<https://leetcode.com/problems/two-sum-ii-input-array-is-sorted/description/>

Given a **1-indexed** array of integers numbers that is already ***sorted in non-decreasing order***, find two numbers such that they add up to a specific target number. Let these two numbers be numbers[index1] and numbers[index2] where 1 <= index1 < index2 < numbers.length.

Return *the indices of the two numbers,* index1 *and* index2*,* ***added by one*** *as an integer array* [index1, index2] *of length 2.*

The tests are generated such that there is **exactly one solution**. You **may not** use the same element twice.

Your solution must use only constant extra space.

**Example 1:**

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

Output: [1,2]

Explanation: The sum of 2 and 7 is 9. Therefore, index1 = 1, index2 = 2. We return [1, 2].

**Example 2:**

Input: numbers = [2,3,4], target = 6

Output: [1,3]

Explanation: The sum of 2 and 4 is 6. Therefore index1 = 1, index2 = 3. We return [1, 3].

**Example 3:**

Input: numbers = [-1,0], target = -1

Output: [1,2]

Explanation: The sum of -1 and 0 is -1. Therefore index1 = 1, index2 = 2. We return [1, 2].

**Constraints:**

* 2 <= numbers.length <= 3 \* 104
* -1000 <= numbers[i] <= 1000
* numbers is sorted in **non-decreasing order**.
* -1000 <= target <= 1000
* The tests are generated such that there is **exactly one solution**.

**Attempt 1: 2023-11-24**

**Solution 1: Binary Search (10 min)**

class Solution {

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

int n = numbers.length;

for(int i = 0; i < n; i++) {

int num = numbers[i];

int val = target - num;

int lo = i + 1;

int hi = n - 1;

while(lo <= hi) {

int mid = lo + (hi - lo) / 2;

if(numbers[mid] == val) {

return new int[]{i + 1, mid + 1};

} else if(numbers[mid] > val) {

hi = mid - 1;

} else {

lo = mid + 1;

}

}

}

return new int[2];

}

}

Time Complexity: O(N\*logN)

Space Complexity: O(1)

**Solution 2: Two Pointers (10 min)**

class Solution {

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

int n = numbers.length;

int i = 0;

int j = n - 1;

while(i < j) {

if(numbers[i] + numbers[j] == target) {

return new int[]{i + 1, j + 1};

} else if(numbers[i] + numbers[j] > target) {

j--;

} else {

i++;

}

}

return new int[2];

}

}

Time Complexity: O(N)

Space Complexity: O(1)

**Refer to**

<https://grandyang.com/leetcode/167/>

这又是一道[Two Sum](http://www.cnblogs.com/grandyang/p/4130379.html)的衍生题，作为LeetCode开山之题，我们务必要把Two Sum及其所有的衍生题都拿下，这道题其实应该更容易一些，因为给定的数组是有序的，而且题目中限定了一定会有解，我最开始想到的方法是二分法来搜索，因为一定有解，而且数组是有序的，那么第一个数字肯定要小于目标值target，那么我们每次用二分法来搜索target - numbers[i]即可，代码如下：

// O(nlgn)

class Solution {

public:

vector<int> twoSum(vector<int>& numbers, int target) {

for (int i = 0; i < numbers.size(); ++i) {

int t = target - numbers[i], left = i + 1, right = numbers.size();

while (left < right) {

int mid = left + (right - left) / 2;

if (numbers[mid] == t) return {i + 1, mid + 1};

else if (numbers[mid] < t) left = mid + 1;

else right = mid;

}

}

return {};

}

};

但是上面那种方法并不efficient，时间复杂度是O(nlgn)，我们再来看一种O(n)的解法，我们只需要两个指针，一个指向开头，一个指向末尾，然后向中间遍历，如果指向的两个数相加正好等于target的话，直接返回两个指针的位置即可，若小于target，左指针右移一位，若大于target，右指针左移一位，以此类推直至两个指针相遇停止，参见代码如下：

// O(n)

class Solution {

public:

vector<int> twoSum(vector<int>& numbers, int target) {

int l = 0, r = numbers.size() - 1;

while (l < r) {

int sum = numbers[l] + numbers[r];

if (sum == target) return {l + 1, r + 1};

else if (sum < target) ++l;

else --r;

}

return {};

}

};