TWO SUM

Source: https://leetcode.com/problems/two-sum/editorial/

PROBLEM OVERVIEW

Given an array of integers *nums* and an integer *target*, the task is to find the índices of the two numbers such that they add up to the target value. It is guaranteed that exactly one such pair exists, and the same element cannot be used twice.

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]

// EXAMPLE 3:
//Input: nums = [3,3], target = 6
//Output: [0,1]
```

CONSTRAINTS

- 2 <= nums.length <= 10^4
- -10^9 <= nums[i] <= 10^9
- -10^9 <= target <= 10^9

APPROACH 1. Brute Force

Algorithm

The brute force approach is simple. Loop through each element *x* and find if there is another value that equals to *target* – *x*.

```
class Solution
{
  public:
    vector<int> twoSum(vector<int>& nums, int target)
    {
      for (int i = 0; i < nums.size()-1; i++)
      {
        for (int j = 1; j < nums.size(); j++)
        {
          if (nums[i] + nums[j] == target && i != j)
            return {i, j};
        }
      }
      return {};
    }
};
```

```
class Solution
{
  public:
    vector<int> twoSum(vector<int>& nums, int target)
    {
      for (int i = 0; i < nums.size()-1; i++)
      {
        for (int j = 1; j < nums.size(); j++)
        {
          if (nums[i] + nums[j] == target && i != j)
            return {i, j};
        }
      }
      return {};
    }
};
```

Complexity Analysis

- **Time complexity: $O(n^2)$.**
  For each element, we try to find its complement by looping through the rest of the array which takes $O(n)$ time. Therefore, the time complexity is $O(n^2)$.
- **Space complexity: $O(1)$.**
  The space required does not depend on the size of the input array, so only constant space is used.

APPROACH 2. Two-pass Hash Table

Intuition

To improve our runtime complexity, we need a more efficient way to check if the complement exists in the array. If the complement exists, we need to get its index. What is the best way to maintain a mapping of each element in the array to its index? A **hash table**.

We can reduce the lookup time from $O(n)$ to $O(1)$ by trading space for speed. A hash table is well suited for this purpose because it supports fast lookup in near constant time. If a collision occurred, a lookup could degenerate to $O(n)$ time. However, lookup in a hash table should be amortized $O(1)$ time as long as the hash function was chosen carefully.

Algorithm

A simple implementation uses two iterations. In the first iteration, we add each element's value as a key and its index as a value to the hash table. Then, in the second iteration, we check if each element's complement (*target – nums[i]*) exists in the hash table. If it does exist, we return current element's index and its complement's index. Beware that the complement mus not be *nums[i]* itself.

```cpp
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target)
    {
        unordered_map<int, int> numMap;
        for (int i = 0; i < nums.size(); i++)
        {
            numMap[nums[i]] = i;
        }

        for (int i = 0; i < nums.size(); i++)
        {
            int aux = target - nums[i];
            if (numMap.count(aux) && numMap[aux] != i)
                return {i, numMap[aux]};
        }
        return {};
    }
};
```

```python
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        hashMap = {}
        for i in range(len(nums)):
            hashMap [nums[i]] = i
        for i in range(len(nums)):
            aux = target - nums[i]
            if aux in hashMap and hashMap[aux] != i:
                return [i, hashMap[aux]]
        return []
```

Complexity Analysis

- **Time complexity: $O(n)$.**
  We traverse the list containing *n* elements exactly twice. Since the hash table reduces the lookup time to $O(1)$, the overall time complexity is $O(n)$.
- **Space complexity: $O(n)$.**

The extra space required depends on the number of items sorted in the hash table, which stores exactly *n* elements.