1. Two Sum

March 5, 2016 hash-table (/articles/?tag=hash-table)

Question | Editorial Solution

Question

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

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

Example:

```
Given nums = [2, 7, 11, 15], target = 9,

Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].
```

Quick Navigation

- Solution
  - Approach #1 (Brute Force) [Accepted]
  - Approach #2 (Two-pass Hash Table) [Accepted]
  - Approach #3 (One-pass Hash Table) [Accepted]

Solution

Approach #1 (Brute Force) [Accepted]

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

```
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 };
            }
        }
    }
    throw new IllegalArgumentException("No two sum solution");
}
```

Complexity Analysis

- Time complexity : $O(n^2)$. For each element, we try to find its complement by looping through the rest of array which takes $O(n)$ time. Therefore, the time complexity is $O(n^2)$.
- Space complexity : $O(1)$.

Approach #2 (Two-pass Hash Table) [Accepted]

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

We reduce the look up time from $O(n)$ to $O(1)$ by trading space for speed. A hash table is built exactly for this purpose, it supports fast look up in *near* constant time. I say "near" because if a collision occurred, a look up could degenerate to $O(n)$ time. But look up in hash table should be amortized $O(1)$ time as long as the hash function was chosen carefully.

A simple implementation uses two iterations. In the first iteration, we add each element's value and its index to the table. Then, in the second iteration we check if each element's complement ($target - nums[i]$) exists in the table. Beware that the complement must not be $nums[i]$ itself!

```
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) };
        }
    }
    throw new IllegalArgumentException("No two sum solution");
}
```

Complexity Analysis:

- Time complexity : $O(n)$. We traverse the list containing n elements exactly twice. Since the hash table reduces the look up time to $O(1)$, the 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) [Accepted]

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

```
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);
    }
    throw new IllegalArgumentException("No two sum solution");
}
```

Complexity Analysis:

- Time complexity : $O(n)$. We traverse the list containing n elements only once. Each look up 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.

‹ Previous (/articles/reverse-linked-list/)
Next › (/articles/first-bad-version/)

FFFate commented 2 weeks ago

@Khue (https://discuss.leetcode.com/uid/239155)
if you had run the code you would find it's right. In fact map.put(K,V) just update the V using the same K which means that there's only one index of the duplicated values in the map.

liscuss.leetcode.com/user/fffate)

Khue commented 2 weeks ago

Agree with Alecsvan on solution #2. Keys need to be unique and if there are repeating number in the array, this will throw an error. I've been working on reversing key and value as a work around, but cannot find an easy way to get a key for hash table using the value, without adding another loop. Suggestions?

liscuss.leetcode.com/user/khue)

Alecsvan commented 3 weeks ago

Approach 2 does not work for duplicated values (e.g. [3,3] 6) as Map cannot contain more than one value per key.

liscuss.leetcode.com/user/alecsvan)

ApyZ commented last month

It cannot be done in nlogn, unless the array is guaranteed to be pre-sorted which is apparently not he case.

liscuss.leetcode.com/user/apyz)

abhishekkarn96 commented last month

can be done in nlogn too

liscuss.leetcode.com/user/abhishekkarn96)

michael.kelly.scott commented last month

The test cases for this problem are obviously not good, because the fastest accepted solution listed on the leaderboard was the brute force algorithm.

liscuss.leetcode.com/user/michael-

Su2017 commented last month

@ChenxiaoNiu (https://discuss.leetcode.com/uid/222140) Why you think the algorithm in approach#3 is finding one element twice? lets say array = {1,3,2,4,8} and target is 9. As we iterate, we get 1 first from the array. The complement is (9-1) =8, we would not find that in the map now. But when we reach 8, we would definitely find 1 in the map.

liscuss.leetcode.com/user/su2017)

ChenxiaoNiu commented last month

Approach #3. How to avoid the algorithm finding one element twice?

liscuss.leetcode.com/user/chenxiaoniu)

View original thread (https://discuss.leetcode.com/topic/29)

Load more comments...