Two Sum

Problem Statement: Given an array of integers nums[] and an integer target, return *indices of the two numbers such that their sum is equal to the target*.

Note: Assume that there is ***exactly* one solution**, and you are not allowed to use the *same* element twice. Example: If target is equal to 6 and num[1] = 3, then nums[1] + nums[1] = target is not a solution.

Example 1:

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

Output: [0,1]

Explanation: Because nums[0] + nums[1] == 9, which is the required target, we return indexes [0,1]. (0-based indexing)

Example 2:

Input Format: nums = [3,2,4,6], target = 6

Output: [1,2]

Explanation: Because nums[1] + nums[2] == 6, which is the required target, we return indexes [1,2].

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1: Naive Approach (Brute Force)

Intuition: For each element, we try to find an element such that the sum of both elements is equal to the given target.

Approach: We traverse through the array, and for each element **i**, we try to find another element amongst the remaining elements, such that the sum of both the elements equals the target.

First Element: **i**

So the required second element will be, **target – i**

If we find both the elements, we break the loop and return the indices.

Dry Run: Given array, nums = [2,1,3,4], target = 4

Using the naive approach, we first select one element and then find the second elements.

For index 0, i = 2

Then, we iterate through index 1 to 3 to find if **target – i,** i.e. 4 – 2 = 2 exists. We find that it does not exist, so we move forward.

For index 1, i=1

Then, we iterate through index 2 to 3 to find if **target – i,** i.e. 4 – 1 = 3 exists. We find that it does exist at index 2, so we store the indices 1 and 2, break the loop, and return the indices.

CODE:

```
public int[] twoSum(int[] nums, int target) {
   int[] res=new int[2];
      Arrays.fill(res,-1);
   for (int i = 0; i < nums.length; ++i) {
         for (int j = i + 1; j < nums.length; ++j) {
                  if (nums[i] + nums[j] == target) {
                           res[0]=i;
                           res[1]=j;
                           break;
                  }
```

```
        }
        if (res[0] != -1)
                break;
    }
    return res;
    }
```

Time Complexity: O(N^2)

Space Complexity: O(1)

Solution 2: Two-Pointer Approach

Intuition: Think about, what if the array is sorted? If the array is sorted, is it possible to reach a sum by traversing the array from both sides simultaneously?

We sort the array, use two variables, each will start from one end of the array, and traverse in both directions till we get the required sum.

Approach: We traverse through the array, and for each element **i**, we try to find another element amongst the remaining elements, such that the sum of both the elements equals the target.

First Element: **i**

So the required second element will be, **target – i**

If we find both the elements, we break the loop and return the indices.

Dry Run: Given array, nums = [2,1,3,4], target = 4

First we sort the array. So nums after sorting is [1,2,3,4]

We take two pointers, i and j. i points to index 0 and j points to index 3.

Now we check if nums[i] + nums[j] == target. In this case, they don't sum up, and nums[i] + nums[j] > target, so we will reduce j by 1.

Now, i = 0, j=2

Here, nums[i] + nums[j] == 1 + 3 == 4, which is the required target, so we store both the elements and break the loop.

Now, we run another loop to find the indices of the elements in the actual array, i.e [2,1,3,4]

Then, return the actual indices, [1,2].

Source code

```
public int[] twoSum(int[] nums, int target) {
    int[] res=new int[2];
    int[] store=Arrays.copyOfRange(nums,0,nums.length);
    Arrays.sort(store);
      int left=0,right=nums.length-1;
      int n1=0,n2=0;

      while(left<right){
      if(store[left]+store[right]==target){

            n1 = store[left];
            n2 = store[right];

            break;

      }
      else if(store[left]+store[right]>target)
            right--;
```

```
        else
                left++;
        }

        for(int i=0;i<nums.length;++i){

        if(nums[i]==n1)
                res[0]=i;
        else if(nums[i]==n2)
                res[1]=i;
        }

           return res;
    }
```

Time Complexity: O(NlogN)

Space Complexity: O(N)

Solution 3: Hashing (Most efficient)

Approach: We can solve this problem efficiently by using **hashing**. We'll use a **hash-map** to see if there's a value **target – i** that can be added to the current array value i to get the sum equals to target. If **target – i** is found in the map, we return the current index, and index stored at **target – nums[index]** location in the map.

This can be done in constant time.

Dry Run: Given array, nums = [2,3,1,4], target = 4

For index 0, i = 2

We try to find if **target – i = 4 – 2 = 2** is present in the map. It is not present in this case, so we put 0 for key 2 in the map.

For index 1, i = 3

We try to find if **target – i = 4 – 3 = 1** is present in the map. We find that it is also not present, so we put 1 for key 3 in the map.

For index 2, i = 1

We try to find if **target – i = 4 – 1 = 3** is present in the map. We find that it is present, so we store index 2 and value stored for key 3 in the map, and break the loop.

And return [1,2].

Source Code

```
public int[] twoSum(int[] nums, int target) {
   int[] res=new int[2];
   HashMap<Integer,Integer> mp=new HashMap<>();

   for (int i = 0; i < nums.length; ++i) {

        if (mp.containsKey(target - nums[i])) {

                res[0]=i;
      res[1 ]=mp.get( target-nums[i]);
                return res;
        }
   mp.put(nums[i],i);
```

```
    }


    return res;
    }
```

Time Complexity: O(N)

Space Complexity: O(N)