* Longest Subarray with Sum K. {ONLY POSITIVE & 0}

⇒ We have an array arr, of size N with a constant K
all elements in arr are "0 or +ve"
⇒ Find the longest subarray with sum K

{N=7, K=3}
{[1 2 3 11 1 1]}

⇒ Brute force :- $T(n^3)$, $S(1)$
Sum of all possible subarrays.

⇒ Better Approch :- $T(n)$, $S(n)$
Using unordered map to store the prefix sum till current element, while traversing the array.

On each Iteration we have ~~curr~~ sum till current element and the prefix sums till each perious element in a unordered map.

- If the currentSum is found equal to K then compare the maxlenght till now with i+1.
- Else if CurrentSum is grater then K then we need to remove some elements from start w/h we will find in Prefix sum, if found then calculate length and compare with maxlen.
- And finally put current sum with current index in map.

- Example :-

	0	1	2	3	4	5
arr =	10	5	2	7	1	9
	i	i	i	i	i	i

K = 15

Sum = 10, 15, 17, 24, 25, 34

maxlen = 0, 2, 4 ← (Ans)

34 → 5
25 → 4
24 → 3
17 → 2
15 → 1
10 → 0

MAP

- NOTE: If there are Negative elements too in array then then the scenerio changes
  : because now as we move right in array the prefix sum is not guaranteed to increase
  ⇒ Suppose prefix sum is 5 Keeps increasing and suddenly a messive -ve number minues the prefix sum back to 5, so the longest Sum will be array till now.
  ⇒ But if we apply the same logic of -ve and 0's here then the "5" prefix sum appears

⇒ multiple times in array so, the index associated to it will update with the latest index where this prefix sum is found. Which is wrong for max len.

⇒ So we will not update ~~the~~ the index with prefix sum even if found again in array. This way we get max len

- Example

arr = | 7 | 2 | 18 | 5 | -9 | 4 |
0 1 2 3 4 5 6

Sum = 7, 9, 10, 13, 18, 9, 13

maxL = 0, 2, 5

Map
18 → 4
13 → 3
10 → 2
9 → 1
7 → 0

∴ Here we have not updated 9's indecies and 13's indecies to keep as many as possible elements in subarray

⇒ OPTIMIZED APPROCH :- T(n), S(1)

:- Using 2 pointers, we will take in instance the sum of subarray ~~from~~ b/w both pointers.

:- if its equal to K then compare its length with previous max lenght and update, finall increase right pointer.

:- if its smaller then k then only increase the right pointer to increase subarray size for more sum.

:- else its grater then K so increase the left pointer to remove elements from subarray and decrease the sum.

There is edge case if left excedes the right then increase right to with it.

:- This whole iteration will run till right pointer is less then "n".