Two Sum

<https://leetcode.com/problems/two-sum/>

1. **Listen**

**Problem Statement:**

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to*target*.

You may assume that each input would have **exactly one solution**, and you may not use the same element twice.

You can return the answer in any order.

**Input:**

1. Array of integers nums
2. Integer target

**Goal:**

Find **two numbers** in nums that add up to target

**Return:**

Return the **indices** of the two numbers in nums that add up to target

1. **Examples**

**Example 1:**

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

**Output:** [0,1]

**Explanation:** Because nums[0] + nums[1] == 9, we return [0, 1].

**Example 2:**

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

**Output:** [1,2]

**Constraints:**

* Each input has **exactly one solution**
* You may not use the same element twice.
* 2 <= nums.length <= 104
* -109 <= nums[i] <= 109
* -109 <= target <= 109
* Only one valid answer exists.

**Test Cases:**

* Array of length 2
* Array of length > 2 both numbers are positive
* Array of length > 2 both numbers are negative
* Array of length > 2 both numbers are mixed

1. **Brute Force**

**Solution 1: Time = O(N^2), Space = O(1)**

Iterate over the array and determine if two numbers are equal.

We test each element, i, against the rest of the array to see if [i] + [some element from i – n-1] adds up to target number.

**Solution 2: Time = O(NlogN), Space = O(N)**

In order to improve on the previous solution, we could use extra space while reducing the runtime by a factor of N.

We can sort the input array.

Use a two-pointer solution: Put one pointer on each end of the array.

A pointer to the front, which will iterate forwards

A pointer at the back, which will iterate backwards

If front + back < target, then move the front pointer up one index.

If front + back > target, then move the back pointer back one index.

This solution will always work when the input array is **already sorted**.

However, if the input array is not sorted, then sorting the values will cause us to lose the original indices of each element.

While this solution is great for finding the two **values** that add up to **target**, the problem description requires us to find the **indices** of the values that add up to target.

Therefore, once we find the values, we need some way to lookup their original indices, because the original indices of each value are **lost after sorting**.

Before we sort and search the array, we can store the input array’s original values and associated indices in a 2D array.

Each element in the 2D array will be an array of size 2. The first being the value (so we can index it later in the two-pointer section), and the second being the index.

We can then sort this array by increasing order using a second argument to Arrays.sort().

1. **Optimize**

**Solution 3: Time = O(N), Space = O(N)**

Iterate over the array.

Use a hashmap to store ‘seen’ values (elements we have iterated over)

If (current element value – target) is present in the hashmap, then we can return the respective indices.

If (current element value – target) is not present in the hashmap, then add the current element to the hashmap as <key = element value, value = index>

1. **Implement**

**Solution 1:**

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

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

{

for(int j = i + 1; i < nums.length; i++)

{

if(nums[i] + nums[j] == target)

return new int[] { i, j };

}

}

return null;

}

**Solution 2:**

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

// store original values : indices

int n = nums.length;

int[][] arr = new int[n][2];

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

arr[i][0] = nums[i];

arr[i][1] = i;

}

Arrays.sort(arr, Comparator.comparingInt(o -> o[0]));

int front = 0, back = nums.length-1;

while(front < back)

{

int sum = arr[front][0] + arr[back][0];

if(sum == target)

return new int[] {arr[front][1], arr[back][1]};

else if(sum < target)

front++;

else

back--;

}

return null;

}

**Solution 3:**

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

HashMap<Integer, Integer> map = new HashMap<>();

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

{

int op1 = nums[i];

int op2 = target - nums[i];

if(map.containsKey(op2))

return new int[]{i, map.get(op2)};

else

map.put(nums[i], i);

}

return null;

}

1. **Test**