Day 57/180 Sliding window problem on String

.

1: Longest substring without repeating characters:

The solution and code both are explained in the video.

2:Smallest distinct window :

The solution and code both are explained in the video.

3:**Smallest window containing 0, 1 and 2**

Logic:

- Creating a sliding window and keeping track of the frequency of the elements.
- using a frequency tracker array which will track the count of the 0, 1 and 2 between the pointers.
- Will decrease the size of the window between the pointers till the requirement that is the count of 0, 1 and 2 is greater than.
- Update the ans with the current window if it is valid.
- Time complexity: **O(size of string)**. We are traversing the string in a while loop till the end of it.
- Space complexity: **O(1).** we have used an array of constant size 3.

```
int smallestSubstring(string S) {
    // Get the length of the input string
    int n = S.size();

    // Initialize the answer variable to a value greater than
the maximum possible length
    int ans = n+1;

    // Initialize two pointers i and j for traversing the string
    int i = 0;
    int j = 0;

    // Create an array to store the count of each character
(assuming characters are '0', '1', and '2')
    int count[3] = {0,0,0};

    // Loop through the string
    while(i < n)
    {
        // Increment the count of the current character at
position i
        count[S[i] - '0']++;
        i++;

        // Check if there are duplicates of any character in the
current substring
        while(j < i && count[S[j]-'0'] > 1)
        {
            // If there are duplicates, decrement the count of
the character at position j and move the pointer j
            count[S[j]-'0']--;
            j++;
        }
        // Check if the current substring contains at least one occurrence of each character ('0', '1', and '2')
        if(count[0] > 0 && count[1] > 0 && count[2] > 0)
        {
            // Update the answer with the minimum length of the substring containing all characters
            ans = min(ans, i - j);
        }
    }

    // If the answer is still greater than the maximum possible length, no valid substring was found
    if(ans == n+1)
    {
        return -1;
    }
    else
    {
        // Return the minimum length of substring containing all characters
        return ans;
    }
}
```

4: Longest K unique characters substring

Logic :

1. Initialization:
   - Initialize two pointers `i` and `j` to the start of the string.

- Initialize variables `ans` to store the length of the longest substring, `size` to keep track of the number of distinct characters, and `mp` as a vector to store the count of each character.

2. <u>Sliding Window:</u>
   - Use a while loop to iterate through the string until either `i` or `j` reaches the end of the string.
   - If the current size of distinct characters (`size`) exceeds k, move the window forward by incrementing `i`.
   - If the size is less than or equal to k:
      - If the count of the character at index `j` is 0, it means this character is not in the current substring. Increment `size` to account for the new distinct character.
      - Increment the count of the character at index `j` in the vector `mp`.
      - Move the window forward by incrementing `j`.
      - If the size becomes equal to k, update `ans` with the maximum length of the current substring (`j - i`).

3. <u>Update Count:</u>
   - If the size exceeds k, decrement the count of the character at index `i` in the vector `mp`.
   - If the count becomes 0, decrement `size` since the character at index `i` is no longer part of the substring.
   - Move the window forward by incrementing `i`.

4. <u>Result:</u>
   - After the loop, check if `ans` is still 0. If it is, no substring with k distinct characters was found, so return -1.
   - Otherwise, return the value of `ans`, which represents the length of the longest substring with at most k distinct characters.

- Time complexity: **O(size of string)**. We are traversing the string in a while loop till the end of it.
- Space complexity: **O(1).** we have used an array of constant size 26.

```
int longestKSubstr(string s, int k) {
    // Initialize pointers, answer variable, and character count
    int i = 0, j = 0, ans = 0;
    int size = 0;
    vector<int> mp(26, 0); // Array to store the count of each
character

    int n = s.length(); // Length of the input string

    // Sliding window approach
    while (i < n && j < n) {
        // If the number of distinct characters exceeds k
        if (size > k) {
            mp[s[i] - 'a']--; // Decrease the count of the character
at the beginning of the window
            if (mp[s[i] - 'a'] == 0) {
                size--; // If count becomes 0, reduce the size of
distinct characters
            }
            i++; // Move the window forward
        } else {
            // If the number of distinct characters is less than or
equal to k
            if (mp[s[j] - 'a'] == 0) {
                size++; // Increase the size of distinct characters
            }
            mp[s[j] - 'a']++; // Increase the count of the character
at the end of the window
            j++; // Move the window forward

            // If the size of distinct characters becomes equal to
k, update the answer
            if (size == k) {
                ans = max(ans, j - i);
            }
        }
    }

    // If no substring with k distinct characters is found, return
-1
    if (ans == 0) {
        return -1;
    } else {
        return ans; // Return the length of the longest substring
with k distinct characters
    }
}
```