Binary Search in JavaScript. A practical Example

Learn what a binary search is with the help of terribly drawn pictures and a code example

### What is a Binary Search?

If you have an array that you need to search, the simplest way might be to use indexOf(), or perhaps a for() loop. Either option will start on the left hand side of the array, and iterate through each item in the Array until your desired

**A Binary Search allows you to search a sorted array by repeatedly splitting the array in half.**

A binary search works by checking if our search value is **more** than, **less**than, or **equal**to the middle value in our array:

* If it’s less than, we can remove the right half of the array.
* If it’s more than, we can remove the left half of the array.
* If it’s equal to, we return the value

The catch here is that the array must be sorted — i.e. the values must be in order for a binary search to work.

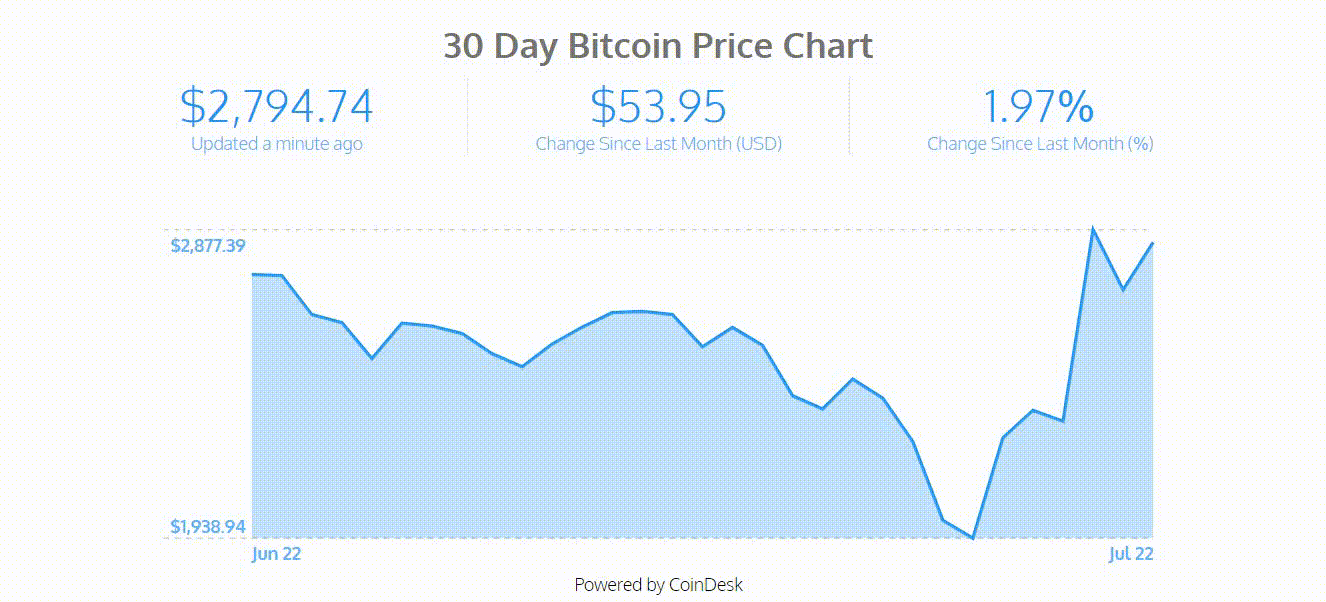
When working with large chunks of data, it is much faster to use a binary search because with each iteration you remove 1/2 of the array’s wrong values, instead of just one wrong value.

### The Project.

Recently, I published an article on [How I built an Interactive 30-Day Bitcoin Price Graph with React and an API](https://codeburst.io/how-i-built-an-interactive-30-day-bitcoin-price-graph-with-react-and-an-api-6fe551c2ab1d).

One of the most important aspects of the chart is the Tool Tip that displays

relevant data as you hover over the chart. As you move your mouse, the chart updates with the data from the closest data point. Here’s a gif example:



How does this work?

I have an array of all of the data point coordinates. The coordinates are within an object, and each object has an svgX key. It looks something like this:

svgData = [  
 {svgX: 1367.844, data...},  
 {svgX: 1684.478, data...},  
 {svgX: 1168.474, data...},  
 {svgX: 1344.854, data...},  
 // etc.  
]

Originally I was using a simple for loop to compare the current X location of the mouse with all of the svgX values in my dataset.

#### For loop.

Here’s what the for loop code looks like. Whichever object has an svgXvalue that is closest to the X location of the mouse — is the data we want to highlight.

const {svgWidth} = this.props;  
let closestPoint = {};  
 for(let i = 0, c = svgWidth; i < svgData.length; i++){  
 if ( Math.abs(svgData[i].svgX — this.state.hoverLoc) <= c ){  
 c = Math.abs(svgData[i].svgX — this.state.hoverLoc);  
 closestPoint = svgData[i];  
 }  
 }

1. First, we get the width of our svg element, and create an empty object to hold our closestPoint.

const {svgWidth} = this.props;  
let closestPoint = {};

2. Then, we loop through the array of data. Each object in our array is compared with X coordinate of the mouse. If the current iteration of the loop has a data point closer to the mouse than any other iteration, that data point is set as our closestPoint.

With a small number of data points, this method is relatively fast. However if we have a large amount of data it is certainly not the most efficient.

#### Binary Search

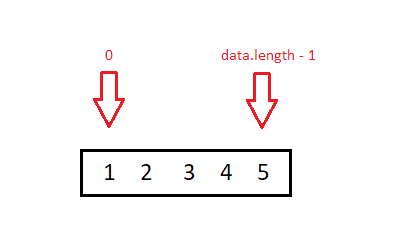
Here’s the binary search code to find the closest data point:

**We need five pieces of information for our binary search:**

1. d — **data**. The array of objects.
2. t — **target**. The location of the mouse (x coordinate).
3. s — **start**. The starting location of this iteration of the binary search.
4. e — **end**. The ending location of this iteration of the binary search.
5. m — **middle**. The middle of the array for this iteration of the binary search.

**I know this is confusing, so lets walk through an example that simplifies the code above. In this example**, **our target value will be 3.7. We’ll be searching an Array of 5 increasing numbers.**

As you can see in the code above on line 9, when we start our binary search we pass in our entire data array, with the mouse target, a starting point of 0, and an ending point equal to the full size of the array:

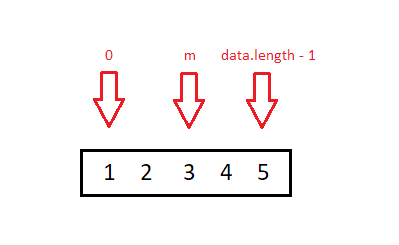


Searching the full array

Once we have our data, the middle of the array is found by adding the starting location and the ending location together then dividing by two. To ensure there are no fractions, the resulting number is rounded down:

const m = Math.floor((s + e)/2);

In this instance we have a starting location of 0, and and ending location of 4. Thus, the middle is (0+4)/2 rounded down = 2:



Finding the middle = d[2]

Recall that for this example our target value is 3.7. After we’ve found the middle point, we check to see if our target value is equal to our middle point. If it is, we return the object and we’re done!

if (t == d[m].svgX) return d[m];

Unfortunately, our middle data point = **3**. And our target equals **3.7**. The two do not match.

Next, we check to see if our ending location minus 1 is equal to our starting location:

if (e — 1 === s) {  
 return   
 Math.abs(d[s].svgX — t) > Math.abs(d[e].svgX — t) ? d[e] : d[s];  
}

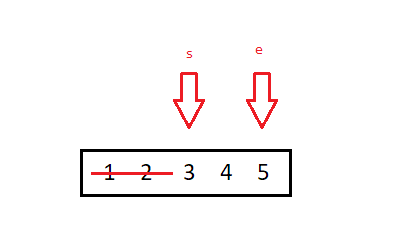
This is our termination case. This case will only prove true when we’ve narrowed down our array to two final values, and our target is between those values. If our target is between those values, we return whichever is closer.

Currently this is false.

Next, we check to see if our target value is greater than our middle value:

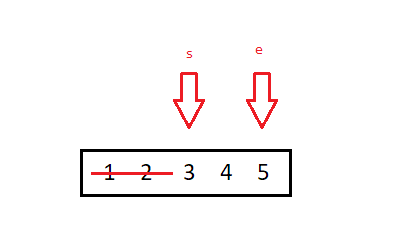
if (t > d[m].svgX) return binarySearch(d,t,m,e);

In this case it is! Target value of **3.7** is greater than middle value of **3**. We can get rid of the first two values in our array:

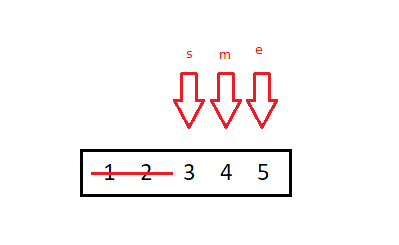


Using recursion we return a new invocation of the binarySearch() function. We pass in our array, the target value of **3.7**, the **middle**value as our starting value, and the ending value as our end value.

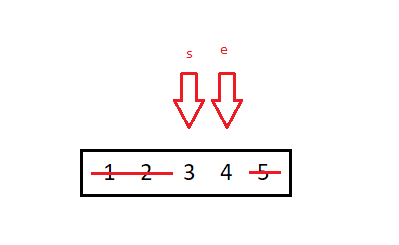
Now we have a brand new invocation of binarySearch() that will search from 2 to data.length-1:



The function finds the new middle point of d[3]:



Since our target vale of **3.7** is less than the middle value of **4**, we get rid of the right hand side of the array, and return a new invocation of the binarySearch() function. This time our starting point remains as d[2], but our ending point changes to our middle point d[3].



We’ve reached our termination case:

if (e — 1 === s) {  
 return   
 Math.abs(d[s].svgX — t) > Math.abs(d[e].svgX — t) ? d[e] : d[s];  
}

Since our ending value minus one equals our starting value, we know the target value is somewhere between the two. We’re going to return whichever data point is closer to the value of **3.7**. Since **4**, (the ending value) is closer, we return the ending value:d[3].

### Speed Test

With smaller data sets, recursion can actually be slower than a for loop. Testing on jsperf with just 10 data points, the recursive function averaged 3% slower than the for loop.

However,**at 10,000 data points, the for loop is 72% slower than the recursive function.**This means the recursive function is nearly twice as fast as the for loop! That’s a massive time savings.

### Closing Notes

Hopefully you can now understand the basics of a binary search in JavaScript!

I publish a few articles and tutorials each week, please consider [entering your email here](https://docs.google.com/forms/d/e/1FAIpQLSeQYYmBCBfJF9MXFmRJ7hnwyXvMwyCtHC5wxVDh5Cq--VT6Fg/viewform) if you’d like to be added to my once-weekly email list.

#### Check out my recent articles:

* [How I built an Interactive 30-Day Bitcoin Price Graph with React, SVG, and an API](https://codeburst.io/how-i-built-an-interactive-30-day-bitcoin-price-graph-with-react-and-an-api-6fe551c2ab1d)
* The [3 Best React JS Courses](https://codeburst.io/3-best-react-js-courses-f31cabae4f1e) on the internet.
* And when you’re ready to really dive into web development, check out my [Best Courses for Learning Full Stack Web Development](https://codeburst.io/best-udemy-courses-for-learning-full-stack-web-development-45e2bd3ec28b)