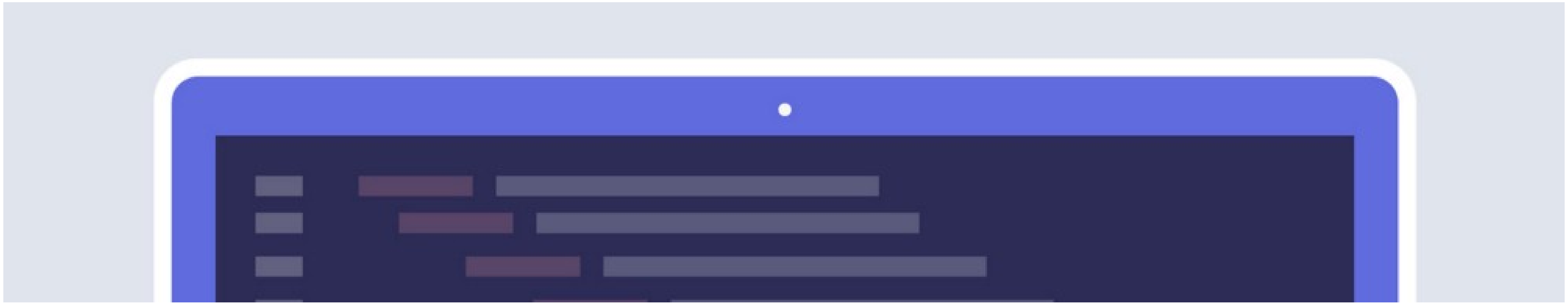
Solving the classic Two-Sum and Three-Sum Problem in JavaScript

Rohan Paul Jun 2, 2019 · 8 min read



First the Two-Sum Problem

A great and classic challenge, is what I stumbled upon in a **Leetcode Problem**. Its a variation of the classic **subset sum problem** in computer science.

Problem Statement — 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].
```

Solution-1-Brute Force — O(n^2)

The above check all the combinations by looping through each element **x** and find if there is another value that equals to target ``target−x``.

```
Time complexity : O(n²)
```

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)** . The space complexity is constant because it doesn't need any temporary buffer to store the data.

So, beware of nested loops — this is where my time complexity begins to grow exponentially.

Solution-2-Much Improved Solution with Hash/Object in O(n) time

On the first line inside the function, notice that I'm declaring a new variable, **numObject**, and setting it equal to an empty hash table ({}). We're going to build out this hash table using the elements from our array as we go.

I’m going to use the numbers in my array to make keys for my new has. And, the index-no of that element i of that array to be the value in the key-value object.

So here, under the first for loop, I am doing a **numsObject[num] = i** which means, I am assigning the given array element value to be the key in the key-value pair of the object / associative-array that I created and the index number of that array element will be the value of the key-value pair.

Then in the second for loop, will check with **hasOwnPropery()** if the key exists. And here the key that I am looking for will be the will be the compliment **(target — x).**

In this way, the look up time is reduced from **O(n) to O(1)** by trading space for speed. A hash table is built to achieve this. 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. Since hash table has average access time **O(1)**, and we only access the array once. The time complexity of this solution is **O(n)**. Since we use the hash table as a temporary buffer, at worst case we need additional **O(n)** storage.

However, notice that in this function we also create a brand new variable (**numObject= {}**). This means that your computer now has to create and set aside a specific place in memory to store that new piece of data. Meaning space complexity will increase.

Performance Test with the below script

Created a random array with 3000 elements and look at the substantial improvement between the two solutions.

. . .

Now lets solve the Three-Sum Problem

A great and classic challenge, **3-Sum** and extremely popular for Developer-interviews. It can be solved with variying level of efficiency and beauty. Its a simple problem on the face of it, but there are a couple things that make it tricky. For evidence on how much 3-Sum is loved, check out this ***Quora thread***

Among some of the unresolved problems in Computer Science on planet earth, there remains this one — Can 3SUM be solved in O(n^{1.99999}) time i.e. better than O(n^2) time ?

It's conjectured to be impossible to find an **O(n^(2−ϵ))** algorithm for solving 3SUM, for any positive constant ϵ. Although, a pretty recent research, shaved off some log factors for the 3SUM problem giving a sub-quadratic deterministic algorithm : **Threesomes, Degenerates, and Love**

Triangles and probably the best known general algo for 3-Sum at the moment.

Solution-1-Brute Force

The requirement is to find all unique triplets in the array. That is, the solution set must not contain duplicate triplets. This means in the resultant array, no number would be repeated. So, I wil achieve this with the following technique.

A> The most simple way to remove duplicates is to stop considering the same value for the same index more than once. Don't let i, j or k refer to the same value in the array twice. Let x refer to some value in the array. Once we have set i=x and found all y, z such that x+y+z=0, we never have to consider setting i=x again.

B> First sort the array. So if there are multiple numbers, like two 5's they will come sequentially. So, whilte iterating for say **i**, I will hit the two sequential **5's** in my two sequential iteration. So I can skip over the second iteration, usingn **continue**.

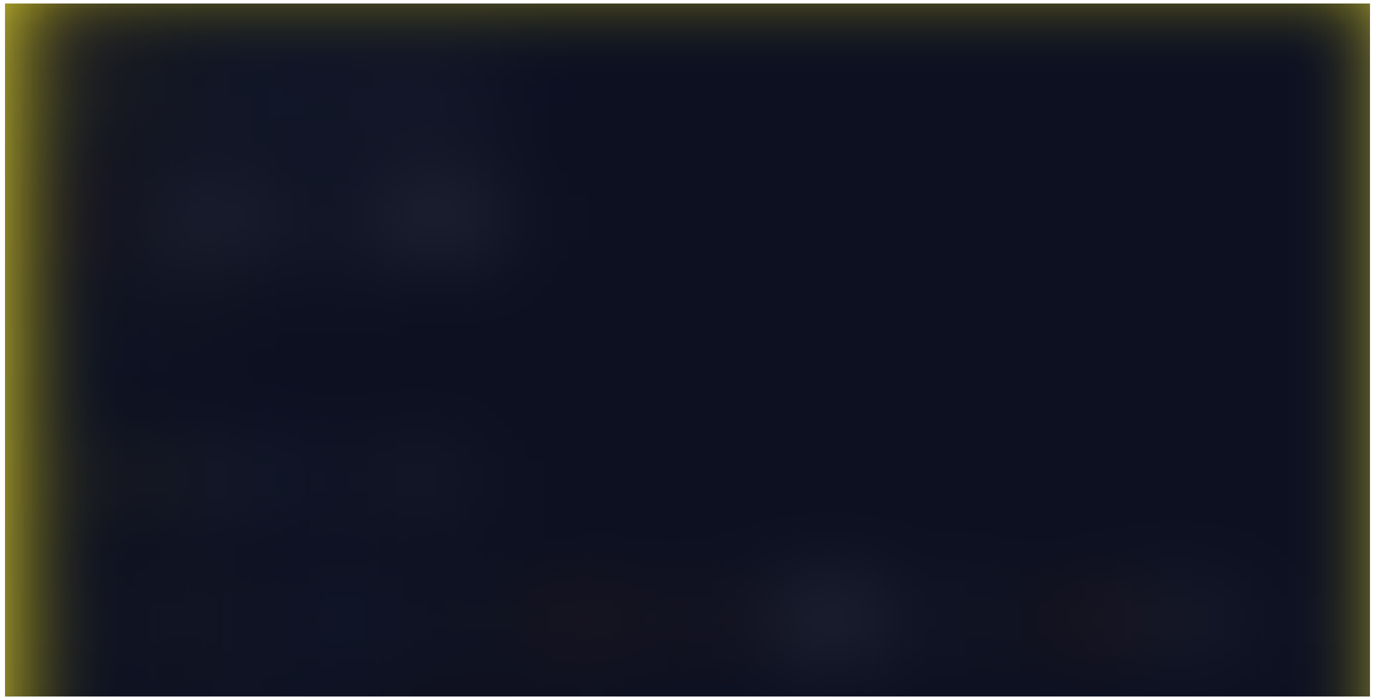
C> So after sorting the array, will check for each iteration that the current element that I am taking is not === previous element. That is if

```
nums[i] === nums[i-1]) continue
```

D> An example below of stopping duplicates..

My First native brute force solution below of traversing the array 3 times checking for all combinations of 3 numbers if their sum produces zero. Although the code produced the correct result in my local machine, but it took so long in Leetcode's exhaustive test-cases, that it gave me "TIME LIMIT EXCEEDED" error there in Leetcode, meaning its not an acceptable solution.

Solution-2- First iteration at finding a 2-pointer-solution that runs in O(n^2) Time



This runs very very substantially faster than the Brute Force one, however, there’s still little room for optimization to the above code. If I put the a line

```
if (arr[aIndex] > 0) return result
```

immediately after I start the first for loop i.e immediately inside

```
for (let indexA = 0; indexA < arr.length — 2; indexA++)
```

in the above code, it will gain some performance, because I will be skipping many iterations as soon as I hit the first element-value of the required result-array to be more than zero.

Solution-3- Even Faster than above after little optimization 2-pointer-solution in O(n^2) Time

The Algorithm

- As a first step, just like the previous solution, since we're dealing with an array of numbers it makes great sense to sort things before diving into algorithm mechanics.
- Instead of 3 nested for loops, here I will use 1 for-loop and one while-loop. Fix the first element as array[i] where i is from 0 to array size — 2. After fixing the first element of triplet, find the other two elements using a **while loop.**
- I will invoke the for-loop

```
for (let indexA = 0; indexA < nums.length — 2; indexA++)
```

- And the first statement in my for-loop will allow the first iteration to run and skip over any subsequent iterations that would lead to the same calulation. So, if my array becomes [1,1,1,3,3] and I am looking at the 0 element 1 — I no more iterate any further.
- Now with each iteration of the first for loop, keeping **a's** value constant, I am looping the values of the next 2 variables **b** and **c** to check if

```
(a + b + c = 0)
```

- **b** starting from index **A + 1** and increasing and **c** starting from the end of the array ***nums.length — 1*** and decreasing.

Since the number array is sorted, the plan is to pull b and c closer together — checking each time for the sum array values to equal 0.

- If I see that the result

```
sum of Triplets i.e a + b + c
```

- is more than zero, I can decrement from right-most elements to reach zero. And if the result is less than zero, I increment the b’s (middle element in the triplet) value to reach zero. Pictorially..

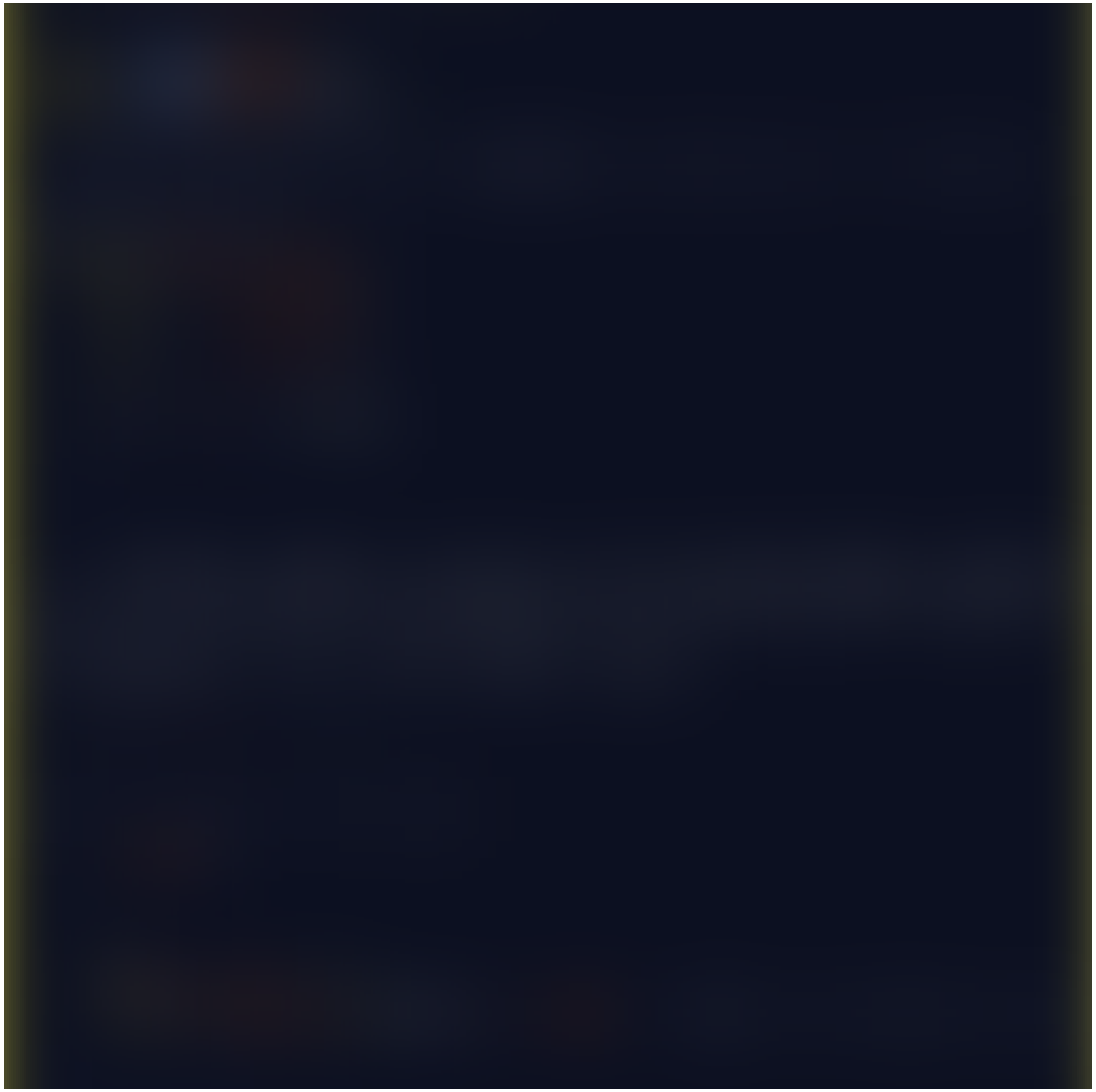
- After all values of **b** and **c** are exhausted, I go back to the next iteration of the for loop for the next value of **a**. Given **a** has the lowest value after sorting the array, and with each value of **a**, I have already iterated through checking for all values of **b** and **c** to check if sum is zero — as soon as

```
a becomes > 0
```

- I can return from the function. Because it means I have reached a point where all the subsequent values of the array will be higher than zero. So no point in iterating further when **a > 0**
- And **a** will only take values upto

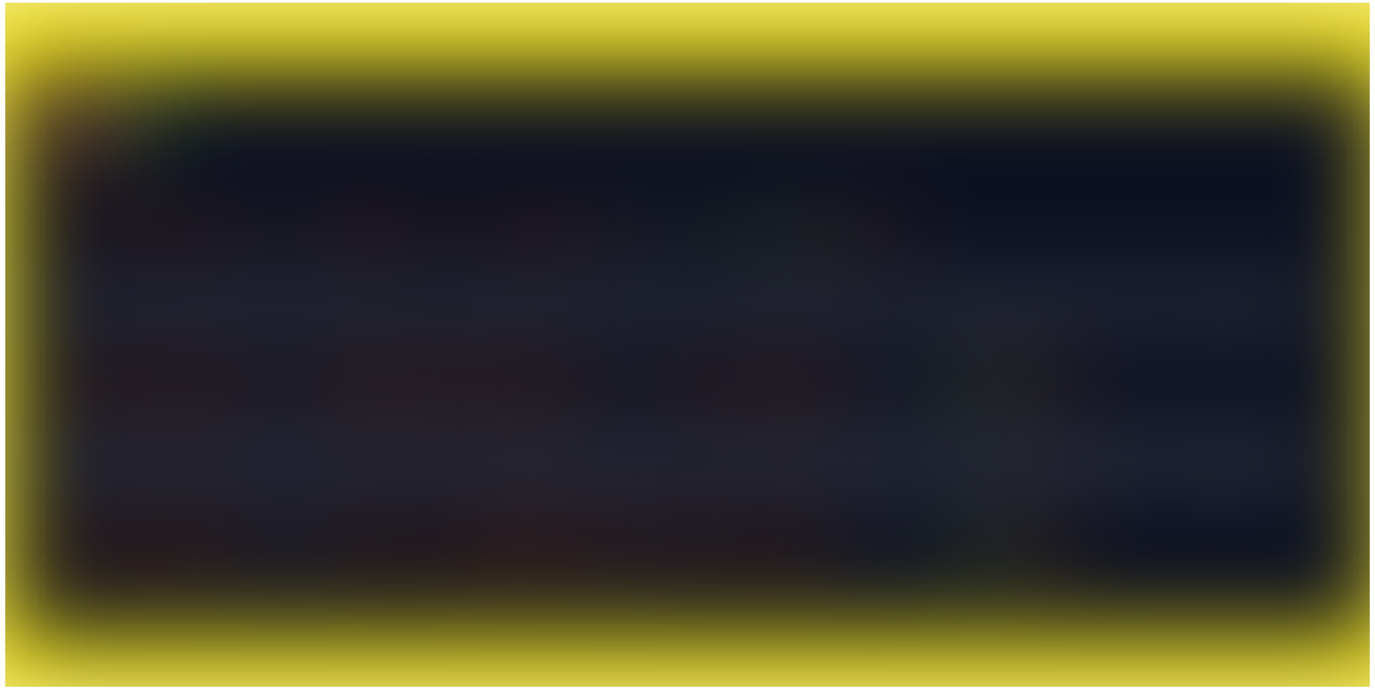
```
(nums.length — 3)
```

- as the last 2 values will be taken by the subsequent 2 variables **b** and **c**



And see the huge performance difference with the 3 approaches to soution, running the code for an array with 2000 randomly generated elements.

And note the great difference of performance from my simple test above.



766 5

Programming JavaScript Algorithms Problem Solving Interview Questions

More from Rohan Paul

Follow

MachineLearning | DataScience | DeepLearning. Previously Frontend Engineer. Ex International Banking Analyst. https://www.linkedin.com/in/rohan-paul-b27285129/

More From Medium

Winds — An in Depth Tutorial on Making Your First Contribution to Open-Source Software

Nick Parsons in HackerNoon.com

Learn Dojo Toolkit — Part 1

Jesus Alvarado in The Startup

The Best 2020 Resources for Your Coding Interview Preparation

Piero Borrelli in JavaScript In Plain English

How Single Page Applications Broke Web Design

Erwin in JavaScript In Plain English

Learning JavaScript: The organized way

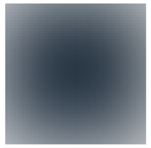
How to get the types you want with type guards

Wyatt Allan in Wyatt Allan

Harry Nicholls in Rangle.io

Easy typed state in React with hooks and Typescript

Sean Matheson



One does not simply “setState”

Amitosh Swain Mahapatra