Two Sum II - Input Array Is Sorted

Created	@2025年4月13日 下午1:50
Question Type	Two Pointers
Difficulty	Medium
LeetCode Question Link	https://leetcode.com/problems/two-sum-ii-input-array-is-sorted/

1. Question Self-understanding:

1.1 Description:

The goal is to find two indices in a sorted (non-decreasing) integer array such that the values at those indices sum to a specified target.

1.2 Input:

The input is a list of integers in non-decreasing order.

1.3 Input Assumption

- There must be two numbers in the array whose sum equals the target value.
- There is exactly one valid solution.
- You may not use the same element twice.

1.4 Output:

The output should be a list containing two indices (1-based) where the values add up to the target.

1.5 Example:

```
Input: numbers = [2,7,11,15], target = 9
Output: [1,2]
```

1.6 Other Q&A:

- **Question**: Can the array be empty?

  Answer: No. This would contradict the assumption that there are two numbers summing to the target.

- **Question**: Can both numbers be the same value?

  Answer: Potentially yes, because the array is sorted in non-decreasing order (e.g., [2,2], target = 4).

2. Attempt 1:

2.1 Thought:

Because the array is sorted and a valid solution is guaranteed, one approach is to use a hash map to store each number (or its complement) along with its index. While iterating through the array:

1. Calculate the complement of the current number relative to the target.
2. Check if the complement is already in the hash map.
3. If it is, return the solution.
4. Otherwise, store the complement of the current number in the hash map.

2.2 Pseudo-Code: (Ignore this part. This is a draft of code for thinking purposes.)

```
class Solution:
    def twoSum(self, numbers: List[int], target: int) → List[int]:
        # Initialize a hash map
        # For each index i and number num in numbers:
        #   if num is in hash map:
        #       return [hash_map[num] + 1, i + 1]
        #   else:
        #       hash_map[target - num] = i
```

2.3 Implementation through python:

```
class Solution:
    def twoSum(self, numbers: List[int], target: int) → List[int]:
        # Initial hash_map
        prev_map = {}
        # Initialize the final result list
        # for-loop the numbers
        for i in range(len(numbers)):
            # check if the number is in the hash_map
            if numbers[i] in prev_map:
                return [prev_map[numbers[i]] + 1, i + 1]
            # add the number to the hash_map
            prev_map[target - numbers[i]] = i
```

2.4 Time Complexity and Space Complexity

2.4.1 Time Complexity:

- O(n), where n is the length of the list, because we iterate once through the list.

2.4.2 Space Complexity:

- O(n) in the worst case, because we might store information about every element in the hash map.

3. Attempt 2:

3.1 Thought:

How can we optimize, and in what direction?

- Although the worst-case scenario requires checking all numbers (giving a worst-case time complexity of O(n), we should consider the space complexity. In the current approach, many entries in the hash map may never be used because there is only one solution. Thus, most of the stored elements won't actually be retrieved later.
- Why keep all these unused elements? Right now, we go from the front to the back, meaning we need to store previous elements in case a later value pairs with one of them to form the target sum. But is there a way to determine that a certain value will not contribute to a future pair? For instance, if the sum of the current number and the last number is already larger or smaller than the target, we might decide how to move our pointers accordingly.
- Could we track both ends instead of just one?
    - **Yes.**

Observation: If the current number increases, the number from the back will decrease.

- Can we use this property to reduce space usage by only checking two numbers at a time, rather than storing additional data?
    - **Yes.**

Continuing with this idea:

- **When does the left-hand side pointer increase?**

  - It increases if the number at the back is not large enough to reach the target.
- **When does the right-hand side pointer decrease?**
  - It decreases if the number at the front is too large for the target.
- **Is it possible for the left pointer to decrease and the right pointer to increase during the algorithm?**
  - No. Once the left pointer moves right because the back number was not large enough, the right pointer only moves inward. Similarly, the right pointer never moves outward again once it starts decreasing.

3.2 Pseudo-Code: (Ignore this part. This is a draft of code for thinking purposes.)

```
class Solution:
    def twoSum(self, numbers: List[int], target: int) → List[int]:
        # left, right = 0, len(numbers) - 1
        # while numbers[left] + numbers[right] != target:
        #    if numbers[left] + numbers[right] < target:
        #        left += 1
        #    else:
        #        right -= 1
        # return [left + 1, right + 1]
```

3.3 Implementation through python:

```
class Solution:
    def twoSum(self, numbers: List[int], target: int) → List[int]:
        #Intialize the left and right pointers
        left = 0
        right = len(numbers) - 1
        # while-loop to find the two numbers
        while numbers[left] + numbers[right] != target:
            # check if the sum is less than the target
            if numbers[left] + numbers[right] < target:
                left += 1
            # check if the sum is greater than the target
            else:
                right -= 1
        return [left + 1, right + 1]
```

3.4 Time Complexity and Space Complexity

3.4.1 Time Complexity:

- O(n), where n is the length of the list, because we iterate once through the list.

3.4.2 Space Complexity:

- O(1) since we only use two pointers as extra variables.