Has Target Sum Lab Solution

(https://github.com/learn-co-curriculum/phase-1-algorithms-has-target-sum-solution) (https://github.com/learn-co-curriculum/phase-1-algorithms-has-target-sum-solution/issues/new)

Learning Goals

- Practice algorithmic problem solving
- Identify the Big O of an algorithm
- Evaluate multiple solutions to a problem

Introduction

Hopefully you were able to get a working solution going for this problem! It's a tricky one, but the more you practice your process, the easier it will be to solve similar problems in the future.

Once again, we'll walk through a couple solutions to this problem using the problem solving approach and discuss the tradeoffs for each solution. Here's the original problem statement:

> Write a function called `hasTargetSum` that receives two arguments:
>
> - an `array` of integers
> - a `target` integer
>
> The function should return true if any pair of numbers in the array adds up to the target number.

Video Walkthrough

1. Rewrite the Problem in Your Own Words

We'll start by rewriting the problem in a different way to make sure we understand it:

> I need to make a `hasTargetSum` function that checks if two numbers in an array add up to some target number. For example, if the array is `[1,2,3,4]` and the target number is `6` , I know that `2` and `4` add up to `6` , so I should return `true` . If I have the same array and the target is `10` , no two numbers in the array add up to `10` , so I should return `false` .

Note that this description of the problem highlights the inputs and output (return value), and gives us some ideas to explore later in our pseudocode.

2. Write Your Own Test Cases

Next, let's write a few test cases. Once again, let's consider what **edge cases** (https://www.geeksforgeeks.org/dont-forget-edge-cases/) might come up. Can our algorithm handle negative numbers? What if more than one pair of numbers adds up to the target? What about arrays with just a single number?

We can add this code to the `index.js` file to test our function later:

```
if (require.main === module) {
  console.log("Expecting: true");
  console.log("=>", hasTargetSum([3, 8, 12, 4, 11, 7], 10));

  console.log("");

  console.log("Expecting: true");
  console.log("=>", hasTargetSum([22, 19, 4, 6, 30], 25));

  console.log("");

  console.log("Expecting: false");
  console.log("=>", hasTargetSum([1, 2, 5], 4));

  console.log("");

  console.log("");
  // Negative numbers?
  console.log("Expecting: true");
  console.log("=>", hasTargetSum([-7, 10, 4, 8], 3));

  console.log("");
  // Multiple pairs?
  console.log("Expecting: true");
  console.log("=>", hasTargetSum([1, 2, 3, 4], 5));

  console.log("");
  // Single numbers?
  console.log("Expecting: false");
  console.log("=>", hasTargetSum([4], 4));
}
```

3. Pseudocode

Now that we understand the problem and have a sense of some cases our algorithm can handle, we can write some pseudocode as an intermediate step before writing out our solution.

```
iterate over the array of numbers
  for the current number, identify a complementary number that adds to our target
  (for example: if our number is 2, and the target is 5, the complementary number is 3)
  iterate over the remaining numbers in the array
    check if any of the remaining numbers is the complement
      if so, return true
if we reach the end of the array, return false
```

With that in mind, let's write out our solution!

4. Code

Here's the code we are given to start, with the pseudocode added in as comments:

```
function hasTargetSum(array, target) {
  // iterate over the array of numbers
  //   for the current number, identify a complementary number that adds to our target
  //   (for example: if our number is 2, and the target is 5, the complementary number is 3)
  //   iterate over the remaining numbers in the array
  //     check if any of the remaining numbers is the complement
  //       if so, return true
  // if we reach the end of the array, return false
}
```

Let's fill in the code based on this pseudocode:

```
function hasTargetSum(array, target) {
  // iterate over the array of numbers
  for (let i = 0; i < array.length; i++) {
    // for the current number, identify a complementary number that adds to our target
    // (for example: if our number is 2, and the target is 5, the complementary number is 3)
    const complement = target - array[i];
    // iterate over the remaining numbers in the array
    for (let j = i + 1; j < array.length; j++) {
      // check if any of the remaining numbers is the complement
      // if so, return true
      if (array[j] === complement) return true;
    }
  }
  // if we reach the end of the array, return false
  return false;
}
```

Now's a good time to check if our implementation passes all of our test cases. Running `node index.js` will check that all the test cases match our expectations. If our code works, it's time to refactor!

5. Make It Clean and Readable

We can clean up our code a bit by removing the comments, but overall there's not a whole lot here to change. Removing comments isn't necessary, but it should help to see how easily our code can be read and understood without comments — do the variable names make sense? Are we performing any operations that aren't obvious at a glance?

```
function hasTargetSum(array, target) {
  for (let i = 0; i < array.length; i++) {
    const complement = target - array[i];
    for (let j = i + 1; j < array.length; j++) {
      if (array[j] === complement) return true;
    }
  }
  return false;
}
```

In this case, our code looks pretty good! Let's talk about how to optimize it given what we now know about Big O and time/space complexity.

6. Optimize

Let's start by determining our algorithm's time complexity with Big O. We can break down the steps like this:

```
function hasTargetSum(array, target) {
  for (let i = 0; i < array.length; i++) {
    // n steps (depending on the length of the input array)
    const complement = target - array[i];
    for (let j = i + 1; j < array.length; j++) {
      // n * n steps (nested loop!)
      if (array[j] === complement) return true;
    }
  }
  // 1 step
  return false;
}
```

Since our **inner loop** performs `n` iterations for every iteration of the **outer loop**, we end up with `O(n * n)` or `O(n²)` (quadratic) for the *time complexity* of this algorithm.

In terms of *space complexity*, the amount of memory needed grows linearly with the size of the input array. We don't need to create any additional data structures to store information. So we end up with `O(n)` .

How can we optimize? Is there any way we can avoid a quadratic runtime?

Think back to the different versions of our sock-finding algorithm. One way we were able to make our algorithm more efficient was by changing the way our data was stored. By using an **object** instead of an **array** to store data, we can improve the runtime of *searching* through data from `O(n)` to `O(1)`:

```
// O(n) runtime
function findSock(array) {
  for (const item of array) {
    if (item === "sock") return "sock";
  }
}

// O(1) runtime
function findSock(object) {
  if (object.sock) return "sock";
}
```

How can we apply this approach to our `hasTargetSum` problem? As we iterate through the numbers in the array, we can create a **new object** as part of our algorithm to keep track of all the numbers we've already seen as **keys** on the object, so we can quickly check if the object has a key that we're looking for. Then, on the next iteration, we can see if one of the numbers we've added as a key to our object is a complement to the number we're iterating over (if it adds up to the target number).

Let's see what this alternate approach would look like in pseudocode:

```
create an object to keep track of all the numbers we've seen
iterate over the array of numbers
  for the current number, identify a complementary number that adds to our target
  (for example: if our number is 2, and the target is 5, the complementary number is 3)
  check if any of the keys in our object is the complement to the current number
    if so, return true
  save the current number as the key on our object so we can check it later against other numbers
if we reach the end of the array, return false
```

Notice that in this version of the pseudocode, we only end up iterating through the array one time instead of having any nested iteration!

Here's what that solution looks like:

```
function hasTargetSum(array, target) {
  // create an object to keep track of all the numbers we've seen
  const seenNumbers = {};
  // iterate over the array of numbers
  for (const number of array) {
    // for the current number, identify a complementary number that adds to our target
    // (for example: if our number is 2, and the target is 5, the complementary number is 3)
    const complement = target - number;
    // check if any of the keys in our object is the complement to the current number
    // if so, return true
    if (seenNumbers[complement]) return true;
    // save the current number as the key on our object so we can check it later against other numbers
    seenNumbers[number] = true;
  }
  // if we reach the end of the array, return false
  return false;
}
```

See if this passes our tests! Then, let's check the time complexity of this new version:

```
function hasTargetSum(array, target) {
  // 1 step
  const seenNumbers = {};
  for (const number of array) {
    // n steps
    const complement = target - number;
    // n steps
    if (seenNumbers[complement]) return true;
    // n steps
```

```
      seenNumbers[number] = true;
    }
    // 1 step
    return false;
  }
```

Here, we end up with `O(3n + 2)` steps, which can be simplified to `O(n)` . So this refactor has improved the time complexity significantly!

In terms of space complexity, we are creating a new data structure `seenNumbers` that will grow in proportion to the size of the input array, so our space complexity gets a bit worse — we are now also using `O(n)` space.

As one last refactor, we can replace our object with another built-in JavaScript data structure that will accomplish a similar task of letting us keep track of a list of elements and access them with `O(1)` lookup time: a **`Set`** **(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set)** .

> `Set` objects are collections of values. You can iterate through the elements of a set in insertion order. A value in the `Set` **may only occur once**; it is unique in the `Set` 's collection.

A `Set` is a very useful data structure to know for solving algorithm problems — any time you need to keep track of a list of unique values, it's a good data structure to consider! In our case, it's a bit more straightforward to use than an object, since an object is really meant for storing key-value pairs, and all we really care about are the keys. Here's how we can use a `Set` :

```
  function hasTargetSum(array, target) {
    const seenNumbers = new Set(); // initialize an empty Set
    for (const number of array) {
      const complement = target - number;

      // .has returns true if the Set includes the complement
      if (seenNumbers.has(complement)) return true;

      // .add adds the number to the Set
      seenNumbers.add(number);
    }
    return false;
  }
```

Our final solution has:

- Time complexity: `O(n)`
- Space complexity: `O(n)`

Conclusion

When solving algorithm problems, coming up with a more efficient solution often involves reframing the problem, and recognizing patterns that you may have seen before in similar problems. The second solution we came up with isn't something a lot of developers come up with on the first attempt, and it's better to get a slow (but working!) solution on paper before trying to optimize. But the more practice you get, the more you'll see these common patterns emerge and be able to apply different strategies to solve problems more efficiently.

Resources

The problem we just solved is a variation of a very common algorithm problem: the **Two Sum** (https://leetcode.com/problems/two-sum/) problem. See if you can solve that version of the problem using a similar technique to what we came up with for this problem. Here are a few videos that talk through solutions to this algorithm:

- **techsith** (https://www.youtube.com/watch?v=_CoCO62fn_E)

  (https://www.youtube.com/watch?v=_CoCO62fn_E)
- **Web Dev Simplified** (https://www.youtube.com/watch?v=lvyh3V4QoIA)

(https://www.youtube.com/watch?v=Ivyh3V4QoIA)

- Terrible Whiteboard (https://www.youtube.com/watch?v=U8B984M1VcU)

  (https://www.youtube.com/watch?v=U8B984M1VcU)
- Carla Notarobot (https://www.youtube.com/watch?v=JBDS8Dc2sBw)

  (https://www.youtube.com/watch?v=JBDS8Dc2sBw)