WEEK-07 DAY-3
Wednesday - *Sorting Algorithms Analyses*

Bubble Sort Analysis

Bubble Sort manipulates the array by swapping the position of two elements. To implement Bubble Sort in JS, you'll need to perform this operation. It helps to have a function to do that. A key detail in this function is that you need an extra variable to store one of the elements since you will be overwriting them in the array:

```
function swap(array, idx1, idx2) {
  let temp = array[idx1];     // save a copy of the first value
  array[idx1] = array[idx2];  // overwrite the first value with the second valu
  array[idx2] = temp;         // overwrite the second value with the first valu
}
```

Note that the swap function does not create or return a new array. It mutates the original array:

```
let arr1 = [2, 8, 5, 2, 6];
swap(arr1, 1, 2);
arr1; // => [ 2, 5, 8, 2, 6 ]
```

Bubble Sort JS Implementation

Take a look at the snippet below and try to understand how it corresponds to the conceptual understanding of the algorithm. Scroll down to the commented version when you get stuck.

```
function bubbleSort(array) {
  let swapped = true;

  while(swapped) {
    swapped = false;

    for (let i = 0; i < array.length - 1; i++) {
      if (array[i] > array[i+1]) {
        swap(array, i, i+1);
        swapped = true;
      }
    }
  }

  return array;
}
```

```
// commented
function bubbleSort(array) {
  // this variable will be used to track whether or not we
```

```
    // made a swap on the previous pass. If we did not make
    // any swap on the previous pass, then the array must
    // already be sorted
    let swapped = true;

    // this while will keep doing passes if a swap was made
    // on the previous pass
    while(swapped) {
      swapped = false;  // reset swap to false

      // this for will perform a single pass
      for (let i = 0; i < array.length; i++) {

        // if the two value are not ordered...
        if (array[i] > array[i+1]) {

          // swap the two values
          swap(array, i, i+1);

          // since you made a swap, remember that you did so
          // b/c we should perform another pass after this one
          swapped = true;
        }
      }
    }

    return array;
  }
```

Time Complexity: O(n^2)

Picture the worst case scenario where the input array is completely unsorted. Say it's sorted in fully decreasing order, but the goal is to sort it in increasing order:

- n is the length of the input array
- The inner `for` loop along contributes *O(n)* in isolation
- The outer while loop contributes *O(n)* in isolation because a single iteration of the while loop will bring one element to its final resting position. In other words, it keeps running the while loop until the array is fully sorted. To fully sort the array we will need to bring all n elements into their final resting positions.
- Those two loops are nested so the total time complexity is O(n * n) = O(n^2).

It's worth mentioning that the best case scenario is when the input array is already fully sorted. This will cause our for loop to conduct a single pass without performing any swap, so the `while` loop will not trigger further iterations. This means best case time complexity is *O(n)* for bubble sort. This best case linear time is probably the only advantage of bubble sort. Programmers are usually interested only in the worst-case analysis and ignore best-case analysis.

Space Complexity: O(1)

Bubble Sort is a constant space, O(1), algorithm. The amount of memory consumed by the algorithm does not increase relative to the size of the input array. It uses the same amount of memory and create the same amount of variables regardless of the size of the input, making this algorithm quite space efficient. The space efficiency mostly comes from the fact that it mutates the input array in-place. This is known as a **destructive sort** because it "destroys" the positions of the values in the array.

When should you use Bubble Sort?

Nearly never, but it may be a good choice in the following list of special cases:

- When sorting really small arrays where run time will be negligible no matter what algorithm you choose.
- When sorting arrays that you expect to already be nearly sorted.
- At parties

Selection Sort Analysis

Since a component of Selection Sort requires us to locate the smallest value in the array, let's focus on that pattern in isolation:

```
function minumumValueIndex(arr) {
    let minIndex = 0;

    for (let j = 0; j < arr.length; j++) {
        if (arr[minIndex] > arr[j]) {
            minIndex = j;
        }
    }

    return minIndex;
}
```

Pretty basic code right? We won't use this explicit helper function to solve selection sort, however we will borrow from this pattern soon.

Selection Sort JS Implementation

We'll also utilize the classic swap pattern that we introduced in the bubble sort. To refresh:

```
function swap(arr, index1, index2) {
  let temp = arr[index1];
  arr[index1] = arr[index2];
  arr[index2] = temp;
}
```

Now for the punchline! Take a look at the snippet below and try to understand how it corresponds to our conceptual understanding of the selection sort algorithm. Scroll down to the commented version when you get stuck.

```
function selectionSort(arr) {
  for (let i = 0; i < arr.length; i++) {
    let minIndex = i;

    for (let j = i + 1; j < arr.length; j++) {
      if (arr[minIndex] > arr[j]) {
        minIndex = j;
      }
    }

    swap(arr, i, minIndex);
  }
  return arr;
}
```

```
// commented
function selectionSort(arr) {
    // the `i` loop will track the index that points to the first element of th
    //    this means that the sorted region is everything left of index i
    //    and the unsorted region is everything to the right of index i
    for (let i = 0; i < arr.length; i++) {
        let minIndex = i;

        // the `j` loop will iterate through the unsorted region and find the i
        for (let j = i + 1; j < arr.length; j++) {
            if (arr[minIndex] > arr[j]) {
                minIndex = j;

            }
        }

        // after we find the minIndex in the unsorted region,
        // swap that minIndex with the first index of the unsorted region
        swap(arr, i, minIndex);
    }
    return arr;
}
```

Time Complexity Analysis

Selection Sort runtime is O(n^2) because:

- n is the length of the input array
- The outer loop i contributes O(n) in isolation, this is plain to see
- The inner loop j is more complicated, it will make one less iteration for every iteration of i.
    - for example, let's say we have an array of 10 elements, `n = 10`.
    - the first full cycle of `j` will have 9 iterations
    - the second full cycle of `j` will have 8 iterations
    - the third full cycle of `j` will have 7 iterations
    - ...
    - the last full cycle of `j` will have 1 iteration
    - This means that the inner loop j will contribute roughly O(n / 2) on average
- The two loops are nested so our total time complexity is O(n * n / 2) = O(n^2)

You'll notice that during this analysis we said something silly like O(n / 2). In some analyses such as this one, we'll prefer to drop the constants only at the end of the sketch so you understand the logical steps we took to derive a complicated time complexity.

Space Complexity Analysis: O(1)

The amount of memory consumed by the algorithm does not increase relative to the size of the input array. We use the same amount of memory and create the same amount of variables regardless of the size of our input. A quick indicator of this is the fact that we don't create any arrays.

When should we use Selection Sort?

There is really only one use case where Selection Sort becomes superior to Bubble Sort. Both algorithms are quadratic in time and constant in space, but the point at which they differ is in the *number of swaps* they make.

Bubble Sort, in the worst case, invokes a swap on every single comparison. Selection Sort only swaps once our inner loop has completely finished traversing the array. Therefore, Selection Sort is optimized to make the least possible number of swaps.

Selection Sort becomes advantageous when making a swap is the most expensive operation in your system. You will likely rarely encounter this scenario, but in a situation where you've built (or have inherited) a system with suboptimal write speed ability, for instance, maybe you're sorting data in a specialized database tuned strictly for fast read speeds at the expense of slow write speeds, using Selection Sort would save you a ton of expensive operations that could potential crash your system under peak load.

Though in industry this situation is very rare, the insights above make for a fantastic conversational piece when weighing technical tradeoffs while strategizing solutions in an interview setting. This commentary may help deliver the impression that you are well-versed in system design and technical analysis, a key indicator that someone is prepared for a senior level position.

Insertion Sort Analysis

Take a look at the snippet below and try to understand how it corresponds to our conceptual understanding of the Insertion Sort algorithm. Scroll down to the commented version when you get stuck:

```
function insertionSort(arr) {
  for (let i = 1; i < arr.length; i++) {
    let currElement = arr[i];
    for (var j = i - 1; j >= 0 && currElement < arr[j]; j--) {
      arr[j + 1] = arr[j];
    }
    arr[j + 1] = currElement;
  }
  return arr;
}
```

```
function insertionSort(arr) {
    // the `i` loop will iterate through every element of the array
    // we begin at i = 1, because we can consider the first element of the arra
    // trivially sorted region of only one element
    // insertion sort allows us to insert new elements anywhere within the sort
    for (let i = 1; i < arr.length; i++) {
        // grab the first element of the unsorted region
        let currElement = arr[i];

        // the `j` loop will iterate left through the sorted region,
        // looking for a legal spot to insert currElement
        for (var j = i - 1; j >= 0 && currElement < arr[j]; j--) {
            // keep moving left while currElement is less than the j-th element

            arr[j + 1] = arr[j];
            // the line above will move the j-th element to the right,
            // leaving a gap to potentially insert currElement
        }
        // insert currElement into that gap
        arr[j + 1] = currElement;

    }
    return arr;
}
```

There are a few key pieces to point out in the above solution before moving forward:

1. The outer `for` loop starts at the 1st index, not the 0th index, and moves to the right.
2. The inner `for` loop starts immediately to the left of the current element, and moves to the left.
3. The condition for the inner `for` loop is complicated, and behaves similarly to a while loop!
    - It continues iterating to the left toward `j = 0`, *only while* the `currElement` is less than `arr[j]`.
    - It does this over and over until it finds the proper place to insert `currElement`, and then we exit the inner loop!
4. When shifting elements in the sorted region to the right, it *does not* replace the value at their old index! If the input array is `[1, 2, 4, 3]`, and `currElement` is 3, after comparing 4 and 3, but before inserting 3 between 2 and 4, the array will look like this: `[1, 2, 4, 4]`.

If you are currently scratching your head, that is perfectly okay because when this one clicks, it clicks for good.

If you're struggling, you should try taking out a pen and paper and step through the solution provided above one step at a time. Keep track of `i`, `j`, `currElement`, `arr[j]`, and the input `arr` itself *at every step*. After going through this a few times, you'll have your "ah HA!" moment.

Time and Space Complexity Analysis

Insertion Sort runtime is O(n^2) because:

In the **worst case scenario** where our input array is entirely unsorted, since this algorithm contains a nested loop, its run time behaves similarly to `bubbleSort` and `selectionSort`. In this case, we are forced to make a comparison at each iteration of the inner loop. Not convinced? Let's derive the complexity. We'll use much of the same argument as we did in `selectionSort`. Say we had the worst case scenario where are input array is sorted in full decreasing order, but we wanted to sort it in increasing order:

- n is the length of the input array
- The outer loop i contributes O(n) in isolation, this is plain to see
- The inner loop j is more complicated. We know j will iterate until it finds an appropriate place to insert the `currElement` into the sorted region. However, since we are discussing the case where the data is already in decreasing order, the element must travel the maximum distance to find it's insertion point! We know this insertion point to be index 0, since every `currElement` will be the next smallest of the array. So:
    - the 1st element travels 1 distance to be inserted
    - the 2nd element travels 2 distance to be inserted
    - the 3rd element travels 3 distance to be inserted
    - ...
    - the n-1th element travels n-1 distance to be inserted
    - This means that our inner loop j will contribute roughly O(n / 2) on average
- The two loops are nested so our total time complexity is O(n * n / 2) = O(n^2)

Space Complexity: O(1)

The amount of memory consumed by the algorithm does not increase relative to the size of the input array. We use the same amount of memory and create the same amount of variables regardless of the size of our input. A quick indicator of this is the fact that we don't create any arrays.

When should you use Insertion Sort?

Insertion Sort has one advantage that makes it absolutely supreme in one special case. Insertion Sort is what's known as an "online" algorithm. Online algorithms are great when you're dealing with *streaming data*, because they can sort the data live *as it is received*.

If you must sort a set of data that is ever-incoming, for example, maybe you are sorting the most relevant posts in a social media feed so that those posts that are most likely to impact the site's audience always appear at the top of the feed, an online algorithm like Insertion Sort is a great option.

Insertion Sort works well in this situation because the left side of the array is always sorted, and in the case of nearly sorted arrays, it can run in linear time. The absolute best case scenario for Insertion Sort is when there is only one unsorted element, and it is located all the way to the right of the array.

Well, if you have data constantly being pushed to the array, it will always be added to the right side. If you keep your algorithm constantly running, the left side will always be sorted. Now you have linear time sort.

Otherwise, Insertion Sort is, in general, useful in all the same situations as Bubble Sort. It's a good option when:

- You are sorting really small arrays where run time will be negligible no matter what algorithm we choose.
- You are sorting an array that you expect to already be nearly sorted.

Merge Sort Analysis

You needed to come up with two pieces of code to make merge sort work.

Full code

```
function merge(array1, array2) {
  let merged = [];

  while (array1.length || array2.length) {
    let ele1 = array1.length ? array1[0] : Infinity;
    let ele2 = array2.length ? array2[0] : Infinity;

    let next;
    if (ele1 < ele2) {
      next = array1.shift();
    } else {
      next = array2.shift();
    }

    merged.push(next);
  }

  return merged;
}

function mergeSort(array) {
  if (array.length <= 1) {
    return array;
  }

  let midIdx = Math.floor(array.length / 2);
  let leftHalf = array.slice(0, midIdx);
  let rightHalf = array.slice(midIdx);

  let sortedLeft = mergeSort(leftHalf);
  let sortedRight = mergeSort(rightHalf);

  return merge(sortedLeft, sortedRight);
}
```

Merging two sorted arrays

Merging two sorted arrays is simple. Since both arrays are sorted, we know the smallest numbers to always be at the front of the arrays. We can construct the new array by comparing the first elements of both input arrays. We remove the smaller element from it's respective array and add it to our new array. Do this until both input arrays are empty:

```
function merge(array1, array2) {
  let merged = [];

  while (array1.length || array2.length) {
    let ele1 = array1.length ? array1[0] : Infinity;
    let ele2 = array2.length ? array2[0] : Infinity;

    let next;
    if (ele1 < ele2) {
      next = array1.shift();
    } else {
      next = array2.shift();
    }

    merged.push(next);
  }

  return merged;
}
```

Remember the following about JavaScript to understand the above code.

- `0` is considered a falsey value, meaning it acts like `false` when used in Boolean expressions. All other numbers are truthy.

- `Infinity` is a value that is guaranteed to be greater than any other quantity
- `shift` is an array method that removes and returns the first element

Here's the annotated version.

```
// commented
function merge(array1, array2) {
  let merged = [];

  // keep running while either array still contains elements
  while (array1.length || array2.length) {
    // if array1 is nonempty, take its the first element as ele1
    // otherwise array1 is empty, so take Infinity as ele1
    let ele1 = array1.length ? array1[0] : Infinity;

    // do the same for array2, ele2
    let ele2 = array2.length ? array2[0] : Infinity;

    let next;
    // remove the smaller of the eles from it's array
    if (ele1 < ele2) {
      next = array1.shift();
    } else {
      next = array2.shift();
    }

    // and add that ele to the new array
    merged.push(next);
  }

  return merged;
}
```

By using `Infinity` as the default element when an array is empty, we are able to elegantly handle the scenario where one array empties before the other. We know that any actual element will be less than `Infinity` so we will continually take the other element into our merged array.

In other words, we can safely handle this edge case:

```
merge([10, 13, 15, 25], []);  // => [10, 13, 15, 25]
```

Nice! We now have a way to merge two sorted arrays into a single sorted array. It's worth mentioning that `merge` will have a `O(n)` runtime where `n` is the combined length of the two input arrays. This is what we meant when we said it was "easy" to merge two sorted arrays; linear time is fast! We'll find fact this useful later.

Divide and conquer, step-by-step

Now that we satisfied the merge idea, let's handle the second point. That is, we say an array of 1 or 0 elements is already sorted. This will be the base case of our recursion. Let's begin adding this code:

```
function mergeSort(array) {
    if (array.length <= 1) {
        return array;
    }
    // ....
}
```

If our base case pertains to an array of a very small size, then the design of our recursive case should make progress toward hitting this base scenario. In other words, we should recursively call `mergeSort` on smaller and smaller arrays. A logical way to do this is to take the input array and split it into left and right halves.

```
function mergeSort(array) {
    if (array.length <= 1) {
        return array;
    }

    let midIdx = Math.floor(array.length / 2);
    let leftHalf = array.slice(0, midIdx);
    let rightHalf = array.slice(midIdx);

    let sortedLeft = mergeSort(leftHalf);
    let sortedRight = mergeSort(rightHalf);
    // ...
}
```

Here is the part of the recursion where we do a lot of hand waving and we take things on faith. We know that `mergeSort` will take in an array and return the sorted version; we assume that it works. That means the two recursive calls will return the `sortedLeft` and `sortedRight` halves.

Okay, so we have two sorted arrays. We want to return one sorted array. So `merge` them! Using the `merge` function we designed earlier:

```
function mergeSort(array) {
    if (array.length <= 1) {
        return array;
    }

    let midIdx = Math.floor(array.length / 2);
    let leftHalf = array.slice(0, midIdx);
    let rightHalf = array.slice(midIdx);

    let sortedLeft = mergeSort(leftHalf);
    let sortedRight = mergeSort(rightHalf);

    return merge(sortedLeft, sortedRight);
}
```

Wow. that's it. Notice how light the implementation of `mergeSort` is. Much of the heavy lifting (the actually comparisons) is done by the `merge` helper.

`mergeSort` is a classic example of a "Divide and Conquer" algorithm. In other words, we keep breaking the array into smaller and smaller sub arrays. This is the same as saying we take the problem and break it down into smaller and smaller subproblems. We do this until the subproblems are so small that we trivially know the answer to them (an array length 0 or 1 is already sorted). Once we have those subanswers we can combine to reconstruct the larger problems that we previously divided (merge the left and right subarrays).

Time and Space Complexity Analysis

Time Complexity: O(n log(n))

- n is the length of the input array
- We must calculate how many recursive calls we make. The number of recursive calls is the number of times we must split the array to reach the base case. Since we split in half each time, the number of recursive calls is `O(log(n))`.
    - for example, say we had an array of length `32`
    - then the length would change as `32 -> 16 -> 8 -> 4 -> 2 -> 1`, we have to split 5 times before reaching the base case, `log(32) = 5`
    - in our algorithm, **log(n)** describes how many times we must halve **n** until the quantity reaches 1.
- Besides the recursive calls, we must consider the while loop within the `merge` function, which contributes `O(n)` in isolation
- We call `merge` in every recursive `mergeSort` call, so the total complexity is **O(n * log(n))**

Space Complexity: O(n)

Merge Sort is the first non-O(1) space sorting algorithm we've seen thus far.

The larger the size of our input array, the greater the number of subarrays we must create in memory. These are not free! They each take up finite space, and we will need a new subarray for each element in the original input. Therefore, Merge Sort has a linear space complexity, O(n).

When should you use Merge Sort?

Unless we, the engineers, have access in advance to some unique, exploitable insight about our dataset, it turns out that O(n log n) time is *the best* we can do when sorting unknown datasets.

That means that Merge Sort is fast! It's way faster than Bubble Sort, Selection Sort, and Insertion Sort. However, due to its linear space complexity, we must always weigh the trade off between speed and memory consumption when making the choice to use Merge Sort. Consider the following:

- If you have unlimited memory available, use it, it's fast!
- If you have a decent amount of memory available and a medium sized dataset, run some tests first, but use it!
- In other cases, maybe you should consider other options.

Quick Sort

Let's begin structuring the recursion. The base case of any recursive problem is where the input is so trivial, we immediately know the answer without calculation. If our problem is to sort an array, what is the trivial array? An array of 1 or 0 elements! Let's establish the code:

```
function quickSort(array) {
    if (array.length <= 1) {
        return array;
    }
    // ...
}
```

If our base case pertains to an array of a very small size, then the design of our recursive case should make progress toward hitting this base scenario. In other words, we should recursively call `quickSort` on smaller and smaller arrays. This is very similar to our previous `mergeSort`, except we don't just split the array down the middle. Instead we should arbitrarily choose an element of the array as a pivot and partition the remaining elements relative to this pivot:

```
function quickSort(array) {
    if (array.length <= 1) {
        return array;
    }

    let pivot = array.shift();
    let left = array.filter(el => el < pivot);
    let right = array.filter(el => el >= pivot);
    // ...
```

Here is what to notice about the partition step above:

1. the pivot is an element of the array; we arbitrarily chose the first element
2. we removed the pivot from the master array before we filter into the left and right partitions

Now that we have the two subarrays of `left` and `right` we have our subproblems! To solve these subproblems we must sort the subarrays. I wish we had a function that sorts an array...oh wait we do, `quickSort`! Recursively:

```
function quickSort(array) {
    if (array.length <= 1) {
        return array;
    }

    let pivot = array.shift();
    let left = array.filter(el => el < pivot);
    let right = array.filter(el => el >= pivot);

    let leftSorted = quickSort(left);
    let rightSorted = quickSort(right);
    // ...
```

Okay, so we have the two sorted partitions. This means we have the two subsolutions. But how do we put them together? Think about how we partitioned them in the first place. Everything in `leftSorted` is **guaranteed** to be less than everything in `rightSorted`. On top of that, `pivot` should be placed after the last element in `leftSorted`, but before the first element in `rightSorted`. So all we need to do is to combine the elements in the order "left, pivot, right"!

```
function quickSort(array) {
    if (array.length <= 1) {
        return array;
    }

    let pivot = array.shift();
    let left = array.filter(el => el < pivot);
    let right = array.filter(el => el >= pivot);

    let leftSorted = quickSort(left);
    let rightSorted = quickSort(right);

    return leftSorted.concat([pivot]).concat(rightSorted);
}
```

That last `concat` line is a bit clunky. Bonus JS Lesson: we can use the spread `...` operator to elegantly concatenate arrays. In general:

```
let one = ['a', 'b']
let two = ['d', 'e', 'f']
let newArr = [ ...one, 'c', ...two  ];
newArr; // =>  [ 'a', 'b', 'c', 'd', 'e', 'f' ]
```

Utilizing that spread pattern gives us this final implementation:

```
function quickSort(array) {
    if (array.length <= 1) {
        return array;
    }

    let pivot = array.shift();
    let left = array.filter(el => el < pivot);
    let right = array.filter(el => el >= pivot);

    let leftSorted = quickSort(left);
    let rightSorted = quickSort(right);

    return [ ...leftSorted, pivot, ...rightSorted ];
}
```

Quicksort Sort JS Implementation

That code was so clean we should show it again. Here's the complete code for your reference, for when you `ctrl+F "quicksort"` the night before an interview:

```
function quickSort(array) {
    if (array.length <= 1) {
        return array;
    }

    let pivot = array.shift();
    let left = array.filter(el => el < pivot);
    let right = array.filter(el => el >= pivot);

    let leftSorted = quickSort(left);
    let rightSorted = quickSort(right);

    return [ ...leftSorted, pivot, ...rightSorted ];
}
```

Time and Space Complexity Analysis

Here is a summary of the complexity.

Time Complexity

- Avg Case: O(n log(n))
- Worst Case: O(n^2)

The runtime analysis of `quickSort` is more complex than `mergeSort`

- n is the length of the input array
- The partition step alone is `O(n)`
- We must calculate how many recursive calls we make. The number of recursive calls is the number of times we must split the array to reach the base case. This is dependent on how we choose the pivot. Let's analyze the best and worst case:
    - **Best Case:** We are lucky and always choose the median as the pivot. This means the left and right partitions will have equal length. This will halve the array length at every step of the recursion. We benefit from this halving with `O(log(n))` recursive calls to reach the base case.
    - **Worst Case:** We are unlucky and always choose the min or max as the pivot. This means one partition will contain everything, and the other partition is empty. This will decrease the array length by 1 at every step of the recursion. We suffer from `O(n)` recursive calls to reach the base case.
- The partition step occurs in every recursive call, so our total complexities are:
    - **Best Case:** O(n * log(n))

    - **Worst Case:** O(n^2)

Although we typically take the worst case when describing Big-O for an algorithm, much research on `quickSort` has shown the worst case to be an exceedingly rare occurrence even if we choose the pivot at random. Because of this we still consider `quickSort` an efficient algorithm. This is a common interview talking point, so you should be familiar with the relationship between the choice of pivot and efficiency of the algorithm.

Just in case: A somewhat common question a student may ask when studying `quickSort` is, "If the median is the best pivot, why don't we always just choose the median when we partition?" Don't overthink this. To know the median of an array, it must be sorted in the first place.

Space Complexity

Our implementation of `quickSort` uses `O(n)` space because of the partition arrays we create. There is an in-place version of `quickSort` that uses `O(log(n))` space. `O(log(n))` space is not huge benefit over `O(n)`. You'll also find our version of `quickSort` as easier to remember, easier to implement. Just know that a `O(logn)` space `quickSort` exists.

When should you use Quick Sort?

- When you are in a pinch and need to throw down an efficient sort (on average). The recursive code is light and simple to implement; much smaller than `mergeSort`.
- When constant space is important to you, use the in-place version. This will of course trade off some simplicity of implementation.

If you know some constraints about dataset you can make some modifications to optimize pivot choice. Here's some food for thought. Our implementation of `quickSort` will always take the first element as the pivot. This means we will suffer from the worst case time complexity in the event that we are given an already sorted array (ironic isn't it?). If you know your input data to be mostly already sorted, randomize the choice of pivot - this is a very easy change. Bam. Solved like a true engineer.

Binary Search

We'll implement binary search recursively. As always, we start with a base case that captures the scenario of the input array being so trivial, that we know the answer without further calculation. If we are given an empty array and a target, we can be certain that the target is not inside of the array:

```
function binarySearch(array, target) {
    if (array.length === 0) {
        return false;
    }
    // ...
}
```

Now for our recursive case. If we want to get a time complexity less than `O(n)`, we must avoid touching all n elements. Adopting our dictionary strategy, let's find the middle element and grab references to the left and right halves of the sorted array:

```
function binarySearch(array, target) {
    if (array.length === 0) {
        return false;
    }

    let midIdx = Math.floor(array.length / 2);
    let leftHalf = array.slice(0, midIdx);
    let rightHalf = array.slice(midIdx + 1);
    // ...
}
```

It's worth pointing out that the left and right halves do not contain the middle element we chose.

Here is where we leverage the sorted property of the array. If the target is less than the middle, then the target must be in the left half of the array. If the target is greater than the middle, then the target must be in the right half of the array. So we can narrow our search to one of these halves, and ignore the other. Luckily we have a function that can search the half, its `binarySearch`:

```
function binarySearch(array, target) {
    if (array.length === 0) {
        return false;
    }

    let midIdx = Math.floor(array.length / 2);
    let leftHalf = array.slice(0, midIdx);
    let rightHalf = array.slice(midIdx + 1);

    if (target < array[midIdx]) {
        return binarySearch(leftHalf, target);
    } else if (target > array[midIdx]) {
        return binarySearch(rightHalf, target);
    }
    // ...
}
```

We know `binarySeach` will return the correct Boolean, so we just pass that result up by returning it ourselves. However, something is lacking in our code. It is only possible to get a false from the literal `return false` line, but there is no `return true`. Looking at our conditionals, we handle the cases where the target is less than middle or the target is greater than the middle, but what if the product is **equal** to the middle? If the target is equal to the middle, then we found the target and should `return true`! This is easy to add with an `else`:

```
function binarySearch(array, target) {
    if (array.length === 0) {
        return false;
    }

    let midIdx = Math.floor(array.length / 2);
    let leftHalf = array.slice(0, midIdx);
    let rightHalf = array.slice(midIdx + 1);

    if (target < array[midIdx]) {
        return binarySearch(leftHalf, target);
    } else if (target > array[midIdx]) {
        return binarySearch(rightHalf, target);
    } else {
        return true;
    }
}
```

To wrap up, we have confidence of our base case will eventually be hit because we are continually halving the array. We halve the array until it's length is 0 or we actually find the target.

Binary Search JS Implementation

Here is the code again for your quick reference:

```
function binarySearch(array, target) {
    if (array.length === 0) {
        return false;
    }

    let midIdx = Math.floor(array.length / 2);
    let leftHalf = array.slice(0, midIdx);
    let rightHalf = array.slice(midIdx + 1);

    if (target < array[midIdx]) {
        return binarySearch(leftHalf, target);
    } else if (target > array[midIdx]) {
        return binarySearch(rightHalf, target);
    } else {
        return true;
    }
}
```

Time and Space Complexity Analysis

The complexity analysis of this algorithm is easier to explain through visuals, so we **highly encourage** you to watch the lecture that accompanies this reading. In any case, here is a summary of the complexity:

Time Complexity: O(log(n))

- n is the length of the input array
- We have no loops, so we must only consider the number of recursive calls it takes to hit the base case
- The number of recursive calls is the number of times we must halve the array until it's length becomes 0. This number can be described by `log(n)`
    - for example, say we had an array of 8 elements, `n = 8`
    - the length would halve as `8 -> 4 -> 2 -> 1`
    - it takes 3 calls, `log(8) = 3`

Space Complexity: O(n)

Our implementation uses n space due to half arrays we create using slice. Note that JavaScript `slice` creates a new array, so it requires additional memory to be allocated.

When should we use Binary Search?

Use this algorithm when the input data is sorted!!! This is a heavy requirement, but if you have it, you'll have an insanely fast algorithm. Of course, you can use one of your high-functioning sorting algorithms to sort the input and *then* perform the binary search!