Pair with Target Sum (easy)

We'll cover the following

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++

```
class PairWithTargetSum {

  public static int[] search(int[] arr, int targetSum) {
    // TODO: Write your code here
    return new int[] { -1, -1 };
  }
}
```

Test | Save | Reset

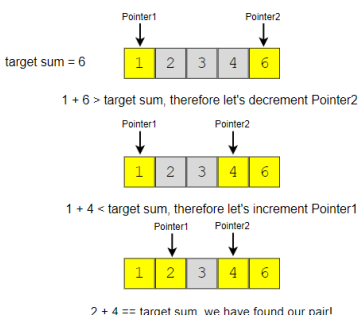
Solution

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:



Code

Here is what our algorithm will look like:

Java Python3 C++ JS

```java
class PairWithTargetSum {

  public static int[] search(int[] arr, int targetSum) {
    int left = 0, right = arr.length - 1;
    while (left < right) {
      int currentSum = arr[left] + arr[right];
      if (currentSum == targetSum)
        return new int[] { left, right }; // found the pair

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

  public static void main(String[] args) {
    int[] result = PairWithTargetSum.search(new int[] { 1, 2, 3, 4, 6 }, 6);
    System.out.println("Pair with target sum: [" + result[0] + ", " + result[1] + "]");
    result = PairWithTargetSum.search(new int[] { 2, 5, 9, 11 }, 11);
    System.out.println("Pair with target sum: [" + result[0] + ", " + result[1] + "]");
  }
}
```

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

```java
import java.util.HashMap;

class PairWithTargetSum {

  public static int[] search(int[] arr, int targetSum) {
    HashMap<Integer, Integer> nums = new HashMap<>(); // to store numbers and their indices
    for (int i = 0; i < arr.length; i++) {
      if (nums.containsKey(targetSum - arr[i]))
        return new int[] { nums.get(targetSum - arr[i]), i };
      else
        nums.put(arr[i], i); // put the number and its index in the map
    }
    return new int[] { -1, -1 }; // pair not found
  }

  public static void main(String[] args) {
    int[] result = PairWithTargetSum.search(new int[] { 1, 2, 3, 4, 6 }, 6);
    System.out.println("Pair with target sum: [" + result[0] + ", " + result[1] + "]");
    result = PairWithTargetSum.search(new int[] { 2, 5, 9, 11 }, 11);
    System.out.println("Pair with target sum: [" + result[0] + ", " + result[1] + "]");
  }
}

```

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**.