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

Solution Article

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 $target - x$.

Implementation

C++ | Java | C | C# | JavaScript | Go | Python3 | TypeScript — Copy

```java
class Solution {
    public int[] twoSum(int[] nums, int target) {
        for (int i = 0; i < nums.length; i++) {
            for (int j = i + 1; j < nums.length; j++) {
                if (nums[j] == target - nums[i]) {
                    return new int[] { i, j };
                }
            }
        }
        // If no valid pair is found, return an empty array instead of null
        return new int[] {};
    }
}
```

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. I say "near" because 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 must not be $nums[i]$ itself!

Implementation

C++ | Java | C | C# | JavaScript | Go | Python3 | TypeScript — Copy

```java
class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            map.put(nums[i], i);
        }
        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];
            if (map.containsKey(complement) && map.get(complement) != i) {
                return new int[] { i, map.get(complement) };
            }
        }
        // In case there is no solution, return an empty array
        return new int[] {};
    }
}
```

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 stored in the hash table, which stores exactly n elements.

Approach 3: One-pass Hash Table

Algorithm

It turns out we can do it in one-pass. While we are iterating and inserting elements into the hash table, we also look back to check if current element's complement already exists in the hash table. If it exists, we have found a solution and return the indices immediately.

Implementation

C++ | Java | C | C# | JavaScript | Go | Python3 | TypeScript — Copy

```java
class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];
            if (map.containsKey(complement)) {
                return new int[] { map.get(complement), i };
            }
            map.put(nums[i], i);
        }
        // Return an empty array if no solution is found
        return new int[] {};
    }
}
```

Complexity Analysis

- Time complexity: $O(n)$.
  We traverse the list containing n elements only once. Each lookup in the table costs only $O(1)$ time.
- Space complexity: $O(n)$.
  The extra space required depends on the number of items stored in the hash table, which stores at most n elements.