Sliding Window and Two Pointers

Introduction

Sliding window:

The Sliding window technique is used to find subarrays in an array that satisfy specific criteria. It is a subset of dynamic programming.

The technique can be applied to a problem where we have to find the maximum or minimum value for a function that calculates the answer repeatedly for a set of values from the array.

The major advantage of this technique is that it reduces the time required to complete a task(related to a fixed subarray). Problems that would usually take **$O(N^3)$** or **$O(N^2)$** time to solve with brute force can be solved in **O(N)** time using this approach because it avoids repetitive calculations, resulting in improved runtime efficiency.

Two pointers :

The name two pointers do justice in this case, as it is exactly as it sounds. It's the use of two different pointers (usually to keep track of array or string indices) to solve a problem involving said indices with the benefit of saving time and space. It is really an easy and effective technique that is typically used for searching pairs in a sorted array.

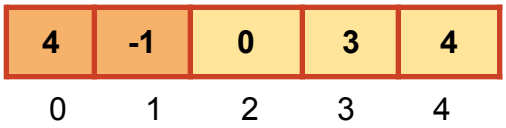
Max Sum of k consecutive elements.

Problem Statement:

You are given an array of length **N** and we have to find out the sum of **K** consecutive elements from the whole array.

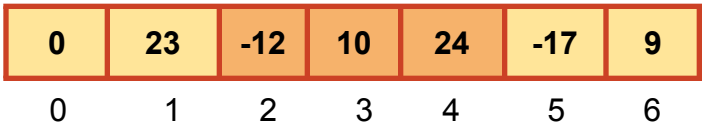
Example:

1. ***ARR*[*5*] = {*4, -1, 0, 3, -4*}**
   ***K* = 2**



Maximum sum = ***3*** (Obtained by adding values on index ***0*** and ***1***).

2. ***ARR*[*7*] = { *0, 23, -12, 10, 24, -17, 9* }**
   K* = *3



Maximum sum = ***22*** (Obtained by adding values on indexes ***2,3*** and ***4***).

Algorithm:

We assume the array to be a window of length **N** and a block of the array as the windowpane of a fixed length **K**.

The pane slides over the window, performing operations on the sub-arrays and subsequently storing the required result in a variable.

- The first **K** elements are added, and their sum is stored in the variable **curSum**, which denotes the sum of the current elements in the block. This is the first sum and is considered to be maximum initially. So, it is stored in the variable **maxSum**.

- Since the size of the windowpane is **K**, it slides one place to the right, and the **curSum** is updated, removing the first element of the previous block and including

the last element of the current block.

- If the current sum is larger than the maximum, the **maxSum** is updated, and the above step is repeated till it reaches the last block of the array.

Code:

```
int slidingWindow(vector<int> &arr, int n, int k)
{
    if (n < k)
    {
        return -1;
    }
    int maxSum = 0;
    int curSum = 0;
    for (int i = 0; i < k; i++)
    {
        curSum += arr[i];
    }

    maxSum = curSum;
    for (int i = k; i < n; i++)
    {
        curSum += arr[i] - arr[i - k];
        maxSum = max(maxSum, curSum);
    }
    return maxSum;
}
```

Time-complexity: O(N) where N is the size of the array.
Since this approach uses a single for loop, which iterates over the array from 1 to N, the time complexity is given by O(N).

Space complexity: O(1)
As we are using extra space only in the form of variables. Thus, the space complexity is constant and given by **O(1).**

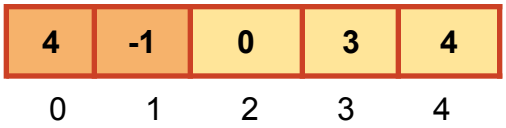
2 Sum in Sorted Array

Problem Statement:

We are given a sorted array in non-decreasing order; we need to return **2** indices values from the array whose value adds up to the given target value. If a pair exists in the array, whose sum is equal to the **target value**, then return those pairs otherwise print **-1**.

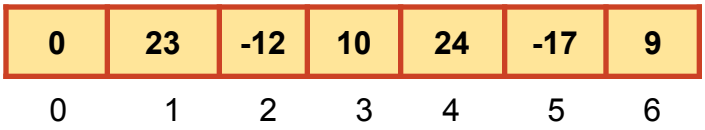
Example:

1. **ARR[5] = {4, -1, 0, 3, -4}**
   target_value = 3



Target_value exists (Obtained by adding values on index ***0*** and ***1***). So, the answer is **0 1**

2. **ARR[7] = { 0, 23, -12, 10, 24, -17, 9 }**
   K = 3



Target_value doesn't exist therefore the answer is -1.

Algorithm:

We can solve this problem using the 2-pointers approach as the array is in sorted order. Let's consider the array **arr** of size **N.**

- First of all, we take two pointers **left** and **right**

- Initially, the left pointer is pointing to the starting of the array **(left=0)** and the right pointer is pointing to the end of the array **(right=N-1).**
- Then, we run a while loop until the **left** is less than the **right**.
- In each execution, we check whether the sum of elements at index **left** and **right (arr[left] + arr[right])** is less than, equal to, or more than our **target value**.
- If the sum of elements is equal to the **target value** we simply return both indices **(left and right).**
- If the sum of elements is less than the target value we simply increment the **left** pointer **(left++)** and if the sum is greater than the target value we decrement the **right** pointer (**right--)**
- If our loop ends without returning any value it means there is no such pair whose sum is equal to our **target value**. In that case, we simply return **-1**.

Code:

```
vector<int>pairSum(int arr[], int N,
                           int target_value)
{
    int left, right;
    vector<int>ans;

    left = 0;
    right = N - 1;
    bool flag=0;
    while (l < r) {
        if (arr[l] + arr[r] == target_value){
           ans.push_back(left);
           ans.push_back(right);
           flag=1;
           break;
        }
        else if (arr[l] + arr[r] < target_value)l++;
        else r--;
    }
    if(!flag)ans.push_back(-1);
    return ans;
}
```

Time-complexity: O(N) where N is the size of the array.
Since this approach uses a single while loop, using two pointers left and right in the worst case the loop will run N times therefore the complexity is given by O(N).

Space complexity: O(1)
We are using extra space as variables and a vector whose maximum size can be 2 therefore the space complexity is O(1).

Longest substring with at most 2 distinct characters.

Problem Statement:
You are given a string as an input, and you have to find the longest substring in the given input such that the substring consists of at most 2 distinct characters.

Algorithm:

- To solve the above problem, we can utilize a 2 pointers approach.
- Let's imagine we're looking for a substring between the indexes **curr_start** and **curr_ end**, which both correspond to index 0 at the start.
- Continue to add one character to the substring at a time by moving the **curr_end** pointer to the right, ensuring that the condition of the substring having at most two unique characters at a time is valid by adding the new character, and if the condition is not valid at any point, start moving the **curr_start** pointer to the right until the condition is valid again.
- Do **curr_end** - **curr_start** at each step to maintain track of the length of the maximum substring.
- To check whether the substring has only **2** unique characters, maintain an array of size **26** (26 alphabets) and an array filled with the count of characters in the string to the respective index. 0^{th} index for **'a'** and 25^{th} index for **'z'**.

Code:

```
void longestSubstring(string s){
    int n=s.size();
```

```
    vector<int>f(26);
    int start=0,total=0,maxlen=0;
    for(int end=0;end<n;end++){
        if(f[s[end]-'a']==0){
            f[s[end]-'a']++;
            total++;
        }
        else f[s[end]-'a']++;
        if(total<=2) maxlen=max(maxlen,end-start+1);
        else{
            while(start<n && total>2){
                f[s[start]-'a']--;
                if(f[s[start]-'a']==0) total--;
                start++;
            }
        }
    }
    cout<<maxlen<<"\n";
}
```

Time Complexity: O (N), where N is the number of characters.

As we are traveling the whole array only once hence the time complexity is linear **O(N).**

Space Complexity: **O (1)**
We are using extra space as variables and a vector whose maximum size can be 26 (for frequency) therefore the space complexity is O(1).