Time Complexity of JavaScript's Built In Methods

05 APRIL 2015 on paulo diniz, diniz, time complexity, big o, algorithms, algorithm, methods, time complexity built in method

While preparing for technical interviews, I've come across a few questions regarding time complexity for algorithms and I've noticed an interesting trend in the answers from less experienced programmers. If you are not familiar with time complexity for algorithms I recommend you first read this blog post https://www.interviewcake.com/big-o-notation-time-and-space-complexity. When analyzing the time complexity of an algorithm, it is common to not consider that some of the methods being used also added to the time complexity of the algorithm. A few of these examples that are easier to tell, are using JavaScript's .map, or .each. A couple of the harder ones to spot are when you're using .indexOf or .slice or even .shift. Just in case you don't believe me, here's a simplified version of what's actually happening when you call .indexOf.

```
Array.prototype.indexOf = function (element){
  for (var x = 0, count = this.length; x < count; x++){
    if(this[x] === element){
      return x;
    }
```

```
  }
  return -1;
};
```

Now, if you actually wrote out this code you would immediately see that you are using a for loop which adds another N complexity to your algorithm's time complexity. It is easy to forget this when you simply use the method .indeOf. The same goes for .map or .each which are also adding another loop through your data. Things get a little more interesting with .slice and .shift or .unshift.
When you use .slice you are not simply removing certain items from an array, you are actually recreating an entirely new array with only the data that you want. So despite not having to write it yourself, in the background JavaScript is still doing the following:

```
Array.prototype.slice = function (begin){
  var newArray = [];
  for (var x = begin, count = this.length; x < count;
x++){
    newArray.push(this[x]);
  }
  return newArray
};
```

In order to understand the issue with .shift and .unshift all you have to do is look at MDN's description of the .shift method "The shift method removes the element at the zeroeth index and shifts the values at consecutive indexes down, then returns the removed value. If the length property is 0, undefined is returned." Notice that once that element is removed, the index

of every subsequent element has to be modified. The same thing happens with .unshift when a new element is inserted. Now that you are aware that these functions do have a time complexity of their own, why would you keep using them? Well for that I give you three reasons:

1. They are easier to use then having to write your own for loops.
2. They are easier to read then nested for loops.
3. And perhaps the most important reason is that these methods have been optimized by the JavaScript engines that run them and functions designed using them are usually faster.

So despite these methods bringing time complexities of their own it is still best practice to use them whenever possible in order to make your code more readable and more optimized.

Paulo Diniz

Paulo Diniz

Paulo Diniz Software Engineer B.A. from University of California Los Angeles – Political Science Applications Programming Certificate - University of California Los Angeles https://github.com/pdiniz

Los Angeles *https://github.com/pdiniz13*

READ THIS NEXT

AlaSQL

I am currently working on a project to integrate Meteor and SQL. As an initial version of this project,...

YOU MIGHT ENJOY

JavaScript's EVAL's Secret Feature

If you are not familiar yet with JavaScript's eval method let's take a look at what it...