Week 7 Quiz

Reset Quiz

HIDE GOT IT! What does "MVP" stand for in the context of planning and documenting the work in your github repo?

1. Most vital product
2. Most valuable product
3. Maximum visible product
4. Minimum viable product

#4 EXPLANATION
Your MVP is the minimum viable product that it takes to "go to market", even if "market" just means sharing it with other people.

HIDE GOT IT! What is "starring" at GitHub?

1. It is appearing in a video on GitHub
2. It is "liking" a repository by clicking on a star icon
3. It is "friending" another developer on GitHub
4. It is creating a star icon to put in your README

#2 EXPLANATION
Look at the top of the repository pages. Those stars mean popularity! Don't you want to be popular? Get people to "star" your repository for fame and success! (:shrug:)

HIDE GOT IT! Which three of the following are ways App Academy students and grads bolster their perceived understanding of the software engineering world via GitHub?

1. Following other developers on GitHub
2. Asking peers in the industry to look at their repositories and "star" them
3. Having only one repository
4. Regularly watering your "green gardens" through frequent commits

#1, #2, #4 :EXPLANATION
Having a single repository is like showing up for school without being prepared. Many focused repositories, even if they're old, can show that you're interested in this craft of software engineering.

HIDE GOT IT! Which represents correct markdown syntax for adding a multiline code snippet to your README?

1. Use three single quotes followed by the word "javascript", a newline, your code, and then three more single quotes.
2. Use three backticks followed by the word "javascript", a newline, your code, and then three more backticks.
3. Use three underscores followed by the word "javascript", a newline, your code, and then three more underscores.
4. Use three double quotes followed by the word "javascript", a newline, your code, and then three more double quotes.

#2 EXPLANATION
It's all about the backticks. Those indicate that it should be considered code.

HIDE GOT IT! What is the main reason for having a strong GitHub profile?

1. GitHub is how 80% of job-seeking software engineers find work.
2. GitHub doesn't require any additional description of projects and therefore makes it very easy for people to read your code.
3. GitHub is an online community of developers who can support one another's job search, contribute ideas for projects, and learn from one another's previous projects and experiences.
4. GitHub provides unlimited access to all code bases and therefore gives the opportunity to fork limitless code.

#3 EXPLANATION
If code is about people, then GitHub is the social platform on which that happens. Your profile on GitHub speaks about who you are, what your interests are in, and shows the type of code you can create.

HIDE GOT IT! Order the common complexity classes according to their growth rate:

Name	Big O Notation
Linear Logorithmic	O(1)
Factorial	O(n)
Polynomial	O(log(n))
Linear	O(n!)
Constant	$O(m^n)$
Exponential	O(n*log(n))
Logarithmic	$O(n^m)$

ANSWER

Name	Big O Notation
Constant	O(1)
Logarithmic	O(log(n))
Linear	O(n)
Linear Logarithmic	O(n*log(n))
Polynomial	$O(n^m)$
Exponential	$O(m^n)$
Factorial	O(n!)

HIDE GOT IT! What is the Big-O notation for the Polynomial complexity class?

$O(n^m)$

HIDE GOT IT! What is the Big-O notation for the Exponential complexity class?

$O(m^n)$

HIDE GOT IT! What is the Big-O notation for the Logarithmic complexity class?

O(log(n))

HIDE GOT IT! What is the Big-O notation for the Linear complexity class?

O(n)

HIDE GOT IT! What is the Big-O notation for the Loglinear complexity class?

O(n*log(n))

HIDE GOT IT! What is the Big-O notation for the Constant complexity class?

O(1)

HIDE GOT IT! Order the following complexity classes from most efficient to most complex:

1. $O(m^n)$: exponential
2. $O(n^m)$: polynomial
3. O(n!) : factorial
4. O(1) : constant
5. O(n) : linear
6. O(n * log(n)) : logLinear
7. O(log(n)) : logarithmic

1. O(1) : constant
2. O(log(n)) : logarithmic
3. O(n) : linear
4. O(n * log(n)) : loglinear
5. $O(n^m)$: polynomial
6. $O(m^n)$: exponential
7. O(n!) : factorial

HIDE GOT IT! What is the time complexity of the following code?

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

Constant : O(1)

HIDE GOT IT! What is the time complexity of the following code?

```
    function myFunction(n) {
        for (let i = 1; i <= 100; i++) {
            console.log(i);
        }
    }
```

O(1) : Constant; Why? Because there are a fixed number of loops in the for loop. The for loop is not dependent upon n.

HIDE GOT IT! What is the time complexity of the following code?

```
    function myFunction(n) {
        if (n <= 1) return;
        myFunction( n / 2);
    }
```

logarithmic: O(log(n)) : because the recursion will half the size of n each time

HIDE GOT IT! What is the time complexity of the following code?

```
    function myFunction(n) {
        let i = n;
        while (i > 1) {
            i /= 2;
        }
    }
```

logarithmic: O(log(n)) : Because the while loop will half the size of n each time.

HIDE GOT IT! What is the time complexity of the following code?

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

Linear : O(n) : Because the for loop will iterate n times

HIDE GOT IT! What is the time complexity of the following code?

```
    function myFunction(n) {
        if (n===1) return;
        myFunction(n - 1)
    }
```

Linear : O(n) : Because the recursive call will be made n times.

HIDE GOT IT! What is the time complexity of the following code?

```
    function myFunction(n) {
        if (n <= 1) return;

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

        myFunction(n/2);
        myFunction(n/2);
    }
```

O(n * log(n)) : LogLinear because the for loop will run n times; then the myFunction will be called recursively 2(log(n)) so we can drop the constant and consider log(n). Combining the for loop time complexity of O(n) and the recursive calls of log(n) we have O(n*log(n));

HIDE GOT IT! What is the time complexity of the following code?

```
    function myFunction(n) {
        for (let i = 1; i <= n; i++) {
            for (let j = 1; j <= n; j++) {}
        }
    }
```

Polynomial : O(n^2) : The outer loop will run n times; but for every time the outler loop runs, the inner loop will run n times; n * n = n^2 so we have O(n^2)

HIDE GOT IT! What is the time complexity of the following code?

```
    function myFunction(n) {
        for (let i = 1; i <= n; i++) {
            for (let j = 1; j <= n; j++) {
                for (let k = 1; k <= n; k++) {}
            }
        }
    }
```

Polynomial O(n^3) : Because the outer for (i) will run n times, and for each time the outer loop runs the inner (j) will run n times - which is n * n - and there's another inner loop (k) that will run n times - so we have n*n*n which is n^3 so we have O(n^3)

HIDE GOT IT! What is the time complexity of the following code?

```
    function myFunction(n) {
        if (n === 1) return;
        myFunction(n - 1);
        myFunction(n - 1);
    }
```

Exponential : O(2^n) : Because each call will make two more recursive calls so they will run a total of 2^n times, or O(2^n)

HIDE GOT IT! What is the time complexity of the following code?

```
    function myFunction(n) {
        if (n === 0) return;
        myFunction(n - 1);
        myFunction(n - 1);
        myFunction(n - 1);
    }
```

Exponential : O(3^n) : Because each call will make 3 more recursive calls, n times.

HIDE GOT IT! What is the time complexity of the following code?

```
    function myFunction(n) {
        if (n === 1) return;

        for (let i = 1; i <= n; i++) {
            myFunction(n - 1);
        }
    }
```

Factorial : O(n!) : The code is recursive, but the number of recursive calls made in a a single stack frame depends on the input. So in this case, the for loop executes n times; but within the for loop the recursive call is made n-1 times. So you have n * n-1 * n-2 * n-3 ... or O(n!);

HIDE GOT IT! Identify the type of sort, the time complexity, and the space complexity of the following code:

```
    function swap(array, idx1, idx2) {
        [array[idx1], array[idx2]] = [array[idx2], array[idx1]]
    }

    function whichSort(array) {
        let swapped = true;

        while (swapped) {
            swapped = false;

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

Bubble Sort. Time complexity O(n^2); Space Complexity of O(1)

HIDE GOT IT! Identify the type of sort as well as the time and space complexity of the following code:

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

    function whichSort(list) {
        for (let i = 0; i < list.length; i++) {
            let min = i;

            for (let j = i+1; j < list.length; j++) {
                if (list[j] < list[min]) {
                    min = j;
                }
            }
            if (min !== i) {
                swap(list, i, min);
            }
        }
    }
```

Selection Sort : Time complexity O(n^2) and O(1) space complexity

HIDE GOT IT! Identify the type of sort, as well as the time and space complexity of the following code:

```
    function mySort(list) {
        for (let i = 1; i < list.length; i++) {
            value = list[i];
            hole = i;

            while (hole > 0 && list[hole - 1] > value) {
                list[hole] = list[hole - 1];
                hole--;
            }
            list[hole] = value;
        }
    }
```

Insertion Sort. Time Complexity: O(n^2), Space complexity: O(1)

HIDE GOT IT! Identify the type of sort, as well as its time and space complexity for the code below:

```
    function doSomething(array1, array2) {
        let result = []
        while (array1.length && array2.length) {
            if (array1[0] < array2[0]) {
                result.push(array1.shift());
            } else {
                result.push(array2.shift());
            }
        }
        return [...result, ...array1, ...array2];
    }

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

        const mid = Math.floor(array.length / 2);
        const left = sortSomething(array.slice(0, mid));
        const right = sortSomething(array.slice(mid));

        return doSomething(left, right);
    }
```

Merge Sort : Time Complexity O(n log(n)) since we split the array in half each time, the number of calls is O(log(n). The while loop inside the doSomething (aka merge) method will go through all of the array - n times. So it's O(n * log(n));

Space complexity is O(n), because we are copying the array n times. [Yes, there are two copies, but each is half of the array.]

HIDE GOT IT! Identify the type of sort, the time and space complexity of the sort in the following code:

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

        let pivot = array.shift();

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

        let sortedLeft = mySort(left);
        let sortedRight = mySort(right);

        return [...sortedLeft, pivot, ...sortedRight];
    }
```

QuickSort: Time complexity of O(n^2) because sort on the left is O(n) and sort on the right is O(n).

Space complexity of Quick Sort is O(n);

HIDE GOT IT! Identify the type of sort below, and it's time and space complexities:

```
    function anotherSearch(list, target) {
        if (list.length === 0) return false;

        let mid = Math.floor(list.length / 2);

        if (list[mid] === target) {
            return true;
        } else if (list[mid] > target) {
            return anotherSearch(list.slice(0, mid), target);
        } else {
            return anotherSearch(list.slice(mid + 1), target);
        }
    }
```

Binary Search - Time complexity O(log(n));

Space complexity O(n)

HIDE GOT IT! Transform the following fibonacci function to use memoization to reduce the time complexity from polynomial time:

```
    const fibonacci = (n) => {
        if ((n === 1) || (n === 2)) {
            return 1;
        }
        return fibonacci(n-1) + fibonacci(n - 2);
    }

    const fibonacci = (n, memo = {0: 0, 1:1} ) => {
    if (n in memo) return memo[n];
    memo[n] = fibonacci(n-1, memo) + fibonacci(n - 2, memo);
    return memo[n];
};
```

HIDE GOT IT! Apply tabulation to the iterative version of fibonacci to make it less than polynomial time:

```
    function fibonacci(n) {
        let previouspreviousNumber = 0;
        let previousNumber = 0;
        let currentNumber = 1;

        for (let i = 1; i < n; i++) {
            previouspreviousNumber = previousNumber;
            previousNumber = currentNumber;
            currentNumber = previouspreviousNumber + previousNumber;
        }
        return currentNumber;
    }


    function fibonacci(n) {
        let table = new Array(n);
        table[0] = 0;
        table[1] = 1;

        for (let i = 2; i <= n; i += 1) {
            table[i] = table[i-1] + table[i-2];
        }
        return table[n];
    }
```

HIDE GOT IT! Explain the complexity of and write a function that performs insertion sort on an array of numbers

Time complexity: O(n^2); The outer loop i contributes O(n) in isolation. The inner while loop 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); We use the same amount of memory and create the same amount of variables regardless of the size of our input.

```
    function insertionSort(list) {
        for (let i = 1; i < list.length; i++) {
            value = list[i];
            hole = i;

            while (hole > 0 && list[hole - 1] > value) {
                list[hole] = list[hole - 1];
                hole--;
            }
            list[hole] = value;
        }
    }
```

HIDE GOT IT! Explain the complexity of and write a function that performs merge sort on an array of numbers.

Time Complexity: O(n * log(n)); Since we split the array in half each time, the number of recursive calls is O(log(n)). The while loop within the merge function contributes O(n) in isolation and we call that for every recursive mergeSort call.
Space Complexity: O(n) : We will create a new subarray for each element in the original input.

```
    function merge(array1, array2) {
        let result = [];
        while (array1.length && array2.length) {
            if (array1[0] < array2[0]) {
                result.push(array1.shift());
            } else {
                result.push(array2.shift());
            }
        }
        return [...result, ...array1, ...array2];
    }

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

        const mid = Math.floor(array.length / 2);
        const left = mergeSort(array.slice(0, mid));
        const right = mergeSort(array.slice(mid));

        return merge(left, right);
    }
```

HIDE GOT IT! Explain the complexity of and write a function that performs quick sort on an array of numbers.

Time Complexity: Avg Case: O(n log(n)) : The partition step alone is O(n). We are lucky and always choose the median as the pivot. This will halve the array length at every step of the recursion for O(log(n)).

Worst Case: O(n^2): 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, yielding O(n).

Space Complexity: O(n) : Our implementation of quickSort uses O(n) space because of the partition arrays we create.

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

        let pivot = array.shift();

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

        let sortedLeft = quickSort(left);
        let sortedRight = quickSort(right);

        return [...sortedLeft, pivot, ...sortedRight];
    }
```

HIDE GOT IT! Explain the complexity of and write a function that performs a binary search on a sorted array of numbers.

Time Complexity: O(log(n)) : The number of recursive calls is the number of times we must halve the array until it's length becomes 0.

Space Complexity: O(n) : Our implementation uses n space due to half arrays we create using slice.

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

        let mid = Math.floor(list.length / 2;

        if (list[mid] === target) {
            return true;
        } else if (list[mid] > target) {
            return binarySearch(list.slice(0, mid, target);
        } else {
            return binarySearch(list.slice(mid+1), target);
        }
    }
```

HIDE GOT IT! For a linked list comprised of a Node class and a List class, what properties will the list and node require?

List: head, tail; length;
Node: value; next;

HIDE GOT IT! For a linked list class implementation, what methods will the linked list class require?

1. constructor(value)
2. addHead(value)

3. addTail(value)
4. insertAt(idx)
5. removeTail()
6. removeHead()
7. remove(idx)
8. contains(value)
9. set(value, idx)
10. size

HIDE GOT IT! What type of data structure is a stack?

1. FIFO: First In First Out
2. LIFO: Last In Last Out

LIFO : Last in (goes on top of the stack), Last out (removed from top of the stack)

HIDE GOT IT! When implementing a stack, define the properties needed on both the Node and Stack classes.

Node: value; next;
Stack: top; length;

HIDE GOT IT! When implementing a stack, define the methods you will need for the stack.

1. push(val)
2. pop()
3. size()

HIDE GOT IT! What type of data structure is a queue?

1. LIFO : Last In First Out
2. FIFO : First in First Out

FIFO : First in First Out;

HIDE GOT IT! What properties would you need on the Node and Queue classes?

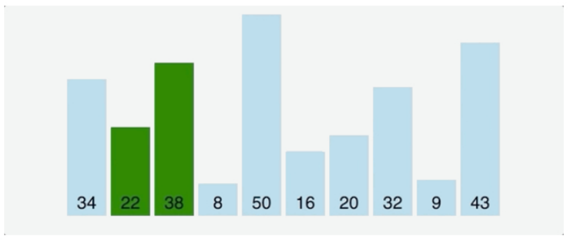
Node: value; next;
Queue: front; back; length;

HIDE GOT IT! What methods would you need on the queue class?

1. enqueue(value) : to add the new node to the back of the queue
2. dequeue: removes a value from the front of the queue
3. size: Returns the size of the queue

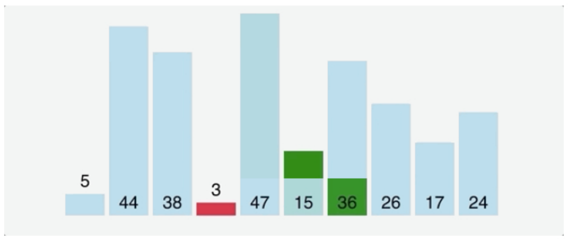
HIDE GOT IT! Explain in words the algorithm for bubbleSort

bubbleSort will look at every element of the array, comparing it to the next element. If the next element is smaller, we will swap that element. We continue that to the end of the array. Each time we swap we set swapped to true. We continue iterating through the array until we've made it all the way through the array without swapping any elements.



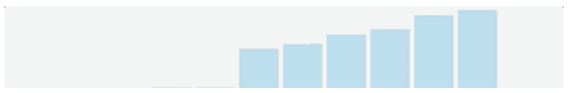
HIDE GOT IT! Explain in words how the selection sort works.

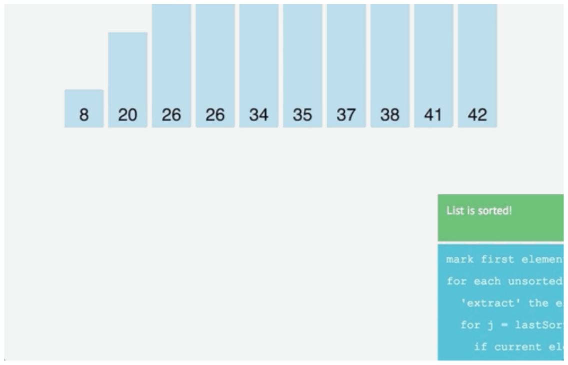
Selection Sort is very similar to Bubble Sort. The major difference between the two is that Bubble Sort bubbles the largest elements up to the end of the array, while Selection Sort selects the smallest elements of the array and directly places them at the beginning of the array in sorted position. Selection sort will utilize swapping just as bubble sort did.



HIDE GOT IT! Explain in words what insertion sort does.

Insertion Sort is similar to Selection Sort in that it gradually builds up a larger and larger sorted region at the left-most end of the array. However, Insertion Sort differs from Selection Sort because this algorithm does not focus on searching for the right element to place (the next smallest in our Selection Sort) on each pass through the array. Instead, it focuses on sorting each element in the order they appear from left to right, regardless of their value, and inserting them in the most appropriate position in the sorted region.

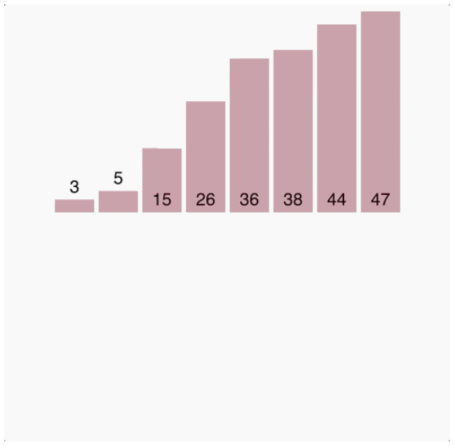




```
mark first elemen
for each unsorted
  'extract' the e
  for j = lastSor
    if current el
```

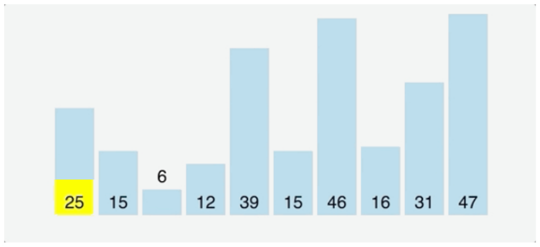
HIDE GOT IT! Describe merge sort in words

Merge sort is based on the premise that it's easy to merge elements of two sorted arrays into a single sorted array. This screams recursion, with your base case being an empty array. Also, if there's only one element in the array you can consider that array as sorted.



HIDE GOT IT! Describe the merge sort high-level steps

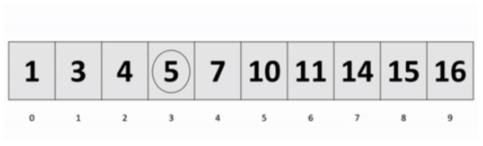
1. choose an element called "the pivot", how that's done is up to the implementation
2. take two variables to point left and right of the list excluding pivot
3. left points to the low index
4. right points to the high
5. while value at left is less than pivot move right
6. while value at right is greater than pivot move left
7. if both step 5 and step 6 does not match swap left and right
8. if left >= right, the point where they met is new pivot
9. repeat, recursively calling this for smaller and smaller arrays



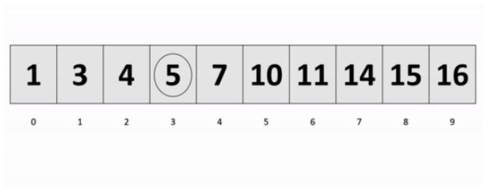
HIDE GOT IT! Describe binary search in words

Binary search only works on sorted trees or lists. Cut the list in half, look at the end element of the left half and if it's greater than what you're looking for, choose your next search for the right element. Otherwise, choose your lefthalf for your next search. Rinse and repeat.

Take for example, using the binary search algorithm on an array of integers:



Looking for the element 5, will cut your search space in 1/2 with each attempt:


1 3 4 5 7 10 11 14 15 16
0 1 2 3 4 5 6 7 8 9