Suppose you need to access information stored in a list. The standard algorithm we have considered in class is just a sequential search. But if the list is kept in sorted order, then we can use a faster algorithm, called binary search. Below, we describe both algorithms.

**Linear/Sequential Search**

- A method for finding a target value within a list
- The list does not need to be in sorted order.
- It sequentially checks each element of the list for the target value until a match is found or until all the elements have been searched.

**Example**

```javascript
function findIndex(values, target) {
    for(var i = 0; i < values.length; ++i){
        if (values[i] == target) {
            return i;
        }
    }
    return -1;
}
```

Worse case scenario: loop through entire list to find the number (either item is last or isn’t found)

Complexity: O(n)
Binary Search

Suppose you have a sorted list of integers. How could you find a particular integer in that list? What about a binary search? The way a binary search works is you look at the middle element and determine whether the item would be in the first half or the second half of the list. Then search the appropriate half, recursively.

If searching for 23 in the 10-element array:

<table>
<thead>
<tr>
<th>2</th>
<th>5</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>23</th>
<th>38</th>
<th>56</th>
<th>72</th>
<th>91</th>
</tr>
</thead>
<tbody>
<tr>
<td>23 &gt; 16, take 2nd half</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>23</td>
<td>38</td>
<td>56</td>
<td>72</td>
<td>91</td>
</tr>
<tr>
<td>23 &lt; 56, take 1st half</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>23</td>
<td>38</td>
<td>56</td>
<td>72</td>
<td>91</td>
</tr>
<tr>
<td>Found 23, Return 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```java
private static boolean binarySearch( int[] a, int start, int stop, int goal ) {
    if (start == stop) {
        return a[start] == goal;
    }
    int half_dist = (stop-start)/2;
    if (a[start+half_dist] >= goal)
        return binarySearch( a, start, start+half_dist, goal );
    else
        return binarySearch( a, start+half_dist+1, stop, goal );
}

public static boolean binarySearch( int[] a, int goal ) {
    if (a.length == 0)
        return false;
    return binarySearch( a, 0, a.length-1, goal );
}
```

Note that if we use `start == stop` to detect an array of length 1, then we have to use another way to detect an array of length 0. In the above code, I chose to put that test for an empty array in the public method.

If we start with an array of length 8, then incorrect guesses reduce the size of the reasonable portion to 4, then 2, and then 1. Once the reasonable portion contains just one element, no further guesses occur; the guess for the 1-element portion is either correct or
incorrect, and we're done. So with an array of length 8, binary search needs at most four guesses.

What do you think would happen with an array of 16 elements? If you said that the first guess would eliminate at least 8 elements, so that at most 8 remain, you're getting the picture. So with 16 elements, we need at most five guesses.

By now, you're probably seeing the pattern. Every time we double the size of the array, we need at most one more guess. Suppose we need at most \( m \) guesses for an array of length \( n \). Then, for an array of length \( 2n \), the first guess cuts the reasonable portion of the array down to size \( n \), and at most \( m \) guesses finish up, giving us a total of at most \( m+1 \) guesses.

Let's look at the general case of an array of length \( n \). We can express the number of guesses, in the worst case, as "the number of times we can repeatedly halve, starting at \( n \), until we get the value 1, plus one." But that's inconvenient to write out. Fortunately, there's a mathematical function that means the same thing as the number of times we repeatedly halve, starting at \( n \), until we get the value 1: the base-2 logarithm of \( n \). We write it as \( \log_2 n \).

Complexity: \( O(\log n) \)