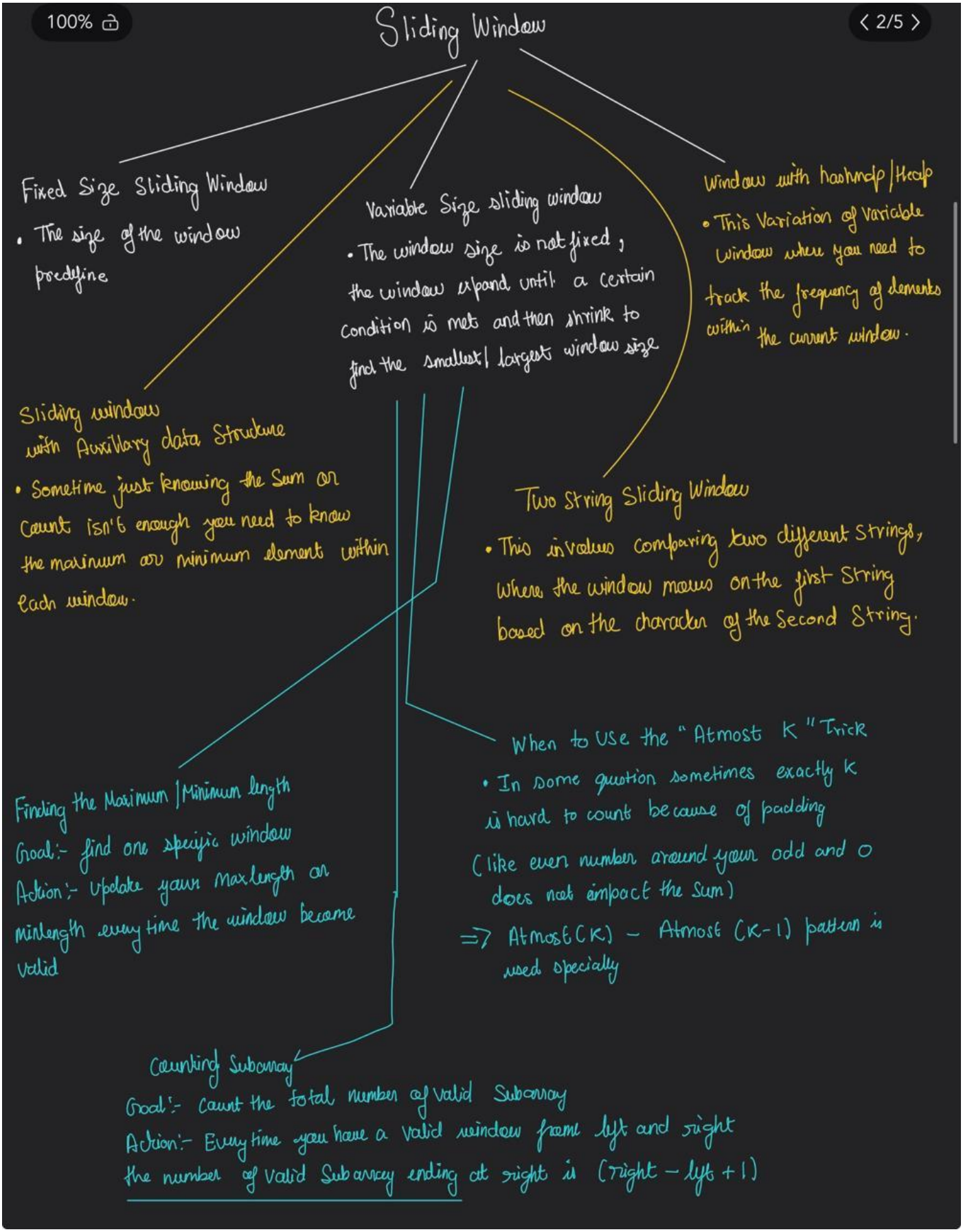
Sliding Window Notes By Aman



1. Fixed-Size Sliding Window

In this pattern, the size of the "window" is predefined (e.g., "find the maximum sum of **k** consecutive elements").

- **How it works:** You maintain a window of size `k`. As you move from left to right, you add the next element and remove the first element of the previous window.
- **Key Indicator:** The problem mentions a specific length or size (e.g., "subarray of size k").
- **Common Problems:**
  - Maximum sum subarray of size k.
  - Find all anagrams in a string.
  - First negative integer in every window of size k.

2. Variable-Size Sliding Window (Flexible Window)

The window size is not fixed. Instead, the window expands until a certain condition is met, and then it shrinks to find the smallest/largest valid window.

- **How it works:** 1. Use two pointers: `start` and `end`. 2. Expand `end` to include elements until the condition is violated. 3. Once violated, shrink `start` until the condition is met again.
- **Key Indicator:** "Find the longest/shortest subarray where the sum/condition is..."
- **Common Problems:**
  - Longest Substring Without Repeating Characters.
  - Smallest Subarray with a Sum greater than X.
  - Longest Subarray with at most K distinct characters.

3. Sliding Window with Hash Map / Frequency Array

This is a variation of the variable window where you need to track the frequency of elements within the current window.

- **How it works:** You use a `Dictionary` or a `HashMap` to store the count of characters/numbers currently in the window. This allows you to check constraints like "at most k unique characters" in $O(1)$ time.
- **Common Problems:**
  - Longest Repeating Character Replacement.
  - Minimum Window Substring (The "Hard" level classic).
  - Subarrays with K Different Integers.

4. Sliding Window with Auxiliary Data Structures (Deque/Heap)

Sometimes, just knowing the sum or count isn't enough; you need to know the **maximum or minimum** element within each window.

- **How it works:** You use a **Monotonic Deque** (Double-ended queue) to keep track of indices of elements in a way that the maximum (or minimum) is always at the front.
- **Common Problems:**
  - Sliding Window Maximum (Maximum of all subarrays of size k).
  - Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit.

5. Two-String Sliding Window

This involves comparing two different strings, where the window moves on the first string based on the characters of the second string.

- **How it works:** Usually involves a frequency map of the "target" string. You expand the window on the "source" string until all characters of the target are covered.
- **Common Problems:**
  - Minimum Window Substring.
  - String Permutation (Check if $S1$ contains a permutation of $S2$).

Pattern Type	Key Characteristic	Typical Data Structure
Fixed Size	Window size k is constant.	Simple variables (Sum/Max).
Variable (Longest)	Find max window satisfying condition.	Two Pointers.
Variable (Shortest)	Find min window satisfying condition.	Two Pointers.
Frequency Based	Counting occurrences within window.	HashMap / Frequency Array.
Monotonic Window	Finding Min/Max in each window.	Deque (Double-ended queue).

The Sliding Window Technique is a problem-solving technique that is used to

1.Running Average: Use a sliding window to efficiently calculate the average of a fixed-size window as new elements arrive in a stream of data.

2.Formulating Adjacent Pairs: Sliding windows are useful when you need to process adjacent pairs of elements in an ordered data structure, allowing you to easily access and operate on neighboring elements.

3.Target Value Identification: When you want to find a specific target value or combination of values in an array, a sliding window can help by adjusting the window size and efficiently searching for the desired value or subarrays that meet specific criteria.

4.Longest/Shortest/Most Optimal Sequence: Sliding windows are handy when you need to find the longest, shortest, or most optimal sequence that satisfies a given condition in a collection. By sliding a window through the collection and tracking relevant information within it, you can identify the desired sequence more efficiently than scanning the entire collection.

The main idea behind the sliding window technique is to convert **two nested loops into a single loop**. Usually, the technique helps us to reduce the time complexity from **O(n^2) or O(n^3) to O(n)**.
This is done by **maintaining a sliding window**, which is a **subarray of the original array** that is of a fixed size. The algorithm then iterates over the original array, updating the sliding window as it goes. This allows the algorithm to keep track of a contiguous sequence of elements in the original array, **without having to iterate over the entire array multiple times.**

Both fixed and variable window sliding window problems can use the techniques of hashing, two pointers, and sliding window optimization.
a.Hashing is a common technique for tracking the elements in a sliding window. This is because a hash table can quickly and efficiently look up the presence of an element in the window.
b.Two pointers is another common technique for tracking the elements in a sliding window. This is because two pointers can easily track the start and end of the window.
c.Sliding window optimization is a technique that combines hashing and two pointers to improve the performance of the sliding window algorithm. This is done by using hashing to quickly look up the presence of an element in the window, and using two pointers to track the start and end of the window.
The choice of technique for solving a sliding window problem depends on the specific problem

and the constraints of the problem. For example, if the sliding window is small, then hashing may be a good choice. However, if the sliding window is large, then two pointers may be a better choice.

Lets discuss How to identify Fixed and variable size Window

1.Fixed Window:

In a fixed window problem, **we have a predefined window size that remains constant** throughout the problem-solving process.
The template for solving a fixed window problem involves maintaining two pointers, **low and high, that represent the indices of the current window.**
The process involves iterating over the array or sequence, adjusting the window as necessary, and performing computations or operations on the elements within the window.
Here's the template

```
fixed_window()
{
  int low = 0, high = 0, windowsize = k;
  while (i < sizeofarray)
  {
    // Step 1: Create a window that is one element smaller than the desired window size
    if (high - low + 1 < windowsize)
    {
      // Generate the window by increasing the high index
      high++;
    }
    // Step 2: Process the window
    else
    {
      // Window size is now equal to the desired window size
      // Step 2a: Calculate the answer based on the elements in the window
      // Step 2b: Remove the oldest element (at low index) from the window for the next window

      // Proceed to the next window by incrementing the low and high indices
    }
  }
```

}

Example on above Format

Q->Given an array arr[] and an integer K, the task is to calculate the sum of all subarrays of size K.

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
   int n, k;
   cin >> n >> k;

   // Input array
   vector<int> v(n, 0);
   for (int i = 0; i < n; i++)
      cin >> v[i];

   int i = 0, j = 0; // Window indices
   int sum = 0; // Current window sum

   while (j < n)
   {
      if (j - i + 1 < k)
      {
         // Expand the window by adding element at index j to the sum
         sum += v[j];
         j++;
      }
      else
      {
```

```
        // Window size is now equal to the desired window size
        // Calculate the answer for the window
        sum += v[j];
        cout << sum << endl;

        // Move the window by incrementing indices i and j
        sum -= v[i];
        i++, j++;
      }
    }
}
```

2 variable window

In a variable window problem,**the window size is not fixed and can change dynamically based on certain conditions or criteria**. The template for solving a variable window problem involves maintaining two pointers, start and end, which represent the indices of the current window.

Initialize the window indices: Start by initializing the start and end pointers to the first element of the sequence or array.

Expand the window: Check a condition to determine whether to expand the window. If the condition is satisfied, increment the end pointer to expand the window size.

Process the window: Once the window size meets the desired criteria or condition, perform the required computations or operations on the elements within the window.

Adjust the window size: If the window size exceeds the desired criteria, adjust the window by moving the start pointer. Iterate or loop until the window size matches the desired criteria, and update the window accordingly.

```
variable_window()
{
  int start = 0, end = 0;
  while (end < n)
  {
    // Perform calculations or operations within the window

    /* Case 1: Expand the window
```

```
    If the window size is less than the desired value (k), increase the end index
    */
    if (end - start + 1 < k)
    {
        end++;
    }

    /* Case 2: Window of desired size
    If the window size is equal to the desired value (k), process the window and calculate the answer
    */
    else if (end - start + 1 == k)
    {
        // Perform the required calculations or operations to obtain the answer
        // Store the answer in a variable (ans)

        end++;
    }

    /* Case 3: Reduce the window size
    If the window size is greater than the desired value (k), adjust the window by moving the start index
    */
    else if (end - start + 1 > k)
    {
        while (end - start + 1 > k)
        {
            // Remove calculations or operations involving the element at the start index

            start++;
        }
```

```
        // Check if the window size becomes equal to the desired value (k) after adjustment
        if (end - start + 1 == k)
        {
           // Perform calculations or operations and store the answer if necessary
        }

        end++;
      }
   }

   // Return the final answer (ans)
}
```

Eg on Above format

1. [longest-substring-without-repeating-characters]

```
class Solution {
public:
   int lengthOfLongestSubstring(string s) {
      int i = 0, j = 0, ans = 0;
      map<char, int> m; // Map to track characters in the current window

      while (j < s.size()) {
         m[s[j]]++; // Add current character to the map
         int windowSize = j - i + 1; // Calculate the current window size

         if (m.size() == windowSize) {
            ans = max(ans, windowSize); // Update the maximum length of the substring
            j++; // Expand the window by moving the end pointer
         }
         else {
```

```
        while (m.size() < windowSize) {
            m[s[i]]--; // Remove characters from the start of the window
            if (m[s[i]] == 0)
                m.erase(s[i]);
            i++; // Move the start pointer to adjust the window
            windowSize = j - i + 1; // Update the window size
        }

        if (m.size() == windowSize) {
            ans = max(ans, windowSize); // Update the maximum length of the substring
        }

        j++; // Expand the window by moving the end pointer
      }
    }

    return ans; // Return the length of the longest substring
  }
};
```

1. Variable-Size: Longest Window (Expansion)

In these problems, you expand the right pointer as much as possible and shrink the left pointer only when the condition is violated to find the **maximum** length.

- **Longest Substring Without Repeating Characters:** Uses a Frequency Map/HashSet to track unique characters.
- **Max Consecutive Ones III:** You expand your window and "allow" up to k zeros. Once you exceed k, you shrink.
- **Fruit Into Baskets:** This is essentially "Longest Subarray with at most 2 distinct elements."
- **Longest Substring with At Most K Distinct Characters:** A direct generalization of "Fruit Into Baskets."
- **Longest Repeating Character Replacement:** You track the frequency of the most frequent character in the window and ensure `(window_size - max_freq) <= k`.

2. Variable-Size: Counting Subarrays (The "Exact" Pattern)

These are tricky. When asked for "Exactly K," you often solve it by calculating `(At most K) - (At most K-1)`.

- **Binary Subarray with Sum:** Finding subarrays that sum to k.
- **Count Number of Nice Subarrays:** Finding subarrays with exactly k odd numbers (same logic as Binary Subarray Sum).
- **Subarrays with K Different Integers:** Another classic "At most K" minus "At most K-1" problem.
- **Number of Substrings Containing All Three Characters:** You count how many valid windows end at the current `right` pointer.

3. Variable-Size: Shortest/Minimum Window

In these, you expand until the condition is **satisfied**, then shrink as much as possible to find the **minimum**.

- **Minimum Window Substring:** You expand until all target characters are found, then shrink to find the smallest possible substring.
- **Minimum Window Subsequence:** Similar to the above, but the characters must appear in a specific order (often involves a sliding window followed by a "backwards" pass to optimize).

4. The "Circular" or "Fixed-Ends" Pattern

This is a slight variation where the "window" isn't necessarily in the middle, but rather what remains after you remove a middle section.

- **Maximum Points You Can Obtain from Cards:** Since you can only take cards from the ends, this is equivalent to finding a **fixed-size sliding window** of size `(Total_Cards - K)` that has the **minimum sum** in the middle of the array.

Question	Pattern	Key Detail
Max Points from Cards	Fixed Size (Inverse)	Find min-sum window of size $(N - K)$.
Longest Substring (No Repeat)	Variable (Longest)	Use Map for last seen index.
Max Consecutive Ones III	Variable (Longest)	Flip at most K zeros.
Fruit Into Baskets	Variable (Longest)	Max 2 distinct characters.
Repeating Char Replacement	Variable (Longest)	Track `maxFreq` in window.
Substrings with 3 Characters	Variable (Counting)	Count valid windows ending at `right`.
Binary Subarray Sum	Variable (Counting)	Use `AtMost(K) - AtMost(K-1)`.
Nice Subarrays	Variable (Counting)	Treat Odd as 1, Even as 0.
Minimum Window Substring	Variable (Shortest)	Shrink until condition is barely met.

Questions On Fixed window size

1. Substrings-of-size-three-with-distinct-characters
2. Substring-with-concatenation-of-all-words
3. Maximum-number-of-vowels-in-a-substring-of-given-length
4. Maximum-number-of-occurrences-of-a-substrin
5. Maximum Average Subarray I
6. Maximum Points You Can Obtain from Cards
7. Find-all-anagrams-in-a-string
8. K Radius Subarray Averages
9. Number of Sub-arrays of Size K and Average Greater than or Equal to Threshold

Questions On variable window size

1. Longest Substring Without Repeating Characters
2. Longest Repeating Character Replacement
3. Sliding Window Maximum
4. Minimum Window Substring
5. Minimum Size Subarray Sum
6. Minimum Consecutive Cards to Pick Up
7. Maximum Erasure Value
8. Fruit Into Baskets
9. Count Number of Nice Subarrays
10. Arithmetic Slices
11. Subarrays-with-k-different-integers