Learning Objectives - Big O analysis

Big O notation is ONLY concerned with performance relative to its input size.

Big O notation describes an algorithm's worst case. Big O describes how the runtime of an algorithm scales with the amount of data it has to work on

We can measure both time and space, but are mostly concerned with time (memory is cheap and abundant)

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

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!)

2. Identify the complexity classes of common sort methods

Algorithm	Runtime Complexity	Memory Efficiency
Bubble Sort	O(n^2)	O(1)
Selection Sort	O(n^2)	O(1)
Insertion Sort	O(n^2)	O(n)
Merge Sort	O(n * log(n))	O(n)
Quick Sort	O(n^2)	O(n)

3. Identify complexity classes of code

`O(1)` - Do a known number of things, these don't grow with input size.

```
const firstThing = li => { // n would be input size
  return li[0];
};
```

```
const threeHundredThousandTimesLog = name => {
  for (let i = 0; i < 300000; i++) {
    console.log(name);
  }
};
```

`O(log n)` - Typically "divide and conquer" type algorithms

```
const splitInHalf = n => {
  if (n <= 1) return n;

  return splitInHalf(n / 2);
}
```

`O(n)` - Where we do a fixed number of things per item in the input

```
const printAll = li => {
  li.forEach(ele => {
    console.log(ele);
  })
};
```

```
const find = (li, value) => {
  for (let i = 0; i < li.length; i++) {
    if (li[i] === value) return true;
  }

  return false
}
```

```
const printALot = (li) => {
  for (let i = 0; i < li.length; i++) {
    for (let j = 0; j < 300000; j++) {
      console.log(li[i]);
    }
  }
}
```

`O(n log n)`

```
const splitButIterate = (li) => { // [1,2,3,4,5,6,7,8]
  if (li.length < 2) return li;
  const midIdx = li.length / 2;

  splitButIterate(li.slice(0, midIdx)); // 1,2,3,4
  splitButIterate(li.slice(midIdx)); // 5,6,7,8

  li.forEach(ele => console.log(ele))
};
```

`O(n^2)`

```
const dreadedDubs = (li) => {
  for (let i = 0; i < li.length; i++) {
    for (let j = 0; j < li.length; j++) {
      print(j);
    }
  }
}
```

`O(2^n)`

```
const twoN = (n) => {
  if (n == 1) return n;

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

`O(n!)`

```
const factorial = (n) => {
  if (n === 1) return n;

  for (let i = 0; i < n; i++) {
    factorial(n - 1);
  }
};
```

Memoization and Tabulation Learning Objectives

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

```
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];
};
```

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

```
function tabulatedFib(n) {
  // create a blank array with n reserved spots
  let table = new Array(n);

  // seed the first two values
  table[0] = 0;
  table[1] = 1;

  // complete the table by moving from left to right,
  // following the fibonacci pattern
  for (let i = 2; i <= n; i += 1) {
    table[i] = table[i - 1] + table[i - 2];
  }

  return table[n];
}

console.log(tabulatedFib(7));      // => 13
```

Sorting Algorithms Learning Objectives

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

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

function bubbleSort(array) {
  let swapped = false

  while (!swapped) {
    swapped = true;

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

Time Complexity: O(n^2)
The inner for loop contributes O(n) in isolation. In the worst case scenario, the while loop will need to run n times to bring all n elements into their final resting positions.

Space Complexity: O(1)
Bubble sort uses the same amount of memory and create the same amount of variables regardless of the size of the input.

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

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

function selectionSort(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);
    }
  }
}
```

Time Complexity: O(n^2)
The outer loop i contributes O(n) in isolation. 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).

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.

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

```
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;
  }
}
```

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.

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

```
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);
}
```

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.

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

```
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];
}
```

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 O(log(n)).

- *Worst Case: O(n2)*
  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 O(n).

Space Complexity: O(n)

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

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

```
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);
  }
}
```

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.

Lists, Stacks and Queues Learning Objectives

1. Explain and implement a List

A Linked List is a list made of up individual nodes, each node containing a value
and a reference to the next node (and optionally the previous node) in the list.

The List itself only needs to keep track of the head node (and optionally the tail node if you want to add things to the end of the list in constant time)
of the list, since it can follow the references on the head (or tail) node to get to any
other node. The list can optionally also keep track of the length of the list.

Linked Lists enable the following operations:

- `addToTail` - Adds a new node to the tail of the list
- `addToHead` - Adds a new node to the head of the list
- `insertAt` - Inserts a new node at a certain position in the list
- `removeTail` - Removes the tail node from the list
- `removeHead` - Removes the head node from the list
- `remove` - Removes a node from a certain position in the list
- `contains` - Searches the list for a node with a particular value
- `get` - Gets the node at a specific position
- `set` - Sets the value of a node at a specific position
- `size` - Gets the length of the Linked List

A Node could be expressed as a class like this:

```
class Node {
    constructor(value, next) {
        this.value = value;
        this.next = next;
    }
}
```

while a basic Linked List class could look like this:

```
class LinkedList {
    constructor() {
        this.head = null;
        this.tail = null;
        this.length = 0;
    }

    addToTail(val) {
        const newNode = new Node(val);

        if (!this.head) {
            this.head = newNode;
        } else {
            this.tail.next = newNode;
        }

        this.tail = newNode;
        this.length++;
        return this;
    }

    removeTail() {
        if (!this.head) return undefined;
        let current = this.head;
        let newTail = current;
        while (current.next) {
            newTail = current;
            current = current.next;
        }
        this.tail = newTail;
        this.tail.next = null;
        this.length--;
        if (this.length === 0) {
            this.head = null;
            this.tail = null;
        }
        return current;
    }

    addToHead(val) {
        let newNode = new Node(val);
        if (!this.head) {
            this.head = newNode;
            this.tail = newNode;
        } else {
            newNode.next = this.head;
            this.head = newNode;
        }
        this.length++;
        return this;
    }

    removeHead() {
        if (!this.head) return undefined;
        const currentHead = this.head;
        this.head = currentHead.next;
        this.length--;
        if (this.length === 0) {
            this.tail = null;
        }
        return currentHead;
    }

    contains(target) {
        let node = this.head;
        while (node) {
            if (node.value === target) return true;
            node = node.next;
        }
        return false;
    }

    get(index) {
        if (index < 0 || index >= this.length) return null;
        let counter = 0;
        let current = this.head;
        while (counter !== index) {
            current = current.next;
            counter++;
        }
        return current;
    }

    set(index, val) {
        const foundNode = this.get(index);
        if (foundNode) {
            foundNode.value = val;
            return true;
        }
        return false;
    }

    insert(index, val) {
        if (index < 0 || index > this.length) return false;
        if (index === this.length) return !!this.addToTail(val);
        if (index === 0) return !!this.addToHead(val);

        const newNode = new Node(val);
        const prev = this.get(index - 1);
        const temp = prev.next;
        prev.next = newNode;
        newNode.next = temp;
        this.length++;
        return true;
    }

    remove(index) {
        if (index < 0 || index >= this.length) return undefined;
        if (index === 0) return this.removeHead();
        if (index === this.length - 1) return this.removeTail();
        const previousNode = this.get(index - 1);
        const removed = previousNode.next;
        previousNode.next = removed.next;
        this.length--;
        return removed;
    }

    size() {
        return this.length;
    }
}
```

2. Explain and implement a Stack

A Stack is a Last In First Out (LIFO) Data structure.

You can usually perform the following operations on a stack:

- `push` a value onto the top of the stack
- `pop` a value off the top of the stack
- `size` get the size of the stack

The example below uses the same Node class as our Linked List.

You could also use a plain array to implement a stack (and people often do)

```
class Stack {
  constructor() {
    this.top = null;
    this.length = 0;
  }

  push(val) {
    const newNode = new Node(val);
    if (!this.top) {
      this.top = newNode;
    } else {
      const temp = this.top;
      this.top = newNode;
      this.top.next = temp;
    }
    return ++this.length;
  }

  pop() {
    if (!this.top) {
      return null;
    }
    const temp = this.top;
    this.top = this.top.next;
    this.length--;
    return temp.value;
  }

  size() {
    return this.length;
  }
}
```

3. Explain and implement a Queue

Again this uses the same Node class as the Linked List and Stack.

You could also use a plain array to implement a queue (and people often do)

Queues are a First In First Out (FIFO) data structure.

Queues usually implement the following operations:

- `enqueue` Adds a value to the back of the queue
- `dequeue` Removes a value from the front of the queue
- `size` Gets the size of the queue

```javascript
class Queue {
  constructor() {
    this.front = null;
    this.back = null;
    this.length = 0;
  }

  enqueue(val) {
    const newNode = new Node(val);
    if(!this.front) {
      this.front = newNode;
      this.back = newNode;
    } else {
      this.back.next = newNode;
      this.back = newNode;
    }
    return ++this.length;
  }

  dequeue() {
    if (!this.front) {
      return null;
    }
    const temp = this.front;
    if (this.front === this.back) {
      this.back = null;
    }
    this.front = this.front.next;
    this.length--;
    return temp.value;
  }

  size() {
    return this.length;
  }
}
```