GitHub, Big-O, and Intro to Data Structures and Algorithms (Week 7) - Learning Objectives

Assessment Structure

- 2 hours
- Mixture of multiple choice (10-15) and VSCode (3-5) problems, each with multiple specs.
    - Coding problems will have specs to run (`npm test`) and check your work against
- Standard assessment procedures
    - You will be in an individual breakout room
    - Use a single monitor and share your screen
    - Only have open those resources needed to complete the assessment:
        - Zoom
        - VSCode
        - Browser with AAO and Progress Tracker (to ask questions)
        - Approved Resources for this assessment:
        - MDN: https://developer.mozilla.org/en-US/docs/Web/JavaScript

GitHub (W7D1) - Learning Objectives

GitHub

1. You will be able to participate in the social aspects of GitHub by starring repositories, following other developers, and reviewing your followers

- Doing these basic activities shows that you are not only an active programmer, but active in the overall community, which is a positive attribute during the job search.

2. You will be able to use Markdown to write code snippets in your README files

- Great cheatsheet for markdown syntax: https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet

3. You will craft your GitHub profile and contribute throughout the course by keeping your "gardens green"

- Make commits to your projects throughout the day.
- Push your projects up to your own repository at the end of the day.

4. You will be able to identify the basics of a good Wiki entries for proposals and minimum viable products

- Design documents give a roadmap for how you envision the project developing.
- They serve as a planning tool as well as a resource during development.
- Minimum Viable Products (MVPs) indicate the features that need to be implemented in order to have a completed application. They also indicate your stepping stones for completing the project.

5. You will be able to identify the basics of a good project README that includes technologies at the top, images, descriptions and code snippets

- Displaying all of these within your README gives someone a preview of the project.
- It highlights what it looks like, describes its main features, and discusses at a high-level how it was implemented.

Big-O, Memoization, and Tabulation (W7D2) - Learning Objectives

Big-O

1. Order the common complexity classes according to their growth rate

- The following are in order from smallest growth (most efficient) to largest:
    - constant O(1)
    - logarithmic O(log n)
    - linear O(n)
    - loglinear, linearithmic, quasilinear O(n log n)
    - polynomial O(n^c) -> O(n^2), O(n^3)
    - exponential O(c^n) -> O(2^n), O(3^n)
    - factorial O(n!)

2. Identify the complexity classes of common sort methods

Sort Name	Time Complexity	Space Complexity
bubble	O(n^2)	O(1)
selection	O(n^2)	O(1)
insertion	O(n^2)	O(1)
merge	O(n log n)	O(n)
quick	*O(n log n)/O(n^2)	*O(n)/O(log n)

- *quick sort's complexities are a little more complicated
    - We are generally only concerned with the worst-scenario when we talk Big-O.
    - With quick sort, the worst case is exceedingly rare (only occurs when our pivot for each round happens to be the next element, resulting in us having to choose n pivot points)
    - Because it is so rare that this occurs, most people will use consider quick sort to be closer to O(n log n) time complexity.
    - We also have two space complexities listed. The version that we used in class creates a new array, resulting in O(n) space. With some tweaking, we can sort in place, modifying the original array and cutting our space complexity to O(log n), which is just a result of the stack frames that we have to create. It's good to know this method exists, but you will not need to create or identify this version.

3. Identify complexity classes of code

- Important takeaway here is being able to connect code patterns with complexities
    - Doing an exact number of calculations (independent of input) -> constant O(n)

```
function constant_1(n) {
  return n * 2 + 1;
}
```

- Looping an exact number of times (independent of input) -> constant O(n)

```
function constant_2(n) {
  for (let i = 1; i <= 20; i++) {
    console.log(i);
  }
}
```

- Recursive calls that divide the input -> logarithmic O(log n)

```
function logarithmic(n) {
  console.log(n);
  if (n <= 1) return;
  logarithmic(n / 2);
}
```

- Loops that depend on the size of the input -> linear O(n)

```
function linear_1(n) {
  for (let i = 1; i <= n; i++) {
    console.log(i);
  }
}
```

- Recursive calls that depend on the size of the input (decrementing instead of dividing) -> linear O(n)

```
function linear_2(n) {
  console.log(n);
  if (n === 1) return;
  linear_2(n - 1);
}
```

- Looping through input on each stack frame, while recursively dividing our data (commonly seen in sorts like merge and quick sort) -> loglinear O(n log n)

```

```

```
function loglinear(n) {
  if (n <= 1) return;
  for (let i = 1; i <= n; i++) { // n calculations in each stack frame
    console.log(n);
  }
  loglinear(n / 2); // log n number of stack frames
  loglinear(n / 2);
}
```

- Nesting loops that depend on the size of the input -> polynomial O(n^c)

```
// O(n^2)
function quadratic(n) {
  for (let i = 1; i <= n; i++) {
    for (let j = 1; j <= n; j++) {
      console.log(`${i}, ${j}`);
    }
  }
}

// O(n^3)
function cubic(n) {
  for (let i = 1; i <= n; i++) {
    for (let j = 1; j <= n; j++) {
      for (let k = 1; k <= n; k++) {
        console.log(`${i}, ${j}, ${k}`);
      }
    }
  }
}
```

- Branching out on each recursive call, with the number of calls dependent on the size of the input -> exponential O(c^n)

```
// O(2^n)
function exponential_2n(n) {
  console.log(n);
  if (n === 1) return;
  exponential_2n(n - 1);
  exponential_2n(n - 1);
}

// O(3^n)
function exponential_3n(n) {
  console.log(n);
  if (n === 1) return;
  exponential_3n(n - 1);
  exponential_3n(n - 1);
```

```
    exponential_3n(n - 1);
  }
```

- When both the number of recursive calls and the number of branches made in the calls are dependent on the size of the input -> factorial O(n!)

```
function factorial(n) {
  console.log(n);
  if (n === 1) return;
  for (let i = 1; i <= n; i++) { // Here we're making n branches on this
frame
    factorial(n - 1); // Since we are decrementing, we're making n stack
frames
  }
}
```

Memoization and Tabulation

1. Apply memoization to recursive problems to make them less than polynomial time.

- Main steps for memoizing a problem:
  1. Write out the brute force recursion
  2. Add the memo object as an additional argument
     - Keys on this object represent input, values are the corresponding output
  3. Add a base condition that returns the stored value if the argument is already in the memo
  4. Before returning a calculation, store the result in the memo for future use
- Example of a standard and memoized fibonacci:

```
// Standard Implementation
// Time Complexity: O(2^n)
//   - For each call to fib, we have to make two branches, with n levels to this
tree
function fib(n) {
  if (n === 1 || n === 2) return 1;
  return fib(n - 1) + fib(n - 2);
}

// Using memoization
// Time Complexity: O(n)
//   - We only ever have to calculate fib(x) one time, every other time that we
use it in a larger problem, the result is immediately accessible in our memo
//   - The memo removes the repeated trees of calculations from our original code
function fib(n, memo = {}) {
  if (n in memo) return memo[n]; // If we already calculated this value, return it
  if (n === 1 || n === 2) return 1;

  // Store the result in the memo first before returning
  // Make sure to pass the memo in to your calls to fib!
```

```
  memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

2. Apply tabulation to iterative problems to make them less than polynomial time.

- Main considerations for using tabulation:
    - Figure out how big is your table
        - Typically going to be base on input size (number in fibonacci, length of string in wordBreak)
    - What does my table represent?
        - You are generally building up your answer.
        - In fibonacci, we used the table to store the fib number at the corresponding index.
        - In stepper, we stored the boolean of whether it was possible to get to that location.
    - What initial values do I need to seed?
        - Consider what your end result should be (boolean, number, etc.).
        - Your seed data is generally going to be the immediate answer that we know from our base condition.
        - In fibonacci, we knew the first two numbers of the series.
        - In stepper, we knew that it was possible to get to our starting location, so we seeded it as true, defaulting the rest to false so that we could later change its value if we could make that step.
    - How do I iterate and fill out my table?
        - We typically need to iterate over or up to our input in some way in order to update and build up our table until we get our final result.
        - In fibonacci, we iterated up to our input number in order to calculate the fib number at each step.
        - In stepper, we iterated over each possible stepping location. If we could have made it to that point from our previous steps (ie that index was true in our table), we continued updating our table by marking the possible landing spots as true.
- Example of a tabulated fibonacci:

```
// Using tabulation
// Time Complexity: O(n)
//   - We are iterating through an array directly related to the size of the input
and performing a constant number of calculations at each step (adding our two
previous values together and storing the result in the array).
function fib(n) {
  // We create a table to track our values as we build them up
  // We're making it n+1 here so that table[n] lines up with the nth fib number
  // This is because arrays are zero-indexed.
  // We could have used an array of length n, but we would have to remember that
  // the nth fib number would then be stored at table[n-1]. Completely doable,
  // but I think making them line up is more intuitive.
  let table = new Array(n + 1);
  // Seed our table with our starting values.
  // Again, if we had a table of length n, we could have seeded table[0] = 1
  // and table[1] = 1 and had the same final result with our indices shifted.
```

```
  table[0] = 0;
  table[1] = 1;

  // Iterate through our input and fill out our table as we go.
  for (let i = 2; i < table.length; i++) {
    table[i] = table[i - 1] + table[i - 2];
  }

  // At the end, the final entry in our table is the result we are looking for.
  // The table holds all of the sub-answers that we used to get to this result.
  return table[n];
}
```

Sorting Algorithms (W7D3) - Learning Objectives

Sorting Algorithms

1. Explain the complexity of and write a function that performs bubble sort on an array of numbers.

- Time Complexity: O(n^2)
  - In our worst case, our input is in the opposite order. We have to perform n swaps and loop through our input n times because a swap is made each time.
- Space Complexity: O(1)
  - We are creating the same number of variables with an exact size, independent of our input. No new arrays are created.
- Code example for bubbleSort:

```
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]) {
        let temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
        // The above three lines could also be in a helper swap function
        // swap(array, i, i+1);
        swapped = true;
      }
    }
  }

  return array;
}
```

2. Explain the complexity of and write a function that performs selection sort on an array of numbers.

- Time Complexity: O(n^2)
  - Our nested loop structure is dependent on the size of our input.
  - The outer loop always occurs n times.
  - For each of those iterations, we have another loop that runs (n - i) times. This just means that the inner loop runs one less time each iteration, but this averages out to (n/2).
  - Our nested structure is then T(n * n/2) = O(n^2)
- Space Complexity: O(1)
  - We are creating the same number of variables with an exact size, independent of our input. No new arrays are created.
- Code example for selectSort:

```
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;
      }
    }

    let temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
    // The above three lines could also be in a helper swap function
    // swap(arr, i, minIndex);
  }
  return arr;
}
```

3. Explain the complexity of and write a function that performs insertion sort on an array of numbers.

- Time Complexity: O(n^2)
  - Our nested loop structure is dependent on the size of our input.
  - The outer loop always occurs n times.
  - For each of those iterations, we have another loop that runs a maximum of (i - 1) times. This just means that the inner loop runs one more time each iteration, but this averages out to (n/2).
  - Our nested structure is then T(n * n/2) = O(n^2)
- Space Complexity: O(1)
  - We are creating the same number of variables with an exact size, independent of our input. No new arrays are created.
- Code example for insertionSort:

```
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;
}
```

4. Explain the complexity of and write a function that performs merge sort on an array of numbers.

- Time Complexity: O(n log n)
  - Our mergeSort function divides our input in half at each step, recursively calling itself with smaller and smaller input. This results in log n stack frames.
  - On each stack frame, our worst case scenario is having to make n comparisons in our merge function in order to determine which element should come next in our sorted array.
  - Since we have log n stack frames and n operations on each frame, we end up with an O(n log n) time complexity
- Space Complexity: O(n)
  - We are ultimately creating n subarrays, making our space complexity linear to our input size.
- Code example for mergeSort:

```
// The merge function is what is combining our sorted sub-arrays
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;
}

// The mergeSort function breaks apart our input into smaller sub-arrays until we
have an input of length <= 1, which is inherently sorted.
// Once we have a left and right subarray that's sorted, we can merge them
```

```
together to get our sorted result of this sub-problem, passing the sorted version
back up the call stack.
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);
}
```

5. Explain the complexity of and write a function that performs quick sort on an array of numbers.

- Time Complexity: Average O(n log n), Worst O(n^2)
    - In our worst case, the pivot that we select results in every element going into either the left or right array. If this happens we end up making n recursive calls to quickSort, with n comparisons at each call.
    - In our average case, we pick something that more evenly splits the arrays, resulting in approximately log n recursive calls and an overall complexity of O(n log n).
    - Quick sort is unique in that the worst case is so exceedingly rare that it is often considered an O(n log n) complexity, even though this is not technically accurate.
- Space Complexity: Our implementation O(n), Possible implementation O(log n)
    - The partition arrays that we create are directly proportional to the size of the input, resulting in O(n) space complexity.
    - With some tweaking, we could implement an in-place quick sort, which would eliminate the creation of new arrays. In this case, the log n stack frames from the recursion are the only proportional amount of space that is used, resulting in O(log n) space complexity.
- Code example for quickSort:

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

  let pivot = array.shift();
  // This implementation uses filter, which returns a new array with any element
that passes the criteria (ie the callback returns true).
  // We also could have iterated over the array (array.forEach(el => ...)) and
pushed each value into the appropriate left/right subarray as we encountered it.
  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 ];
  // We also could have concatenated the arrays instead of spreading their
contents
  // return leftSorted.concat([pivot]).concat(rightSorted);
}
```

6. Explain the complexity of and write a function that performs a binary search on a sorted array of numbers.

- Time Complexity: O(log n)
    - With each recursive call, we split our input in half. This means we have to make at most log n checks to know if the element is in our array.
- Space Complexity: Our implementation O(n), Possible implementation O(1)
    - We have to make a subarray for each recursive call. In the worst case (we don't find the element), the total length of these arrays is approximately equal to the length of the original (n).
    - If we kept references to the beginning and end index of the portion of the array that we are searching, we could eliminate the need for creating new subarrays. We could also use a while loop to perform this functionality until we either found our target or our beginning and end indices crossed. This would eliminate the space required for recursive calls (adding stack frames). Ultimately we would be using the same number of variables independent of input size, resulting in O(1).
- Code example for binarySearch and binarySearchIndex:

```
// Returns simply true/false for presence
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;
  }
}

// Returns the index or -1 if not found
function binarySearchIndex(array, target) {
  if (!array.length) return -1;

  const midIdx = Math.floor(array.length / 2);
  const midEl = array[midIdx];
```

```
  if (target < midEl) {
    return binarySearchIndex(array.slice(0, midIdx), target);
  } else if (target > midEl) {
    // Since our recursive call will have new indices for the subarray, we have to
adjust the return value to align it with the indices of our original array.
    // If the recursive call returns -1, it was not found and we can immediately
return -1
    // If it was found in the subarray, we have to add on the number of elements
that were removed from the beginning of our larger original array.
    // For example, if we try to find 15 in an array of [5, 10, 15]:
      // - Our first call to binarySearchIndex will check our middle element of 10
      // - Since our target is greater, we will recursively call our search on
elements to the right, being the subarray [15]
      // - On our recursive call we found our target! It's index in this call is
0.
      // - When we return 0 to where binarySearchIndex was called, we need to
adjust it to line up with this larger array (the 0th element of this larger array
is 5, but our target was at the 0th index of the subarray)
      // - Since we sliced off 2 elements from the beginning before making our
recursive call, we add 2 to the return value to adjust it back to line up with our
original array.
    const idxShift = binarySearchIndex(array.slice(midIdx + 1), target);
    return idxShift === -1 ? -1 : idxShift + midIdx + 1;
  } else {
    return midIdx;
  }
}
```

Lists, Stacks, and Queues (W7D4) - Learning Objectives

Lists, Stacks, and Queues

1. Explain and implement a Linked List.

- A linked list are a collection of ordered data that track three main components:
  - head: beginning of the list
  - tail: end of the list
  - length: count of the number of elements in the list
- The main differences between lists and arrays are that a list does not have random access or indices to signify where in the list an element is.
  - The only references to elements that we have in a list are the head and the tail.
  - If we want an element in the middle of the list, we would have to traverse the list until we encountered it.
- The two main types of linked lists that we talked about are Singly Linked Lists and Doubly Linked Lists.
  - Singly Linked Lists are composed of nodes that only have a reference to the next node in the list. We can only traverse the list in one direction.
  - Doubly Linked Lists are composed of nodes that have a reference to both the next node and the previous node in the list. This allows us to traverse both forwards and backwards.
- Methods of a linked list that we should know are:

    - addToTail: Adds a new node to the end of the list.
    - addToHead: Adds a new node to the front of the list.
    - insertAt: Adds a new node at the specified position (we need to traverse to that point, then update pointers)
    - removeTail: Removes the last node of the list.
    - removeHead: Removes the first node of the list.
    - removeFrom: Removes the node at the specified position.
    - contains: Traverses the list and returns a boolean to indicate if the value was found at any node.
    - get: Returns a reference to the node at the specified position.
    - set: Updates the value of the node at the specified position.
    - size: Returns the current length of the list.
- Time complexities for these methods:
    - Accessing a node: O(n), because we may have to traverse the entire list.
    - Searching a list: O(n), because we may have to traverse the entire list.
    - Inserting a value: O(1), under the assumption that we have a reference to the node that we want to insert it after/before. If we don't have this reference we would first have to access it (O(n) from above), but the actual creation is O(1)
    - Deleting a node: O(1), for the same reasons as insertion. If we first need to find the previous and next nodes, we would need to access them (O(n) from above), but the actual deletion is O(1)
- Be able to implement a Singly Linked List and a Doubly Linked List. This would require you to use a Node class with a value instance variable and an instance variable that points to the next (and possibly previous) Node instance(s). You should then be able to interact with these Nodes to perform all of the actions of a Linked List, as we defined above.

2. Explain and implement a Stack.

- A Last In First Out (LIFO) Abstract Data Type (ADT).
    - LIFO: The last element put into the stack is the first thing removed from it. Think of it as a can of Pringles or a pile of dishes.
    - ADT: The actual implementation of the stack can vary as long as the main principles and methods associated with them are abided by. We could use Nodes like we did with Linked Lists, we could use an Array as an underlying instance variable as long as the methods we implement only interact with it in the way a stack should be interacted with, etc.
- Methods of a Stack we should know are:
    - push: Adds an element to the top of the stack.
    - pop: Removes an element from the top of the stack.
    - peek: Returns the value of the top element of the stack.
    - size: Returns the number of elements in the stack.
- Time Complexities:
    - Adding an element: O(1), since we are always adding it to the top and the addition doesn't affect any other elements.
    - Removing an element: O(1), we're always taking the top element of the stack.
    - Finding or Accessing a particular element: O(n), since we can only interact with our stack by removing elements from the top, we may have to remove every element to find what we're looking for.

3. Explain and implement a Queue.

- A First In First Out (LIFO) Abstract Data Type (ADT).
    - LIFO: The first element put into the queue is the first thing removed from it. Think of it as if you are waiting in line at a store, first come, first serve.
    - ADT: The actual implementation of the queue can vary as long as the main principles and methods associated with them are abided by. We could use Nodes like we did with Linked Lists, we could use an Array as an underlying instance variable as long as the methods we implement only interact with it in the way a queue should be interacted with, etc.
- Methods of a Queue we should know are:
    - enqueue: Adds an element to the back of the queue.
    - dequeue: Removes an element from the front of the queue.
    - peek: Returns the value of the front element of the queue.
    - size: Returns the number of elements in the queue.
- Time Complexities:
    - Adding an element: O(1), since we are always adding it to the back. If we are using Nodes instead of a simple array, keeping a reference to the last node allows us to immediately update these pointers without having to do any traversal.
    - Removing an element: O(1), we're always taking the front element of the queue.
    - Finding or Accessing a particular element: O(n), since we can only interact with our queue by removing elements from the front, we may have to remove every element to find what we're looking for.

Heaps (W7D5) - Learning Objectives

Heaps

1. What is a max (or min) heap?

- A tree that has partially ordered nodes.
- A heap must be a complete tree.
- Each parent is greater than its children in a max heap, or less than them in a min heap.
- Overall, the root is then either the maximum or minimum value of all of the nodes.

2. How is a binary heap different from a binary search tree?

- Both represent trees with parents that have a maximum of two children.
- The order of children is important in a BST. The left is less than the parent and the right is greater than the parent.
- The only stipulation in a binary heap is that the parent is greater than (max heap) or less than (min heap) the children. There is no distinction between what must go left vs. right other than maintining a complete tree.

3. What is a complete tree? How does it relate to heaps?

- A complete tree is balanced, with the bottom row having nodes as far left as possible.
- This consequently means there are no gaps in the tree.
- Heaps must be complete trees. This lends to an easy representation using basic JavaScript.

4. What is a commmon way to represent heaps in JavaScript?

- We can use an array. Since there are no gaps in the heap, we can add elements to an array and they will represent the next valid position to be filled in the heap.

```
/*

            40
         /      \
       32          24
     /    \       /   \
   30      9    20     18
   /  \
 2    7

*/

// our heap: [null, 40, 32, 24, 30,  9, 20, 18,  2,  7]
//  indices: [   0,  1,  2,  3,  4,  5,  6,  7,  8,  9]
```

- We typically use a `null` at index 0 because it allows for simpler calculations of child/parent indices, but this is not a strict requirement.

5. In this representation, how can we find any particular node's parent or children?

- The left child of an element at index i is going to be at index (2 * i)
- The right child of an element at index i is going to be at index (2 * i + 1)
- Working backwards, the parent of an element at index i is at index Math.floor(i / 2)

6. What processes do we need to follow when we insert an element into a heap?

- To maintain the heap structure, we insert it as far left as possible, then sift the newly inserted value up if necessary until it reaches an appropriate position. This sifting would be necessary if we are currently larger than our parent in a max heap, or smaller in a min heap.
- When considering an array format, this would mean we can simply push our new element onto the end of the array, then compare our value with our parentIdx's value until a swap does not occur.
- From our max_heap project:

```
insert(val) {
  this.array.push(val);
  this.siftUp(this.array.length - 1);
}

siftUp(idx) {
  // If we are already the root, we cannot sift up further
  if (idx === 1) return;

  let parentIdx = this.getParent(idx);

  // If our value is greater than our parents, swap, then call siftUp again
  // on the new location to see if we need to do further swaps.
```

```
  if (this.array[parentIdx] < this.array[idx]) {
    [ this.array[parentIdx], this.array[idx] ] = [ this.array[idx],
this.array[parentIdx] ];
    this.siftUp(parentIdx);
  }
}
```

7. What processes do we need to follow when we remove then root of a heap?

- To maintain the heap structure, we replace our root with the node that is as far left as possible, then sift the newly replaced value down if necessary until it reaches an appropriate position. This sifting would be necessary if we are currently smaller than either child in a max heap, or greater in a min heap.
- When considering an array format, this would mean we can pop our last element off of the array, then replace `array[1]` with this value (we hold on to the replaced value to return at the end of our algorithm). We compare our value with each childIdx's value until a swap does not occur.
- From our max_heap project:

```
deleteMax() {
  // Since our first element is `null` we take the element at index 1.
  // We want to keep null in the array if there are no other elements,
  // which is why we are returning null instead of popping for a length of 1.
  if (this.array.length === 2) return this.array.pop();
  if (this.array.length === 1) return null;

  // Since we're overwriting our index 1, we keep a reference to its value
  // so that we can return it later.
  let max = this.array[1];
  // We reassign the root of our heap to be the last element in our array.
  // Using .pop() removes that element from the end for us as well.
  this.array[1] = this.array.pop();
  // We check to see if the element that took our root's spot needs to be
  // sifted down into an appropriate position.
  this.siftDown(1);
  // After our sifting is done, our heap has been reorganize into a valid
  // configuration. We can now return the value that we originally removed.
  return max;
}

siftDown(idx) {
  let ary = this.array;
  let leftIdx = this.getLeftChild(idx);
  let rightIdx = this.getRightChild(idx);
  let leftVal = ary[leftIdx];
  let rightVal = ary[rightIdx];

  // If we do not have a child, leftVal or rightVal would be `undefined`.
  // We can't perform comparisons to `undefined` so we reassign them to be
  // -Infinity, which will always result in our value being greater, indicating
  // we are in a correct position (we can't sift down when we're already a leaf)
  if (leftVal === undefined) leftVal = -Infinity;
  if (rightVal === undefined) rightVal = -Infinity;
```

```
  // If we are greater than both of our children, we are in our final spot.
  if (ary[idx] > leftVal && ary[idx] > rightVal) return;

  // If one of our children is greater, we made it past the previous conditional.
  // We determine which child is greater, then assign that index as the the
  // one that we need to swap with.
  let swapIdx;
  if (leftVal < rightVal) {
    swapIdx = rightIdx;
  } else {
    swapIdx = leftIdx;
  }

  // We swap our current element with our largest child.
  [ ary[idx], ary[swapIdx] ] = [ ary[swapIdx], ary[idx] ];
  // We invoke siftDown again with the new index to see if we need to sift
further.
  this.siftDown(swapIdx);
}
```

8. Given an array, determine if it represents a max (or min) heap.

- For the array to represent a max heap, it would need to be complete (no `undefined` values, which would indicate a gap) and each parent would have to be greater than its children.
- We can recursively call our function to see if each node is greater than its children.
- Reference the is_heap problem for code example