**TREE QUESTIONS**

Table of Contents

[Theory 2](#_Toc208212261)

[2. Variable-Size Window 3](#_Toc208212262)

[3. Substring/Anagram Matching 3](#_Toc208212263)

[4. Advanced Sliding Window 3](#_Toc208212264)

[LEVEL 1: **Fixed-Size Window** 4](#_Toc208212265)

[1. Maximum Average Subarray 4](#_Toc208212266)

[2. First negative in every window of size k 4](#_Toc208212267)

[3. Sliding Window Maximum 4](#_Toc208212268)

[4. Count Occurrences of Anagrams 4](#_Toc208212269)

[LEVEL 2: **Variable-Size Window** 5](#_Toc208212270)

[5. \_\_\_\_\_\_\_\_ 5](#_Toc208212271)

[1. Max Consecutive Ones III 5](#_Toc208212272)

[2. Number of Substrings Containing All Three Characters 5](#_Toc208212273)

[3. Longest substring without repeating characters 6](#_Toc208212274)

[4. Longest Repeating Character Replacement 6](#_Toc208212275)

[5. Fruits into baskets 7](#_Toc208212276)

[LEVEL 3: **Advanced Sliding Window** 9](#_Toc208212277)

[1. Binary subarray with sum 9](#_Toc208212278)

[2. Count Subarrays With Score Less Than K 9](#_Toc208212279)

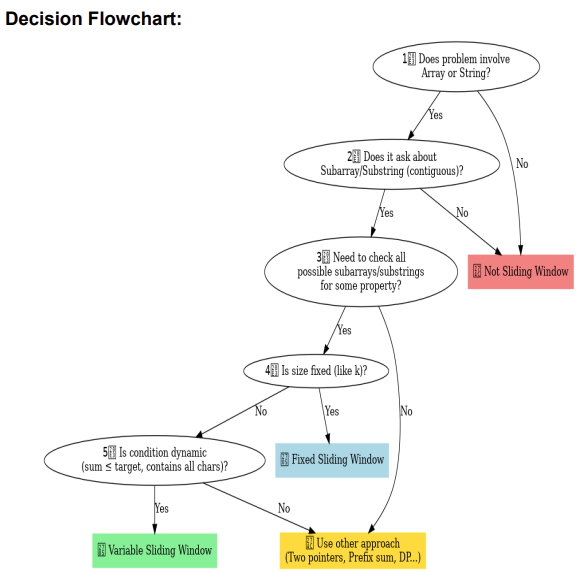
[**SOLUTIONS:** 11](#_Toc208212280)

[**LEVEL 1:** 11](#_Toc208212281)

# Theory

Think of sliding window as looking at a **moving segment (window)** of an array or string.  
Instead of checking every possible subarray/substring separately (which is slow, O(n^2)), we **reuse information from the previous window** when sliding forward.

1. **Fixed Sliding Window:** Keep a window of fixed size k, compute its value (sum/avg/etc.), and as the window slides, **remove the element going out** and **add the element coming in** in O(1) time.
2. **Variable Sliding Window:** Expand the window step by step until the condition is satisfied, then **shrink from the left** while keeping the condition valid, ensuring the smallest/largest valid window is tracked



**When to use Sliding Window:**

* **Sequential input** (array/string) → contiguous part matters.
* **Range-based question** (sum, max, min, avg, count).
* **Contiguity constraint** (subarray/substring/continuous segment).
* **Fixed size window** (e.g., subarray of size k).
* **Variable size window** (expand/shrink until condition holds).

The sliding window technique is ideal for problems involving contiguous sequences, such as substrings or subarrays.

* [Sliding Window Maximum](https://leetcode.com/problems/sliding-window-maximum/)
* [Longest Subarray of 1's After Deleting One Element](https://leetcode.com/problems/longest-subarray-of-1s-after-deleting-one-element/)[LeetCode](https://leetcode.com/problems/sliding-window-maximum/?utm_source=chatgpt.com)

### 2. **Variable-Size Window**

These problems require dynamically adjusting the window size based on certain conditions.

* [Minimum Size Subarray Sum](https://leetcode.com/problems/minimum-size-subarray-sum/)
* [Longest Repeating Character Replacement](https://leetcode.com/problems/longest-repeating-character-replacement/)
* [Permutation in String](https://leetcode.com/problems/permutation-in-string/)
* [Longest Substring with At Most K Distinct Characters](https://leetcode.com/problems/longest-substring-with-at-most-k-distinct-characters/)

### 3. **Substring/Anagram Matching**

These problems focus on finding substrings or anagrams within a larger string.

* [Minimum Window Substring](https://leetcode.com/problems/minimum-window-substring/)
* [Find All Anagrams in a String](https://leetcode.com/problems/find-all-anagrams-in-a-string/)
* [Substring with Concatenation of All Words](https://leetcode.com/problems/substring-with-concatenation-of-all-words/)
* [Check Inclusions](https://leetcode.com/problems/permutation-in-string/)
* [Longest Substring with At Most Two Distinct Characters](https://leetcode.com/problems/longest-substring-with-at-most-two-distinct-characters/)[LeetCode](https://leetcode.com/problems/minimum-window-substring/?utm_source=chatgpt.com)

### 4. **Advanced Sliding Window**

These problems require more complex data structures or logic within the sliding window.

* [Sliding Window Median](https://leetcode.com/problems/sliding-window-median/)
* [Max Sum of Rectangle No Larger Than K](https://leetcode.com/problems/max-sum-of-rectangle-no-larger-than-k/)
* [Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit](https://leetcode.com/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/)
* [Subarrays with K Different Integers](https://leetcode.com/problems/subarrays-with-k-different-integers/)[LeetCode](https://leetcode.com/problems/sliding-window-median/description/?envId=9rt1jt27&envType=list&utm_source=chatgpt.com)

# LEVEL 1: **Fixed-Size Window**

### Maximum Average Subarray

Link: <https://leetcode.com/problems/maximum-average-subarray-i/>

### First negative in every window of size k

Link: <https://www.geeksforgeeks.org/problems/first-negative-integer-in-every-window-of-size-k3345/1>

### Sliding Window Maximum

Link: <https://leetcode.com/problems/sliding-window-maximum/>

### Count Occurrences of Anagrams

Link: <https://www.geeksforgeeks.org/problems/count-occurences-of-anagrams5839/1>

# LEVEL 2: **Variable-Size Window**

### Longest Subarray with Sum K

Link: <https://www.naukri.com/code360/problems/longest-subarray-with-sum-k_6682399>

### Max Consecutive Ones III

Link: <https://leetcode.com/problems/max-consecutive-ones-iii/description/>

### Longest substring without repeating characters

Link: <https://leetcode.com/problems/longest-substring-without-repeating-characters/description/>

### Longest Repeating Character Replacement

Link: <https://leetcode.com/problems/longest-repeating-character-replacement/description/>

### Fruits into baskets

Link: <https://leetcode.com/problems/fruit-into-baskets/description/>

# LEVEL 3: **Advanced Sliding Window**

### Binary subarray with sum

Link:<https://leetcode.com/problems/max-consecutive-ones-iii/description/>

Solution: <https://www.youtube.com/watch?v=j4JDr4-jvo4>

#helper(x) => subarrays(means continuous) with sum <=x

class Solution:

    def numSubarraysWithSum(self, nums: List[int], goal: int) -> int:

        def helper(x):

            if x<0: return 0

            l,curr,count = 0,0,0

            for r in range(len(nums)):

                curr += nums[r]

                while(curr>x):

                    curr-=nums[l]

                    l+=1

                count += (r-l+1)

            return count

        return helper(goal) - helper(goal-1)

### Count Subarrays With Score Less Than K

Link:<https://leetcode.com/problems/count-subarrays-with-score-less-than-k/description/>

class Solution:

    def countSubarrays(self, nums: List[int], k: int) -> int:

        l,curr,count = 0,0,0

        for r in range(len(nums)):

            curr +=nums[r]

            while(curr\*(r-l+1) >=k):

                curr -= nums[l]

                l+=1

            print(nums[l:r+1])

            count += r-l+1

        return count

### Number of Substrings Containing All Three Characters

Link:<https://leetcode.com/problems/number-of-substrings-containing-all-three-characters/description/>

#if \_ \_ \_ this is substring with all 3 chars, then all characters before it will also have all characters,

#so just find left most index of this string, and add that in count

class Solution:

    def numberOfSubstrings(self, s: str) -> int:

        lastseen={}   #keys=a,b,c

        count=0

        n = len(s)

        for i in range(n):

            #put new key or modify exisiting one

            lastseen[s[i]] = i

            if len(lastseen)==3:

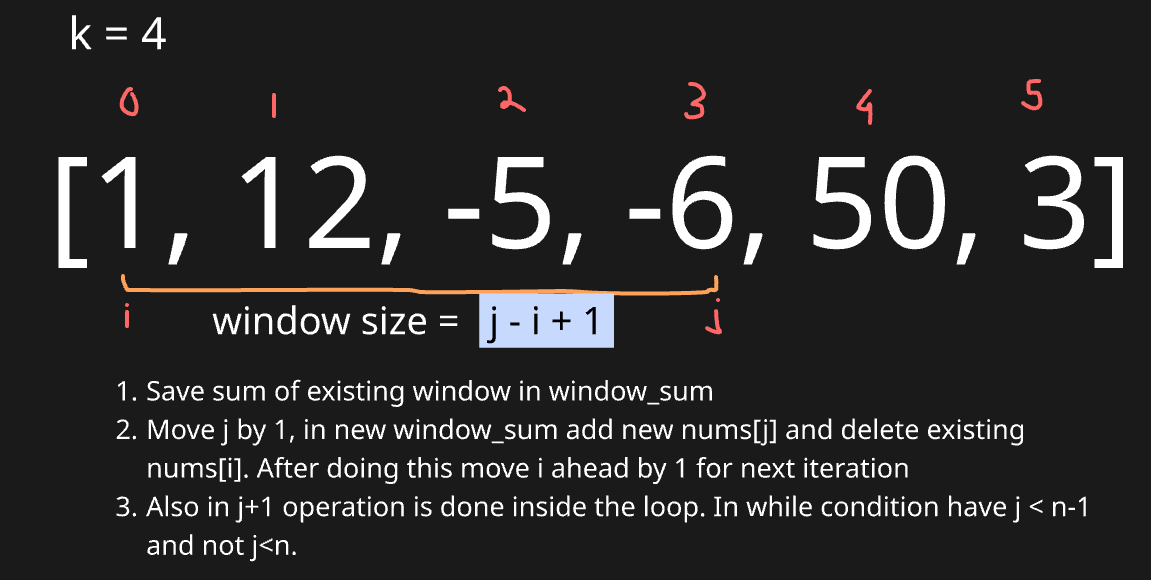
                count = count + (1 + min(lastseen['a'], lastseen['b'], lastseen['c']))

        return count

# **SOLUTIONS:**

## **LEVEL 1:**

1. Maximum Average Subarray



# Equivalent to finding subarray with maximum sum with size K

class Solution:

    def findMaxAverage(self, nums: List[int], k: int) -> float:

        n = len(nums)

        # Step 1: Initialize window [0 : k-1]

        i, j = 0, k - 1   # i = left, j = right

        window\_sum = sum(nums[i:j+1])

        max\_sum = window\_sum

        # Step 2: Slide window using while loop

        while j < n - 1:

            j += 1                      # move right end forward

            window\_sum += nums[j]       # add new element

            window\_sum -= nums[i]       # remove old element

            i += 1                      # shift left end forward for next iteration

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

        # Step 3: Return max average

        return max\_sum / k

1. First Negative in every window of size k

class Solution:

    def firstNegInt(self, arr, k):

        n = len(arr)

        i , j = 0, k-1

        queue = []   #can even use deque to track negative

        ans = []

        # Step 1: Check for initial window

        for x in range(k):

            if arr[x]<0:

                queue.append(arr[x])

        ans.append(queue[0] if queue else 0)

        # Step 2: Check for other windows

        while j<n-1:

            j+=1   #move j forward

            if arr[j]<0:

                queue.append(arr[j])

            #perform work on existing ith element

            #Step 3: Check if queue[0] element is to be removed from window

            if queue and arr[i]==queue[0]:

                queue.pop(0)

            i+=1  #move i forward

            #to check if negative exists, append

            ans.append(queue[0] if queue else 0)

        return ans

**Standard Flow of solutions:**

* Create initial window and get answer from it
* Check for other windows:
  + Increase end index
  + Add new element at end (jth element)
  + Remove the start element (ith element)
  + Try to consolidate the array formed in answer
  + Increase start index

1. Sliding Window Maximum

**Approach**: Use a **deque to track indices of potential maxima** in the current window in decreasing order.  
The **front of the deque** always represents the maximum of the current window.  
Slide the window by adding the new element at the right and removing elements that are smaller or out of the window from the deque.

from collections import deque

class Solution:

    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:

        n = len(nums)

        dq = deque()   # stores indices of useful elements (in decreasing order)

        ans = []

        i, j = 0, k - 1

        # Step 1: Process the first window of size k

        for idx in range(k):

            while dq and nums[dq[-1]] < nums[idx]:

                dq.pop()

            dq.append(idx)

        ans.append(nums[dq[0]])  # maximum of first window

        # Step 2: Slide the window across the array

        while j < n - 1:

            j += 1  # expand window by moving right end

            # Maintain decreasing order in deque

            while dq and nums[dq[-1]] < nums[j]:

                dq.pop()

            dq.append(j)

            # Remove element if it goes out of the window

            if dq[0] == i:

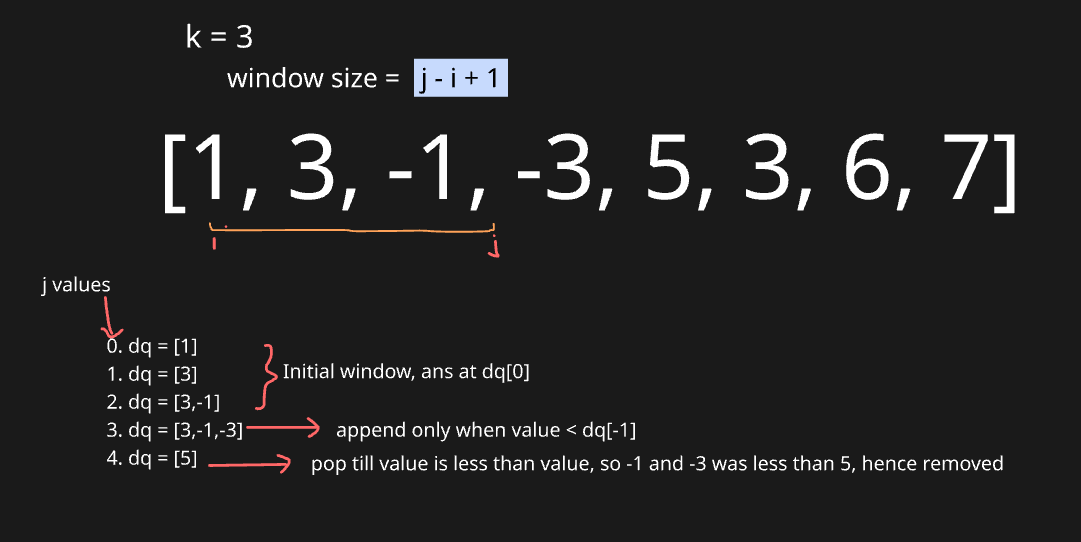
                dq.popleft()

            ans.append(nums[dq[0]])  # max for current window

            i += 1  # shrink window from left

        return ans

* **Initialize first window** – Use deque to store indices in decreasing order of values for the first elements.
* **Record first max** – The front of the deque (dq[0]) holds the maximum of the first window.
* **Slide window** – Move the right end j forward and expand the window by one element.
* **Maintain deque order** – Pop indices from back if nums[j] is greater, ensuring deque stores potential maxima.
* **Remove out-of-window elements & record max** – Pop from front if index leaves window, then append nums[dq[0]] to results.



\*\***Note**\*\* In all above codes, we are writing duplicate code for handling initial window. This can be taken cared in while loop itself with few code modificaitons

from collections import deque

class Solution:

    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:

        n = len(nums)

        dq = deque()   # will store indices, not values

        ans = []

        i, j = 0, 0  # i = window start, j = window end

        while j < n:

            # Step 1: Maintain decreasing order in deque

            while dq and nums[dq[-1]] < nums[j]:

                dq.pop()

            dq.append(j)

            # Step 2: Remove elements that are out of the window

            if dq[0] < i:

                dq.popleft()

            # Step 3: If window size reached k, record the maximum

            if j - i + 1 == k:

                ans.append(nums[dq[0]])  # front of deque = max in window

                i += 1  # slide window start

            j += 1  # always expand window

        return ans

1. Count Occurrences of Anagrams

from collections import defaultdict

d1 = defaultdict(int, {'a': 1, 'b': 2})

d2 = defaultdict(int, {'a': 1, 'b': 2, 'c': 0})

print(d1 == d2)   # ❌ False

Even though c: 0 is basically what d1['c'] would come if accessed, Python doesn’t treat them as equal.

from collections import defaultdict

class Solution:

    def search(self,pat, txt):

        k = len(pat)

        n = len(txt)

        count = 0

        i ,j = 0,k-1

        dd\_pat,dd\_txt = defaultdict(int), defaultdict(int)

        for v in pat:

            dd\_pat[v] += 1

        for v in txt[:j+1]:

            dd\_txt[v] += 1

        if dd\_pat == dd\_txt:

            count+=1

        while j<n-1:

            j+=1

            dd\_txt[txt[j]] += 1   #add new element in dd

            dd\_txt[txt[i]] -= 1    #remove ith element in dd

            flag=True             #compare 2 default dicts

            for v in dd\_pat.keys():

                if dd\_pat[v]!=dd\_txt[v]:

                    flag = False

                    break

            if flag: count+=1

            i+=1

        return count

## **LEVEL 2:**

1. Longest Subarray with sum K

Here in place of giving window size we are given fixed condition: so it is variable length sliding window.

**Approach**: We use a **sliding window** with two pointers i and j. Expand j to grow the window sum, and whenever the sum exceeds k, shrink from i until it’s ≤ k. If the sum equals k, update the maximum length.

\*\***Note**\*\*: It only works when elements are non-negative

Remember: **length of window = j-i+1**

def longestSubarrayWithSumK(arr: [int], k: int) -> int:

    n = len(arr)

    ans = 0

    i = j = 0

    s = 0

    while j < n:

        # expand window by including arr[j]

        s += arr[j]

        # shrink window until sum <= k

        while s > k and i <= j:

            s -= arr[i]

            i += 1

        # check if current window sum equals k

        if s == k:

            ans = max(ans, j - i + 1)

        j += 1

    return ans

1. Max Consecutive Ones |||

**Approach**: We use a **sliding window** with two pointers (left and right).  
Expand right to include elements, counting how many zeros are inside.  
If the window has more than k zeros, shrink it from the left until valid again.  
Track the maximum valid window length throughout the process.

class Solution:

    def longestOnes(self, nums: List[int], k: int) -> int:

        n = len(nums)

        i,j=0,0

        zc=0              #zeros count in window

        ans=0    #stores max length

        while j<n:

            if nums[j]==0:     #always expand window from right

                zc+=1

            while zc>k and i<=j:   #shrink window till zero count < k

                if nums[i]==0:

                    zc-=1

                i+=1

            ans = max(ans,j-i+1) #always check for max length window

            j+=1

        return ans

1. Longest substring without repeating characters

* Use two pointers i and j to maintain a sliding window over the string.
* Track character frequencies within the window using a defaultdict(int).
* If a character’s count exceeds 1, move i forward while decrementing counts to remove duplicates.
* Update ans with the maximum window size j - i + 1 whenever the window is valid.

from collections import defaultdict

class Solution:

    def lengthOfLongestSubstring(self, s: str) -> int:

        n = len(s)

        i,j = 0,0

        dd = defaultdict(int)

        ans = 0

        while j<n:

            val = s[j]

            dd[val] += 1

            while dd[val]>1 and i<=j:

                dd[s[i]]-=1

                i+=1

            ans = max(ans,j-i+1)

            j+=1

        return ans

1. Longest Repeating Character Replacement

**Approach**: A hashmap tracks the frequency of characters within the current window. If the condition (window size - max frequency) > k becomes true, it means more than k replacements are needed, so the window is shrunk from the left by moving i. Throughout this process, the maximum valid window length is updated and returned as the result.

from collections import defaultdict

class Solution:

    def characterReplacement(self, s: str, k: int) -> int:

        n=len(s)

        i,j = 0,0

        dd = defaultdict(int)

        ans = 0

        while j<n:

            dd[s[j]] += 1

#max(count.values()) this will find the maximum frequency of a character in the hash again

            if j - i + 1 - max(dd.values()) > k:

                dd[s[i]] -= 1

                i += 1

            ans = max(ans, j - i + 1)

            j+=1

        return ans

**Dry Run:**

**s = "AABAB", k = 1**  
i=0, j=0, dd={}, ans=0

* j=0 'A' → dd={'A':1}, 1-1=0≤1 → ans=1
* j=1 'A' → dd={'A':2}, 2-2=0≤1 → ans=2
* j=2 'B' → dd={'A':2,'B':1}, 3-2=1≤1 → ans=3
* j=3 'A' → dd={'A':3,'B':1}, 4-3=1≤1 → ans=4
* j=4 'B' → dd={'A':3,'B':2}, 5-3=2>1 → shrink i=1, dd={'A':2,'B':2}, ans=4

**Result: 4** (max window length)

1. Fruit Into Baskets

**Approach**: Two pointers i and j define the window, and a hash map tracks the count of each fruit type inside it. As new fruits are added from the right, if the number of distinct fruit types exceeds two, the left pointer i moves forward while decreasing counts and removing any fruit whose count becomes zero. Throughout this process, the maximum window length is updated and returned as the result.

#Problem is equivalent to: find length of longest subsequence having atmost 2 type of numbers

from collections import defaultdict

class Solution:

    def totalFruit(self, fruits: List[int]) -> int:

        n = len(fruits)

        i,j=0,0

        dd = defaultdict(int)  #{fruit: it's count}

        ans=0

        while(j<n):

            dd[fruits[j]] +=1

            while len(dd.keys())>2:

                dd[fruits[i]]-=1

           #remove element from left whose count becomes 0 first on traversing left to right

                if dd[fruits[i]]==0:

                    del dd[fruits[i]]

                i+=1

            ans = max(ans,j-i+1)

            j+=1

        return ans