### Sliding Window Pattern

#### What is the Sliding Window Pattern?

The sliding window pattern is a technique used to solve problems that involve a contiguous subarray or substring within an array or string. It involves maintaining a window that slides over the array/string to find the optimal solution without reprocessing elements unnecessarily. This helps in reducing the time complexity significantly from O(n^2) to O(n) in many cases.

#### How is it Used?

The sliding window pattern generally works in the following way:

1. Initialize the window with a starting point.

2. Expand or contract the window based on the problem's requirements.

3. Keep track of the best solution while the window slides over the array or string.

#### Where to Use (Conditions)?

The sliding window pattern is useful when:

- You need to find the maximum/minimum/sum/average of a subarray or substring.

- The problem involves contiguous elements.

- The problem can be broken down into overlapping subproblems.

#### Example Problem: Maximum Sum Subarray of Size K

\*\*Problem Statement:\*\* Given an array of integers and a number `k`, find the maximum sum of a subarray of size `k`.

#### Example to Understand

Consider the array `[2, 1, 5, 1, 3, 2]` and `k = 3`.

1. Start with the first window `[2, 1, 5]` which has a sum of `8`.

2. Slide the window one element to the right to `[1, 5, 1]` and update the sum to `7`.

3. Continue this until the end of the array.

The maximum sum of any subarray of size `k` is `10` from the subarray `[5, 1, 3]`.

def maximum\_sum\_of\_k\_size\_subarray(arr,k)->int:

    max\_sum=0

    window\_sum=0

    start=0

    for i in range(k):

        window\_sum+=arr[i]

    max\_sum=window\_sum

    for i in range(k,len(arr)):

        window\_sum += arr[i] - arr[start]

        max\_sum = max(max\_sum, window\_sum)

        start+=1

    print("maximum sum-->",max\_sum)

# arr = [2,1,5,1,3,2]

# k = 3

# maximum\_sum\_of\_k\_size\_subarray(arr,k)

if \_\_name\_\_ == "\_\_main\_\_":

    t = int(input("Enter number of test cases: "))

    for i in range(t):

        k = int(input("Enter the window size: "))

        arr = list(map(int,input().split()))

        maximum\_sum\_of\_k\_size\_subarray(arr,k)

# Example usage

arr = [2, 1, 5, 1, 3, 2]

k = 3

print(max\_sum\_subarray(arr, k)) # Output: 10

```

#### C++ Code

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int maxSumSubarray(vector<int>& arr, int k) {

    int max\_sum = 0;

    int window\_sum = 0;

    int start = 0;

    // Calculate the sum of the first window

    for (int i = 0; i < k; ++i) {

        window\_sum += arr[i];

    }

    max\_sum = window\_sum;

    // Slide the window

    for (int end = k; end < arr.size(); ++end) {

        window\_sum += arr[end] - arr[start];

        max\_sum = max(max\_sum, window\_sum);

        start += 1;

    }

    return max\_sum;

}

int main() {

    vector<int> arr = {2, 1, 5, 1, 3, 2};

    int k = 3;

    cout << maxSumSubarray(arr, k) << endl;  // Output: 10

    return 0;

}

#### Problems on LeetCode

1. [Maximum Sum Subarray of Size K (Easy)](<https://leetcode.com/problems/maximum-average-subarray-i/>)

2. [Longest Substring Without Repeating Characters (Medium)](<https://leetcode.com/problems/longest-substring-without-repeating-characters/>)

3. [Permutation in String (Medium)](<https://leetcode.com/problems/permutation-in-string/>)

4. [Minimum Window Substring (Hard)](<https://leetcode.com/problems/minimum-window-substring>/)

5. [Sliding Window Maximum (Hard)](<https://leetcode.com/problems/sliding-window-maximum/>)

By understanding and practicing the sliding window pattern, you'll be able to efficiently solve a wide range of problems that involve contiguous subarrays or substrings.