Understanding the Two Sum Problem

Matthew Henschke Follow

Sep 30, 2020 · 5 min read ★

One popular interview question that may or may not be thrown at you in a technical interview is known as the “Two Sum Problem”. For example, the Two Sum has been known for appearing on Facebook and Google interviews, and was the first Leetcode problem that I successfully solved with all test cases passed. Moreover, my relentless technical interview prep journey began with Two Sum!

The Two Sum Problem can be considered a Leetcode classic that consists of different fundamental solutions. Constructing these solutions involves an

understanding of different techniques that will be discussed further below. Before continuing, I would absolutely recommend that you brush up on arrays, hash maps, time-complexity, and space-complexity in order to fully understand the problem. GeeksForGeeks has well-written articles on different interview problems and concepts and is a great resource for this. Let's now dive into this problem.

Problem Description: Given an array of integers, return indices of the two numbers such that they add up to a specific target.

It is important to note that you CANNOT use the same element in the array twice, but you can assume that there will only be ONE solution for each test case.

Example Test Cases

```
Input: nums = [1,4,10,-3], target = 14
Output: [1,2] or [2,1] # 4 + 10 = 14

Input: nums = [9,5,1,23], target = 10
Output: [0,2] or [2,0] # 9 + 1 = 10

Input: nums = [1,-2,5,10], target = -1
Output: [0,1] or [1,0] # 1 + -2 = -1
```

As you can see from the test cases above, the output returned in each test case is an array of the indices of the two numbers that add up to the target. Note that the order of the indices that are in the solution array is not important for this problem. Now, let's take a look at the different solutions of the Two Sum Problem using the Python3 programming language.

First Approach

The solution seems simple enough, right? We have a nested for loop where we look for the first pair of indices of two numbers that add up to the target value. However, even though this approach is as simple as it gets, is it the most optimal? Let's have a look at the time and space complexities below.

Time Complexity: $O(n^2)$

Space Complexity: O(1)

At first glance, you can see that this solution is favorable in regards to space since the space complexity is constant. It is constant because this algorithm does not rely on any additional data structures such as arrays or dictionaries. Even as the input array can grow bigger for other test cases, it is still not using any extra space.

On the other hand, the time complexity of this solution is a cause for concern with an $\mathbf{O(n^2)}$ run time. For each element in **nums**, this approach attempts to find its complement by iterating through the rest of the array that takes $\mathbf{O(n)}$ time. While this approach may not make a difference when it comes to small inputs, it can have a drastic effect on the run time with larger inputs since time complexity is growing quadratically. So, always be aware of nested for loops as these can be detrimental to the running time of your code!

A More Optimal Solution

Unlike the previous solution, there is no nested for loop used here which is going to make a BIG difference in time complexity later on. However, we are now using a new variable, **hash_map** which is a Python dictionary. We are utilizing **hash_map** as specified by its name as a hash map data structure. We are storing each element in **nums** as keys where their corresponding values are their indices.

For each element that we are iterating through in **nums**, we are storing their complements as keys in the hash map with their values being in the current index. Note that we are only storing these key value pairs if the current element is not a present key in our hash map. If the current element

exists as a key, then we can return an array with the current index and its corresponding value in the hash map since this element was inserted in a previous iteration. Let's now check out the time and space complexities of this new approach.

Time Complexity: O(n)

Space Complexity: O(n)

We have now improved the time complexity of our solution immensely with a run time of **O(n)**. Each element is only being iterated once and ONLY once as we are no longer looping through the entire array for each iteration. Also, the lookup time to see if an element exists in our hash map is near-constant time. It is very near-constant time because a lookup could take **O(n)** time if a collision took place in the hash map. As long as the hash function used by the hash map is chosen carefully where collisions are avoided, a lookup is amortized to **O(1)**.

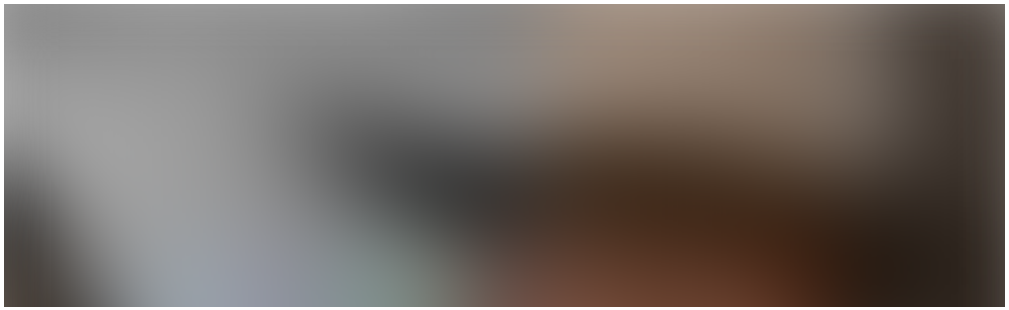
However, this improvement in performance comes at a significant cost. We now have a space complexity of **O(n)** as we are now utilizing a hash map in our solution as a storage buffer for our elements in the input. Now, we are sacrificing space for speed. This is a perfect example of the space-time

tradeoff where we can either increase space and decrease speed or vice-versa!

Conclusion

When tackling interview problems, it is really crucial to keep space complexity and time complexity in mind in regards to developing a solution. Moreover, it is important to ask yourself, “Can I do better?” every time you develop a solution to a technical problem so that you can improve the efficiency of your algorithm whether in speed or space improvements. No one wants to run a program that is very slow and uses a lot of memory; it is not efficient!

Now go give this problem a shot at https://leetcode.com/problems/two-sum/. If you are given this question during one of your interviews, consider yourself very lucky as you now understand the problem on a deeper level! Thank you so much for reading and I will catch you on my next post 😉



Software Engineering Interview Questions Leetcode Easy Two Sum