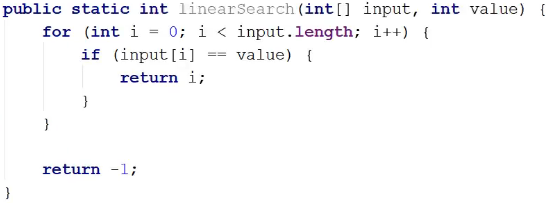
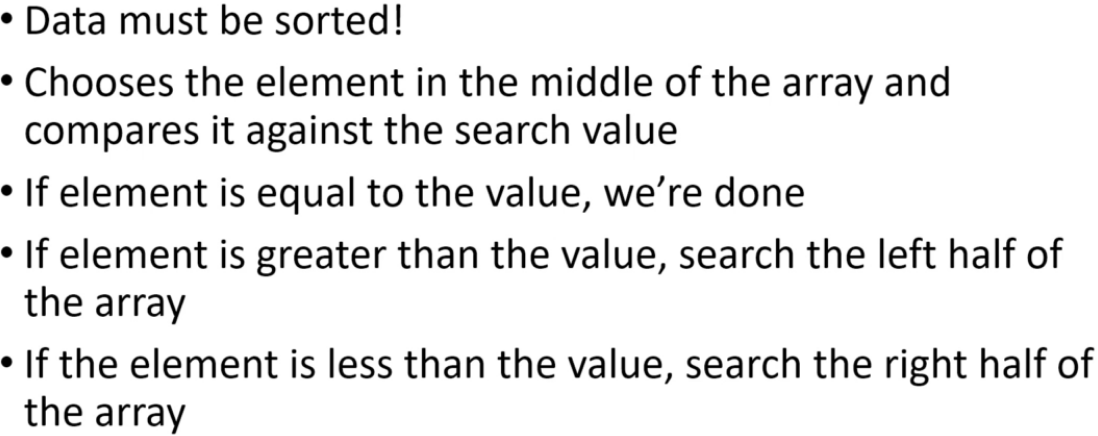
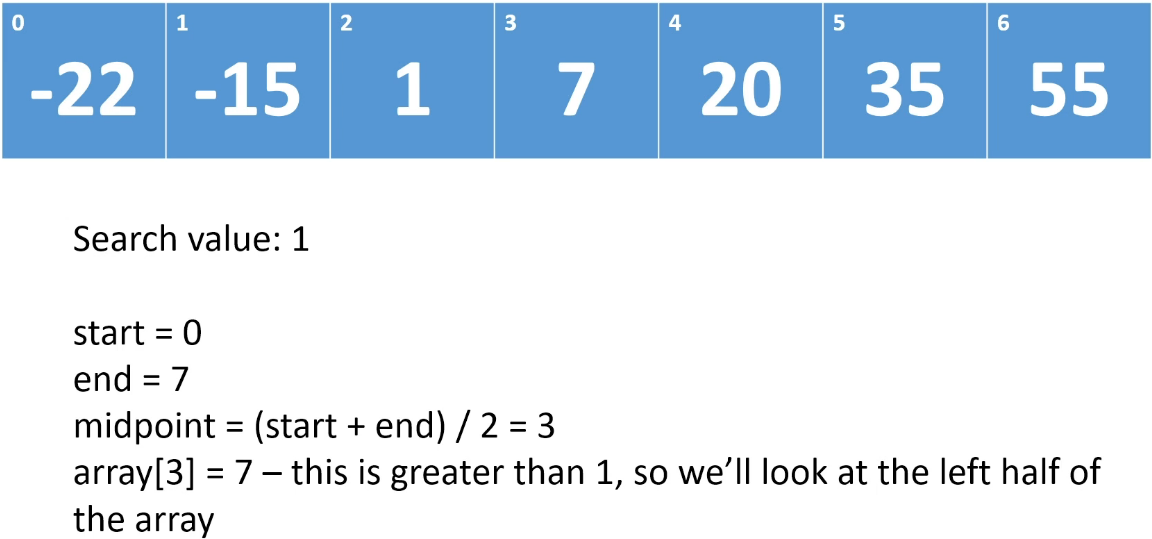
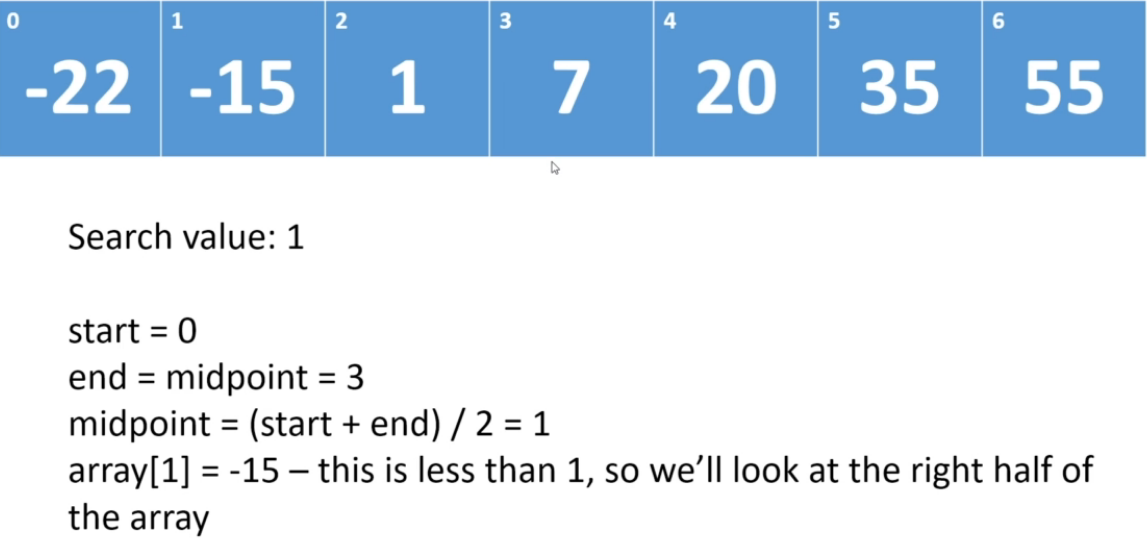
**Introduction to Search Algorithms**  
\* A search algorithm will find a value in a data structure.  
\* The search algorithms we’ll look at will find a value in an array.  
\* You can use them to search Linked Lists and other data structures but we’ll look at arrays.  
\* We’re not going to look at as many algorithms as we did for sort because there is 1 search algorithm that performs really, really well and so we’re going to look at the linear search algorithm and then we’re going to look at the binary search algorithm.

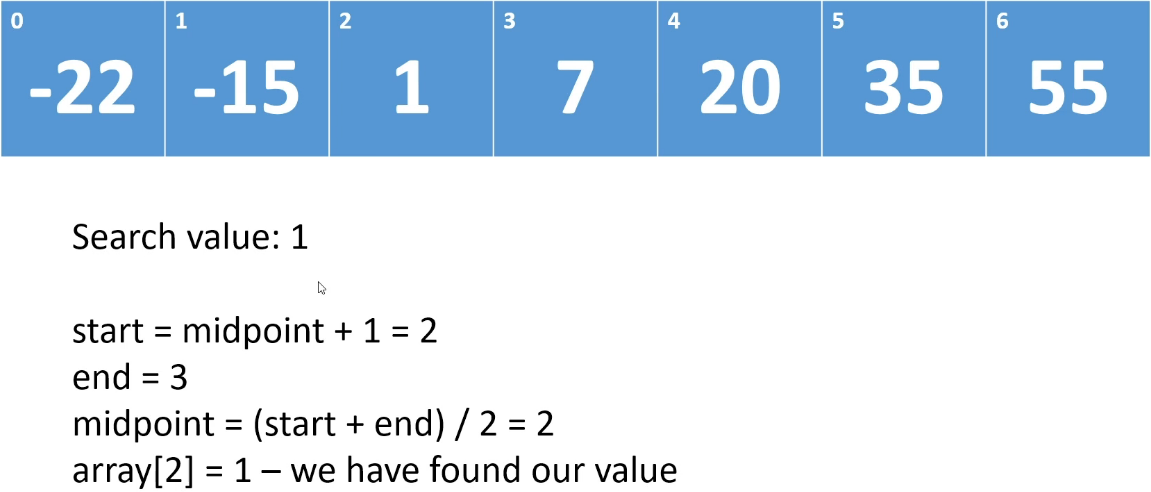
**Linear Search Algorithm**  
**O(n)** => in the worst case we have to traverse the entire array.  
\* It’s the search algorithm that everybody would think of if somebody said to you: write code to find -15 in this array.  
=> You’re going to iterate over the array and compare each value to the value you’re searching for.  
\* This is why it’s called a linear search - you’re basically just incrementing the index by 1 in a linear fashion and you go from the beginning of the array to the end of the array and you’re going to either find the item you’re looking for, or you’re going to hit the end of the array and if you do and the last item in the array isn’t equal to the one you’re looking for, then you can be certain that the array doesn’t contain the value you’re looking for.  


**Binary Search Algorithm**  
**O(logn)**  
\* **Binary Search is pretty much the standard search algorithm**.  
\* BUT it **requires the data that it’s searching be sorted**.

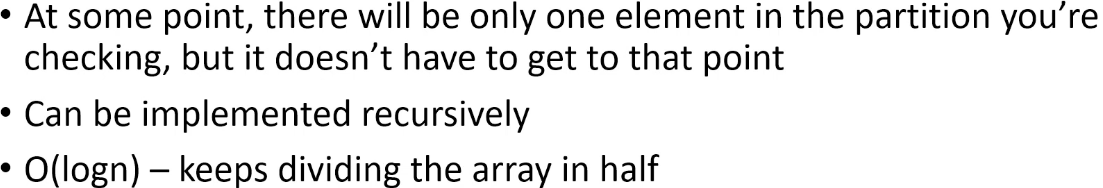


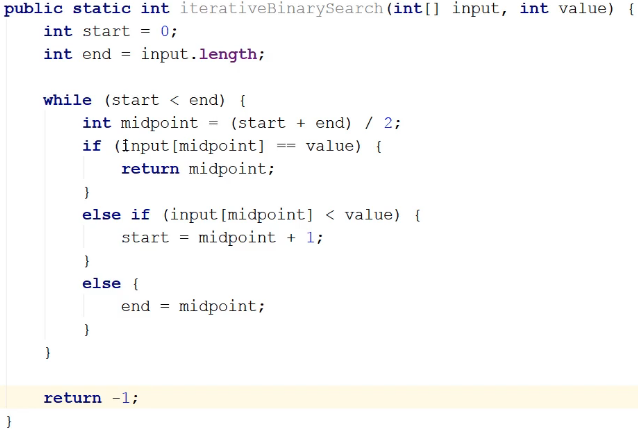
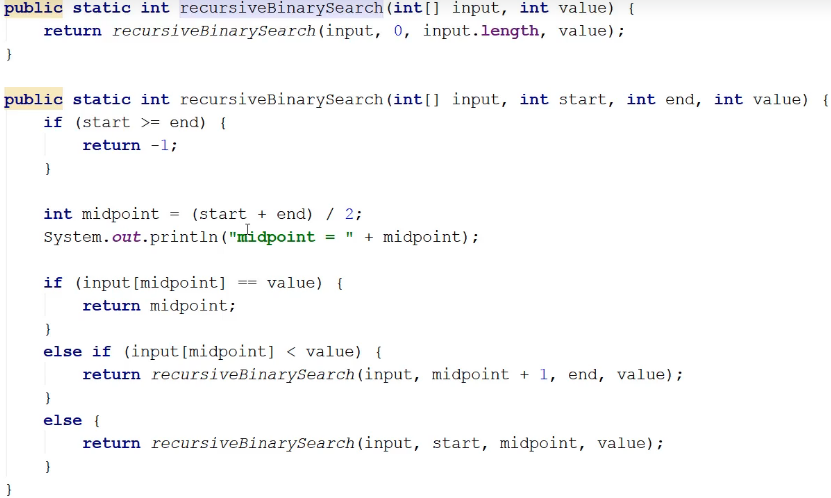
\* That goes for any data structure, if you want to use binary search on a Linked List, then the Linked List has to be sorted or you have to sort it first.  
\* If you know your application is going to be using binary search to do a lot of searches, then what you might consider doing is inserting items into whatever data structure you’re using so that they’re always in sorted order.  
\* So rather than sorting it every single time you want to run a binary search, if you know you’re going to be running a lot of searches, then it might be better to just always make sure that the values in the data structure are always sorted.  
\* The values are sorted and so let’s say the middle of the array is at index 5, if that element is > the searched value than we know that the value we’re searaching for has to be in indices 0 - 4 because the data is sorted and so if the value at index 5 is > the value we’re searaching for, that means that the values from 6 up to the end of the array are also going to be > the one we’re looking for.  
\* We keep search left/right parts until we’re going to end up ultimately either hitting our searched value or getting down to a 1 element partition and that partition is going to equal our search value.  
\* **So with each step we’re dividing the array in half**.  
\* It’s very similar to what we were doing with Merge Sort.  
\* **In our implementaton** **end is always 1 greater than the partition we want to look at**.  






\* In this example, we did as many steps as we would’ve done for Linear Search but if we were looking for a value at the end of the array for example, it would’ve been a lot quicker.  
\* That’s Binary Search, it basically starts out by dividing the array down the middle and then it always looks at the middle element and then depending on what the comparison result is, we’re either finished or we’re then only going to focus on the left or right side, and we keep doing that.



**Binary Search (Implementation)**  
**(Implemented my own version before watching)**\* We’re doing **logical partitioning** - we’re not creating a new array every time we partition.  
\* **If start == end, we’ve searched the entire array and we haven’t found what we’re looking for**.  
\* Iterative:  
  
\* Recursive:  
  
\* **I didn’t have to break it into 2 methods, I could just call it directly but normally what you’d have is a method that takes the entire thing and then you’d call the recursive method**.  
\* **Each call of the recursive method is pretty much equivalent to an iteration of the WHILE loop**.  
\* But instead of just adjusting the start and end points and looping around, we’re adjusting the start and end points and then calling the method.  
\* When changing an iterative method into a recursive one, you’re basically taking a loop and you’re changing it to a recursive method.  
\* **In this case, I think the iterative version is really clear so I would use this iterative version because as we discussed when we went over recursion, the recursive form is usually more expensive because there’s overhead involved with method calls**.  
\* So on the same hardware, same array and everything else, we would expect the iterative method to perform a little better than the recursive method.