Big O Notation

Definition:	Way of estimating the complexity of an algorithm by measuring how it's operations scale as the input size grows
Worst Case:	Big O is only concerned with the worst case scenario
Simplification:	Constants do not matter. Smaller terms do not matter. Always know what n is
Hints:	Arithmetic is constant as is variable assignment and accessing elements in an array or object by index or key;
Common Runtimes:	Constant: O(1) Logarithmic O(log n) Linear O(n) Logarithmic O(n log n) Quadratic O(n2) Exponential O(2n)

Stacks and Queues

Queues:	First in first out. Items only added to the back and only removed from the front. Use linked lists for implementation because removing from the front of an array is O(n)
Stacks:	Last in first out. Items only added and removed from the back (or top like a stack of pancakes). Arrays are fine for stacks because push and pop are both O(1)

Trees

Trees:	A data structure that organizes and stores data in a hierarchical tree like fashion.
Tree Terminology:	Node: basic unit, children: nodes directly below a node, descendants: nodes below a node, parent: node directly above a node, ancestor: node above a node, root: node at the top of the tree, leaf node: node without any children
Tree examples:	Filesystems, HTML DOM, Taxonomy
Types:	n-ary tree's can have any number of children. Binary trees: trees where each node has 0,1 or 2 children

Graphs

Definition:	Like trees except the can contain loops (cycles) also relationships can be directed or undirected.
Termin ology:	Node (Vertex) basic unit, Edge: connects two nodes, Adjacent: two nodes that share an edge, Weight: optional parameter for edges (like price)
Node Class:	just needs this.name and this.adjacent

Graphs (cont)

Graph Class:	this.nodes = new Set( )

Bubble and Merge Sort Example

```
function bubbleSort(arr){
        for(let i = 0; i < arr.length; i++){
                for(let j = 0; j < arr.length-i; j++){
                        console.log(arr);
                        if(arr[j] > arr[j + 1]){
                                let newVal = arr[j + 1];
                                arr[j + 1] = arr[j];
                                arr[j] = newVal;
                        }
                }
        }
}
const merge = (arr1, arr2) => {
   let output = [];
   let arrOneIdx = 0;
   let arrTwoIdx = 0;
    while (arrOneIdx < arr1.length && arrTwoIdx < arr2.length) {
       if (arr1[arrOneIdx] < arr2[arrTwoIdx]) {
            output.push(arr1[arrOneIdx]);
            arrOneIdx++;
       } else {
            output.push(arr2[arrTwoIdx]);
            arrTwoIdx++;
       }
   }
    while(arrOneIdx < arr1.length){
```

Bubble and Merge Sort Example (cont)

```
>    output.push(arr1[arrOneIdx]);
    arrOneIdx ++
  }
  while(arrTwoIdx < arr2.length){
   output.push(arr2[arrTwoIdx]);
    arrTwoIdx ++
  }
  return output;
};
function mergeSort(arr) {
  if (arr.length <= 1) return arr;
  let mid = Math.floor(arr.length / 2);
  let left = mergeSort(arr.slice(0, mid));
  let right = mergeSort(arr.slice(mid));
  return merge(left, right);
}
```

JavaScript Tricky Parts

Closures:	A closure occurs when a function remembers it's scope even after that function has finished executing.

JavaScript Tricky Parts (cont)

Scope:	In JavaScript, scope refers to the context in which variables and functions are declared and can be accessed. There are two main types of scope: global scope (accessible from anywhere in your code) and local scope (limited to a specific function or block of code).
Nested Functions:	When you have a function defined inside another function, the inner function has access to the variables and parameters of the outer function. This is where closures come into play.
Closers revisited:	A closure is created when an inner function references variables from its containing (outer) function, and that inner function is returned or passed around. This allows the inner function to maintain access to the outer function's variables even after the outer function has completed its execution.

JavaScript Tricky Parts (cont)

Closure use cases:	1) Data encapsulation, ie create private variables. 2) Function Factories: use closers create and return functions 3) Callback functions 4) Event Handling
Prototype:	The prototype is an object on every JS object that contains properties and methods that aren't on the object itself, this allows for the sharing and inheritance of functionality, JS classes use the prototype under the hood.
The 'new' keyword:	The new keyword does four things 1) Creates an empty object 2) Set this to be that object 3) Returns the object 4) Creates a link to that objects prototype

Divide and Conquer

Definition:	Given a data set divide and conquer removes a large fraction of the data set at each step
Binary Search:	data must be structured, for example a sorted array

Divide and Conquer (cont)

Tips:	make sure data is structured, if you can solve it quickly with linear search try binary search. Watch out for one of errors.
Runtime:	O(log n) because it cuts the data set in 1/2 each step

Recursion

Definition:	A powerful programing technique that involves a function calling itself
Loops:	All loops can be written with recursion and visa versa
Base Case:	The base case is required, every recursive function needs one it's what tells the function when it's done. Without a base case you'll get a stack overflow.
Use Cases:	Filesystems, Fractals, Parsing, Nested Data

Binary Search Trees

Definition:	A binary tree for efficient searching and sorting.

Binary Search Trees (cont)

Rules:	1) Each node must only have at max two children. 2) The nodes are organized where all nodes in the left subtree are < than the nodes value and all the nodes in the right subtree have values greater than the nodes value.
Traversal:	Typically uses recursion for traversal with either In Order, Pre Order or Post Order Traversal methods.

In order example: traverse(node) {
if (node.left) traverse(node.left);
console.log(node.val);
if (node.right) traverse(node.right);
}

BFS and DFS

```
class treeNode {
    con str uct or(val, children
= []) {
        thi s.val = val;
        thi s.c hildren =
children;
    }
    dep thF irs tFi nd(val) {
       let toVisi tStack =
[this];
        while (toVis itS tac -
k.l ength) {
           let current =
toVisi tSt ack.pop();
            con sol e.l og( " -
Vis iti ng", curren t.val);
           if (current === val)
{
```

BFS and DFS (cont)

```
>       return current.val;
    }
    for (let child of current.children) {
      toVisitStack.push(child);
    }
  }
 }
 breathFirstFind(val) {
  let toVisitQueue = [this];
  while (toVisitQueue.length) {
    let current = toVisitQueue.shift();
    console.log('visiting', current.val);
    if (current === val) {
      return current;
    }
    for (let child of current.children) {
      toVisitQueue.push(child);
    }
  }
 }
}
```

Whiteboarding Process

Step 1:	Listen carefully
Step 2:	Repeat the question back in your own words
Step 3:	Ask clarifying statements like edge cases.
Step 4:	Write test cases

Whiteboarding Process (cont)

Step 5:	Write down requirements; like arguments, what it returns etc...
Step 6:	Write pseudo code
Step 7:	Code
Step 8:	Test your code, be the computer
Step 9:	Try to optimize, clean up code, be prepared to talk about runtime/ Big O
Tips:	Use good variable names, don't brush past tricky parts, limit use of built in methods, take your time they want you to think deeply and approach it methodically it's not a race

Problem Solving Process

1) Understand the problem	restate it in your own words, understand the inputs and outputs
2) Explore Examples	start with simple examples, move to more complex examples
3) Break it down	write out the steps, write pseudocode
4) Solve a simpler problem	if your stuck solve the parts of the problem that you can and come back to the hard part

Problem Solving Process (cont)

5) Use tools	debug, console.log etc....
6) Look back and refactor	clean up code, can you improve performance?

Binary Search Example:

```
function binarySearch(arr, val)
{
   let leftIdx = 0;
   let rightIdx = arr.length -
1;
    while (leftIdx <= rightIdx)
{
       // find the middle value
       let middleIdx = Math.f -
loo r(( leftIdx + rightIdx) /
2);
       let middleVal = arr[mi -
ddl eIdx];
       if (middleVal < val) {
           // middleVal is too
small, look at the right half
            leftIdx = middleIdx
+ 1;
       } else if (middleVal >
val) {
           // middleVal is too
large, look at the left half
            rig htIdx =
middleIdx - 1;
       } else {
           // we found our
value!
            return middleIdx;
       }
   }

   // left and right pointers
crossed, val isn't in arr
    return -1;
}
```

Recursive Examples

```
/* product: calculate the product
numbers. /
function produc t(nums) {
// base case
if(num s.l ength === 0) return 1;
// normal case
return nums[0] * produc t(n ums.sl
}
/* longest: return the length of t
word in an array of words. /
function longes t(w ords) {
   // base case
   if (words.length === 0) return
   // normal case
    const curren tLength = words[
    const remain ing Words = words
    const maxLength = longes t(r e
ords);
    return Math.m ax( cur ren tLe
gth);
}
/* everyO ther: return a string wi
letter. /
function everyO the r(str) {
   // base case
   if (str.l ength === 0) return "
   // normal case
    const curren tLetter = str[0];
    const remain ing Letters = str
    return${curr ent Let ter }${ ev
r ema ini ngL ett ers)};
}
```

Recursive Examples (cont)

> /* *isPalindrome: checks whether a string is a palindrome or not.* /
function isPalindrome(str) {
  // base case
  if (str.length === 0 || str.length === 1) return true;
  // normal case
  if (str[0].toLowerCase() === str[str.length - 1].toLowerCase()) {
    return isPalindrome(str.slice(1, str.length - 1));
  } else return false;
}

Frequency Counter and Multiple Pointers Example

```
function findFreq(string){
       let stringFreq = {};
        for(let char of string){
                str ing Fre -
q[char] = string Fre q[char] + 1
|| 1
       }
        return stringFreq
}
function constr uct Not e(m -
ess aage, letters) {
        const messFreq =
findFr eq( mes saage);
        const lettFreq =
findFr eq( let ters);
        for(let char in
messFreq){
                if( !le ttF -
req [char]) return false;
                if( mes sFr -
eq[ char] > lettFr eq[ char])
return false;
       }
        return true;
 }
function averag ePa ir( array,
target) {
```

Frequency Counter and Multiple Pointers Example (cont)

>   let left = 0;
  let right = array.length - 1;
  while (left < right) {
    const average = right + left / 2;
    if (average === target) return true;
    else if (average > target) right -= 1;
    else left += 1;
  }
  return false;
}

Frequency Counter and Multiple Pointers Example

```
function findFreq(string){
       let stringFreq = {};
        for(let char of string){
                str ing Fre -
q[char] = string Fre q[char] + 1
|| 1
       }
        return stringFreq
}
function constr uct Not e(m -
ess aage, letters) {
        const messFreq =
findFr eq( mes saage);
        const lettFreq =
findFr eq( let ters);
        for(let char in
messFreq){
                if( !le ttF -
req [char]) return false;
                if( mes sFr -
eq[ char] > lettFr eq[ char])
return false;
       }
        return true;
 }
function averag ePa ir( array,
target) {
   let left = 0;
```

Frequency Counter and Multiple Pointers Example (cont)

>   let right = array.length - 1;
  while (left < right) {
    const average = right + left / 2;
    if (average === target) return true;
    else if (average > target) right -= 1;
    else left += 1;
  }
  return false;
}