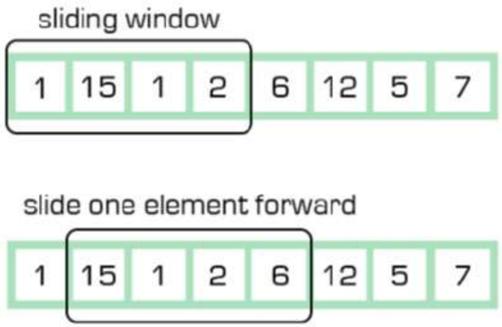
Sliding Window Technique

Table of Contents

1. What is sliding window technique?
2. Fixed size window
   - Sliding Window Maximum (Leetcode-239)
   - Max Sum Subarray of size K (GFG)
3. Variable size window
   - Minimum Size Subarray Sum (Leetcode-209)
   - Binary Subarrays With Sum (Leetcode-930)
4. Two pointer approach
   - Two Sum (Leetcode-1)
5. Optimization approach
   - Minimum Window Substring (Leetcode-76)
6. Master sliding window through this resource

https://leetcode.com/discuss/general-discussion/1122776/summary-of-sliding-window-patterns-for-subarray-substring

Window size = 4



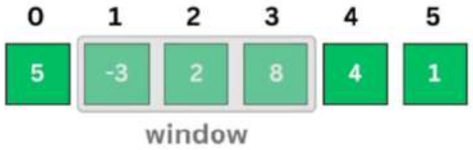
1. What is sliding window technique?

The sliding window is a problem-solving technique that's designed to transform two nested loops into a single loop. It applies to arrays or lists. These problems are painless to solve using a brute force approach in $O(n^2)$ or $O(n^3)$. However, the sliding window technique can reduce the time complexity to $O(n)$.

📝 ***Sliding window variants:***

1. *Fixed size window*
2. *Variable size window*
3. *Two pointer approach*
4. *Optimization approach*

📝 ***Master sliding window through this resource:***
https://leetcode.com/discuss/general-discussion/1122776/summary-of-sliding-window-patterns-for-subarray-substring



Sliding Window Algorithm

2. Fixed size window

Problem 1: Sliding Window Maximum (Leetcode-239)
Problem 2: Max Sum Subarray of size K (GFG)

Fixed Size Sliding Window Approach:

***Note**: Determine window size*
Fixed Size 'K' Window

***Step 1:** Process first 'K' elements*
Initial State

***Step 2:** Process remaining window*
- ***Remove***
- ***Addition***
- ***Store***

1. Sliding Window Maximum (Leetcode-239)

Ex

nums	1	3	-1	-3	5	3	6	7
	0	1	2	3	4	5	6	7

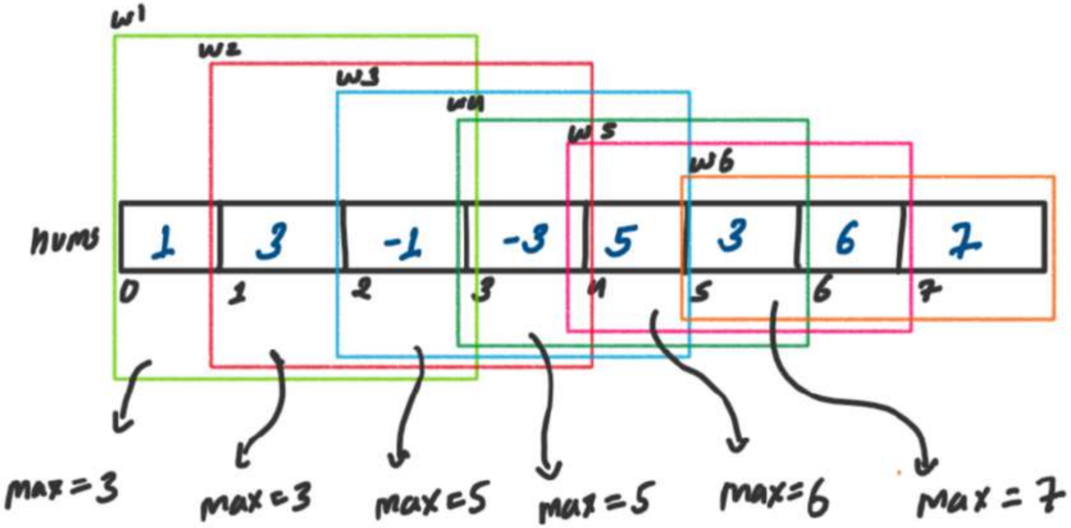
K=3

Fixed size window

Output

3	3	5	5	6	7

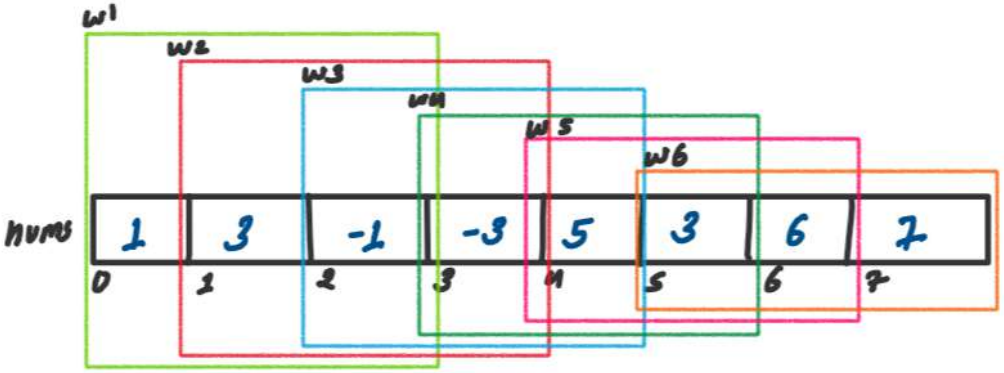
Explanation



K=3

N=size=8

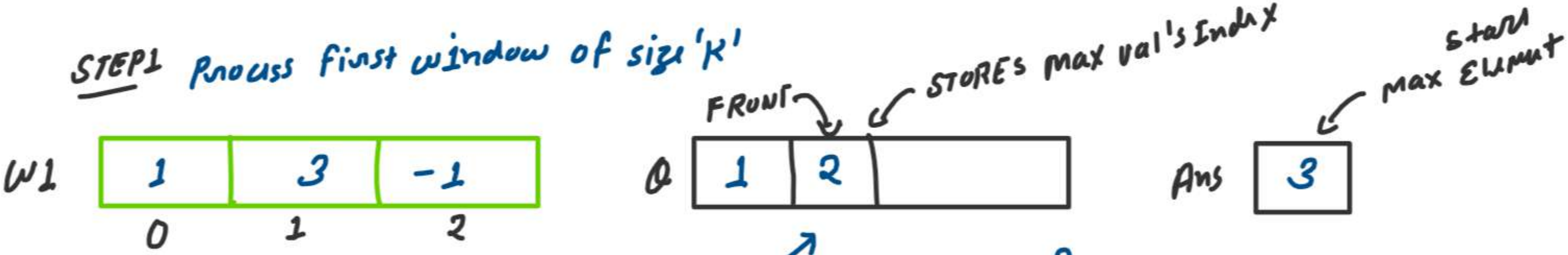
DRY RUN



K=3

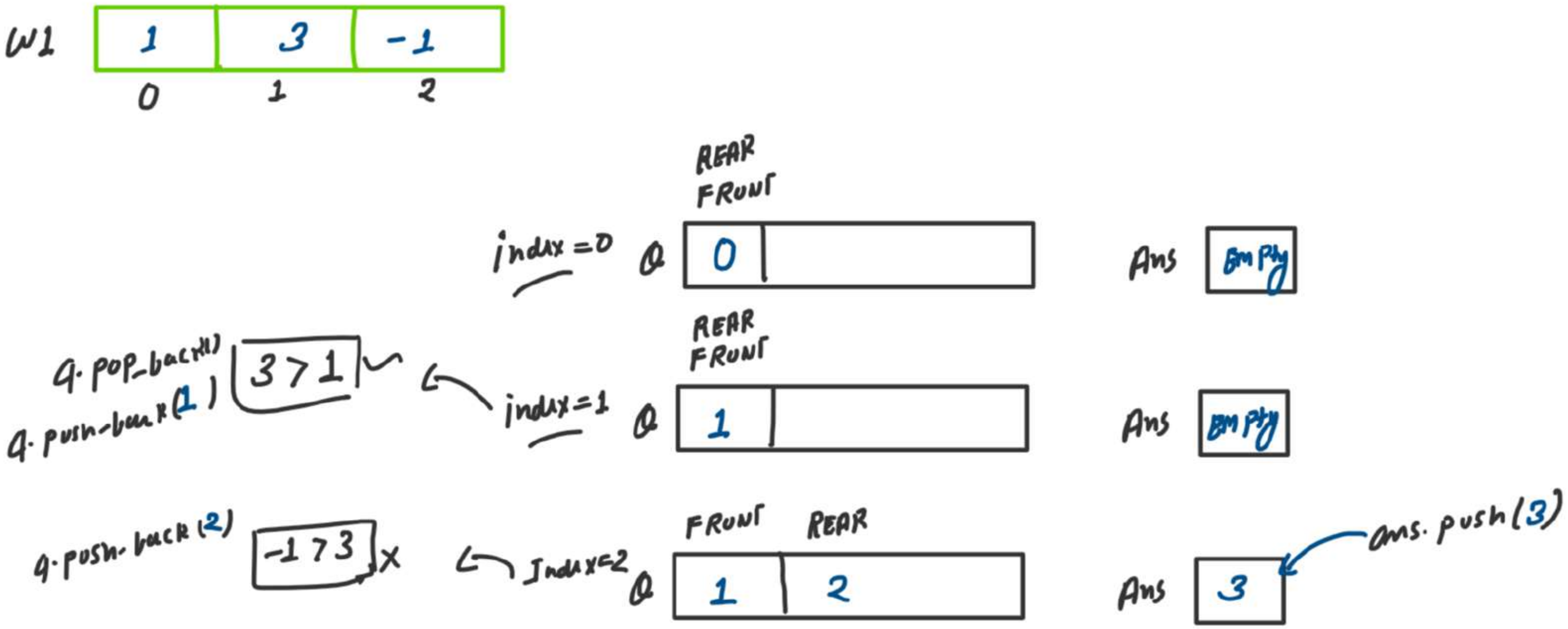
N = size = 8

STEP1 Process first window of size 'K'



YEH kaise fill HO GYA?

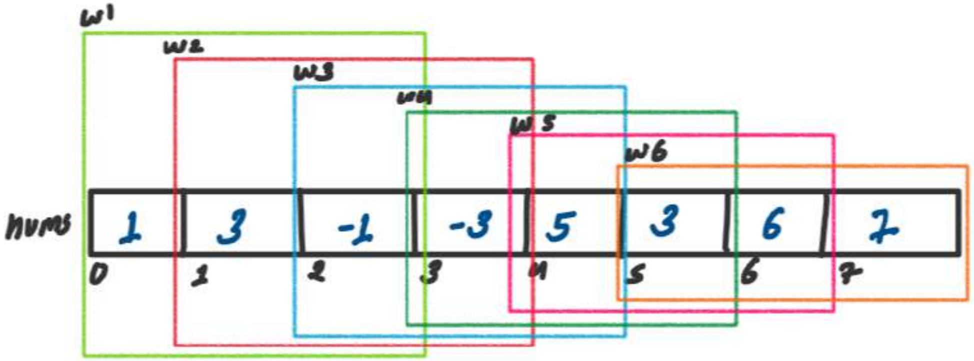
STEP1 Process first window of size 'k'



STEP2 Process Remaining windows

↳ Remove

Insertion



K=3

N = size = 8

W2

3	-1	-3
1	2	3

× I 3 - 1 >= 3

× II -3 > -1

index=3

Q

F		R	
1	2	3	

Ans 3 ← q.push(3)

REMOVE (in queue)

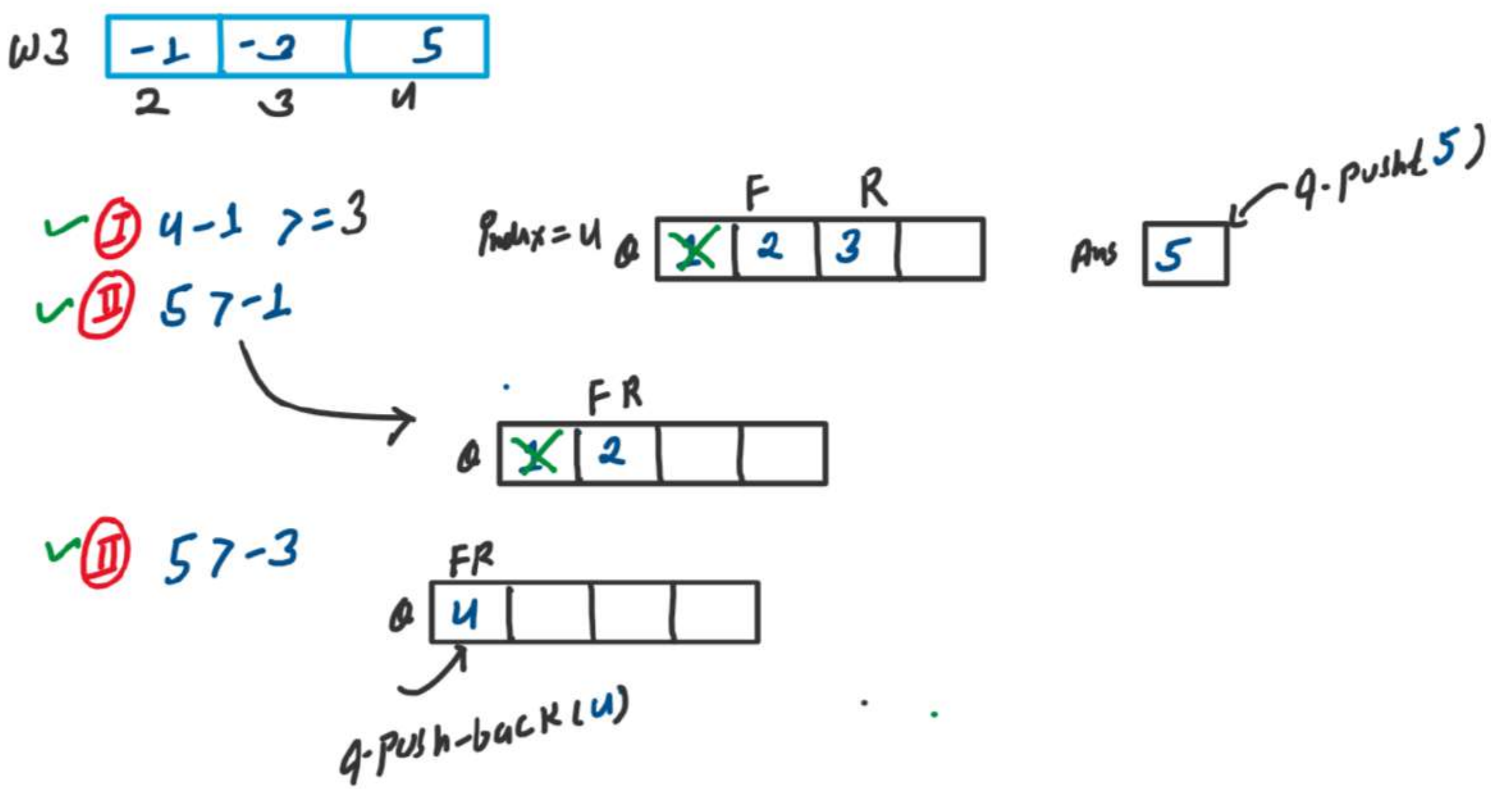
I Out of Range [0,1,2]

II Chotta Element Remove

Index - q.front() >= K

↳ q.pop_front();

q.pop_back()



REMOVE (in queue)

I Out of Range [0,1,2]

II Chotta Element Removed

Index-q.front() >= K

q.pop-front();

q.pop-back();

W4	-3	5	3
	3	4	5

x Ⅰ 5-4 >= 3

x Ⅱ 3 > 5

		F	R		
Index=5	Q	4	5		

Ans | 5 | ← q.push(5)

REMOVE (in queue)

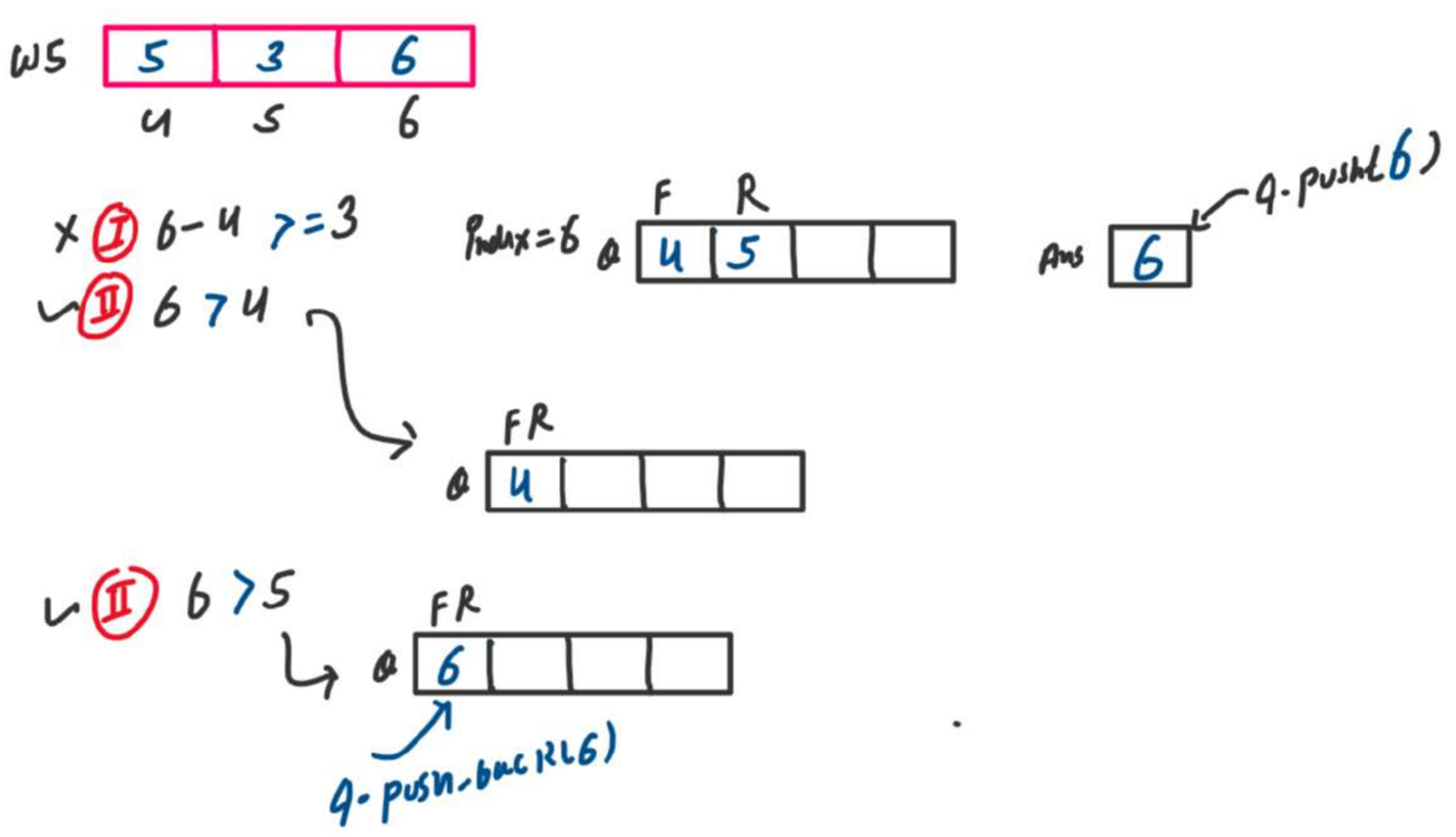
Ⅰ Out of Range [0,1,2]

Ⅱ Chotta Element Removed

Ⅰ → Index - q.front() >= K

↳ q.pop_front();

Ⅱ → q.pop_back();



REMOVE (in queue)

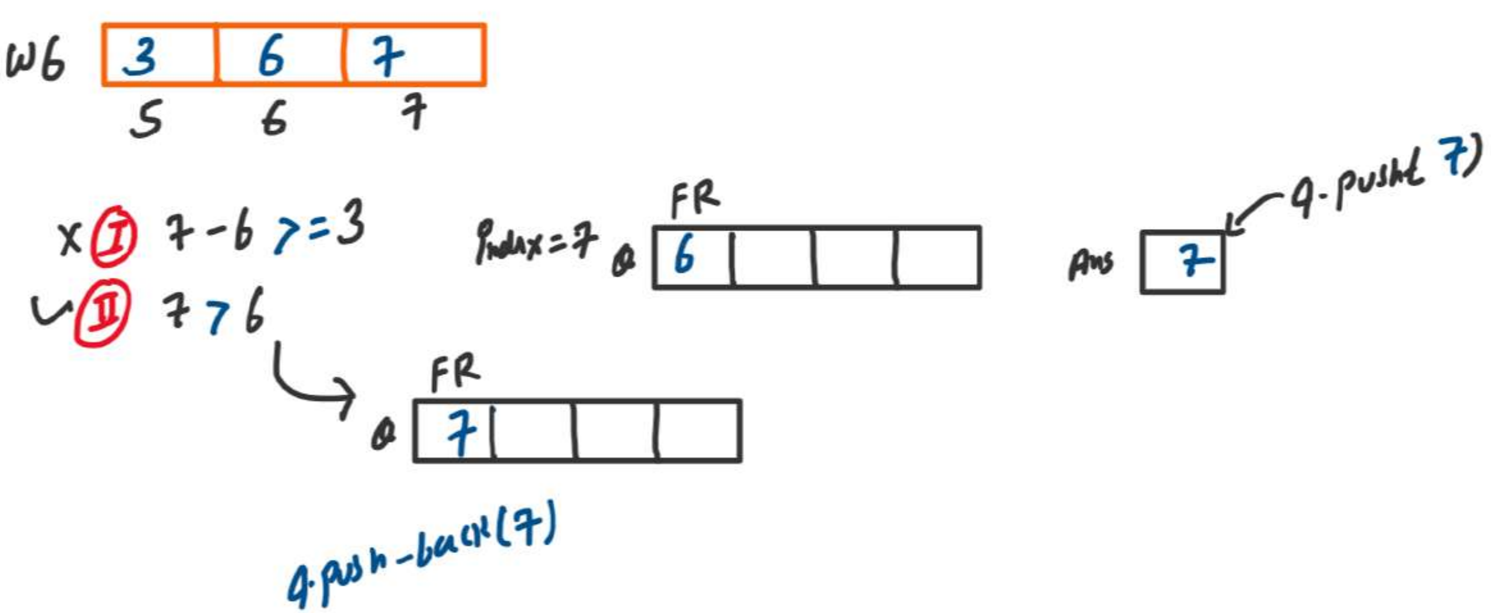
I Out of Range [0,1,2]

II Chotta Element Removed

Index - q.front() >= K

→ q.pop_front();

→ q.pop_back();



REMOVE (in queue)

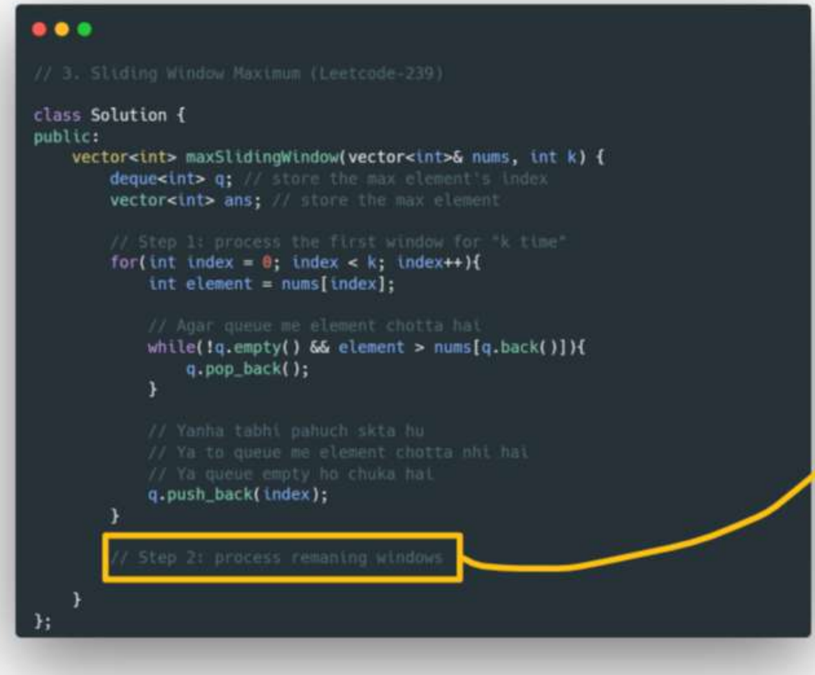
I Out of Range [0, 1, 2]

II Chotta Element Removed

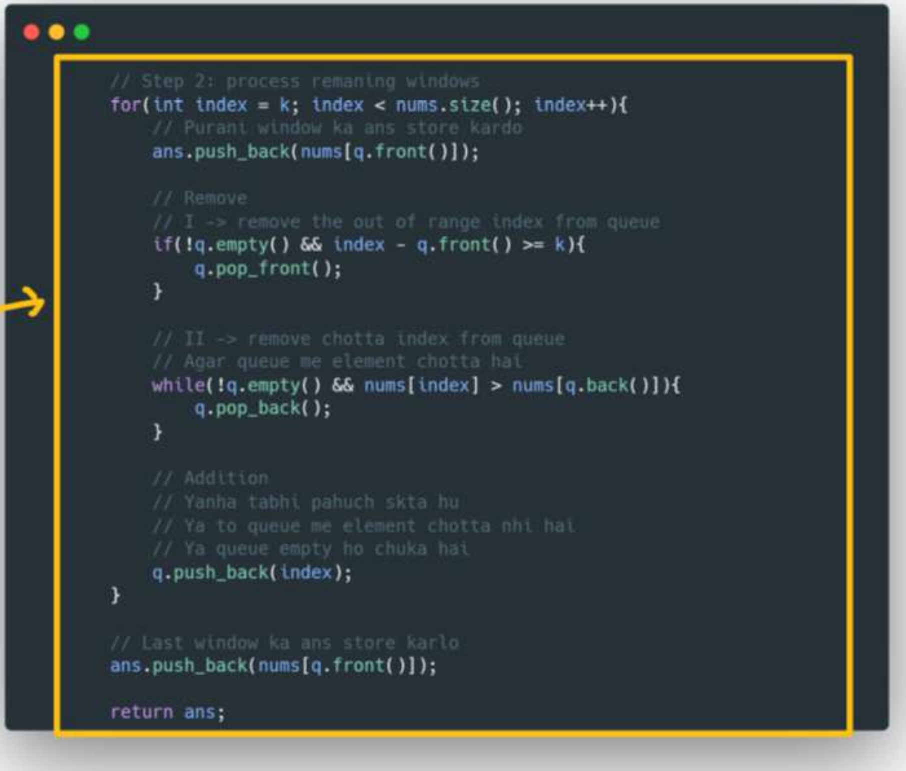
Index - q.front() >= K

q.pop_front();

q.pop_back();



```
// 3. Sliding Window Maximum (Leetcode-239)

class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int> q; // store the max element's index
        vector<int> ans; // store the max element

        // Step 1: process the first window for "k time"
        for(int index = 0; index < k; index++){
            int element = nums[index];

            // Agar queue me element chotta hai
            while(!q.empty() && element > nums[q.back()]){
                q.pop_back();
            }

            // Yanha tabhi pahuch skta hu
            // Ya to queue me element chotta nhi hai
            // Ya queue empty ho chuka hai
            q.push_back(index);
        }

        // Step 2: process remaning windows

    }
};
```



```
        // Step 2: process remaning windows
        for(int index = k; index < nums.size(); index++){
            // Purani window ka ans store kardo
            ans.push_back(nums[q.front()]);

            // Remove
            // I -> remove the out of range index from queue
            if(!q.empty() && index - q.front() >= k){
                q.pop_front();
            }

            // II -> remove chotta index from queue
            // Agar queue me element chotta hai
            while(!q.empty() && nums[index] > nums[q.back()]){
                q.pop_back();
            }

            // Addition
            // Yanha tabhi pahuch skta hu
            // Ya to queue me element chotta nhi hai
            // Ya queue empty ho chuka hai
            q.push_back(index);
        }

        // Last window ka ans store karlo
        ans.push_back(nums[q.front()]);

        return ans;
```

Time Complexity: *O(N),*
where N is size of array

Space Complexity: *O(K),*
where K is the size of the window

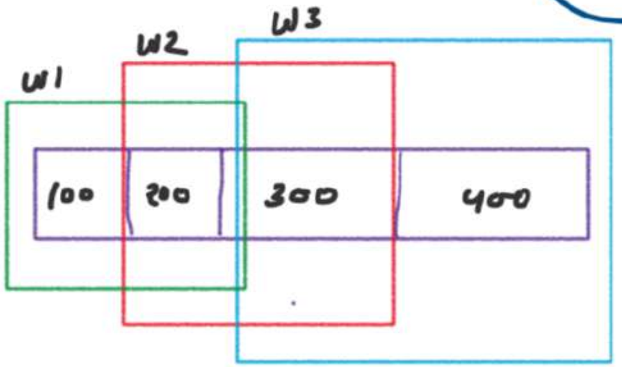
📂 2. Max Sum Subarray of Size K (GFG)

Problem Statement:
Given an array of integers **Arr** of size **N** and a number **K**. Return the **maximum sum** of a subarray of **size K.**

Fixed size window

NOTE: A subarray is a contiguous part of any given array.

Example 1:
Input: N = 4, K = 2, Arr = [100, 200, 300, 400]
Output: 700
Explanation: Arr3 + Arr4 =700, which is maximum.

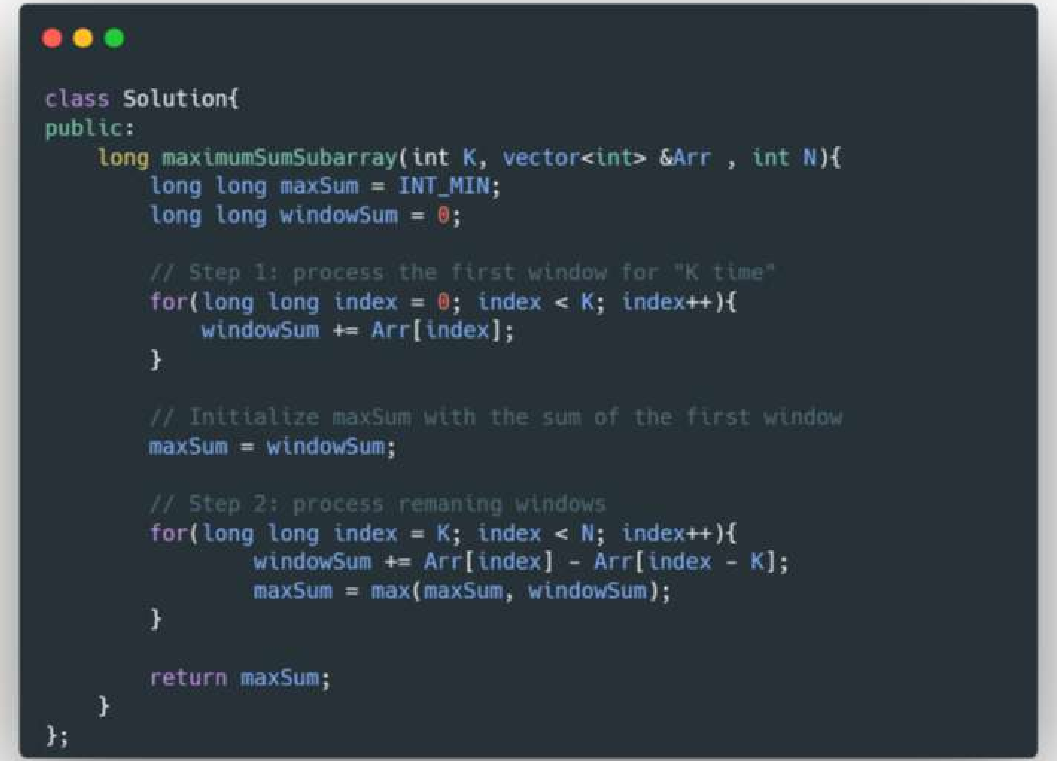


Example 2:
Input: N = 4, K = 4, Arr = [100, 200, 300, 400]
Output: 1000
Explanation: Arr1 + Arr2 + Arr3 + Arr4 =1000, which is maximum.

sum of

w1 = 100 + 200
w2 = 200 + 300
w3 = 300 + 400

max sum
= 300 + 400
= 700



```cpp
class Solution{
public:
    long maximumSumSubarray(int K, vector<int> &Arr , int N){
        long long maxSum = INT_MIN;
        long long windowSum = 0;

        // Step 1: process the first window for "K time"
        for(long long index = 0; index < K; index++){
            windowSum += Arr[index];
        }

        // Initialize maxSum with the sum of the first window
        maxSum = windowSum;

        // Step 2: process remaning windows
        for(long long index = K; index < N; index++){
                windowSum += Arr[index] - Arr[index - K];
                maxSum = max(maxSum, windowSum);
        }

        return maxSum;
    }
};
```

Time Complexity: *O(N)*

Space Complexity: *O(1)*