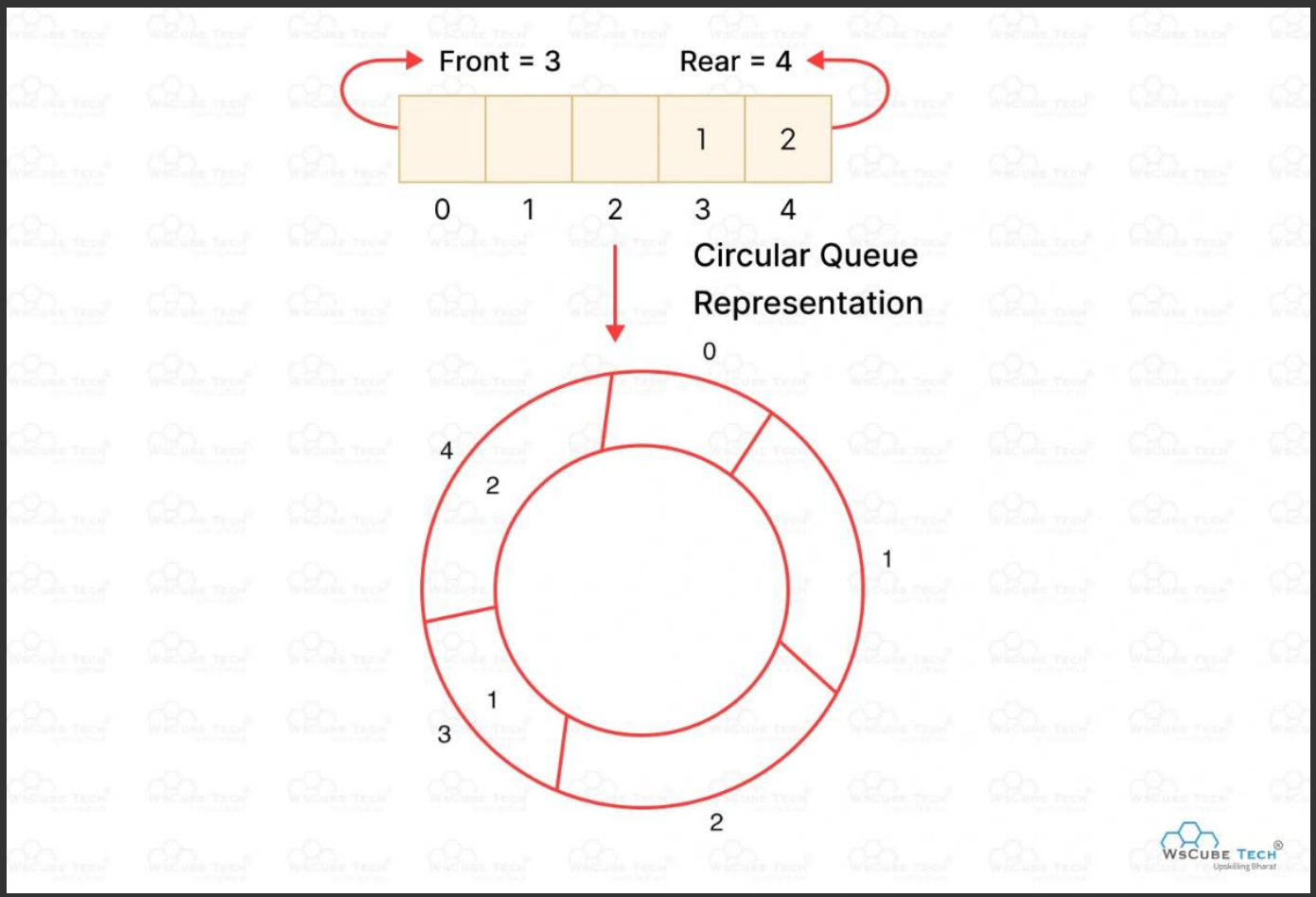
CQ Methods

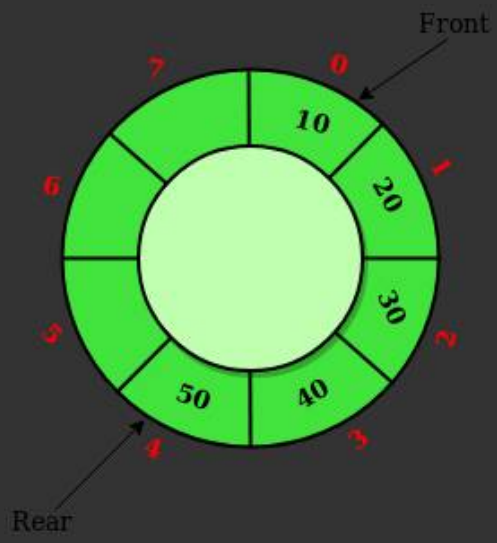
Below is the **DSA-Mastery "Circular Queue Methods Handbook"** — all essential **CQ operations**, clean patterns, edge-case rules, and optimal implementations.

This covers **all method definitions** used in interviews (LC 622, LC 641, system design, CP).

🔵 CIRCULAR QUEUE (CQ) — COMPLETE METHOD SET


Circular Queue Representation
New element can be inserted by incrementing Rear Pointer.
0
2
4
Rear
9
1
3
-2
2
3
1
Front
ScholarHat


Front = 3
Rear = 4
1
2
0
1
2
3
4
Circular Queue Representation
0
4
2
1
1
3
2
WSCUBE TECH



A Circular Queue exposes these key methods:

```
scss

enQueue(x)
deQueue()
Front()
Rear()
isEmpty()
isFull()
size()          (optional)
capacity()      (optional)
```

Below you get each method's **meaning**, **implementation rule**, and **Python template**.

==========

⭐ FULL METHOD SET (with count variable — recommended)

==========

```
python

class MyCircularQueue:
    def __init__(self, k):
        self.q = [0] * k          # fixed-size buffer
        self.cap = k
        self.front = 0            # index of first element
        self.rear = 0             # index for next insert
        self.count = 0            # number of elements
```

1️⃣ enQueue(x)

Insert element at the **rear**.

✔ Rule

- Cannot insert if queue is **full**

- Place at `rear`
- Move `rear = (rear + 1) % cap`
- Increment `count`

✔️ Implementation

```python
def enQueue(self, x):
    if self.count == self.cap:
        return False
    self.q[self.rear] = x
    self.rear = (self.rear + 1) % self.cap
    self.count += 1
    return True
```

2️⃣ deQueue()

Remove element from the **front**.

✔️ Rule

- Cannot remove if **empty**
- Move `front = (front + 1) % cap`
- Decrease `count`

✔️ Implementation

```python
def deQueue(self):
    if self.count == 0:
        return False
    self.front = (self.front + 1) % self.cap
    self.count -= 1
    return True
```

3️⃣ Front()

Return the element at the front (**peek** operation).

✔️ Rule

If empty → return `-1`.

```python
def Front(self):
    return -1 if self.count == 0 else self.q[self.front]
```

4️⃣ Rear()

Return the **last inserted** element.

✔️ Rule

Rear points to the *next empty slot*, so last element is at `(rear - 1 + cap) % cap`.

```python
def Rear(self):
    return -1 if self.count == 0 else self.q[(self.rear - 1) % self.cap]
```

5️⃣ isEmpty()

```python
def isEmpty(self):
    return self.count == 0
```

6️⃣ isFull()

```python
def isFull(self):
    return self.count == self.cap
```

7️⃣ size() (optional but useful in CP)

```python
def size(self):
    return self.count
```

8️⃣ capacity() (optional)

```python
def capacity(self):
    return self.cap
```

==========

⭐ METHOD SET WITHOUT count (1 empty slot trick)

==========

Useful in languages where count is expensive.

✔️ Full queue rule:

```
arduino

(rear + 1) % size == front
```

✔️ Empty queue rule:

```
ini

rear == front
```

Implementation

```
python

class CircularQueue:
    def __init__(self, k):
        self.size = k + 1          # 1 extra space
        self.q = [0] * self.size
        self.front = 0
        self.rear = 0

    def enQueue(self, x):
        if (self.rear + 1) % self.size == self.front:
            return False
        self.q[self.rear] = x
        self.rear = (self.rear + 1) % self.size
        return True

    def deQueue(self):
        if self.rear == self.front:
            return False
        self.front = (self.front + 1) % self.size
        return True
```

==========

⭐ CIRCULAR DEQUE (Double-Ended Queue) METHODS

==========

This is LC **641** style.

```
scss

insertFront(x)
insertLast(x)
deleteFront()
deleteLast()
getFront()
```

```
getRear()
isEmpty()
isFull()
```

Insert at front

```python
def insertFront(self, x):
    if self.count == self.cap:
        return False
    self.front = (self.front - 1 + self.cap) % self.cap
    self.q[self.front] = x
    self.count += 1
    return True
```

Insert at rear

```python
def insertLast(self, x):
    if self.count == self.cap:
        return False
    self.q[self.rear] = x
    self.rear = (self.rear + 1) % self.cap
    self.count += 1
    return True
```

Delete from front

```python
def deleteFront(self):
    if self.count == 0:
        return False
    self.front = (self.front + 1) % self.cap
    self.count -= 1
    return True
```

Delete from rear

```python
def deleteLast(self):
    if self.count == 0:
        return False
    self.rear = (self.rear - 1 + self.cap) % self.cap
    self.count -= 1
    return True
```

==========

⭐ EXTRA METHODS (System Design / CP)

==========

1️⃣ overwrite(x) — replace oldest on full

Used in streaming buffers.

```
python

def overwrite(self, x):
    if self.isFull():
        self.front = (self.front + 1) % self.cap
        self.count -= 1
    self.enQueue(x)
```

2️⃣ peekNextInsert()

```
python

def nextInsertIndex(self):
    return self.rear
```

3️⃣ advancePointer(i)

General pointer wrapping:

```
python

def adv(self, i):
    return (i + 1) % self.cap
```

4️⃣ distance(front, rear) (useful for debugging)

```
python

def distance(self):
    return (self.rear - self.front + self.cap) % self.cap
```

==========

⭐ SUMMARY — ALL CQ METHODS YOU MUST KNOW

Method	Meaning
`enQueue(x)`	Insert at rear
`deQueue()`	Remove from front
`Front()`	Get front element
`Rear()`	Get last element
`isEmpty()`	No elements
`isFull()`	Queue full

Method	Meaning
`size()`	# elements
`capacity()`	Max size
`insertFront()`	CQ deque op
`insertLast()`	CQ deque op
`deleteFront()`	CQ deque op
`deleteLast()`	CQ deque op
`overwrite(x)`	Replace oldest (streaming)

🎁 Want “Circular Queue 25 Problems”, “CQ Patterns + Templates”, or “CQ Interview Notebook”?

Just say **CQ 25 Problems**, **CQ Patterns**, or **CQ Handbook**.