Assignment based on Sub-array and sliding window concept

Problem statement 1

Given an array of integers and a window size k, for each window of size k, print the first negative integer in that window. If a window does not contain a negative integer, print 0.
Concept and Sliding Window Logic

1. Use a queue (Deque) to store the indices of negative numbers in the current window.
2. Slide the window from left to right:

o Add the current element's index to the queue if it is negative.
o Remove elements out of the current window from the queue.
o The element at the front of the queue is the first negative number for the current window.
o If the queue is empty, it means no negative number is present in that window → Output 0.

Example:
Input: arr = [12, -1, -7, 8, -15, 30, 16, 28], k = 3
Windows:
[12, -1, -7] → First negative: -1
[-1, -7, 8] → First negative: -1
[-7, 8, -15] → First negative: -7
[8, -15, 30] → First negative: -15
[-15, 30, 16] → First negative: -15
[30, 16, 28] → No negative → 0
Output: [-1, -1, -7, -15, -15, 0]

Problem statement 2

Given an integer array nums and an integer k, return a list of the maximum values in each contiguous subarray (window) of size k.
Approach: Sliding Window Using Deque
To solve this in O(n) time, we use a Deque (double-ended queue) to store indices of useful elements in the current window:

- The front of the deque always holds the index of the maximum for the current window.
- While sliding the window:

o Remove elements from the back if they are less than the current element (they can never be maximum).
o Remove elements from the front if they are outside the current window.

Input: nums = [1, 3, -1, -3, 5, 3, 6, 7], k = 3
Windows and maximums:
[1, 3, -1] → 3
[3, -1, -3] → 3
[-1, -3, 5] → 5
[-3, 5, 3] → 5
[5, 3, 6] → 6
[3, 6, 7] → 7
Output: [3, 3, 5, 5, 6, 7]

Problem statement 3

Count occurrences of anagrams Variable-size
Given a text txt and a pattern pat, count the number of occurrences of anagrams of pat in txt.
Key Concepts
This is a variable-size sliding window problem where:

- We use a window of size equal to the pattern's length.
- We slide the window across the text and for each window:
  - o Check if the substring is an anagram of the pattern.

Optimized Approach Using HashMap

1. Use a Map<Character, Integer> to store character frequency of the pattern.
2. Use a sliding window of size pat.length().
3. For each character in the current window:
   - o Maintain a count of characters matched.
   - o If all characters are matched → we found an anagram.
4. As the window slides, update the frequency map accordingly.

Problem statement 4

Given a string s, find the length of the longest substring without repeating characters.
Sliding Window Approach (Variable-Size)
We use the sliding window technique where:

- We expand the window by moving the right pointer.
- If a duplicate character is found, we shrink the window from the left until the substring has no duplicates.

Tools:
We use a Set<Character> from Java Collections to:

- Store characters in the current window.
- Quickly check for duplicates.

Example
Input: s = "abcabcbb"
Step-by-step:
"abc" → valid (length 3)
"abca" → 'a' repeated → move left
New window: "bca" → valid
Max length: 3
Output: 3

Problem statement 5

Problem: Minimum Size Subarray Sum (Sum ≥ Target)
Given an array of positive integers nums and an integer target, find the minimal length of a contiguous subarray of which the sum is greater than or equal to target. If no such subarray exists, return 0.
Approach: Variable-Size Sliding Window
Key Idea:
Use two pointers start and end to form a sliding window:

- Expand the window by moving end and add nums[end] to the running sum.

- Once the sum is ≥ target, shrink the window by moving start forward to find the minimum length

Input: target = 7, nums = [2,3,1,2,4,3]
Step-by-step:
[2,3,1,2] → sum = 8 → valid → length = 4
Shrink window: [3,1,2] = 6 (too small)
Add next: [3,1,2,4] = 10 → valid → shrink to [4,3] = 7 → valid → length = 2
Output: 2
Java Source Code (Using Collection Framework)
We'll use:

- Basic structures like ArrayList optionally (not required here),
- But mainly focus on using the sliding window pattern.

Problem statement6

Given a string s and an integer k, find the length of the longest substring of s that contains at most k distinct characters.
Approach: Variable-Size Sliding Window + HashMap
Key Idea:

- Use a sliding window with two pointers: left and right.
- Use a HashMap<Character, Integer> to store the frequency of characters currently in the window.
- Expand the window by moving right and adding characters to the map.
- If the window contains more than k distinct characters, shrink the window from the left by moving left and updating the map.
- Track the maximum window size that contains at most k distinct characters.

Example:
Input: s = "eceba", k = 2
Step-by-step:
- Window: "ece" → distinct chars = {e, c} → length 3
- Expand to "eceb" → distinct chars = {e, c, b} → 3 > 2 → shrink from left
- After shrinking: "ceb" (distinct chars = 3 still), shrink again
- Window: "eb" → distinct chars = {e, b} → length 2
- Max length found = 3 ("ece")

Output: 3

Problem statement7

Given a string s and an integer k, you can replace at most k characters in the string to make all characters in the substring the same. Find the length of the longest substring containing the same character you can get by performing this replacement.
Approach: Variable-Size Sliding Window with Frequency Map
Key idea:

- Maintain a sliding window [left, right].
- Track frequencies of characters in the current window using a HashMap<Character, Integer>.
- Keep track of the count of the most frequent character in the window (maxFreq).
- The window is valid if:

(window size) - maxFreq ≤ k

i.e., the number of characters we need to replace to make the entire window the same is ≤ k.
- If invalid, shrink the window from the left.
- Keep track of the maximum window size while moving right.

Example
Input: s = "AABABBA", k = 1

Step-by-step:
Window "AABA":
 - freq: {A=3, B=1}
 - maxFreq = 3
 - window size = 4
 - replacements needed = 4 - 3 = 1 ≤ k → valid

Expand:
Window "AABABB":
 - freq: {A=3, B=3}
 - maxFreq = 3
 - window size = 6
 - replacements needed = 6 - 3 = 3 > k → shrink window

Shrink:
Remove 'A' at left:
Window "ABABB":
 - freq: {A=2, B=3}
 - maxFreq = 3
 - replacements = 5 - 3 = 2 > k → shrink more

Continue shrinking until the window is valid.
Max valid window length found = 4 ("AABA" or "ABBA")

Problem statement8

Maximum Number of Vowels in a Substring of Given Length
Given a string s and an integer k, return the maximum number of vowels in any substring of length k.
Key Concepts:
- Use a sliding window of size k.
- Keep track of the number of vowels in the current window.
- Slide the window by one character at a time and update the vowel count.
- Use Set<Character> from the Java Collection Framework to easily check if a character is a vowel.

Example
Input: s = "abciiidef", k = 3
Output: 3
Explanation: The substring "iii" contains 3 vowels.

Problem statement 9

Problem: Permutation in String
Description: Given two strings s1 and s2, write a function that returns true if s2 contains any permutation of s1.A permutation of s1 is any rearrangement of its letters.
The task is to check if any substring of s2 is a permutation of s1.
Approach (Using Java Collection Framework):
We'll use:
- HashMap<Character, Integer> to store the character frequency of s1 and each window of s2 (of length equal to s1).
- A sliding window to iterate through s2.

At each step:
- Compare the character frequency of the window with that of s1.
- If equal → s2 contains a permutation of s1.

Example with source code
Input: s1 = "abc", s2 = "eidbacooo"
Output: true
Explanation: "bac" is a permutation of "abc" and is in s2.

Example10
Problem: Maximum Points You Can Obtain from Cards
Description: You are given an integer array cardPoints where each cardPoints[i] represents the points you get from that card. You can pick k cards total, but you can only pick from either the beginning or the end of the array — not from the middle. Your task is to return the maximum possible score you can obtain by picking exactly k cards.
Approach (Prefix + Suffix + Sliding Window):
Instead of picking cards from random ends, we realize:
- You pick x cards from the front and k - x from the back (for every x from 0 to k).
- We calculate the sum of these selections, and take the maximum.

This can be optimized using a sliding window:
- Start by summing the first k cards (all from front).
- Then, move the window by removing cards from the front and adding from the back.
- Keep track of the maximum sum.

Example with source code
Input: cardPoints = [1, 2, 3, 4, 5, 6, 1], k = 3
Output: 12
Explanation:
Pick the last card (1) and the first two cards (1 + 2), or pick the last three (6 + 1 + 5), etc.
The optimal combination is: 6 + 5 + 1 = 12

Problem statement 10
Problem: Max Consecutive Ones with at Most One 0 Flipped
Description: Given a binary array (only 0s and 1s), find the maximum number of consecutive 1s you can get by flipping at most one 0 to 1.
Approach: Sliding Window (Variable Size) + Collection Framework
We use a variable-size sliding window:
- We slide a window across the array and track the number of zeros inside it.

• If the window has more than 1 zero, we shrink it from the left until it becomes valid (i.e., contains at most 1 zero).
• Track the maximum window length while maintaining this constraint.
We will use:
• A Queue (from Java Collection Framework) to store the indices of zeros — this helps track where the last zero is to flip and adjust the window quickly.
Input: nums = [1,1,0,1,1,1], Output: 6
Explanation: Flip the 0 at index 2 → [1,1,1,1,1,1]

Problem statement 11

Problem: Max Sum of Subarray with Unique Elements
Description:Given an integer array nums, find the maximum sum of any subarray that contains only unique elements (no duplicates).

Approach: Variable-size Sliding Window + HashSet
• Use two pointers (left, right) to represent a sliding window.
• Use a HashSet to track unique elements in the current window.
• Move right forward and add elements to the window if they are not duplicates.
• If a duplicate is encountered, move left forward, removing elements from the set until the duplicate is removed.
• Track the sum of the current window and update the maximum sum.
Example:
Input: nums = [4, 2, 4, 5, 6]
Output: 17
Explanation:
The subarray [2, 4, 5, 6] has all unique elements and sum = 17 (maximum).

Problem statement 12

Problem: Longest Subarray with Sum ≤ k
Description:Given an integer array nums (can contain positive and negative numbers) and an integer k, find the length of the longest contiguous subarray whose sum is less than or equal to k.

In-depth Explanation:
Key Points:
• The subarray must be contiguous.
• The sum of elements in the subarray ≤ k.
• We want the maximum length of such a subarray.
Challenges:
• If all numbers are non-negative, a sliding window approach can be used directly because sums increase when expanding and decrease when shrinking.
• But if nums contain negative numbers, sums can increase or decrease unpredictably, and the simple sliding window approach fails.

When nums are all non-negative:
We can use a variable-size sliding window:
1. Initialize two pointers: left and right at 0.

2. Expand right pointer, adding nums[right] to the current sum.
3. If sum > k, shrink window by moving left forward and subtracting nums[left].
4. Keep track of max window size where sum ≤ k.

When nums can have negative numbers:
The problem becomes trickier.

- Sliding window doesn't work straightforwardly.
- One approach uses prefix sums + TreeMap (Java Collection Framework) to efficiently find subarrays sums that meet the condition.
- We'll use prefix sums + a balanced tree (TreeMap) to solve this in O (n log n).

Example
nums = [2, -1, 2, 3], k = 3
Prefix sums:

```
index: 0   1   2   3   4
sum:   0   2   1   3   6
```

Goal: For each prefix sum, find earliest prefix sum ≥ (current_sum - k)
Longest subarray sum ≤ 3 is length 3 ([2, -1, 2])

Problem statement 13

The "Binary Subarrays With Sum = K" problem asks you to count the number of contiguous subarrays (of variable sizes) in a binary array (contains only 0s and 1s) such that the sum of elements in each subarray is equal to K.
Problem Statement
Input:

- A binary array nums[] (containing only 0s and 1s)
- An integer K

Output:

- Number of subarrays whose sum equals K

Example
nums = [1,0,1,0,1], K = 2
Output: 4
Explanation:
These 4 subarrays sum to 2:

1. [1,0,1] from index 0 to 2
2. [0,1,0,1] from index 1 to 4
3. [1,0,1] from index 2 to 4
4. [1,1] from index 0 and 2 (non-contiguous — not valid)

Optimal Approach: Prefix Sum + HashMap (Using Java Collections)
Idea:

- We use a prefix sum and a HashMap to count the number of times a particular sum occurs.
- For every element in the array, compute the current sum.
- Then check how many times (currentSum - K) has occurred — this tells us how many previous prefixes can be subtracted to get a subarray with sum K.

Problem statement14

Let's walk through the “Subarrays With Product Less Than K” problem in depth, focusing on:
Problem Statement
You are given an array of positive integers and a positive integer k. You need to return the number of contiguous subarrays where the product of all the elements is strictly less than k.

- Variable-size subarrays
- Sliding window + two pointers
- Java implementation using Collection Framework
- Step-by-step explanation with example

Constraints:

- All array elements are positive integers
- k is a positive integer (k > 0)
- You must find variable-sized contiguous subarrays

Example with source code
nums = [10, 5, 2, 6]
k = 100
Output : 8

Explanation:
The subarrays whose product is less than 100:

- [10]
- [5]
- [2]
- [6]
- [10, 5]
- [5, 2]
- [2, 6]
- [5, 2, 6]

Approach: Sliding Window (Two Pointers)
Key Idea:
Use a sliding window to maintain the product of the current window.
When the product becomes >= k, shrink the window from the left until the product is < k.
Why Sliding Window Works:

- Since all elements are positive, the product will never decrease unless you remove elements.
- You only need to count all subarrays ending at right whose product is less than k.

For each index right, number of valid subarrays: right - left + 1

Problem statement 15

Let's dive into the "Maximum Average Subarray of Size K" problem — focusing on:

- Fixed-size subarrays
- Sliding window technique
- Full Java implementation using the Collection Framework (e.g., List)
- Step-by-step explanation and dry run

Problem Statement
Given:
- An array nums[] of integers (can be positive or negative)
- An integer k

Task:
- Find the maximum average of all contiguous subarrays of size k

Example:
nums = [1, 12, -5, -6, 50, 3]
k = 4
Output: 12.75

Explanation:
The contiguous subarray of size 4 with the maximum average is [12, -5, -6, 50]:
- Sum = 51
- Average = 51 / 4 = 12.75

Algorithm: Sliding Window (Fixed-size)
Key Idea:
- Maintain a running sum of a window of size k
- Slide the window by:
  - Subtracting the element that goes out
  - Adding the element that comes in
- Track the maximum sum, then divide by k at the end

Problem statement 16

Find all substrings of length k with no repeated characters Fixed-size
Given a string s and an integer k, your task is to find all substrings of length k that have no repeated characters. This time, you are allowed to use Java's Collection Framework (HashSet, ArrayList, etc.).
Key Concepts
- Substring of length k: A continuous sequence of k characters.
- No repeated characters: Each character in the substring must be unique.
- Use sliding window of size k to process substrings efficiently.
- Use a HashSet to check for unique characters.

Example
s = "abcabcbb";
k = 3;

Output
abc
bca
cab

"abc", "bca", and "cab" are substrings of length 3 with no repeated characters. "cbc" or "cbb" are invalid due to duplicates.

Approach

1. Use a sliding window of size k to extract substrings from s.
2. For each substring:

o Use a HashSet to check whether it contains all unique characters.

3. Store and print only the substrings that satisfy the condition.

Problem statement 17

Count subarrays with sum = k Variable-size
To count the number of subarrays with sum = k in a variable-sized subarray problem, we can use HashMap from Java's Collection Framework to efficiently track prefix sums.
Concept
This problem is solved using the Prefix Sum + HashMap technique:

- Prefix Sum: The sum of all elements in the array up to a certain index.
- Idea: If the sum of elements from index i to j is k, then:

prefixSum[j]−prefixSum[i−1]=k⇒prefixSum[i−1]=prefixSum[j]−kprefixSum[j] - prefixSum[i-1] = k \Rightarrow prefixSum[i-1] = prefixSum[j] - kprefixSum[j]−prefixSum[i−1]=k⇒prefixSum[i−1]=prefixSum[j]−k
So, for every current prefix sum, we check if (currentSum - k) exists in the HashMap.
Algorithm Steps

1. Initialize currentSum = 0.
2. Use HashMap<Integer, Integer> to store prefix sums and their counts.
3. Traverse the array:

o Add current element to currentSum.
o If currentSum == k, increment count.
o Check if currentSum - k exists in the map → if yes, add its frequency to count.
o Update the map with the current prefix sum.