**🧠 Mindset for These Problems**

* These are **constructive algorithms**, not formulaic ones.
* Learn **patterns** and **mold** them according to the problem.

**✅ Four Patterns to Focus On**

**🟡 1. Constant Size Sliding Window**

**🧩 Problem Type:**

“Find maximum sum of K consecutive elements in an array.”

**🔍 Example:**

plaintext

Copy code

Array = [-1, 2, 3, 3, 2, -1]

K = 4

* First window: [-1, 2, 3, 3] → sum = 7
* Next window: [2, 3, 3, 2] → sum = 10
* Next window: [3, 3, 2, -1] → sum = 7

**🧪 Template:**

1. Compute sum of the first K elements.
2. Slide the window one element at a time:
   * Subtract the element going out of the window.
   * Add the new element coming into the window.
3. Track max\_sum.

**🧠 Code Skeleton:**

int sum = 0;

for (int i = 0; i < K; i++) {

sum += arr[i];

}

int maxSum = sum;

for (int i = K; i < n; i++) {

sum = sum - arr[i - K] + arr[i];

maxSum = Math.max(maxSum, sum);

}

**💡 Tips:**

* Use two pointers (L and R) to track window start and end.
* Be careful with bounds (R should not go out of array length).

**🟢 2. Variable Size Sliding Window**

**🧩 Problem Type:**

“Find longest subarray/substring with sum ≤ K (or other conditions).”

**🧪 Key Concepts:**

* Subarray: **Contiguous** elements.
* Start with Brute Force → Better → Optimal.

**🧪 Brute Force Approach:**

* Generate **all possible subarrays** using nested loops.
* For each subarray, compute the sum and check condition.

**🧠 Code Skeleton:**

for (int i = 0; i < n; i++) {

int sum = 0;

for (int j = i; j < n; j++) {

sum += arr[j];

if (sum <= K) {

maxLength = Math.max(maxLength, j - i + 1);

}

}

}

**🔥 Drawback:**

* Time Complexity: **O(n²)** → Not good for large input.

This explanation is a **walkthrough of the sliding window two-pointer technique** used to solve problems involving the **longest subarray** that satisfies a particular condition — specifically when **sum ≤ K**.

Here’s a concise summary with key points and code template based on the explanation:

**✅ Problem Type:**

* Find the **longest subarray** with **sum ≤ K**

**✅ Key Concepts:**

1. **Sliding Window Technique**:
   * Maintains a window using two pointers: left (L) and right (R)
   * You **expand** the window by increasing R
   * You **shrink** the window by increasing L (when the sum exceeds K)
2. **Always**:
   * Start with L = 0, R = 0, sum = 0, maxLen = 0
   * Move R to expand and include more elements
   * Shrink from L to maintain validity when sum > K
   * Update maxLen when you find a valid window

**✅ Code Template (in Java-like pseudocode):**

int left = 0, right = 0;

int sum = 0;

int maxLen = 0;

while (right < n) {

sum += arr[right];

// Shrink until the window becomes valid again

while (sum > K) {

sum -= arr[left];

left++;

}

// Update max length when valid

maxLen = Math.max(maxLen, right - left + 1);

// Expand window

right++;

}

System.out.println("Maximum length = " + maxLen);

**✅ Time and Space Complexity:**

* **Time Complexity**: O(n)
  + R moves from 0 to n-1 → O(n)
  + Each element is removed from the window at most once (L) → O(n)
  + So total: O(2n) = O(n)
* **Space Complexity**: O(1)
  + No extra space unless problem uses data structures like maps/sets

**✅ When to Use This Pattern:**

* Subarray problems involving:
  + **Longest/Shortest subarray**
  + **Sum or condition based on elements**
  + Conditions like: sum ≤ K, product < K, count <= N, etc.

**✅ Tips to Remember:**

* Expansion (right++) happens always
* Shrinking (left++) happens only when the condition is **violated**
* Always update the result **after validating** the window

This is a transcript from a video lecture explaining how to **optimize the sliding window algorithm** when solving array-related problems — especially when you're tasked with:

* **Finding the length of the longest subarray** whose sum is less than or equal to K
* **Counting the number of subarrays** that satisfy certain conditions
* **Avoiding unnecessary shrinking** of the window

**🔍 Key Concepts & Optimizations:**

**✅ Problem Setup:**

You're given an array and a value K. You want to find the **maximum length of a subarray** such that the **sum of its elements is less than or equal to K**.

**1. Initial Approach:**

Using two pointers (L and R) and a sliding window:

* Expand R to include elements.
* If the sum goes beyond K, **shrink** from L until the sum is within limits.

This naive approach can lead to:

* **O(2N)** time complexity because in the worst case both L and R move separately across the array.

**2. Optimization Idea:**

Instead of **shrinking all the way**, shrink **just one step**.

**Example:**

Array: [2, 5, 1, 10, 10]  
K: 14

You start expanding R and building a valid window:

* [2] → sum = 2 ✅
* [2,5] → sum = 7 ✅
* [2,5,1] → sum = 8 ✅
* [2,5,1,10] → sum = 18 ❌

Now, sum > 14.  
Instead of shrinking L repeatedly (which may reduce the current valid subarray too much), just:

* Remove 2 → sum = 16 ❌
* Remove 5 → sum = 11 ✅

This is fine but **costs multiple iterations**.

**🔧 Optimization Technique:**

Only shrink the window **by one** step instead of a full while loop.

🔁 Replace the while with an if.

* If adding the new element makes the sum invalid, just remove the oldest element and continue.
* This avoids O(N) time spent shrinking the window in each iteration.

This change makes the time complexity:

**O(N)** – linear time.

This is possible **only if you're asked for the length**, **not** the subarray itself.

**🧠 Reasoning Behind the Optimization:**

* You already know the **max length** so far.
* If the window sum becomes invalid, shrinking it all the way down to a smaller length (e.g., 2 or 1) is pointless — you’ll have to build it up again to check further.
* Just reduce it **to the previous max valid length** and continue checking longer lengths.

**3. When to Not Use This Optimization:**

If the problem requires:

* **Storing** or **returning** the actual subarray.
* Finding the **subarray with max sum or min sum** satisfying a condition.

Then, you **must shrink fully** using a while loop.

**4. Pattern 3: Counting Subarrays with a Condition**

**Problem:**

Count number of subarrays with sum **equal to K**.

This is tricky to solve using two-pointer because:

* You can't decide when to shrink or expand based on a constant target K.

So we split the problem:

text

Copy code

Count of subarrays with sum ≤ K → X

Count of subarrays with sum ≤ K-1 → Y

Answer = X - Y

This trick works because:

Subarrays with exact sum K = subarrays with sum ≤ K - subarrays with sum ≤ K-1

**5. Pattern 4: Minimum Length / Shortest Window**

These are rare and slightly different:

* You're asked to find the **minimum length** of a subarray satisfying a condition.
* In that case, you might still use two pointers but with a different shrinking strategy.

**✅ Summary of Key Patterns:**

| **Pattern** | **Goal** | **Shrink Logic** | **Can Replace while with if?** |
| --- | --- | --- | --- |
| **1** | Longest subarray with sum ≤ K | Only shrink if necessary | ✅ (if only length is needed) |
| **2** | Return actual subarray | Must shrink to valid subarray | ❌ |
| **3** | Count number of subarrays | Use prefix sum or transformed logic | ❌ (requires more logic) |
| **4** | Minimum/shortest subarray | Shrink aggressively when valid | ❌ |

[**1423. Maximum Points You Can Obtain from Cards**](https://leetcode.com/problems/maximum-points-you-can-obtain-from-cards/)

There are several cards **arranged in a row**, and each card has an associated number of points. The points are given in the integer array cardPoints.

In one step, you can take one card from the beginning or from the end of the row. You have to take exactly k cards.

Your score is the sum of the points of the cards you have taken.

Given the integer array cardPoints and the integer k, return the *maximum score* you can obtain.

**Example 1:**

**Input:** cardPoints = [1,2,3,4,5,6,1], k = 3

**Output:** 12

**Explanation:** After the first step, your score will always be 1. However, choosing the rightmost card first will maximize your total score. The optimal strategy is to take the three cards on the right, giving a final score of 1 + 6 + 5 = 12.

**Example 2:**

**Input:** cardPoints = [2,2,2], k = 2

**Output:** 4

**Explanation:** Regardless of which two cards you take, your score will always be 4.

**Example 3:**

**Input:** cardPoints = [9,7,7,9,7,7,9], k = 7

**Output:** 55

**Explanation:** You have to take all the cards. Your score is the sum of points of all cards.

**Constraints:**

* 1 <= cardPoints.length <= 105
* 1 <= cardPoints[i] <= 104
* 1 <= k <= cardPoints.length

**Code:-** class Solution {

    public int maxScore(int[] cardPoints, int k) {

        int n = cardPoints.length;

        //if array size is 1 and k is also 1 then return first ele.

        if(k == 1 && n == 1) return cardPoints[0];

        //if k > array size ->invalid condition or array is empty return 0

        if(k > n || n == 0) return 0;

        //first take sum of first k consecutive cards which will be our left sliding window

        int sum = 0;

        for(int i = 0; i < k; i++){

            sum += cardPoints[i];

        }

        /\*now assign sum to maxPts as our initial points are equal to sum of first k consecutive ele.\*/

        int maxPts = sum;

        /\*now take for loop for changing size of leftmost and rightmost sliding window to get different set of cards to find all possible combinations\*/

        int l = k - 1;/\*decrease this pointer by 1 after every iteration to shrink leftmost sliding window\*/

        int r = n - 1;//decrease r pointer increase size of rightmost sliding window.

        while(l >= 0){

            sum -= cardPoints[l];//shrink leftmost sliding window to the left

            l--;

            sum += cardPoints[r];//expand rightmost sliding window to the left

            r--;

            maxPts = Math.max(sum, maxPts);

        }

        return maxPts;

    }

}

[**3. Longest Substring Without Repeating Characters**](https://leetcode.com/problems/longest-substring-without-repeating-characters/)

Given a string s, find the length of the **longest** **substring** without duplicate characters.

**Example 1:**

**Input:** s = "abcabcbb"

**Output:** 3

**Explanation:** The answer is "abc", with the length of 3.

**Example 2:**

**Input:** s = "bbbbb"

**Output:** 1

**Explanation:** The answer is "b", with the length of 1.

**Example 3:**

**Input:** s = "pwwkew"

**Output:** 3

**Explanation:** The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

**Constraints:**

* 0 <= s.length <= 5 \* 104
* s consists of English letters, digits, symbols and spaces.

**Code:-**

class Solution {

    public int lengthOfLongestSubstring(String s) {

     int n = s.length();

     /\*as there are 256 characters in english dictionary which have ascii code hence declared array of size 256 to store last index(this is index of element in a string and not in an array) at which that ele in prent in a string to the index of array = ascii code of that character\*/

     int[] lastSeenIndexOfCurrentElementInString = new int[256];

     //initialize every ele with -1 as consider no any ele exist in a string

     for(int i = 0; i < 255; i++){

        lastSeenIndexOfCurrentElementInString[i] = -1;

     }

     int r = 0;/\*to expand a substring if no any duplicate fount in current substring/sliding window\*/

     int l = 0;/\*to shrink a substring if duplicate founfd in a current window imp. note lastSeenIndexOfCurrentElementInString[s.charAt(r)] >= l i.e duplicate shounld be present at index of string which is part of current substring/current sliding window;lastSeenIndexOfCurrentElementInString[s.charAt(r)] => this gives last seen index of current element present before current ele.\*/

     //sliding window can be till last index of given string

     int maxLength = 0;

     while(r < n){

        /\*if current element exist before in a substring i.e. current element's last seen index i.e. ele. present at index in an array = ascii code of current ele. is not = -1\*/

        /\*lastSeenIndexOfCurrentElementInString[s.charAt(r)] => gives last index of current element in a string present before current index r of string\*/

        //if duplicate of current el. is present in a substring

        if(lastSeenIndexOfCurrentElementInString[s.charAt(r)] != -1){

        //if duplicate is only present inside current sliding window i.e betn l and r =>[l,r]

        /\*inner if makes sure that current el is after index l of string i.e intial idx of current sliding window\*/

        if(lastSeenIndexOfCurrentElementInString[s.charAt(r)] >= l){

         /\*then update l and put if after first occurence of duplicate in a string or previously stored lastSeenIndexOfCurrentElementInString\*/

         /\*lastSeenIndexOfCurrentElementInString[s.charAt(r)] => gives index of first occurence of current ele. in a substring\*/

         l = lastSeenIndexOfCurrentElementInString[s.charAt(r)] + 1;

          }

        }

        /\*update last index of current element in a substring as current ele's index in a substring and not index at which it was present before current index\*/

        int length = r - l + 1;//length of current sliding window

        maxLength = Math.max(length,maxLength);

        lastSeenIndexOfCurrentElementInString[s.charAt(r)] = r;

        /\*expand the substring if duplicate not present in a substring,even if present we have removed it using above nested if stmts\*/

        r++;

     }

     return maxLength;

 }

}