Pair with Target Sum (easy)

Problem Statement

Given an array of sorted numbers and a target sum, find a **pair in the array whose sum is equal to the given target**.

Write a function to return the indices of the two numbers (i.e. the pair) such that they add up to the given target.

Example 1:

```
Input: [1, 2, 3, 4, 6], target=6
Output: [1, 3]
Explanation: The numbers at index 1 and 3 add up to 6: 2+4=6
```

Example 2:

```
Input: [2, 5, 9, 11], target=11
Output: [0, 2]
Explanation: The numbers at index 0 and 2 add up to 11: 2+9=11
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
const pair_with_targetsum = function(arr, target_sum) {
  // TODO: Write your code here
  return [-1, -1];
}

```

TEST | SAVE | RESET

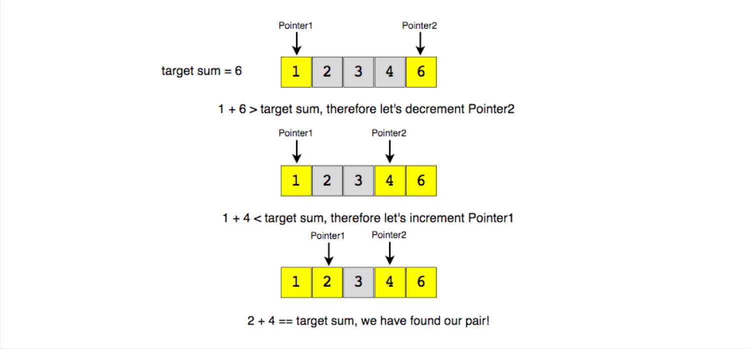
Solution

Since the given array is sorted, a brute-force solution could be to iterate through the array, taking one number at a time and searching for the second number through **Binary Search**. The time complexity of this algorithm will be $O(N * logN)$. Can we do better than this?

We can follow the **Two Pointers** approach. We will start with one pointer pointing to the beginning of the array and another pointing at the end. At every step, we will see if the numbers pointed by the two pointers add up to the target sum. If they do, we have found our pair; otherwise, we will do one of two things:

1. If the sum of the two numbers pointed by the two pointers is greater than the target sum, this means that we need a pair with a smaller sum. So, to try more pairs, we can decrement the end-pointer.
2. If the sum of the two numbers pointed by the two pointers is smaller than the target sum, this means that we need a pair with a larger sum. So, to try more pairs, we can increment the start-pointer.

Here is the visual representation of this algorithm for Example-1:


Pointer1
Pointer2
target sum = 6
1
2
3
4
6
1 + 6 > target sum, therefore let's decrement Pointer2
Pointer1
Pointer2
1
2
3
4
6
1 + 4 < target sum, therefore let's increment Pointer1
Pointer1
Pointer2
1
2
3
4
6
2 + 4 == target sum, we have found our pair!

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

MW
Explore
Tracks
My Courses
Edpresso
Refer a Friend
Create

```
function pair_with_target_sum(arr, targetSum) {
  let left = 0,
    right = arr.length - 1;
  while (left < right) {
    const currentSum = arr[left] + arr[right];
    if (currentSum === targetSum) {
      return [left, right];
    }

    if (targetSum > currentSum) {
      left += 1; // we need a pair with a bigger sum
    } else {
      right -= 1; // we need a pair with a smaller sum
    }
  }
  return [-1, -1];
}


console.log(pair_with_target_sum([1, 2, 3, 4, 6], 6));
console.log(pair_with_target_sum([2, 5, 9, 11], 11));
```

RUN SAVE RESET

Time Complexity

The time complexity of the above algorithm will be $O(N)$, where 'N' is the total number of elements in the given array.

Space Complexity

The algorithm runs in constant space $O(1)$.

An Alternate approach

Instead of using a two-pointer or a binary search approach, we can utilize a **HashTable** to search for the required pair. We can iterate through the array one number at a time. Let's say during our iteration we are at number 'X', so we need to find 'Y' such that "$X + Y == Target$". We will do two things here:

1. Search for 'Y' (which is equivalent to "$Target - X$") in the **HashTable**. If it is there, we have found the required pair.
2. Otherwise, insert "X" in the **HashTable**, so that we can search it for the later numbers.

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
function pair_with_target_sum(arr, targetSum) {
  const nums = {}; // to store numbers and their indices
  for (let i = 0; i < arr.length; i++) {
    const num = arr[i];
    if (targetSum - num in nums) {
      return [nums[targetSum - num], i];
    }
    nums[arr[i]] = i;
  }
  return [-1, -1];
}


console.log(pair_with_target_sum([1, 2, 3, 4, 6], 6));
console.log(pair_with_target_sum([2, 5, 9, 11], 11));
```

RUN SAVE RESET

Time Complexity

The time complexity of the above algorithm will be $O(N)$, where 'N' is the total number of elements in the given array.

Space Complexity

The space complexity will also be $O(N)$, as, in the worst case, we will be pushing 'N' numbers in the **HashTable**.

☑ MARK AS COMPLETED

← Back
Introduction

Next →
Remove Duplicates (easy)

Report an Issue   Ask a Question