Oliver Caldwell's blog

On programming, Vim and Linux. May contain JavaScript.

Searching JavaScript arrays with a binary search

I migrated from Octopress back to WordPress and all of my code snippets exploded, I'm in the process of fixing those, please bear with me. This post also has huge improvements detailed in the comments section ~~that makes me think this post warrants a revisit, I'll link to it from here if so~~. **I've written a new post!** This one includes benchmarks and generative testing (that all passes!). So you can be sure that it works and that any changes made do actually make it faster. This post is therefore **deprecated**. The repository for the new implementation can be found at Olical/binary-search.

We are asking browsers to do more and more as they become more capable. I'm not sure if that's a good thing or not, but that's a heated flame war that should be saved for another day.

There may come a time in your browser (ab)use in which you need to search a ridiculous amount or values within a cat gif processing platform. Or maybe you need to search through all of the elements in a bloated DOM.

You will probably find that `indexOf` doesn't quite cut it in those situations. In fact, if you *do* happen survive the inevitable violent explosion caused by your CPU trying to take the easy way out, you might want to try a JavaScript search method that is better suited for that much data.

This is where you have an excuse to suggest a binary search and blow everyone

else's minds.

What is it

A binary search searches by splitting your array into smaller and smaller chunks until it finds your desired value. Unlike the normal `indexOf` which searches from left to right in a simple iteration. The binary search Wikipedia article explains it best (as always). There are a couple of downsides; It will be slower with smaller data sets (this needs proving) and the array you are searching **needs to be sorted**.

Because a binary search is O(log n), and not O(n) like `indexOf`, it's great for large sets of data. I set up a little jsPerf for my version of the JavaScript implementation; it is searching **100,000** numbers.

The source for my JavaScript binary search implementation is currently held in a gist. The really cool thing about this is that it's only 138 bytes when minified, that's tiny enough to fit inside a tweet.

```
/**
 * Performs a binary search on the host array. This method can either be
 * injected into Array.prototype or called with a specified scope like this:
 * binaryIndexOf.call(someArray, searchElement);
 *
 * @param {*} searchElement The item to search for within the array.
 * @return {Number} The index of the element which defaults to -1 when not found.
 */
function binaryIndexOf(searchElement) {
    'use strict';

    var minIndex = 0;
    var maxIndex = this.length - 1;
    var currentIndex;
    var currentElement;

    while (minIndex <= maxIndex) {
        currentIndex = (minIndex + maxIndex) / 2 | 0;
        currentElement = this[currentIndex];

        if (currentElement < searchElement) {
            minIndex = currentIndex + 1;
        }
```

```
        else if (currentElement > searchElement) {
            maxIndex = currentIndex - 1;
        }
        else {
            return currentIndex;
        }
    }

    return -1;
```

Edit: I've swapped `Math.floor` for `number | 0` at Yehonatan's reccomendation. It's faster sometimes.

Real world use case

Okay, you might not want to search an array of numbers. Say you were working on a JavaScript map implementation, as I will be talking about in my next post, you might need to search an array of objects. As long as this array contains some kind of number you can use as an index whilst sorting, then it can be done.

Say we have a `Model` class that each have a numerical ID or publish time. The first thing you would need to do is sort them. You can either do this with a sorting function like this…

```
models.sort(function (a, b) {
    return a.id < b.id ? -1 : 1;
});
```

Or you can do it the smart way which kills two birds with one stone. You define a `valueOf` method which returns the ID. This value can then be used by the default sort method and, more importantly, within the `binaryIndexOf` method which needs the comparison operators to work correctly in order to find things.

```
Model.prototype.valueOf = function () {
    return this.id;
};

models.sort(); // Smooth!
```

So now that your models are sorted and their greater than and less than comparison operators will work, you can use the `binaryIndexOf` method on it to search through your 100,000 models. Why are you even doing this to a browser, you monster.

```
// You can either execute it with call like this:
var index = binaryIndexOf.call(models, someModel);

// Or you can inject it into the prototype of Array to have a more native feel.
Array.prototype.binaryIndexOf = binaryIndexOf;
var index = models.binaryIndexOf(someModel);
```

Let me know what you think and if you have any improvements or suggestions. I hope this can help to destroy a bottleneck or two (or 100,000).

Edit: Where should you stick it?

doomslice over on reddit suggested that I return the twos compliment in place of -1. As far as I can tell, this means returning the negative version of the last place that was checked. So if you wanted to insert a value and wanted to know where you should put it, you could run the function and use the returned number to splice the value into the array.

This way you can add items without ruining the order. You don't exactly want to go re-sorting potentially thousands of values every time you add something. Here's the modified function I came up with, as well as a working example. It demonstrates finding where to insert an element and then inserting it.

JavaScript HTML CSS Result Edit in JSFiddle

```
/**
 * Performs a binary search on the host array. This method can either be
 * injected into Array.prototype or called with a specified scope like this:
 * binaryIndexOf.call(someArray, searchElement);
 *
 * @param {*} searchElement The item to search for within the array.
 * @return {Number} The index of the element which defaults to -1 when not found.
 */
function binaryIndexOf(searchElement) {
        'use strict';

        var minIndex = 0;
        var maxIndex = this.length - 1;
```

As you can see, because I use the bitwise NOT operator (~, also suggested by doomsice!) on the index during the splice, it will always insert the element in the right place, even if there is an identical value already there! Pretty cool. You do have to be careful though, now you are going to need to check for `index < 0`, and not `index === -1` when you are looking for existence of a value.

Categorized in: From Octopress

Posted on June 8, 2013 by Oliver Caldwell

25 Comments Oliver Caldwell's blog Login

Recommend 4 Share Sort by Best

Join the discussion…

Tantaman · 2 years ago

Should be ~currentIndex not ~maxIndex.

Example:
var a = [1, 2, 3];
a.binaryIndexOf(0);

With your current code, the above will give you 0. It should be -1 since 0 wasn't in the array and needs to be inserted into the 0th index.

a.binaryIndexOf(1.5); will give you -1. It should be -2 since 1.5 should be inserted into the array at index 1.

The magnitude of the returned missing index should always be 1 greater than the actual insertion index otherwise you can't decide between if something is missing in the array or needs to be inserted as the first element in the array.

In other words, if the missing element should be inserted at index 0 you should return -1. If it should be inserted at index 1, you need to return -2 and so on. If you don't do this you confuse the case of "the array is missing the element which belongs at index 0" and "the array found the element at index 0"

4 · Reply · Share ›

wesman ➚ Tantaman · 5 months ago

I think it actually depends on whether you're looking at the end, beginning, or middle of the set. I replaced ~currentIndex with...

```
if (minIndex > currentIndex) {
return ~minIndex;
} else if(maxIndex > currentIndex) {
return ~maxIndex;
}
return ~currentIndex;
```

This started working for me. Am I right?

· Reply · Share ›

Guest ➚ Tantaman · a year ago

Wrong again! It should be ~minIndex

· Reply · Share ›

Yehonatan · 3 years ago

Instead of Math.floor(num) you can use num | 0

1 · Reply · Share ›

Oliver Caldwell Mod ➚ Yehonatan · 3 years ago

Nice, I should really go through the code and drop in a few more clever tricks. Looks like |0 is a little bit faster. Thanks for the tip! http://jsperf.com/math-floor-v...

· Reply · Share ›

Salnikov Stanislav ↗ Oliver Caldwell · 3 months ago

Instead of x / 2 | 0 you can use x >> 1

Reply · Share ›

observerms · 11 days ago

I am learning JavaScript. Your blog gives the best example I have found of using _while technique for a binary search.

QUESTION:
where can I find an explanation of how _this works in Line 19:
19 currentElement = this[currentIndex] ;

Thanks from a beginner
observerms

Reply · Share ›

Oliver Caldwell Mod ↗ observerms · 11 days ago

I've tried to explain it with code. I hope this helps :) https://jsfiddle.net/Olical/f2...

Read up on call / apply for more information. https://developer.mozilla.org/...

Reply · Share ›

Pedro · 18 days ago

function IndexOf(a,v,min,max,mid,f){
min=0,max=a.length,mid=(max+min)>>>1;
while(f=(v!==a[mid]))
if(v<a[mid]) max="(mid=(max+min)">>>1)+1;
else min=(mid=(max+min)>>>1)-1;
return f ? -1: mid;
}

Reply · Share ›

August · a year ago

Wow I love both the information and the diction of this article.

Reply · Share ›

Howard · 2 years ago

corrected....
<html>

<body>

<script>

function binaryIndexOf(searchElement) {

// JLH >> Should be Find the
starting matching elements index.

'use
strict';

var
minIndex = 0;

var
maxIndex = this.length - 1;

see more

Reply · Share ›

Howard · 2 years ago

From Howard
I edit-tidied your code for a job interview.
Two things
1) there is a boundary error on data items where the last two integers > 2^30; (it's not low +high /2 see the code below.
2) The code doesn't handle repeating values. It likely should give the first index (and a length) of the binary search e.g. if the data is [1,2,2,2,2,2,2,2,2,2,3] ... your algorithm will not return 1 (the 2nd item). I assumed a relatively unique set of data so I implemented a linear range scan. A second binary search could also work faster if only the first item is sought, though it's also more useful to know the count of the items.
Thanks
[code]
<html>
<body>
<script>

function binaryIndexOf(searchElement) {
'use strict';

see more

Reply · Share ›

uberbrady · 2 years ago

First off - thank you so much for this nifty piece of code!

I had to use it for something so I ran it and put together some tests (using Mocha). I took a lot of the advice from the comments -

- I took Xiadong Tan's recommendation for returning ~max - that seemed to work

correctly for me.

- I took "J's" note about using rightshift (>>) instead of dividing by zero and flooring the result. Makes sense - and, well, my tests still pass, so it must be OK :)

Here's the gist with the reworked function, and the tests:

https://gist.github.com/uberbr...

Thanks again for this useful code!

Reply · Share ›

J · 2 years ago

Yes, as Xiaodong Tan said, with Oliver's code, return "~Math.max(minIndex, maxIndex)" (or use the ternary operator [?:] instead of Math.max) instead of "~maxIndex" at the end. Also, changing the division by 2 to ">> 1" makes the calculation much faster. Finally, the "resultIndex" variable isn't used, so it can be deleted.
Thank you.

Reply · Share ›

TRON · 2 years ago

It's usually better and easier to work with half open intervals, the same is true for binary halving. Here is a binary search function that does that, additionally this returns the right insert index for the item even if it wasnt found in the array:
function binary_search(list, item, cmp_func) {
var lo = 0;
var hi = list.length;
if (!cmp_func) {
cmp_func = function(a,b) { return (a<b) ?="" -1="" :="" ((a="">b) ? 1 : 0) }
}
while (lo < hi) {
var mid = ((lo + hi) / 2) | 0;
var cmp_res = cmp_func(item, list[mid]);
if (cmp_res == 0)
{
return {
found: true,
index: mid
};

see more

Reply · Share ›

Xiaodong Tan · 2 years ago

Actually ~maxIndex is incorrect if currentElement > searchElement on the last check. A

simple test with an array like ['CA', 'WA'] against 'TX' will show that maxIndex is 0. The correct answer is 1, not 0.
The return value should be ~max(minIndex, maxIndex).

Reply · Share ›

Guest ↗ Xiaodong Tan · a year ago
or just ~minIndex

Reply · Share ›

MinHyeong Kim ↗ Guest · 7 months ago
~maxIndex fine.

Reply · Share ›

Emilio · 2 years ago
Very usefull and well written! Thanks!

Reply · Share ›

Arnor Heidar Sigurdsson · 3 years ago
There's a node.js module for doing a binary search on a sorted array: https://npmjs.org/package/bina...

I built a sorted set that uses that module: https://npmjs.org/package/sset

I also built a sorted array that uses the sorted set internally in dumb way (doesn't try to be an array, though): https://npmjs.org/package/sarr...

Reply · Share ›

Oliver Caldwell Mod ↗ Arnor Heidar Sigurdsson · 3 years ago
Interesting. I was planning on writing something about maps and potentially sets too. I'm trying to do things that I need to learn to plug the gaps in my self taught knowledge whilst helping people. My aim with all of these implementations is to write them with as little overhead as possible.

So they wouldn't be used as a library, which will probably be more powerful and fully tested, but could be dropped into a class as a helper method. Basically I want to focus on the core parts of algorithms and patterns. If I do anything on sets I'll probably drop in a reference to your modules, they look great.

1 Reply · Share ›

NoName · 3 years ago
var a = [1, 2, 3]

console.log(a.indexOf(3)); // 2

console.log(binaryIndexOf.call(a, 3)); // -1