

Advanced Search on Linear Data Structures

LI YIN¹

February 8, 2020

¹www.liyinscience.com

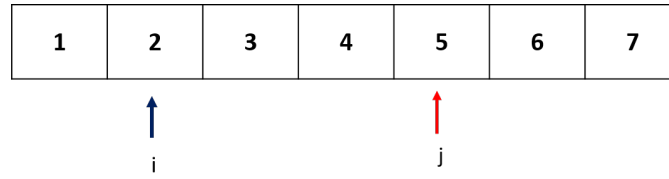


Figure 1: Two pointer Technique

On linear data structures, or on implicit linear state space, either a particular targeted item or a consecutive substructure such as a subarray and substring can be searched.

To find a single item on linear space, we can apply linear search in general, or binary search if the data structure is ordered/sorted with logarithmic cost. In this chapter, we introduce two pointer techniques that are commonly used to solve two types of problems:

1. Searching: To search for an item such as median, a predefined substructure, and a substructure that satisfy certain conditions such as finding the minimum subarray length wherein the subarray equals to a targeted sum. Or find a substructure satisfy a string pattern.
2. Adjusting: To adjust ordering or arrangement of items in the data structure such as removing duplicates from sorted array.

As the name suggests, Two pointers technique involves two pointers that start and move with the following two patterns:

1. Equi-directional: Both pointers start from the beginning of the array, and usually one moves faster and the other slower. Sliding window algorithm can be put into this category.
2. Opposite-directional: One pointer start at the start position and conversely the other pointer starts at the end. These two oppositely posed pointers move toward each other and usually meet in the middle.

In the following sections, we will detail on two-pointer technique exemplified on real interview questions.

0.1 Slow-Faster Pointers

Suppose we have two pointers, i and j , which may or may not start at the start position in the linear data structures, but one move slower (i) and the other faster (j). Two pointers can decide either a pair or a subarray to solve related problems. For the case of subarray, the algorithm is called sliding window algorithm. On the span of the array, and at most of three potential

sub-spaces exist: from start index to i ($[0, i]$), from i to j ($[i, j]$), and from j to the end index ($[j, n]$).

Even though slow-faster pointers technique rarely given formal introduction in book, it is widely used in algorithms. In sorting, Lumuto's partition in the QuickSort used the slow-faster pointers to divide the whole region into three parts according the comparison result to the pivot: Smaller Items region, Larger Items region, and the unrestricted region. In string pattern matching, fixed sliding window and one we will introduce in this chapter.

In this section, we explain how two pointers work on two types of linear data structures: Array and Linked List.

0.1.1 Array

Remove Duplicates from Sorted Array(L26)

Given a sorted array $a = [0, 0, 1, 1, 1, 2, 2, 3, 3, 4]$, remove the duplicates in-place such that each element appears only once and return the new length. Do not allocate extra space for another array, you must do this by modifying the input array in-place with $O(1)$ extra memory. In the given example, there are in total of 5 unique items and 5 is returned.

Analysis We set both slower pointer i and the faster pointer j at the first item in the array. Recall that slow-fast pointers cut the space of the sorted array into three parts, we can define them as:

1. unique items in region $[0, i]$,
2. untouched items in region $[i + 1, j]$,
3. and unprocessed items in region $[j + 1, n)$.

In the process, we compare the items pointed by two pointers, once these two items does not equal, we find an new unique item. We copy this unique item at the faster pointer right next to the position of the slower pointer. Afterwards, we move the slow pointer by one position to remove duplicates of our copied value.

With our example, at first, $i = j = 0$, region one has one item which is naively unique and region two has zero item. Part of the process is illustrated as:

i	j	$[0, i]$	$[i+1, j]$	process
0	0	[0]	[]	item $0==0$, $j+1=1$
0	1	[0]	[0]	item $0==0$, $j+1=2$
0	2	[0]	[0, 1]	item $0!=1$, $i+1=1$, copy 1 to index 1, $j+1=3$
1	3	[0, 1]	[1, 1]	item $1==1$, $j+1=4$
1	4	[0, 1]	[1, 1, 1]	item $1==1$, $j+1=5$

```

1  5    [0, 1]  [1, 1, 1, 2] item 1==2, i+1=2, copy 2 to index 2,
    j+1=6
2  6    [0, 1, 2] [1, 1, 2, 2]

```

The code is given as:

```

1 def removeDuplicates(nums) -> int:
2     i, j = 0, 0
3     while j < len(nums):
4         if nums[i] != nums[j]:
5             # Copy j to i+1
6             i += 1
7             nums[i] = nums[j]
8         j += 1
9     return i + 1

```

After calling the above function on our given example, array a becomes $[[0, 1, 2, 3, 4, 2, 2, 3, 3, 4]]$. Check the source code for the whole visualized process.

Minimum Size Subarray Sum(L209)

Given an array of n positive integers and a positive integer s , find the minimal length of a contiguous subarray of which the $sum \geq s$. If there isn't one, return 0 instead.

Example:

Input: $s = 7$, $nums = [1, 4, 1, 2, 4, 3]$

Output: 2

Explanation: the subarray $[4, 3]$ has the minimal length under the problem constraint.

Analysis In this problem, we need to secure a substructure—subarray—that not only satisfies a condition ($sum \geq s$) but also has the minimal length. Naively, we can enumerate all subarrays and search through them to find the minimal length, which requires at least $O(n^2)$ time complexity using prefix sum. The code is as:

However, we can use two pointers i and j ($i \leq j$) and both points at the first item. In this case, these two pointers defines a subarray $a[i : j + 1]$ and we care the region $[i, j]$. As we increase pointer j , we keep adding positive item into the sum of the subarray, making the subarray sum monotonically increasing. Oppositely, if we increase pointer i , we remove positive item away from the subarray, making the sum of the subarray monotonically decreasing. The detailed steps of two pointer technique in this case is as:

1. Get the optimal subarray for all subproblems(subarries) that start from current i , which is 0 at first. We accomplish this by forwarding

j pointer to include enough items until $sum \geq s$ that we pause and go to the next step. Let's assume pointer j stops at e_0 .

2. Get the optimal subarray for all subproblems(subarrays) that end with current j , which is e_0 at the moment. We do this by forwarding pointer i this time to shrink the window size until $sum \geq s$ no longer holds. Let's assume pointer i stops at index s_0 . Now, we find the optimal solution for subproblems $a[0 : i, 0 : j]$ (denoting subarrays with the start point in range $[0, i]$ and the end point in range $[0, j]$).
3. Now that $i = s_0$ and $j = e_0$, we repeat step 1 and 2.

In our example, we first move j until $j = 3$ with a subarray sum of 8. Then we move pointer i until $i = 1$ when the subarray sum is less than 7. For subarray $[1, 4, 1, 2]$, we find its optimal solution to have a length 3. The Python code is given as:

```

1 def minSubArrayLen(s: int, nums) -> int:
2     i, j = 0, 0
3     acc = 0
4     ans = float('inf')
5     while j < len(nums):
6         acc += nums[j]
7         # Shrink the window
8         while acc >= s:
9             ans = min(ans, j - i + 1)
10            acc -= nums[i]
11            i += 1
12        j += 1
13
14    return ans if ans < float('inf') else 0

```

Because both pointer i and j move at most n steps, with the total operations to be at most $2n$, making the time complexity as $O(n)$. The above question would be trivial if the maximum subarray length is asked.

0.1.2 Minimum Window Substring (L76, hard)

Given a string S and a string T , find all the minimum windows in S which will contain all the characters in T in complexity $O(n)$.

Example:

Input: $S = \text{"AOBECDBANC"} , T = \text{"ABC"}$
 Output: $[\text{"CDBA"} , \text{"BANC"}]$

Analysis Applying two pointers, with the region between pointer i and j to be our testing substring. For this problem, the condition for the window $[i, j]$ it will at most have all characters from T . The intuition is we keep expanding the window by moving forward j until all characters in T is found. Afterwards, we contract the window so that we can find the minimum

'A'	'B'	'C'	count
1	1	1	3

Figure 2: The data structures to track the state of window.

window with the condition satisfied. Instead of using another data structure to track the state of the current window, we can depict the pattern T as a dictionary data structure where all unique characters comprising the keys and with the number of occurrence of each character as value. We use another variable `count` to track how the number of unique characters. In all, they are used to track the state of the moving window in $[i, j]$, with the value of the dictionary to indicate how many occurrence is short of, and the `count` represents how many unique characters is not fully found, and we depict the state in Fig. 2.

Along the expanding and shrinking of the window that comes with the movement of pointer i and j , we track the state with:

- When forwarding j , we encompass $S[j]$ in the window. If $S[j]$ is a key in the dictionary, decrease the value by one. Further, if the value reaches to the threshold 0, we decrease `count` by one, meaning we are short of one less character in the window.
- Once `count=0`, our window satisfy condition for contracting. We then forward i , removing $S[i]$ from the window if it is existing key in the dictionary by increasing this key's value, meaning the window is short of one more character. Once the value reaches to the threshold of 1, we increase `count`.

Part of this process with our example is shown in Fig. 3. And the Python code is given as:

```

1 from collections import Counter
2 def minWindow(s, t):
3     dict_t = Counter(t)
4     count = len(dict_t)
5     i, j = 0, 0
6     ans = []
7     minLen = float('inf')
8     while j < len(s):
9         c = s[j]
10        if c in dict_t:

```

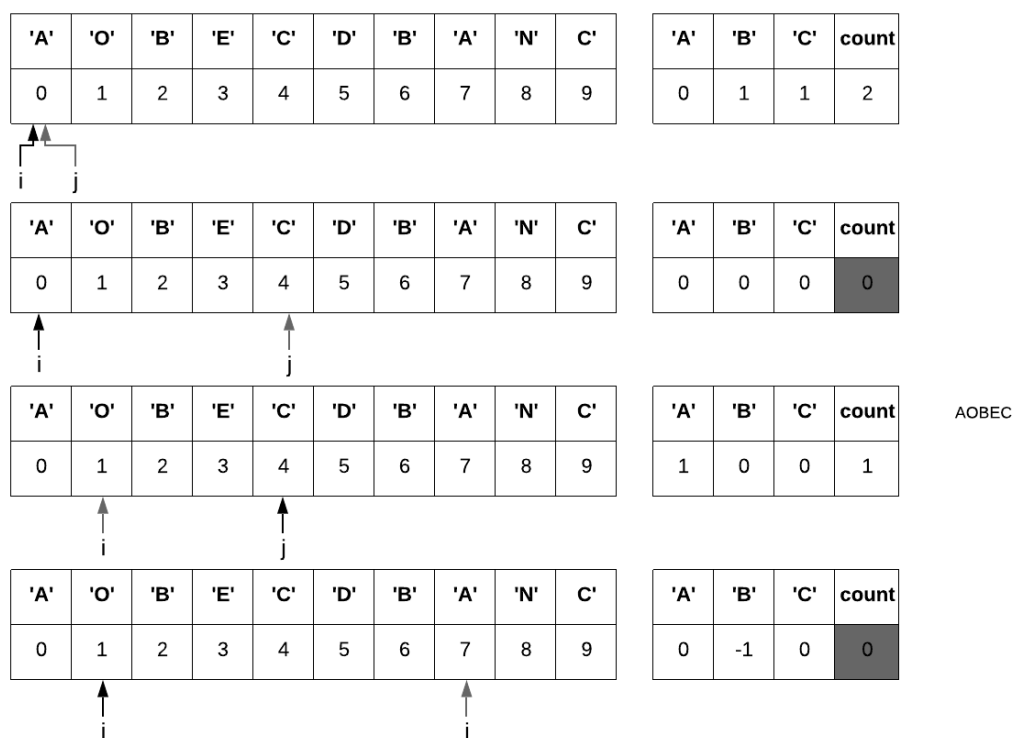


Figure 3: The partial process of applying two pointers. The grey shaded arrow indicates the pointer that is on move.

```

11     dict_t[c] -= 1
12     if dict_t[c] == 0:
13         count -= 1
14     # Shrink the window
15     while count == 0 and i < j:
16         curLen = j - i + 1
17         if curLen < minLen:
18             minLen = j - i + 1
19             ans = [s[i:j+1]]
20         elif curLen == minLen:
21             ans.append(s[i:j+1])
22
23     c = s[i]
24     if c in dict_t:
25         dict_t[c] += 1
26         if dict_t[c] == 1:
27             count += 1
28     i += 1
29
30     j += 1
31     return ans

```

0.1.3 When Two Pointers do not work

Two pointer does not always work on subarray related problems.

What happens if there exists negative number in the array?

Since the sum of the subarray is no longer monotonically increasing with the number of items between two pointers, we can not figure out how to move two pointers each step. Instead (1) we can use prefix sum and organize them in order, and use binary search to find all possible start index. (2) use monotone stack (see LeetCode problem: 325. Maximum Size Subarray Sum Equals k, 325. Maximum Size Subarray Sum Equals k (hard))

What if we are to check the maximum average subarray?

644. Maximum Average Subarray II (hard). Similarly, the average of subarray does not follow a certain order with the moving of two pointers at each side, making it impossible to decide how to make the two pointers.

0.1.4 Linked List

The complete code to remove cycle is provided in google colab together with running examples.

Middle of the Linked List(L876)

Given a non-empty, singly linked list with head node *head*, return a middle node of linked list. When the linked list is of odd length, there exists one and only middle node, but when it is of even length, two exists and we return the second middle node.

Example 1 (odd length):

Input: [1,2,3,4,5]

Output: Node 3 from this list (Serialization: [3,4,5])

Example 2 (even length):

Input: [1,2,3,4,5,6]

Output: Node 4

from this list (Serialization: [4,5,6])

Analysis If the data structure is array, we can compute the position of the middle item simply with the total length. Following this method, if only one pointer is applied, we can first iterate over the whole linked list in $O(n)$ time to get the length. Then we do another iteration to obtain the middle node. $n + \frac{n}{2}$ times of operations needed, making the time complexity $O(n)$.

However, we can apply two pointers simultaneously at the head node, each one moves at different paces: the slow pointer moves one step at a time and the fast moves two steps instead. When the fast pointer reached the end, the slow pointer will stop at the middle. This slow-faster pointers technique requires only $\frac{n}{2}$ times of operations, which is three times faster than our naive method, although the big Oh time complexity still remains $O(n)$.

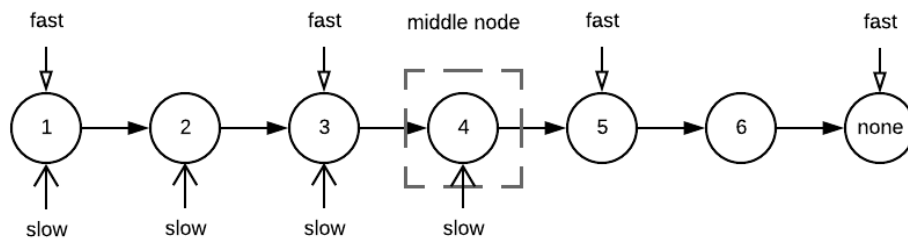


Figure 4: Slow-fast pointer to find middle

Implementation Simply, we illustrate the process of running the two pointers technique on our two examples in Fig. 4. As we can see, when the slow pointer reaches to item 3, the faster pointer is at item 5, which is the last item in the first example that comes with odd length. Further, when the slow pointer reaches to item 4, the faster pointer reaches to the empty node of the last item in the second example that comes with even length. Therefore, in the implementation, we check two conditions in the `while` loop:

1. For example 1: if the fast pointer has no successor (`fast.next==None`), the loop terminates.
2. For example 2: if the fast pointer is invalid (`fast==None`), the loop terminates.

The Python code is as:

```

1 def middleNode(head):
2     slow = fast = head
3     while fast and fast.next:
4         fast = fast.next.next
5         slow = slow.next
6     return slow

```

Floyd's Cycle Detection (Floyd's Tortoise and Hare)

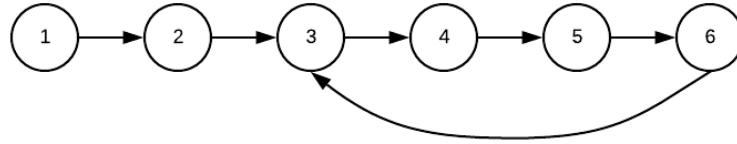


Figure 5: Circular Linked List

When a linked list which has a cycle, as shown in Fig. 5, iterating items over the list will make the program stuck into infinite loop. The pointer starts from the heap, traverse to the start of the loop, and then comes back to the start of the loop again and continues this process endlessly. To avoid being stuck into a “trap”, we have to possibly solve the following three problems:

1. Check if there exists a cycle.
2. Check where the cycle starts.
3. Remove the cycle once it is detected.

The solution encompasses the exact way of slow faster pointers traversing through the linked list as our last example. With the slow pointer iterating one item at a time, and the faster pointer in double pace, these two pointers will definitely meet at one item in the loop. In our example, they will meet at node 6. So, is it possible that it will meet at the non-loop region starts from the heap and ends at the start node of the loop? The answer is No, because the faster pointer will only traverse through the non-loop region once and it is always faster than the slow pointer, making it impossible to meet in this region. This method is called Floyd's Cycle Detection, aka Floyd's Tortoise and Hare Cycle Detection. Let's see more details at how to solve our mentioned three problems with this method.

Check Linked List Cycle(L141) Compared with the code in the last example, we only need to check if the **slow** and **fast** pointers are pointing at the same node: If it is, we are certain that there must be a loop in the list and return **True**, otherwise return **False**.

```

1 def hasCycle(head):
2     slow = fast = head
3     while fast and fast.next:
4         slow = slow.next
5         fast = fast.next.next
6         if slow == fast:
7             return True
8     return False
  
```

Check Start Node of Linked List Cycle(L142) Given a linked list, return the node where the cycle begins. If there is no cycle, return `None`.

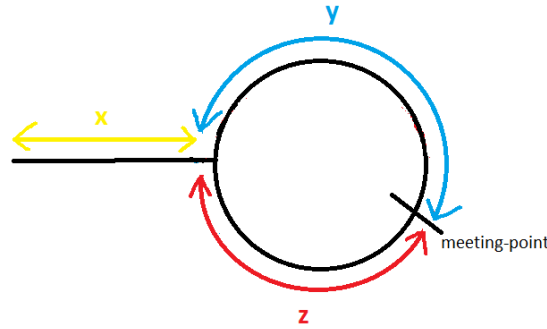


Figure 6: Floyd's Cycle finding Algorithm

For a given linked list, assume the slow and fast pointers meet at node somewhere in the cycle. As shown in Fig. 6, we denote three nodes: head (h , start node of cycle(s), and meeting node in the cycle(m). we denote the distance between h and s to be x , the distance between s and m to be y , and the distance between m and s to be z . Because the faster pointer traverses through the list in double speed, when it meets up with the slow pointer, the distance that it traveled($x + y + z + y$) to be two times of the distance traveled by the slow pointer ($x + y$).

$$x + y + z + y = x + y \quad (1)$$

$$x = z \quad (2)$$

From the above equation, we obtain the equal relation between x and z . the starting node of the cycle from the head is x , and y is the distance from the start node to the slow and fast pointer's node, and z is the remaining distance from the meeting point to the start node. Therefore, after we have detected the cycle from the last example, we can reset the slow pointer to the head of the linked list after. Then we make the slow and the fast pointer both traverse at the same pace—one node at a time—until they meet at a node we stop the traversal. The node where they stop at is the start node of the cycle. The code is given as:

```

1 def detectCycle(head):
2     slow = fast = head
3
4     def getStartNode(slow, fast, head):
5         # Reset slow pointer
6         slow = head
7         while fast and slow != fast:
8             slow = slow.next

```

```

9         fast = fast.next
10        return slow
11
12        while fast and fast.next:
13            slow = slow.next
14            fast = fast.next.next
15            # A cycle is detected
16            if slow == fast:
17                return getStartNode(slow, fast, head)
18
19        return None

```


Remove Linked List Cycle We can remove the cycle by recirculating the last node in the cycle, which in example in Fig. 5 is node 6 to an empty node. Therefore, we have to modify the above code to make the `slow` and `fast` pointers stop at the last node instead of the start node of the loop. This subroutine is implemented as:

```

1 def resetLastNode(slow, fast, head):
2     slow = head
3     while fast and slow.next != fast.next:
4         slow = slow.next
5         fast = fast.next
6     fast.next = None

```

The complete code to remove cycle is provided in google colab together with running examples.

 **What if there has not only one, but multiple cycles in the Linked List?**

0.2 Opposite-directional Pointers

Another variant of two pointers technique is to place these two pointers oppositely: one at the beginning and the other at the end of the array. Through the process, they move toward each other until they meet in the middle. Details such as how much each pointer moves or which pointer to move at each step decided by our specific problems to solve. We just have to make sure when we are applying this technique, we have considered its whole state space, and will not miss out some area which makes the search incomplete.

The simplest example of this two pointers method is to reverse an array or a string around. For example, when the list $a = [1, 2, 3, 4, 5]$ is reversed, it becomes $[5, 4, 3, 2, 1]$. Of course we can simply assign a new list and copy the items in reversed orders. But, with two pointers, we are able to reverse

it in-place and using only $O(\frac{n}{2})$ times of operations through the following code:

```

1 def reverse(a):
2     i, j = 0, len(a) - 1
3     while i < j:
4         # Swap items
5         a[i], a[j] = a[j], a[i]
6         i += 1
7         j -= 1

```

Moreover, binary search can be viewed as an example of opposite-directional pointers. At first, these two pointers are the first and the last item in the array. Then depends on which side of the target compared with the item in the middle, one of the pointers move either forward or backward to the middle point, reducing the search space to half of where it started at each step. We also explore another example with this technique.

Two Sum on Sorted Array(L167)

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.

Input: numbers = [2,7,11,15], target = 9

Output: [1,2]

Explanation: The sum of 2 and 7 is 9. Therefore index1 = 0, index2 = 1.

Analysis If we simply put enumerate all possible pairs, we have to take $O(n^2)$ to solve this problem. However, with the opposite-directional two pointers, it gives out linear performance.

Denote the list as $A = [a_1, a_2, \dots, a_{n-1}, a_n]$, and for the sorted array we have $a_1 \leq a_2 \leq \dots \leq a_{n-1} \leq a_n$. The range of the sum of any two items in the array is within two possible ranges: $[a_1 + a_2, a_1 + a_n]$ and $[a_1 + a_n, a_{n-1} + a_n]$. By placing one pointer i at a_1 and the other j at a_n to start with, we can get $a_1 + a_n$ as the sum. Pointer i can only move forward, accessing larger items. On the other hand, pointer j can only backward, accessing smaller items. Now there are three scenarios according to the comparison between the target and the current sum of the two pointers:

1. If $t == a[i] + a[j]$, target sum found.
2. If $t > a[i] + a[j]$, we have to increase the sum, we can only do this by moving pointer i forward.
3. If $t < a[i] + a[j]$, we have to decrease the sum, we can only do this by moving pointer j backward.

The Python code is as:

```

1 def twoSum(a, target):
2     n = len(a)
3     i, j = 0, n-1
4     while i < j:
5         temp = a[i] + a[j]
6         if temp == target:
7             return [i, j]
8         elif temp < target:
9             i += 1
10        else:
11            j -= 1
12    return []

```

0.3 Follow Up: Three Pointers

Sometimes, manipulating two pointers is not even enough to distinguish different subspaces, we might need to the assistant of one another pointer to make things work.

Binary Subarrays With Sum (L930)

In an array A of 0s and 1s, how many non-empty subarrays have sum S ?

Example 1:
Input: $A = [1, 0, 1, 0, 1]$, $S = 2$
Output: 4
Explanation:
The 4 subarrays are listed below:
 $[1, 0, 1]$, index (0, 2)
 $[1, 0, 1, 0]$, index (0, 3)
 $[0, 1, 0, 1]$, index (1, 4)
 $[1, 0, 1]$, index (2, 4)

Analysis This problem is highly similar to the minimum length subarray problem we encountered before. We naturally start with two pointers i and j , and restrict the subarray in range $[i, j]$ to satisfy condition $sum \leq S$. The window is contracted when the condition is violated. We would have write the following code:

```

1 def numSubarraysWithSum(a, S):
2     i, j = 0, 0
3     win_sum = 0
4     ans = 0
5     while j < len(a):
6         win_sum += a[j]
7         while i < j and win_sum > S:
8             win_sum -= a[i]
9             i += 1
10        if win_sum == S:
11            ans += 1

```

```

12     print('{}, {}'.format(i, j))
13     j += 1
14     return ans

```

However, the above code only returns 3, instead of 4 as shown in the example. By printing out pointers i and j , we can see the above code is missing case (2, 4). Why? Because we are restricting the subarray sum in range $[i, j]$ to be smaller than or equal to S , with the occurrence of 0s that might appear in the front or in the rear of the subarray:

- In the process of expanding the subarray, pointer j is moved one at a time. Thus, even though 0s appear in the rear of the subarray, the counting is correct.
- However, in the process of shrinking the subarray while the restriction is violated ($sum > S$), we stop right away once $sum \leq S$. And in the code, we end up only counting it as one occurrence. With 0s at the beginning of the subarray, such as the subarray $[0, 1, 0, 1]$ with index 1 and 4, there count should be two instead of one.

The solution is to add another pointer i_h to handle the missed case: When the $sum = S$, count the total occurrence of 0 in the front. Compared with the above solution, the code only differs slightly with the additional pointer and one extra `while` loop to deal the case. Also we need to pay attention that $i_h \leq j$, otherwise, the `while` loop would fail with example with only zeros and a targeting sum 0.

```

1 def numSubarraysWithSum(a, S):
2     i, i_h, j = 0, 0, 0
3     win_sum = 0
4     ans = 0
5     while j < len(a):
6         win_sum += a[j]
7         while i < j and win_sum > S:
8             win_sum -= a[i]
9             i += 1
10        # Move i_h to count all zeros in the front
11        i_h = i
12        while i_h < j and win_sum == S and a[i_h] == 0:
13            ans += 1
14            i_h += 1
15
16        if win_sum == S:
17            ans += 1
18        j += 1
19    return ans

```

We noticed that in this case, we have to explicitly restrict $i < j$ and $i_h < j$ due to the special case, while in all our previous examples, we do not have to.

0.4 Summary

Two pointers is a powerful tool for solving problems on linear data structures, such as “certain” subarray and substring problems as we have shown in the examples. The “window” secluded between the two pointers can be viewed as sliding window: It can move slide forwarding with the forwarding the slower pointer. Two important properties are generally required for this technique to work:

8	5	10	7	9	4	15	12	90
8	5	10	7	9	4	15	12	90
8	5	10	7	9	4	15	12	90

Figure 7: Sliding Window Property

1. Sliding window property: Either we move the faster pointer j forward by one, or move the slower pointer i , we can get the state of current window in $O(1)$ cost knowing the state of the last window.

For example, given an array, imagine that we have a fixed size window as shown in Fig. 7, and we can slide it forward one position at a time, compute the sum of each window. The bruteforce solution would be of $O(kn)$ complexity where k is the window size and n is the array size by using two nested `for` loops: one to set the starting point, and the other to compute the sum in $O(k)$. However, the sum of the current window (S_c) can be computed from the last window (S_l), and the items that just slid out and in as a_j and a_i respectively. Then $S_c = S_l - a_i + a_j$. Getting the state of the window between two pointers in $O(1)$ as shown in the example is our called Sliding Window Property.

Usually, for an array with numerical value, it satisfies the sliding window property if we are to compute its sum or product. For substring, as shown in our minimum window substring example, we can get the state of current window referring to the state of the last window in $O(1)$ with the assist of dictionary data structure. In substring, this is more obscure, and the general requirement is that the state of the substring does not relate to the order of the characters(anagram-like state).

2. Monotonicity: For subarray sum/product, the array should only comprise all positive/negative values so that the prefix sum/product has

monotonicity: moving the faster pointer and the slower pointer forward results into opposite change to the state. The same goes for the substring problems where we see from the minimum window substring example the change of the state: `count` and the value of the dictionary is monotonic, and each either increases or decreases with the moving of two pointers.

0.5 Exercises

1. 3. Longest Substring Without Repeating Characters
2. 674. Longest Continuous Increasing Subsequence (easy)
3. 438. Find All Anagrams in a String
4. 30. Substring with Concatenation of All Words
5. 159. Longest Substring with At Most Two Distinct Characters
6. 567. Permutation in String
7. 340. Longest Substring with At Most K Distinct Characters
8. 424. Longest Repeating Character Replacement