How to Solve Two Sum in JavaScript

Solve the classic interview problem in linear time without sorting.

Jordan Moore Follow
Nov 24, 2019 · 7 min read ★

Two Sum — It's possibly one of the most prolific algorithm questions in existence, and as such, should be practiced extensively.

Even though it is a common question, there are many variants to this problem that could trip up a developer in an interview.

One constraint that could be used in an interview would be to disallow **sorting the input array**. This constraint is a great way to assess if a candidate has a thorough understanding of data structures and their possible time complexities.

This article will go into a deep dive on **Two Sum**. The following is the problem description that we'll use:

> *Given an array of integers, return* ***an array containing the indices*** *of the two numbers that add up to a specific target. It may be assumed that each input would have* **exactly** *one solution, and the* same *element may* **not** *be used twice. Lastly, the array cannot be sorted.*

. . .

Understanding the Problem

The first line of the problem description is very specific: given an array, return the indexes of the two numbers that add up to a certain target. Additionally, it's clear that the return value should be an **array** with both of the indexes included.

It’s certain that an **array of integers** is given, but this description is not clear on whether or not a target can be `null` or a data type besides a `number`. For instance, the **target** might possibly be `'3'`, which is a `string` but could be coerced into a `number`.

In an interview, these would be great questions to ask an interviewer. For the purposes of this question, assume that **target** will always be an integer of `number` data type.

The second sentence explains that there is only **one** solution in a given array for a set **target** number. So, the first pair of numbers whose sum is equal to the target number can be returned without raising an exception.

Additionally, the same element cannot be used twice. This means that an element cannot be added to itself to reach the target number.

Finally, the final line indicates that the array cannot be sorted.

Now that the problem has been explained, break it down into small requirements. This will be helpful for creating an algorithm.

1. Given an array of integers and an integer.
2. Find the two numbers whose sum is equal to the integer input, and then return their indexes.
3. The return value should be an array with the indexes stored inside of it.
4. The same element at a given index cannot be used twice.
5. There is only one solution for the given array and target integer.
6. No sorting allowed.

Test Cases

This problem is actually pretty straightforward, so covering a wide amount of edge cases isn’t necessary; however, there is one edge case to consider, and that edge case is: **negative integers**.

The problem description doesn't say that the integers must be positive. So, a test case with negative elements in the array and a case with a negative target number would be wise to include in the test suite.

One may include as many tests as needed, but these are the test cases for this article:

1. `array = [3, 2, 4], target = 6`
2. `array = [6, 2, 3, 9, -5, 5, 7, 2], target = 1`
3. `array = [2, -3, 1, -5], target = -3`

It's time to plan an algorithm.

> *Note:* `n` *will represent the* `length` *property of* `array`.

. . .

The Brute Force Algorithm isn't Adequate

Since the essence of this problem is to find the **pair** of elements that add up to a certain number, the logic of this problem is easy to follow:

1. For `currentElement` in `array`, check the sum of `currentElement` and `nextElement` against the target. If `sum === target`, return the indices of `currentElement` and `nextElement`.
2. Iterate through the `array`, checking for each element plus every element with a higher index until the correct pair is found.

This is a **brute force** solution. It is not an efficient solution, as it requires potentially iterating over the `array` many times before finding the desired pair.

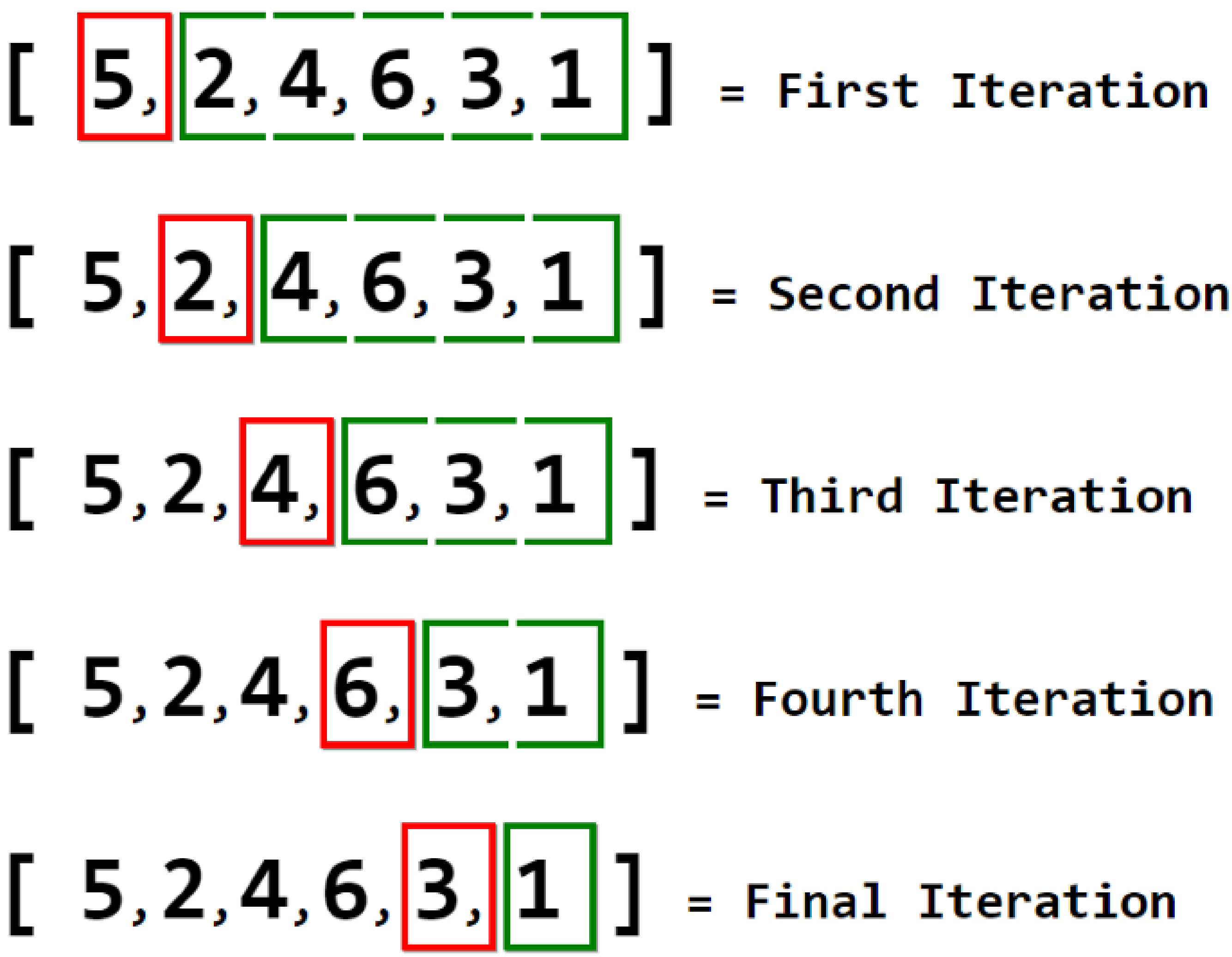
In most situations, there will be a test case which has an array with thousands of elements. Now, imagine that the elements whose `sum === target` are at indexes `[array.length - 2, array.length - 1]`.

In order to reach those two elements, `n-1` elements from `array` must be iterated through on every iteration of the loop for `currentElement`.

> *Remember,* `currentElement` *must be summed with every element* ***after*** *it in the* `array`*, since the elements before the* `currentElement` *have already been checked.*

This nested loop would result in thousands of unneeded iterations.

Here is a diagram detailing the number of iterations required to check through an entire `array`. Keep in mind, this is a very small input; imagine this type of algorithm on a **huge** input.



The **red** window represents the `currentElement`. The **green** window represents all of the elements which must be added with `currentElement` to check for `sum === target`. For each element in the green window there is one iteration.

Expressed in mathematical terms, there are `n**2` possible pairs in an `array`. So, the longest (worst case) this algorithm could take is `O(n**2)` time, which is incredibly inefficient.

As one can see, the brute force solution is not ideal for large `array` inputs.

There must be a way to minimize the number of iterations it takes to find the correct pair. Is there a data structure that allows for constant `O(1)` lookup time?

Hash Table

A **hash table** is a collection of key and value pairs. A **hash function** is used to map values to indexes, also known as keys. These key and value pairs allow for constant lookup time.

In JavaScript, we can use an `object` to create a hash table. Here's one example of a simple hash table:

```
1  let values = [4, 2, -1, 5, 6];
2  let hashTable = {};
3
4  values.forEach((value, index) => hashTable[value] = index);
5
6  hashTable[4];   // 0
7  hashTable[-10]; // undefined
8  hashTable[-1];  // 2
9  hashTable[6];   // 4
```


As shown by lines 6–9, it is possible to access the `index` of each element from `values` by referencing the `value` in `hashTable`.

The main benefit of using a hash table is the `O(1)` lookup time. Instead of needing to loop through a list of values to find the correct one, once the values are mapped to some sort of unique key, the value can be referenced immediately with the use of its key.

Creating a More Efficient Algorithm

So, it's a wise idea to use a hash table for Two Sum; however, what is the right way to use one for this problem?

The brute force solution has two loops, one nested inside of the other. The loop that takes longer is the nested loop since it goes through `array` several times. The nested loop was colored **green** in the picture above.

If there was only the first loop, represented by the **red** window in the picture above, the time complexity would be `O(n)`, since in the worst-case scenario `array` would need to be iterated through completely to find the correct pair.

It's a good idea to eliminate the nested loop. Could the nested loop be replaced with a hash table?

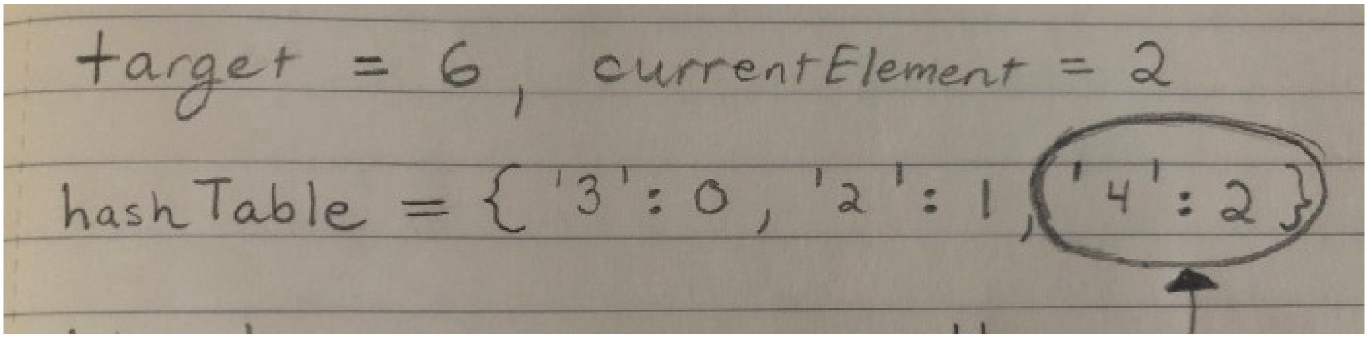
Yes — a hash table actually works perfectly alongside the single loop.

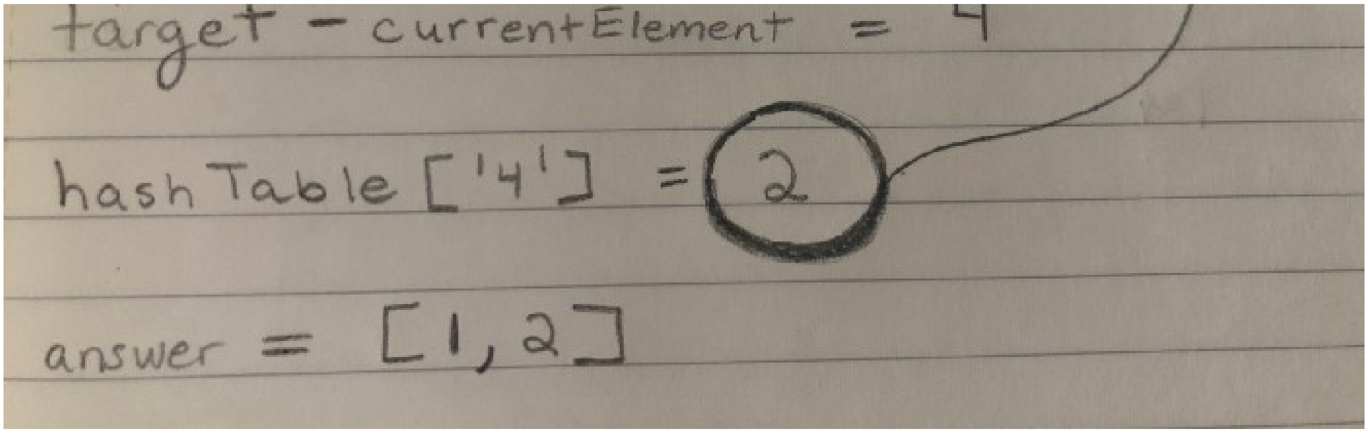
So, the loop will iterate through `array` one element at a time. That means that `currentElement` must be checked against every other element in `array`. How could this be implemented with a hash table?

The number that would pair with `currentElement` is the **difference** between `currentElement` and `target`. So, store `target – currentElement` in `currentDifference`. Using the lookup feature of the hash table, check if `currentDifference` is a **key** in the hash table.

If it exists in the hash table, that means it is an element of `array`.

Here is a drawing which details this step of the algorithm. For the purposes of this drawing, the inputs are: `array = [3, 2, 4], target = 6`. Additionally, `currentElement = 2` in the loop.





Sometimes, pen and paper is necessary.

The answer for these inputs is `[1, 2]`, since `2` and `4` at indices `1` and `2` respectively comprise the pair of numbers which sum to `target = 6`.

The algorithm espoused above successfully found the correct element by looking up `currentDifference` in `hashTable` to find if that element if it exists. Then, it returned the index of `hashTable[currentDifference]` and that was used in the return value, `[1, 2]`. `1` represents the index of `currentElement` in `array`.

The only caveat to this algorithm, however, is that the element must not be combined with itself to match `target`. A simple check of `currentElementIndex !== hashTable[currentDifference]` would eliminate that edge case though.

So, the final algorithm is:

1. Create an object containing the key-value pairs of the element and its index, respectively.
2. Iterate through `array`. For `currentElement`, compute `currentDifference`.
3. If `currentDifference` exists in `hashTable` and `currentElementIndex !== hashTable[currentDifference]`, return the indices of each element.
4. If `currentDifference` does not exist or the indices of both elements are equal, move to the next element in the `array`.

. . .

Implementing the Algorithm

First, declare a function `twoSum` and declare a variable to store the object in.

```
function twoSum(nums, target) {
  numsIndexes = {};
};
```

twoSum.js hosted with ❤ by GitHub view raw

The next step is to loop through the `nums` array. Do this in a `for` loop so `break`ing early is possible.

In this loop, also check if the `currentDifference` exists in the hash table: if it does, `return` the correct indices. In the `else` case, assign the current element as a property on `numsIndexes` and point its value at the index, `i`.

```
var twoSum = function(nums, target) {
  numsIndexes = {};

  for (let i = 0; i < nums.length; i += 1) {
    let currentDifference = target - nums[i];

    if (numsIndexes[currentDifference] !== undefined && numsIndexes[currentDifference] !== i)
      return [i, numsIndexes[currentDifference]];
    } else {
      numsIndexes[nums[i]] = i;
    }
  }
};
```

updated_ts.js hosted with ❤ by GitHub view raw

It's time to test this solution on LeetCode, to ensure that large test cases can be executed successfully.

Success Details ›

Runtime: 44 ms, faster than 99.51% of JavaScript online submissions for Two Sum.

Memory Usage: 35 MB, less than 27.69% of JavaScript online submissions for Two Sum.

Almost 100%!

99.51%! That's pretty fast. Also, the entire test suite was executed successfully.

...

Conclusion

This article was information and algorithm-heavy, so excellent job on getting to this point.

As one can see, utilizing a hash table in one's algorithm can greatly decrease the time complexity of it. The strategy for using the hash table in this problem transfers well to many other problems, so keep it close while tackling difficult questions.

Happy coding!

Programming JavaScript Software Engineering