Sliding Window

The sliding window technique involves fixing a window size on an array or string and moving this window through the data structure iteratively. It's particularly useful for solving problems that involve finding subarrays, substrings, or specific patterns within a sequence.

How to use Sliding Window Technique?

The general use of the Sliding window technique can be demonstrated as follows:

Find the size of the window required
Compute the result for 1st window, i.e. from the start of the data structure
Then use a loop to slide the window by 1, and keep computing the result window by window.

Utility of Sliding Window Technique:

- Subarray:
Efficiently finds contiguous sections within an array meeting specific criteria (sum, unique elements, etc.) by sliding a window through the array.

- Substring:
Effectively identifies consecutive character sequences in strings that fulfill defined conditions (unique characters, specific patterns) using a sliding window approach.

- Largest/Smallest Sum:
Quickly determines the subarray with the largest or smallest sum by intelligently adjusting the window to maximize or minimize the sum.

Overall Benefit:
The sliding window technique streamlines the process of identifying desired subarrays, substrings, or sum-related patterns within data structures, enhancing efficiency and optimization in problem-solving scenarios.

Q. Given an array of integers Arr of size N and a number K. Return the maximum sum of a subarray of size K.(A subarray is a contiguous part of any given array.)
Input:
N = 4, K = 2
Arr = [100, 200, 300, 400]
Output:
700
Explanation:
Arr3 + Arr4 =700,

which is maximum.

Code:

```
public long maximumSumSubarray(int K, List<Integer> Arr, int N) {
    long sum = 0;
    int i = 0;
    while (i < K) {
        sum += Arr.get(i++);
    }
    long ans = sum;
    while (i < N) {
        sum -= Arr.get(i - K);
        sum += Arr.get(i);
        ans = Math.max(ans, sum);
        i++;
    }
    return ans;
}
```

Time: O(N), to traverse the array
Space: O(1), only constant space being used

Explanation: take a window of size k first take the sum of first k elements the keep it as ans then keep moving forward add the new element remove the first element and ans will be max of this sum or ans thus at the end ans will be storing the max sum subarray of size k

Q. 1052. Grumpy Bookstore Owner
There is a bookstore owner that has a store open for n minutes. Every minute, some number of customers enter the store. You are given an integer array customers of length n where customers[i] is the number of the customer that enters the store at the start of the ith minute and all those customers leave after the end of that minute.

For some minutes, the bookstore owner is grumpy. You are given a binary array grumpy where grumpy[i] is 1 if the bookstore owner is grumpy during the ith minute, and is 0 otherwise.

When the bookstore owner is grumpy, the customers of that minute are not satisfied, otherwise, they are satisfied.

The bookstore owner knows a secret technique to keep themselves not grumpy for minutes consecutive minutes, but can only use it once.

Return the maximum number of customers that can be satisfied throughout the day.
Example 1:

Input: customers = [1,0,1,2,1,1,7,5], grumpy = [0,1,0,1,0,1,0,1], minutes = 3
Output: 16
Explanation: The bookstore owner keeps themselves not grumpy for the last 3 minutes.
The maximum number of customers that can be satisfied = 1 + 1 + 1 + 1 + 7 + 5 = 16.
Example 2:

Input: customers = [1], grumpy = [0], minutes = 1
Output: 1

Code:

```
public int maxSatisfied(int[] cs, int[] grumpy, int X) {
    int satisfied = 0, maxAddSatisfied = 0, addSatisfied = 0;

    for (int i = 0; i < cs.length; ++i) {
        satisfied += (grumpy[i] == 0) ? cs[i] : 0;
        addSatisfied += (grumpy[i] == 1) ? cs[i] : 0;

        if (i >= X) {
            addSatisfied -= (grumpy[i - X] == 1) ? cs[i - X] : 0;
        }

        maxAddSatisfied = Math.max(maxAddSatisfied, addSatisfied);
    }

    return satisfied + maxAddSatisfied;
}
```

Time: O(n), to traverse the vector
Space: O(1), only constant space being used

Explanation: When the owner is not grumpy, we count all customers as satisfied.

We then use the sliding window to count additionally satisfied customers (add_satisfied) if the owner start 'behaving' at minute i. We track the maximum additional satisfied customers in m_add_satisfied.

Finally, return satisfied + m_add_satisfied as the result.

Q. Given an array and a positive integer k, find the first negative integer for each window(contiguous subarray) of size k. If a window does not contain a negative integer, then print 0 for that window.
Input : arr[] = {-8, 2, 3, -6, 10}, k = 2
Output : -8 0 -6 -6
First negative integer for each window of size k
{-8, 2} = -8
{2, 3} = 0 (does not contain a negative integer)
{3, -6} = -6
{-6, 10} = -6
Input : arr[] = {12, -1, -7, 8, -15, 30, 16, 28} , k = 3
Output : -1 -1 -7 -15 -15 0

Code:

```
void printFirstNegativeInteger(int arr[], int k, int n) {
    int firstNegativeIndex = 0;
    int firstNegativeElement;

    for (int i = k - 1; i < n; i++) {

        while ((firstNegativeIndex < i)
                && (firstNegativeIndex <= i - k
                    || arr[firstNegativeIndex] >= 0)) {
            firstNegativeIndex++;
        }

        if (arr[firstNegativeIndex] < 0) {
            firstNegativeElement = arr[firstNegativeIndex];
        } else {
            firstNegativeElement = 0;
        }
        System.out.print(firstNegativeElement + " ");
    }
}
```

Time Complexity: O(n) , traversing the array

Auxiliary Space: O(1), constant space being used only

Explanation: The idea is to have a variable firstNegativeIndex to keep track of the first negative element in the k sized window. At every iteration, we skip the elements which no longer fall

under the current k size window (firstNegativeIndex <= i – k) as well as the non-negative elements(zero or positive).

Q. (Leetcode 209) Minimum Size Subarray Sum
Given an array of positive integers nums and a positive integer target, return the minimal length of a
subarray
whose sum is greater than or equal to the target. If there is no such subarray, return 0 instead.

Example 1:

Input: target = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: The subarray [4,3] has the minimal length under the problem constraint.
Example 2:

Input: target = 4, nums = [1,4,4]
Output: 1
Example 3:

Input: target = 11, nums = [1,1,1,1,1,1,1,1]
Output: 0

Code:

```
class Solution {
    public int minSubArrayLen(int target, int[] nums) {
        int l = 0;
        int ans = Integer.MAX_VALUE;
        int total = 0;
        for (int r = 0; r < nums.length; r++) {
            total += nums[r];
            while (total >= target) {
                ans = Math.min(ans, r - l + 1);
                total -= nums[l];
                l++;
            }
        }
        return (ans == Integer.MAX_VALUE) ? 0 : ans;
    }
}
```

Time complexity: O(n),The reason that the Time Complexity is not affected by this internal while loop is because we are never allowing he while loop to go past our right pointer and there are also times where it will never run. The worst case would be that the left pointer reads every index that the right pointer does. This would give us a Time Complexity of O(2n) which reduces to O(n).
Space complexity: O(1), only const space beign used

Explanation:
When approaching a sliding window problem there are 3 things that may be needed:
a left pointer
a right pointer
a way to hold the data of our current window
I have a left and right pointer and I use the variable "total" to hold the data of what is inside of my sliding window.

I have loop to increment my right pointer along the vector. However, I know that if the total is >= the target it is not possible for me to increment the right pointer and find another valid subarray since it would be longer than the current subarray and we are trying to find the minimum length subarray.

This is when I switch to my next objective: Minimize my current subarray. This is why I have the internal while loop that increments the left pointer and updates the "ans" variable which is the answer to the problem.

Q. (Leetcode 1493) Longest Subarray of 1's After Deleting One Element
Given a binary array nums, you should delete one element from it.

Return the size of the longest non-empty subarray containing only 1's in the resulting array.
Return 0 if there is no such subarray.

Example 1:

Input: nums = [1,1,0,1]
Output: 3
Explanation: After deleting the number in position 2, [1,1,1] contains 3 numbers with value of 1's.
Example 2:

Input: nums = [0,1,1,1,0,1,1,0,1]
Output: 5
Explanation: After deleting the number in position 4, [0,1,1,1,1,1,0,1] longest subarray with value of 1's is [1,1,1,1,1].

Code:

```
class Solution {
    public int longestSubarray(int[] nums) {
        int n = nums.length;

        int left = 0;
        int zeros = 0;
        int ans = 0;

        for (int right = 0; right < n; right++) {
            if (nums[right] == 0) {
                zeros++;
            }
            while (zeros > 1) {
                if (nums[left] == 0) {
                    zeros--;
                }
                left++;
            }
            ans = Math.max(ans, right - left + 1 - zeros);
        }
        return (ans == n) ? ans - 1 : ans;
    }
}
```

Time complexity: O(N) Each element in the array will be iterated over twice at max. Each element will be iterated over for the first time in the for loop; then, it might be possible to re-iterate while shrinking the window in the while loop. No element can be iterated more than twice.
Space complexity: O(1)

Explanation:
This code finds the longest subarray with only 1s after removing at most one element (0 or 1). It uses a sliding window approach with left and right pointers. As it moves through the array, the right pointer tracks encountered zeros. The left pointer adjusts the window to allow at most one zero. It calculates subarray lengths and updates the longest length found so far. In the end, if the entire array qualifies, it adjusts the length for the allowed deletion and returns the result.

Q. (Leetcode1004) Max Consecutive Ones III
Given a binary array nums and an integer k, return the maximum number of consecutive 1's in the array if you can flip at most k 0s.

Example 1:

Input: nums = [1,1,1,0,0,0,1,1,1,1,0], k = 2
Output: 6
Explanation: [1,1,1,0,0,1,1,1,1,1,1]
Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.

Example 2:

Input: nums = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], k = 3
Output: 10
Explanation: [0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1]
Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.

Code:

```
  public int longestOnes(int[] A, int K) {
   int i = 0, j;
   for (j = 0; j < A.length; ++j) {
       if (A[j] == 0) K--;
       if (K < 0 && A[i++] == 0) K++;
   }
   return j - i;
}
```

Time: O(n), traverse the array
Space: O(1), constant space being used

Explanation: For each A[j], try to find the longest subarray.
If A[i] ~ A[j] has zeros <= K, we continue to increment j.
If A[i] ~ A[j] has zeros > K, we increment i (as well as j).

Q. (leetcode 713) Subarray Product Less Than K
Given an array of integers nums and an integer k, return the number of contiguous subarrays where the product of all the elements in the subarray is strictly less than k.
Example 1:

Input: nums = [10,5,2,6], k = 100
Output: 8
Explanation: The 8 subarrays that have product less than 100 are:
[10], [5], [2], [6], [10, 5], [5, 2], [2, 6], [5, 2, 6]
Note that [10, 5, 2] is not included as the product of 100 is not strictly less than k.
Example 2:

Input: nums = [1,2,3], k = 0
Output: 0

Code:

```
class Solution {
    public int numSubarrayProductLessThanK(int[] nums, int k) {
        if (k == 0) return 0;
        int cnt = 0;
        int pro = 1;
        for (int i = 0, j = 0; j < nums.length; j++) {
            pro *= nums[j];
            while (i <= j && pro >= k) {
                pro /= nums[i++];
            }
            cnt += j - i + 1;
        }
        return cnt;
    }
}
```

Time: O(n), to traverse the array
Space: O(1), only constant space being used

Explanation: The idea is always to keep a max-product-window less than K;
Every time shift window by adding a new number on the right(j), if the product is greater than k, then try to reduce numbers on the left(i), until the subarray product fits less than k again, (subarray could be empty);
Each step introduces x new subarrays, where x is the size of the current window (j + 1 - i);
example:
for window (5, 2), when 6 is introduced, it adds 3 new subarrays: (5, (2, (6)))