# 167. Two Sum II - Input Array Is Sorted

Medium Array Two Pointers Binary Search

## Problem Description

Given a sorted array of integers `numbers` in non-decreasing order, the objective is to find two distinct numbers within the array that sum up to a particular `target`. The array is "1-indexed," meaning that indexing begins at 1, not 0 as it usually does in programming languages. We need to return the indices of these two numbers such that `index1 < index2`, incremented by one to account for the 1-indexing, in the format of an array `[index1, index2]`.

## Intuition

When presented with a sorted array, we have a significant advantage because the nature of the sorting gives us directional choices. If we are looking for two numbers that add up to the target, then as we pick any two numbers, we can immediately know if we need a larger or a smaller number by comparing their sum to the target.

**Intuition for Two Pointers Solution:** Two pointers can efficiently solve this problem since the array is already sorted. Start the pointers at opposite ends (`i` at the start and `j` at the end). If their sum is too small, we move the `i` pointer to the right to increase the sum. If their sum is too large, we move the `j` pointer to the left to decrease the sum. This is possible because the array is sorted, so moving the `i` pointer to the right will only increase the sum, and moving the `j` pointer to the left will only decrease it. We continue this process until the pointers' sum equals the target, at which point we have found the solution. Since there is exactly one solution, this approach will always work.

This solution is effective and intuitive because it takes advantage of the sorted nature of the array and eliminates the need for nested loops, which would increase the time complexity. Thus, our approach results in a linear time solution.

## Solution Approach

The solution utilizes a well-known pattern in algorithm design known as the "two-pointer technique." Here's how we implement this pattern to solve our problem:

1. **Initialization**: We start with two pointers, `i` and `j`. The pointer `i` is initialized to 0, the beginning of the array, and `j` to `len(numbers) - 1`, the end of the array.
2. **Iteration**: We enter a loop where `i` moves from the start towards the end, and `j` moves from the end towards the start. The loop continues as long as `i` is less than `j`, ensuring we only work with pairs where `index1 < index2`.
3. **Sum and Compare**: In each iteration, we calculate the sum `x` of `numbers[i]` and `numbers[j]`. We then compare `x` with the `target`.
4. **Decision Making**:
   - If `x` equals the `target`, we have found the correct indices. Since the problem specifies 1-based indices, we return `[i + 1, j + 1]`.
   - If `i` is less than the target, it means we need a larger number to reach the target. We increment `i` (`i += 1`) to move to the next larger number, because the array is sorted in non-decreasing order.
   - If `x` is greater than the target, we need a smaller number. We decrement `j` (`j -= 1`) to consider the next smaller number for a similar reason.

This approach only uses constant extra space, thus adhering to the space complexity constraints of the problem.

Note: The solution provided in the reference uses two different approaches: `[Binary Search](/problems/binary-search-speedrun)` and `[Two Pointers](/problems/two_pointers_intro)`. The binary search approach is not reflected in the implementation provided, as binary search would introduce a `log n` factor to the time complexity, making the overall complexity `O(n log n)`, which is less efficient than the two-pointer approach that operates in `O(n)` time.

### Example Walkthrough

Let's walk through a small example to illustrate the solution approach described above.

Imagine we have the following sorted array `numbers` and we are given the `target` 9:

```
numbers: [1, 2, 4, 4, 6]
target: 9
```

#### Initialization

According to the two-pointer technique, we start with:

- Pointer `i` at `numbers[0]` (the first element).
- Pointer `j` at `numbers[4]` (the last element).

#### Iteration and Decision Making

We will now iterate and use our decision-making process:

1. In the first iteration, `numbers[i]` is 1 and `numbers[j]` is 6. The sum equals 7, which is less than the target 9.
   - We increment `i` to move to the next larger number.
2. Now `i` points to `numbers[1]` (2) and `j` still points to `numbers[4]` (6). The sum is 8, which is again less than the target.
   - We increment `i` once more.
3. `i` now points to `numbers[2]` (4) and `j` to `numbers[4]` (6). Their sum is 10, which is greater than the target.
   - We decrement `j` to move to the next smaller number.
4. `i` remains pointing to `numbers[2]` (4), and `j` moves to `numbers[3]` (also 4). The sum is 8, which is less than the target.
   - We increment `i`.
5. Now both `i` and `j` point to `numbers[3]` (the second 4 in the array). The sum of `numbers[i]` and `numbers[j]` is 8, which is less than the target, but since `i` cannot be equal to `j`, we increment `i`.
6. After the previous step, `i` exceeds `j`, and thus the loop ends without finding the exact pair. However, for this example walkthrough, the correct pair is the two 4s at positions 3 and 4 which we incidentally skipped, due to our strict reading of 'distinct numbers'. If by 'distinct numbers' the original statement meant distinct indices (which are holding possibly equal values, like the two 4s in this example), we should increment `i` again after step 4. Let's correct this:
7. After correcting, `i` will point to `numbers[3]` and `j` to `numbers[4]`, both are 4. Their sum is now 8, which is exactly our target of 8.
   - We return `[i + 1, j + 1]` which corresponds to `[3 + 1, 4 + 1]` or `[4, 5]` in a 1-indexed array.

There we have our solution following the two-pointer approach: indices `[4, 5]` of the input array `numbers` contain the numbers that sum up to the `target` 9.

Note: The hypothetical array and target were chosen for illustrative purposes. The actual input array and target might not have this same issue regarding 'distinct numbers'.

## Solution Implementation

### Python

```python
# Define a class named Solution
class Solution:
    # Define a method that finds the two indices of the numbers that add up to the target sum
    def twoSum(self, numbers: List[int], target: int) -> List[int]:
        # Initialize two pointers, one at the beginning and one at the end of the array
        left_pointer, right_pointer = 0, len(numbers) - 1

        # Iterate through the array until the two pointers meet
        while left_pointer < right_pointer:
            # Calculate the sum of the two numbers at the current pointers
            current_sum = numbers[left_pointer] + numbers[right_pointer]

            # If the sum equals the target, return the indices (1-based)
            if current_sum == target:
                return [left_pointer + 1, right_pointer + 1]

            # If the sum is less than the target, move the left pointer to the right to increase the sum
            if current_sum < target:
                left_pointer += 1
            # If the sum is greater than the target, move the right pointer to the left to decrease the sum
            else:
                right_pointer -= 1
        # Return an empty list if no two numbers sum up to the target (though the problem guarantees a solution)
        return []
```

### Java

```java
class Solution {

    public int[] twoSum(int[] numbers, int target) {

        // Initialize pointers for the two indices to be checked
        int left = 0;                   // Starting from the beginning of the array
        int right = numbers.length - 1; // Starting from the end of the array

        // Loop continues until the correct pair is found
        while (left < right) {
            // Calculate the sum of the elements at the left and right indices
            int sum = numbers[left] + numbers[right];

            // Check if the sum is equal to the target
            if (sum == target) {
                // Return the indices of the two numbers.
                // Incremented by one to match the problem's one-based indexing requirement
                return new int[] {left + 1, right + 1};
            }

            // If the sum is less than the target, increment the left index to increase the sum
            if (sum < target) {
                left++;
            } else {
                // If the sum is greater than the target, decrement the right index to decrease the sum
                right--;
            }
        }

        // The problem statement guarantees that exactly one solution exists.
        // so the following statement is unreachable. This return is used to satisfy the syntax requirements.
        return new int[] {-1, -1};
    }
}
```

### C++

```cpp
#include <vector> // Include the vector header for using the vector container.

// Define our own Solution class.
class Solution {
public:
    // `twoSum` function to find the indices of the two numbers from `numbers` vector that add up to a specific `target`.
    std::vector<int> twoSum(std::vector<int>& numbers, int target) {
        // Initialize two pointers: one at the start (`left`) and one at the end (`right`) of the vector.
        int left = 0, right = numbers.size() - 1;

        // Loop until the condition is true, which is an indefinite loop here because we expect to always find a solution.
        while(true) {
            // Calculate the sum of the elements at the `left` and `right` pointers.
            int sum = numbers[left] + numbers[right];

            // If the sum is equal to the target, return the indices of the two numbers.
            // The problem statement may assume that indices are 1-based, so we add 1 to each index.
            if (sum == target) {
                return {left + 1, right + 1};
            }

            // If sum is less than the target, we move the `left` pointer to the right to increase the sum.
            if (sum < target) {
                left++;
            // If the sum is greater than the target, we move the `right` pointer to the left to decrease the sum.
            } else {
                right--;
            }
        }

        // Note: The initial implementation assumes there will always be a solution before the loop ends,
        // and as such doesn't have a mechanism to return a value if there is no solution. This may be something
        // to consider in a real-world application.
    }
};
```

### TypeScript

```typescript
/**
 * Finds two numbers in a sorted array whose sum equals a specific target number.
 *
 * @param {number[]} numbers - A sorted array of numbers.
 * @param {number} target - The target sum to find.
 * @returns {number[]} An array containing the 1-based indices of the two numbers that add up to the target.
 */
function twoSum(numbers: number[], target: number): number[] {
    // Initialize two pointers, one at the start and the other at the end of the array.
    let startIndex = 0;
    let endIndex = numbers.length - 1;

    // Iterate through the array until the two pointers meet.
    while (startIndex < endIndex) {
        // Calculate the sum of the values at the two pointers.
        const sum = numbers[startIndex] + numbers[endIndex];

        // If the sum is equal to the target, return the indices (1-based).
        if (sum === target) {
            return [startIndex + 1, endIndex + 1];
        }

        // If the sum is less than the target, move the start pointer to the right.
        if (sum < target) {
            ++startIndex;
        } else {
            // If the sum is greater than the target, move the end pointer to the left.
            --endIndex;
        }
    }

    // If no two numbers sum up to the target, return an empty array
    // In the context of this question, a solution is guaranteed so this line is unlikely to be reached.
    return [];
}
```

```python
# Define a class named Solution
class Solution:
    # Define a method that finds the two indices of the numbers that add up to the target sum
    def twoSum(self, numbers: List[int], target: int) -> List[int]:
        # Initialize two pointers, one at the beginning and one at the end of the array
        left_pointer, right_pointer = 0, len(numbers) - 1

        # Iterate through the array until the two pointers meet
        while left_pointer < right_pointer:
            # Calculate the sum of the two numbers at the current pointers
            current_sum = numbers[left_pointer] + numbers[right_pointer]

            # If the sum equals the target, return the indices (1-based)
            if current_sum == target:
                return [left_pointer + 1, right_pointer + 1]

            # If the sum is less than the target, move the left pointer to the right to increase the sum
            if current_sum < target:
                left_pointer += 1
            # If the sum is greater than the target, move the right pointer to the left to decrease the sum
            else:
                right_pointer -= 1
        # Return an empty list if no two numbers sum up to the target (though the problem guarantees a solution)
        return []
```

## Time and Space Complexity

The algorithm uses a two-pointer approach to find two numbers that add up to the target value. The pointers start at the beginning and end of the sorted list and move towards each other.

The time complexity of the algorithm is `O(n)` because in the worst case, the left pointer `i` might have to move all the way to the second to last element, and the right pointer `j` might move to the second element. Each pointer can traverse the list at most once, which results in a linear time complexity with respect to the length of the input list `numbers`.

The space complexity of the algorithm is `O(1)`. This is because the algorithm only uses a constant amount of extra space: the two pointers `i` and `j`, and the variable `x` for the current sum. No additional space that is dependent on the input size is used, which keeps the space complexity constant.