Two Pointer: (For Searching)

***Purpose for reduce time complexity**

Variants of two pointer

Opposite directional : One pointer starts from the beginning while the other pointer starts from the end. They move toward each other until they both meet or some condition satisfy.

Equi-directional: Both start from the beginning, One slow-runner and the other fast-runner.

Coding Problems based On Opposite directional :

167. Two Sum II - Input array is sorted
125. Valid Palindrome
283. Move Zeroes
344. Reverse String
27. Remove Element

Coding Problems based On Equi-directional:

Find the maximum sum of any contiguous subarray of size k.
Find middle node of a linked list
141. Linked List Cycle
3. Longest Substring Without Repeating Characters
26. Remove Duplicates from Sorted Array

#167. Two Sum II - Input array is sorted

Given an array of integers that is already ***sorted in ascending order***, find two numbers such that they add up to a specific target number.

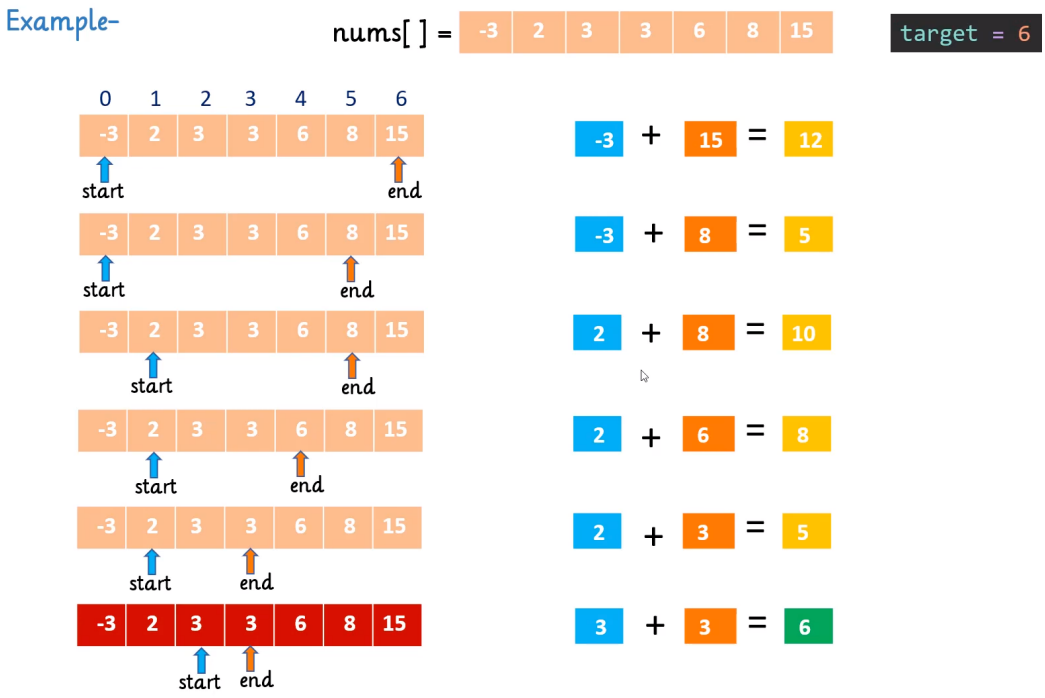
The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2.

Example 1:

```
Input: nums = [2, 7, 11, 15], target = 9,
Output: [1,2]
Explanation: The sum of 2 and 7 is 9. Therefore index1 = 1, index2 = 2.
```

Example 2:

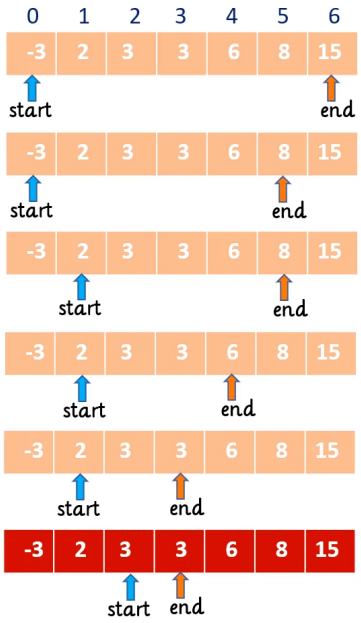
```
Input: nums = [-3, 2, 3, 3, 6, 8, 15], target = 6,
Output: [3,4]
Explanation: The sum of 3 and 3 is 6. Therefore index1 = 3, index2 = 4.
```



Example-

nums[ ] = | -3 | 2 | 3 | 3 | 6 | 8 | 15 |

target = 6



Two Pointer Technique :

```java
static int[] twoSum(nums[],target) {
    int start = 0;
    int end = nums.length - 1;
    int result[] = new int[2];

    while (start < end) {
        int sum = nums[start] + nums[end];
        if (sum == target) {
            result[0] = start + 1;
            result[1] = end + 1;
            break;
        } else if (sum < target) {
            start++;
        } else {
            end--;
        }
    }
    return result;
}
```

Problem Statement **(Equi- directional)**:

Given an array of integers n and a positive number k, find the maximum sum of any contiguous subarray of size k.

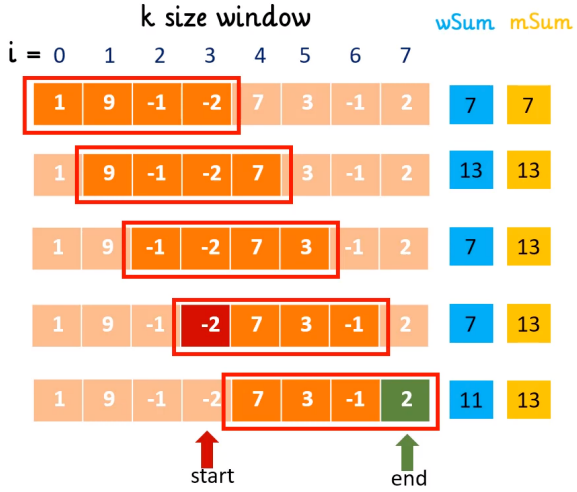
Example 1

```
Input: [2, 1, 5, 1, 3, 2], k=3
Output: 9
Explanation: Subarray with maximum sum is [5, 1, 3].
```

Example 2

```
Input: [1, 9, -1, -2, 7, 3, -1, 2], k=4
Output: 13
Explanation: Subarray with maximum sum is [9, -1, -2, 7].
```

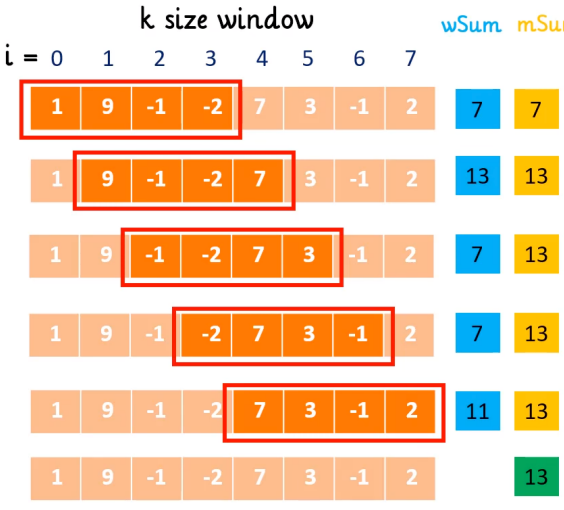
Example- A[ ] = 1 9 -1 -2 7 3 -1 2 k=4




```
windowSum += A[end] - A[start];
```

```
windowSum = 7 + 2 - (-2)
windowSum = 7 + 2 + 2
windowSum = 11
```

Example- A[ ] = 1 9 -1 -2 7 3 -1 2 k=4



Optimal Solution: -

```
getMaxSumSubArrayOfSizeKM2(A[],k) {
    int windowSum = 0, maxSum = 0;
    int start=0,end=0;;

    while(end<k) {
        windowSum += A[end++];
    }

    while(end<A.length) {
        windowSum += A[end++] - A[start++];
        maxSum = Math.max(maxSum, windowSum);
    }
    return maxSum;
}
```