Two Sum II - Input array is sorted

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.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.

You may assume that each input would have exactly one solution.

```
Input: numbers={2, 7, 11, 15}, target=9
Output: index1=1, index2=2
```

Solution 1

Without HashMap, just have two pointers, A points to index 0, B points to index len - 1, shrink the scope based on the value and target comparison.

```java
public int[] twoSum(int[] num, int target) {
    int[] indice = new int[2];
    if (num == null || num.length < 2) return indice;
    int left = 0, right = num.length - 1;
    while (left < right) {
        int v = num[left] + num[right];
        if (v == target) {
            indice[0] = left + 1;
            indice[1] = right + 1;
            break;
        } else if (v > target) {
            right --;
        } else {
            left ++;
        }
    }
    return indice;
}
```

written by titanduan3 original link here

Solution 2

I know that the best solution is using two pointers like what is done in the previous solution sharing. However, I see the tag contains "binary search". I do not know if I misunderstand but is binary search a less efficient way for this problem.

Say, fix the first element A[0] and do binary search on the remaining n-1 elements. If cannot find any element which equals target-A[0], Try A[1]. That is, fix A[1] and do binary search on A[2]~A[n-1]. Continue this process until we have the last two elements A[n-2] and A[n-1].

Does this gives a time complexity lg(n-1) + lg(n-2) + ... + lg(1) ~ O(lg(n!)) ~ O(nlgn). So it is less efficient than the O(n) solution. Am I missing something here?

The code also passes OJ.

```
vector<int> twoSum(vector<int> &numbers, int target) {
    if(numbers.empty()) return {};
    for(int i=0; i<numbers.size()-1; i++) {
        int start=i+1, end=numbers.size()-1, gap=target-numbers[i];
        while(start <= end) {
            int m = start+(end-start)/2;
            if(numbers[m] == gap) return {i+1,m+1};
            else if(numbers[m] > gap) end=m-1;
            else start=m+1;
        }
    }
}
```

written by morrischen2008 original link here

Solution 3

My algorithm is O(n), but runs 8ms, I am just wondering whether there is more efficient algorithm?

written by applelooking original link here