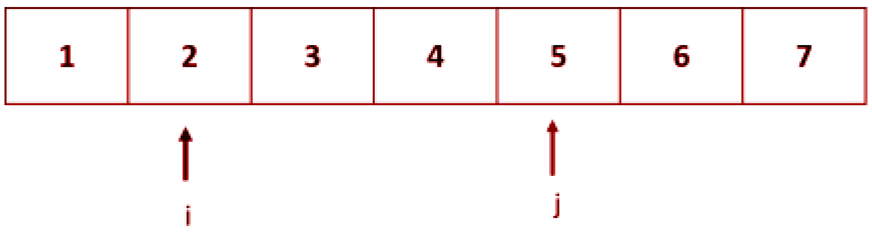
Two Pointers

Two pointers is an algorithmic technique where, as the name suggests, we maintain two pointers to keep track of two indices in the array. The primary condition for using the Two Pointers technique is **monotonicity**. When we can prove that based on a certain condition the pointer is either going to move left or right and by how much, two pointers provide an efficient way to solve problems.

1. **Equi-directional**: Both start from the beginning: we have fixed and variable sliding window algorithms.

2. **Opposite-directional**: One at the start and the other at the end, they move close to each other and meet somewhere in between, (-> <-).



Equi Directional Two Pointers Technique

Here, we maintain a sliding window of flexible length whose endpoints are stored in our moving pointers. We keep one pointer always behind or at the other and use the other one to expand the window size.

- **MINIMUM SUBARRAY WITH PRODUCT AT LEAST P**

Problem Statement: Given an array A consisting of natural numbers, and a number P, find the length of the minimum length subarray whose product is >= P.

Example: Array : 1 2 3 4 5 6
P: 20
Output: 2
Explanation: The subarray 5-6 is the minimum length subarray with product >=20.

Approach:

We know that the product of the subarray is monotonically increasing as the size of the subarray increases. Therefore, we place a 'window' with left and right as i and j at the first item first. The steps are as follows:

- Get the optimal subarray starting from current i, 0: Then we first move the j pointer to include enough items that product[0:j+1]>=P, this is the process of getting the optimal subarray that starts with 0. And assume j stops at ed

- Get the optimal subarray that ends with current j, e0: we shrink the window size by moving the i pointer forward so that we can get the optimal subarray that ends with current j and the optimal subarray starts from s0.

- Now, we find the optimal solution for subproblem [0:i,0:j](the start point in range [0, i] and endpoint in range [0,j]. Start from next i and j, and repeat steps 1 and 2.

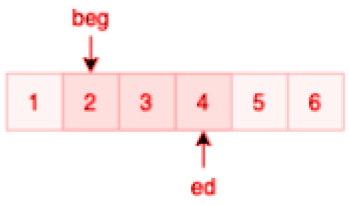


Fig.3 A Subarray with product >=20.

The code looks as follows:

```
function minSubarray(A, P):
        //  store the length of the array
        n = A.length
        /*
                maintain 2 pointers one for the left end and the other for the right end
                of the subarray
        */
        int i = 0, j = 0
        //  We maintain the Product of the current window in the variable curP
        curP = 1
        //  final answer
        ans  = infinity
        //  move the left pointer rightwards
```

```
while(i < n)
      /*
            while we do not reach the required product keep expanding the
            window
      */
      while(j < n and curP <P)
            curP *= A[j]
            j++
      //   we update the ans with the window length
      if(curP >= P)
            ans = min(ans, j  - i + 1)
      //   move the left pointer and update its contribution to the product
      curP /= A[i]
      i++
return ans
```

The above process is a standard flexible window size algorithm, and it is a complete search that searches all the possible result space. Both j and i pointer move at most n, it makes the total operations to be at most 2*n, which we get time complexity as O(n).

Opposite - directional Two Pointers Technique

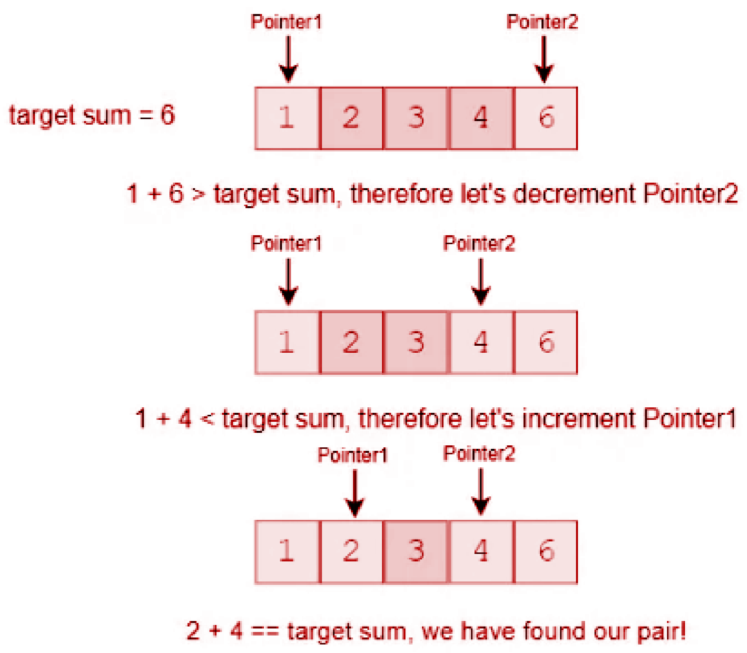
We start with 2 pointers where one is usually placed at the beginning of the array and the other one is placed at the end of the array.
Two-pointers are usually used for searching a pair in the array. There are cases where the data is organized in a way that we can search all the result space by placing two pointers each at the start and rear of the array and move them to each other and eventually meet and terminate the search process. The search target should help us decide which pointer to move at that step. This way, each item in the array is guaranteed to be visited at most one time by one of the two pointers, thus making the time complexity be O(n). Binary search uses the technique of two pointers too, the left and right pointer together decide the current searching space, but it erases half searching space at each step instead.

- **TWO SUM**

Problem Statement: Given a sorted array and a number S, find the indices of any two elements which sum up to exactly S, or report that such a pair does not exist.

Example: Array: 1 2 3 4 5 6
Target Sum: 6
Output: 1, 3
Explanation: Elements at indices 1 and 3 (i.e 2 and 4 respectively sum upto the target sum = 6).

Approach:



Pointer1
Pointer2
target sum = 6
1 + 6 > target sum, therefore let's decrement Pointer2
Pointer1
Pointer2
1 + 4 < target sum, therefore let's increment Pointer1
Pointer1
Pointer2
2 + 4 == target sum, we have found our pair!

Due to the fact that the array is sorted which means in the array [s,s1 ..., e1, e], the sum of any two integers is in the range of [s+s1, e1+e]. By placing two pointers, v1 and v2, that start from s and e, we started the search space from the middle of the possible range. [s+s1, s+e, e1+e]. Compare the target t with the sum of the two pointers v1 and v2:

1. S == v1 + v2: found
2. v1 + v2 < S: we need to move to the right side of the space from v1, i.e we increase v1 to get a larger value.
3. v1 + v2 > S: we need to move to the left side of the space from v2, i.e we decrease v2 to get a smaller value.

The code looks as follows:

```
function twoSum(A, S):
        //  store the length of the array
        n = A.length
        /*
        maintain 2 pointers one for the left end and the other for the right end of
        the subarray.
*/
int i = 0, j = n - 1
//  final answer pair
ans  = {-1 , -1}
//  while the left pointer is less than the right pointer
while(i <= j)
    //  calculate the current Sum of 2 pointers
    curSum = A[i] + A[j]
    if(curSum == S)
        //  we have found a valid pair so return the indices
        ans = {i, j}
        return ans
    else if (curSum > S)
        //  move the right pointer left as the sum is greater
        j -= 1
    else
        //  move the left pointer right as the sum is smaller
        i += 1
return ans
```

Since both the pointers move to the right and left at most n times the total increment and decrement operators are bounded by 2*n. Hence the Time complexity is O(n). The space complexity is O(1) since we are using constant space to maintain the pointers.