Revision - Two Pointers

Introduction

In two pointer technique, we iterate two pointeres across an array to search for a pair of indices satisfying some condition in linear time.
Since each pointer iterate over the array just once, the time complexity of two pointer techniques are usually O(N), where N is the size of the array. The time complexity can change depending on the operations you perform during iteration of the pointers.
The main idea to solve any particular question using two pointers are:

1. Can we define two pointers with a certainity of what exact condition we are trying to construct.
2. We know with certainity in which direction we need to update the two pointers.

Question - 1

Given a sorted integer array and an integer K. Find any pair (i, j) such that A[i] + A[j] = K.

Example: Consider the array A = [1, 2, 8, 9, 12], and K = 11.
Here, 2 + 9 = 11, so our answer would be (1, 3).

Solution:

Brute force: In our brute force solution, we would simply iterate over all pairs (i, j), check if A[i] + A[j] = K, and return the answer accordingly.

Pseudo Code:

```
function(A, K):
    N = length of A
    for i from 0 to N - 1:
        for j from i + 1 to N - 1:
            if A[i] + A[j] is equal to K:
                return (i, j)

    No answer found
    return (-1, -1)
```

- Time complexity: $O(N^2)$
- Space complexity: O(1)

Here, we can use two pointer technique to solve this problem. We come through the following observation:

1. The array is sorted.

We can maintain two pointers:

1. left pointer initially pointing to index 0.
2. right pointer initially pointing to index n - 1.

Now, we can move these pointers based on the following conditions:

- If the sum is equal to K, we simply return the (left, right) pairs whose sum is equal to K.
- If the sum is greater than K, then we are sure that we need to reduce the sum. How can we reduce it?
- Incrementing left pointer will simply increase the sum. So, we need to decrement the right pointer as it will result in decreasing the sum.
- Similar is the case when the sum is less than K.
- We were able to use two pointer technique as we are certain of the condition we have constructed and the direction of two pointers movement.

Psedo Code:

```
function(A, K):
    n = length of A
    left = 0
    right = n - 1
    while left < right:
        if A[left] + A[right] is equal to K:
            return (left, right)
        else if A[left] + A[right] > K:
            right = right - 1
        else:
            left = left + 1

    if we come out of the loop, we are sure that we did not got any pairs.

    return (-1, -1)
```

- Time complexity: O(N)
- Space complexity: O(1)

Question - 2

Given an array of positive elements. Check if there is a subarray with sum K.

Example: Consider the array A = [1, 3, 15, 10, 2, 4] and K = 27

Solution:

Brute force: In the brute force solution, we can think about finding the sum of all possible subarrays and check if the sum is equal to K. The time complexity would be $O(N^3)$. We can optimize it by using a prefix array.

Pseudo Code:

```
function(A, K):
    N = length of A
    prefix = A
    for i from 1 to N - 1:
        prefix[i] = prefix[i] + A[i]

    for i from 1 to N - 1:
        for j from i + 1 to N - 1:
            if prefix[j] - prefix[i - 1] == K:
                handle i - 1 == -1 separately
                return true

    we didn't got any subarray
    return false
```

Observation:

- The prefix array which we construct is an increasing array.

Here, we can think of maintaining a subarray which can be a possible answer.
Since a subarray can be defined using its end points so we can think about two pointer techniques.

We can maintain two pointers left and right each initially at index 0.
Now, the conditions and directions of movement of the pointers are explained below:

The algorithm works as follows:

- We are trying to maintain the sum of the subarray which is enclosed by left and right pointer.
- Whenever we increment the right pointer, we simply add A[right] to the sum, as we have added another element to our maintained subarray.
- Now, we will keep on removing the elements at left pointer until our sum is greater than K.
- After removing all such elements, we are sure that the subarray which we have maintained has a sum less than or equal to K, as we have removed all the left elements which were making the sum greater than K.

- After removing left elements, we simply check if the sum is equal to K, and return true if it is.

Pseudo Code:

```
function(A, K):
    n = length of A
    right = 0
    left = 0
    sum = 0
    for right from 0 to n - 1:
        sum = sum + A[right]
        while sum > K:
            sum -= A[left]
            left = left + 1
        if sum is equal to K:
            return true

    return false
```

- Time complexity: O(N)
- Space complexity: O(1)