chesterheng / **master-coding-interview**

Master the Coding Interview: Data Structures + Algorithms

www.udemy.com/course/master-the-coding-interview-data-structures-algorithms/

5 stars 5 forks

Star Watch

Code Issues Pull requests Actions Projects Security Insights

master Go to file

 **chesterheng** 243. Coding Problems ... on Jun 8 104

View code

README.md

Master the Coding Interview: Data Structures + Algorithms

Table of contents

Section 1: Introduction

Interview Mind Map

- Getting the Interview
- Big O Notation

- Technical Interviews
- Non Technical Interviews
- Offer + Negotiation

Technical Interview Mind Map

- Data Structures
- Algorithms

Fast Track

- Getting The Interview
- Non Technical Interview
- Offer + Negotiation

Complete

- Everything

Tech Track

- Big O
- How To Solve Problems
- Data Structures
- Algorithms
- Extra Coding Exercises

⬆ back to top

Section 2: Getting More Interviews

- Resume
- LinkedIn
- Portfolio
- Email

⬆ back to top

Resume

Resources

- ResumeMaker.Online
- Resume Cheat Sheet
- Jobscan
- Engineering Resume Templates
- This resume does not exist

Resume

- One Page
- Relevant Skills
- Personalized
- Online Link

⬆ back to top

What If I Don't Have Enough Experience?

Creative Tim Free HTML templates Medium

- GitHub
- Website
- 1 - 2 Big Projects
- Blog

⬆ back to top

Portfolio

- Creative Tim
- HTML5/CSS3 Free Templates
- ZtM-Job-Board
- Landing page templates for startups
- Free Bootstrap Templates & Themes
- 15 Web Developer Portfolios to Inspire You

⬆ back to top

Where To Find Jobs?

Where To Find Jobs?

⬆ back to top

Section 3: Big O

Setting Up Your Environment

- Repl.it
- glot.io
- RunJS

⬆ back to top

Python, C/C++, Golang, Swift and JavaScript Solutions!

- Python
- C/C++
- Golang
- Swift

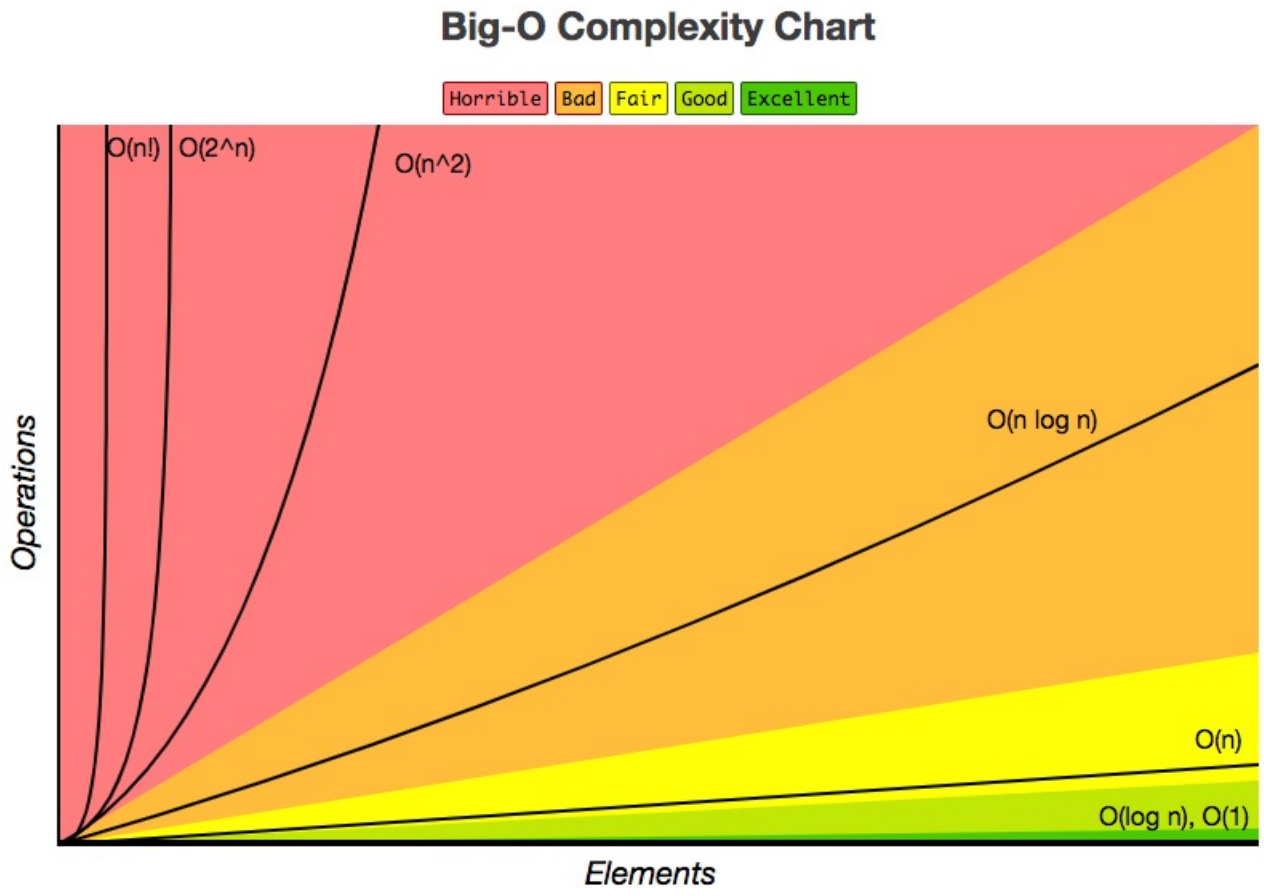
⬆ back to top

What Is Good Code?

What Is Good Code?

- Readable
- Scalable [Big O]
  - x-axis: Elements, y-axis: Operations
  - Excellent, Good: O(log n), O(1)
  - Fair: O(n)
  - Bad: O(nlog n)
  - Horrible: O(n^2), O(2^n), O(n!)

Big O



- Know Thy Complexities
- Big O Algorithm Complexity
- Big O Cheat Sheet
- What is the difference between big oh, big omega and big theta notations?

↑ back to top

O(n)

```
// O(n): Linear time
const fish = ['dory', 'bruce', 'marlin', 'nemo']
const nemo = ['nemo']
const everyone = [
  'dory',
  'bruce',
  'marlin',
  'nemo',
  'gill',
  'bloat',
  'nigel',
  'squirt',
  'darla',
```

```
  'hank',
]
const large = new Array(100000).fill('nemo')

const findNemo = (fish) => {
  let t0 = performance.now()
  for (let i = 0; i < fish.length; i++) {
    if (fish[i] === 'nemo') {
      console.log('Found NEMO!')
    }
  }
  let t1 = performance.now()
  console.log('Call to find Nemo took ' + (t1 - t0) + ' milliseconds.')
}

findNemo(large)
```

↑ back to top

O(1)

```
// O(1): Constant time
const boxes = [0, 1, 2, 3, 4, 5]

const logFirstTwoBoxes = (boxes) => {
  console.log(boxes[0]) // O(1)
  console.log(boxes[1]) // O(1)
}

logFirstTwoBoxes(boxes) // O(2)
```

↑ back to top

Exercise: Big O Calculation

```
// What is the Big O of the below function?
// Hint, you may want to go line by line
const funChallenge = (input) => {
  let a = 10 // O(1)
  a = 50 + 3 // O(1)

  for (let i = 0; i < input.length; i++) {
    anotherFunction() // O(n)
    let stranger = true // O(n)
    a++ // O(n)
  }
  return a // O(1)
```

```
}

// 1 + 1 + 1 + n + n + n
// Big O(3 + 3n)
// O(n)
funChallenge()
```

↑ back to top

Exercise: Big O Calculation 2

```
// What is the Big O of the below function?
// (Hint, you may want to go line by line)
const anotherFunChallenge = (input) => {
  let a = 5 //O(1)
  let b = 10 //O(1)
  let c = 50 //O(1)
  for (let i = 0; i < input; i++) {
    let x = i + 1 //O(n)
    let y = i + 2 //O(n)
    let z = i + 3 //O(n)
  }
  for (let j = 0; j < input; j++) {
    let p = j * 2 //O(n)
    let q = j * 2 //O(n)
  }
  let whoAmI = "I don't know" //O(1)
}

// Big O(4 + 5n)
// Big O(n)
anotherFunChallenge(5)
```

↑ back to top

Simplifying Big O

Rule Book

1. Worst Case
2. Remove Constants
3. Different terms for inputs
4. Drop Non Dominants

↑ back to top

Big O Rule 1 - Worst Case

```
// Worst Case: n
const fish = ['dory', 'bruce', 'marlin', 'nemo']
const nemo = ['nemo']
const everyone = [
  'dory',
  'bruce',
  'marlin',
  'nemo',
  'gill',
  'bloat',
  'nigel',
  'squirt',
  'darla',
  'hank',
]
const large = new Array(100000).fill('nemo')

const findNemo = (fish) => {
  let t0 = performance.now()
  for (let i = 0; i < fish.length; i++) {
    console.log('running')
    if (fish[i] === 'nemo') {
      console.log('Found NEMO!')
      break
    }
  }
  let t1 = performance.now()
  console.log('Call to find Nemo took ' + (t1 - t0) + ' milliseconds.')
}

findNemo(large)
```

back to top

Big O Rule 2 - Remove Constants

```
// Big O(1 + n/2 + 100)
// Big O(n/2 + 101)
// Big O(n/2)
// Big O(n)
const printFirstItemThenFirstHalfThenSayHi100Times = (items) => {
  // O(1)
  console.log(items[0])

  const middleIndex = Math.floor(items.length / 2)
  const index = 0
```

```
  // O(n/2)
  while (index < middleIndex) {
    console.log(items[index])
    index++
  }

  // O(100)
  for (let i = 0; i < 100; i++) {
    console.log('hi')
  }
}
```

↑ back to top

Big O Rule 3 - Different terms for inputs

```
// boxes, boxes2 are 2 different terms for inputs
// Big O(a + b)
const compressBoxesTwice = (boxes, boxes2) => {
  boxes.forEach((box) => console.log(box)) // O(a)
  boxes2.forEach((box) => console.log(box)) // O(b)
}

compressBoxesTwice([1, 2, 3], [4, 5])
```

↑ back to top

O(n^2)

```
// Big O(a * b) - Quadratic Time
const boxes = ['a', 'b', 'c', 'd', 'e']
const logAllPairsOfArray = (array) => {
  for (let i = 0; i < array.length; i++) {
    // O(a)
    for (let j = 0; j < array.length; j++) {
      // O(b)
      console.log(array[i], array[j])
    }
  }
}

logAllPairsOfArray(boxes)
```

↑ back to top

Big O Rule 4 - Drop Non Dominants

```
// Big O(n + n^2)
// Drop Non Dominants -> Big O(n^2)
const printAllNumbersThenAllPairSums = (numbers) => {
  // O(n)
  console.log('these are the numbers:')
  numbers.forEach((number) => console.log(number))

  // O(n^2)
  console.log('and these are their sums:')
  numbers.forEach((firstNumber) =>
    numbers.forEach((secondNumber) => console.log(firstNumber + secondNumber))
  )
}

printAllNumbersThenAllPairSums([1, 2, 3, 4, 5])
```

↑ back to top

What Does This All Mean?

Data Structures + Algorithms = Programs

↑ back to top

O(n!)

Example of O(n!)?

↑ back to top

3 Pillars Of Programming

What is good code?

1. Readable
2. Scalable - Speed (Time Complexity)
3. Scalable - memory (Space Complexity)

↑ back to top

Space Complexity

When a program executes it has two ways to remember things

- Heap - Store variables
- Stack - Keep track of function calls

What causes Space Complexity?

- Variables
- Data Structures
- Function Call
- Allocations

⬆ back to top

Exercise: Space Complexity

```
// Space complexity O(1)
const boooo = n => {
  // space allocation for i is O(1)
  for (let i = 0; i < n.length; i++) {
    console.log('booooo');
  }
}
boooo([1, 2, 3, 4, 5])

// Space complexity O(n)
const arrayOfHiNTimes = n => {
  // space allocation for Data Structures hiArray is O(n)
  const hiArray = [];
  for (let i = 0; i < n; i++) {
    hiArray[i] = 'hi';
  }
  return hiArray;
}
arrayOfHiNTimes(6)
```

⬆ back to top

Section 4: How To Solve Coding Problems

What Are Companies Looking For?

- Analytic Skills
  - How can you think through a problem and analyze things?
- Coding Skills

    - Do you code well, by writing clean, simple, organized, readable code?
- Technical Skills
    - Do you know the fundamentals of the job you're applying for?
    - Do you understand the pros and cons of different solutions?
    - When you should use a certain data structure over the other?
    - Why should we use a certain algorithm over another?
- Communication Skills
    - Does your personality match the companies' culture?
    - Can you communicate well with others?

⬆ back to top

What We Need For Coding Interviews

Interview Cheat Sheet

[Data Structures](Top 8 Data Structures for Coding Interviews and practice interview questions)

Data Structures	Data Structures
Arrays	Queues
Hash tables	Trees
Linked Lists	Tries
Stacks	Graphs

Algorithms
Recursion
Sorting
BFS + DFS (Searching)
Dynamic Programming

⬆ back to top

Exercise: Interview Question

Given 2 arrays, create a function that let's a user know (true/false) whether these two arrays contain any common items.

1. When the interviewer says the question, write down the key points at the top. Make sure you have all the details. Show how organized you are.

```
const array1 = ['a', 'b', 'c', 'x'];
const array2 = ['z', 'y', 'i'];
should return false.

const array1 = ['a', 'b', 'c', 'x'];
const array2 = ['z', 'y', 'x'];
should return true.
```

2. Make sure you double check: What are the inputs? What are the outputs?

```
What are the inputs?
2 parameters - arrays

What are the outputs?
return true or false
```

3. What is the most important value of the problem? Do you have time, and space and memory, etc.. What is the main goal?

```
2 parameters - arrays - no size limit
```

4. Don't be annoying and ask too many questions.
5. Start with the naive/brute force approach. First thing that comes into mind. It shows that you're able to think well and critically (you don't need to write this code, just speak about it).

```
const array1 = ['a', 'b', 'c', 'x'];
const array2 = ['z', 'y', 'a'];

const containsCommonItem = (arr1, arr2) => {
  for (let i=0; i < arr1.length; i++) {
    for ( let j=0; j < arr2.length; j++) {
      if(arr1[i] === arr2[j]) {
        return true;
      }
    }
  }
```

```
  return false
}

containsCommonItem(array1, array2);
```

6. Tell them why this approach is not the best (i.e. O(n^2) or higher, not readable, etc...)

```
Time Complexity - O(a*b)
Space Complexity - O(1)
```

7. Walk through your approach, comment things and see where you may be able to break things. Any repetition, bottlenecks like O(N^2), or unnecessary work? Did you use all the information the interviewer gave you? Bottleneck is the part of the code with the biggest Big O. Focus on that. Sometimes this occurs with repeated work as well.

8. Before you start coding, walk through your code and write down the steps you are going to follow.

```
const array1 = ['a', 'b', 'c', 'x'];
const array2 = ['z', 'y', 'a'];

array1 = obj {
  a: true,
  b: true,
  c: true,
  x: true
}
array2[index] === obj.properties
```

9. Modularize your code from the very beginning. Break up your code into beautiful small pieces and add just comments if you need to.

```
const array1 = ['a', 'b', 'c', 'x'];
const array2 = ['z', 'y', 'a'];
const containsCommonItem2 = (arr1, arr2) => {
  // loop through first array and create object where properties === items in the ar
  // can we assume always 2 params?
  let map = {};
  for (let i=0; i < arr1.length; i++) {
    if(!map[arr1[i]]) {
      const item = arr1[i];
      map[item] = true;
    }
  }
```

```
  // loop through second array and check if item in second array exists on created c
  for (let j=0; j < arr2.length; j++) {
    if (map[arr2[j]]) {
      return true;
    }
  }
  return false
}

containsCommonItem2(array1, array2)

// Time Complexity: O(a + b)
// Space Complexity: O(a)
```

10. Start actually writing your code now. Keep in mind that the more you prepare and understand what you need to code, the better the whiteboard will go. So never start a whiteboard interview not being sure of how things are going to work out. That is a recipe for disaster. Keep in mind: A lot of interviews ask questions that you won't be able to fully answer on time. So think: What can I show in order to show that I can do this and I am better than other coders. Break things up in Functions (if you can't remember a method, just make up a function and you will at least have it there. Write something, and start with the easy part.

```
const array1 = ['a', 'b', 'c', 'x'];
const array2 = ['z', 'y', 'a'];
const arrToMap = arr1 => {
  const map = {};
  for (let i=0; i < arr1.length; i++) {
    if(!map[arr1[i]]) {
      const item = arr1[i];
      map[item] = true;
    }
  }
  return map;
}

const arrInMap = (map, arr2) => {
  for (let j=0; j < arr2.length; j++) {
    if (map[arr2[j]]) {
      return true;
    }
  }
  return false
}

const containsCommonItem2 = (arr1, arr2) => {
```

```
  const map = arrToMap(arr1);
  return arrInMap(map,arr2);
}

containsCommonItem2(array1, array2)
```

11. Think about error checks and how you can break this code. Never make assumptions about the input. Assume people are trying to break your code and that Darth Vader is using your function. How will you safeguard it? Always check for false inputs that you don't want. Here is a trick: Comment in the code, the checks that you want to do… write the function, then tell the interviewer that you would write tests now to make your function fail (but you won't need to actually write the tests).

12. Don't use bad/confusing names like i and j. Write code that reads well.

13. Test your code: Check for no params, 0, undefined, null, massive arrays, async code, etc… Ask the interviewer if we can make assumption about the code. Can you make the answer return an error? Poke holes into your solution. Are you repeating yourself?

```
const arrToMap = (arr1 = []) => {
  const map = {};
  for (let i=0; i < arr1.length; i++) {
    if(!map[arr1[i]]) {
      const item = arr1[i];
      map[item] = true;
    }
  }
  return map;
}

const arrInMap = (map, arr2) => {
  for (let j=0; j < arr2.length; j++) {
    if (map[arr2[j]]) {
      return true;
    }
  }
  return false
}

const containsCommonItem2 = (arr1 = [], arr2 = []) => {
  const map = arrToMap(arr1);
  return arrInMap(map,arr2);
}

containsCommonItem2()
```

14. Finally talk to the interviewer where you would improve the code. Does it work? Are there different approaches? Is it readable? What would you google to improve? How can performance be improved? Possibly: Ask the interviewer what was the most interesting solution you have seen to this problem

```
const containsCommonItem3 = (arr1, arr2)
  => arr1.some(item => arr2.includes(item))

containsCommonItem3(array1, array2)
```

15. If your interviewer is happy with the solution, the interview usually ends here. It is also common that the interviewer asks you extension questions, such as how you would handle the problem if the whole input is too large to fit into memory, or if the input arrives as a stream. This is a common follow-up question at Google, where they care a lot about scale. The answer is usually a divide-and-conquer approach—perform distributed processing of the data and only read certain chunks of the input from disk into memory, write the output back to disk and combine them later.

[↑] back to top

Review Google Interview

```
// [1, 2, 3, 9] Sum = 8, No
// [1, 2, 4, 4] Sum = 8, Yes

// Naive Approach
const hasPairWithSum = (arr, sum) => {
  const len = arr.length;
  for(let i =0; i<len-1; i++){
     for(let j = i+1;j<len; j++){
        if (arr[i] + arr[j] === sum)
            return true;
     }
  }
  return false;
}

hasPairWithSum([1, 2, 3, 9], 8)
hasPairWithSum([1, 2, 4, 4], 8)

// Better Approach
const hasPairWithSum2 = (arr, sum) => {
  const mySet = new Set();
  const len = arr.length;
  for (let i = 0; i < len; i++){
```

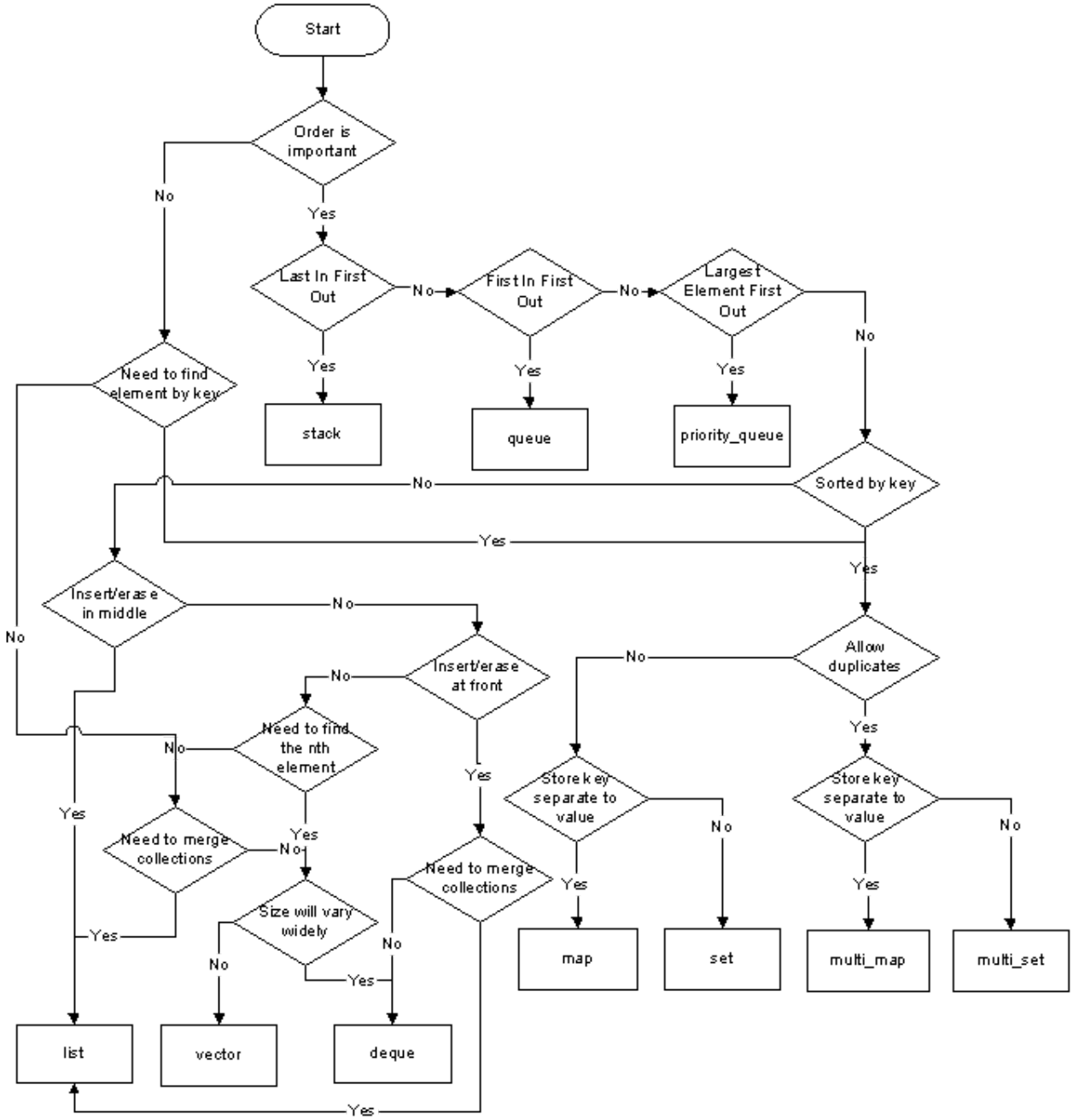
```
    if (mySet.has(arr[i])) {
      return true;
    }
    mySet.add(sum - arr[i]);
  }
  return false;
}

hasPairWithSum2([1, 2, 3, 9], 8)
hasPairWithSum2([1, 2, 4, 4], 8)
```

back to top

Section 5: Data Structures: Introduction

How to choose the right Data Structure?

Choosing the Right Data Structure to solve problems



↑ back to top

Examples of Data Structures in real life

- Data Structures
- Real-Life Examples of Data Structures
- Examples of Data Structures in real life
- Data Structures In The Real World — Linked List
- Real world data structures: tables and graphs in JavaScript
- The Real-Life Applications Of Graph Data Structures You Must Know
- How do I use algorithms and data structure in real life?

↑ back to top

What Is A Data Structure?

- A data structure is a collection of values.
- The values can have relationships among them and they can have functions applied to them.
- All programs are we're modeling real life scenarios.

Question

- How to build one?
- How to Use it?

↑ back to top

How Computers Store Data

- Computer Memory
- Registers and RAM
- CPU: access RAM and Storage for information
- RAM: fast but limited, non-persistent
- Storage: more but slow, persistent

↑ back to top

Data Structures In Different Languages

Data Structures — Language Support (Part 3)

↑ back to top

Operations On Data Structures

- Insertion
- Deletion
- Traversal
- Searching
- Sorting

- Access

[↑ back to top]

Section 6: Data Structures: Arrays

Arrays Introduction

Array vs Object

- Arrays for storing ordered collections.
- Objects for storing keyed collections.

Real life examples

- Cinema Book Challenge

Array

Operation	Big O
lookup	O(1)
push	O(1)
insert	O(n)
delete	O(n)

```
// 4 * 4 = 16 bytes of storage
const strings = ['a', 'b', 'c', 'd'];
strings[2]

const numbers = [1, 2, 3, 4, 5];

//push
strings.push('e');  // O(1)

//pop
strings.pop();  // O(1)
strings.pop();  // O(1)

//unshift
// 'x' will push all elements to their right
// ['x', 'a', 'b', 'c', 'd'];
//   0    1    2    3    4
strings.unshift('x')  // O(n)
```

```
//splice
strings.splice(2, 0, 'alien');  // O(n)
```

back to top

Static vs Dynamic Arrays

JavaScript Array is dynamic

Array Operation	Big O	Dynamic Array	Big O
lookup	O(1)	lookup	O(1)
push	O(1)	append*	O(1) or O(n)
insert	O(n)	insert	O(n)
delete	O(n)	delete	O(n)

back to top

Optional: Classes In Javascript

- Understanding Classes in JavaScript
- Arrow Functions in Class Properties Might Not Be As Great As We Think

```
class Hero {
  constructor(name, level) {
    this.name = name;
    this.level = level;
  }
  greet = () => {
    return `${this.name} says hello.`;
  }
}

class Mage extends Hero {
  constructor(name, level, spell) {
    super(name, level);
    this.spell = spell;
  }
  greet() {
    super.greet()
  }
}
```

```
const hero1 = new Hero('Varg', 1);
const hero2 = new Mage('Lejon', 2, 'Magic Missile');
hero2.greet()
```

back to top

Implementing An Array

```
class MyArray {
  constructor() {
    this.length = 0;
    this.data = {};
  }
  get(index) {
    return this.data[index];  // O(1)
  }

  push(item) {
    this.data[this.length] = item;  // O(1)
    this.length++;
    return this.data;
  }

  pop() {
    const lastItem = this.data[this.length - 1];  // O(1)
    delete this.data[this.length - 1];
    this.length--;
    return lastItem;
  }

  deleteAtIndex(index) {
    const item = this.data[index];
    this.shiftItems(index); // O(n)
    return item;
  }

  shiftItems(index) {
    for (let i = index; i < this.length - 1; i++) {
      this.data[i] = this.data[i + 1];  // O(n)
    }
    delete this.data[this.length - 1];
    this.length--;
  }
}

const myArray = new MyArray();
myArray.push('hi');
myArray.push('you');
myArray.push('!');
```

```
myArray.get(0);
myArray.pop();
myArray.deleteAtIndex(0);
myArray.push('are');
myArray.push('nice');
myArray.shiftItems(0);
```

↑ back to top

Exercise: Reverse A String

```
const reverse1 = (str = '') => {
  if(!str || typeof str != 'string' || str.length < 2 ) return str;

  const backwards = [];
  const totalItems = str.length - 1;
  for(let i = totalItems; i >= 0; i--){
    backwards.push(str[i]);
  }
  return backwards.join('');
}

const reverse2 = (str = '') => str.split('').reverse().join('');
const reverse3 = (str = '') => [...str].reverse().join('');

reverse1('Timbits Hi')
reverse2('Timbits Hi')
reverse3('Timbits Hi')
```

↑ back to top

Exercise: Merge Sorted Arrays

```
const ascendingSort = (a, b) => a - b;
const descendingSort = (a, b) => b - a;

const mergeSortedArrays = (arr1, arr2) => arr1.concat(arr2).sort(ascendingSort);
mergeSortedArrays([0,3,4,31], [3,4,6,30]);
```

↑ back to top

Interview Questions: Arrays

- LeetCode
- Interview Questions

```
// Given an array of integers, return indices of the two numbers such that they add
// You may assume that each input would have exactly one solution, and you may not u

// Given nums = [2, 7, 11, 15], target = 9,
// Because nums[0] + nums[1] = 2 + 7 = 9,
// return [0, 1].

const twoSum = (nums, target) => {
  let low = 0;
  let high = nums.length - 1;
  while(low < high) {
    if(nums[low] + nums[high] === target) return [low, high];
    else if(nums[low] + nums[high] > target) high--;
    else low++;
  }
};

twoSum([2, 3, 4, 5, 6], 10)
```

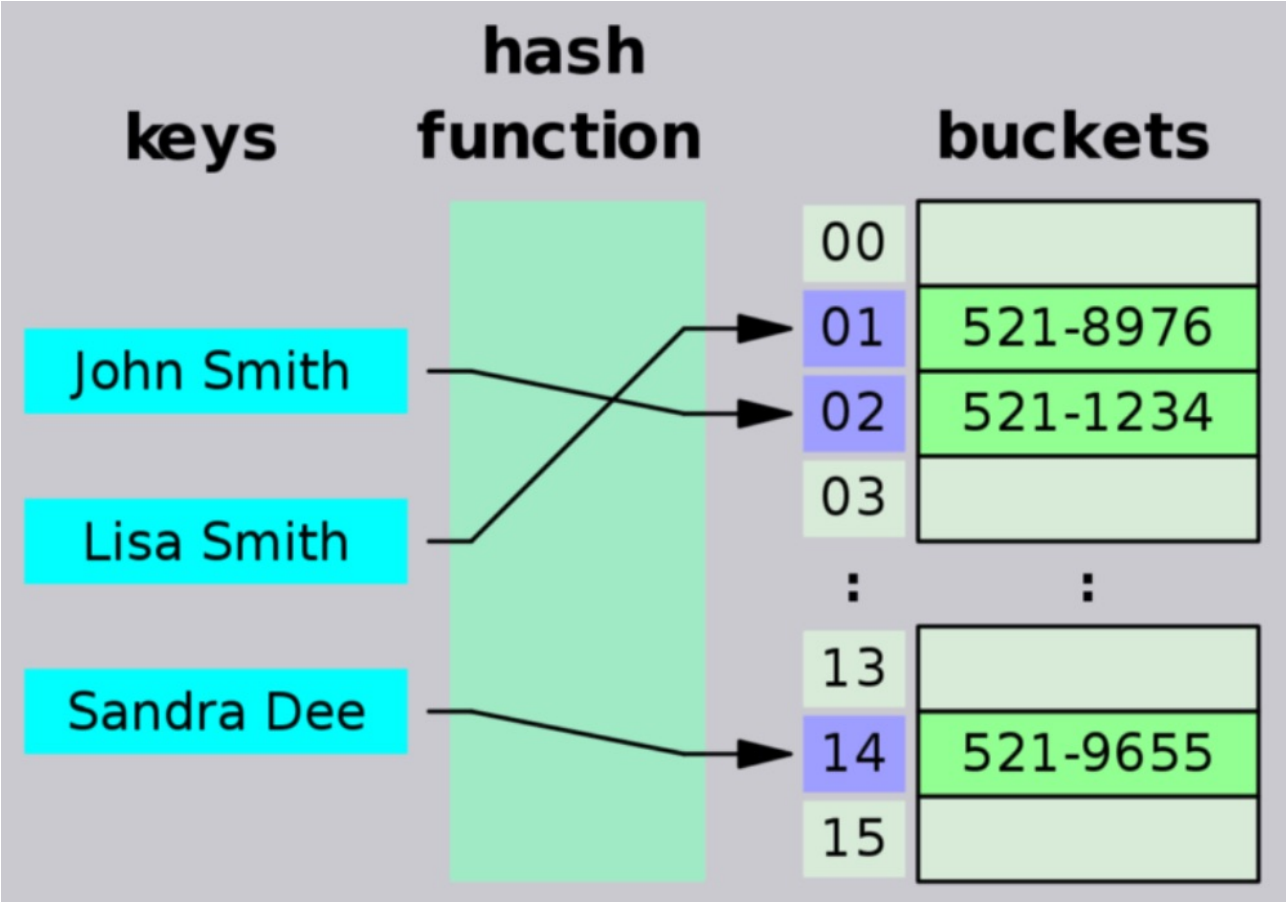
↑ back to top

Section 7: Data Structures: Hash Tables

Hash Tables Introduction

Operation	Big O
insert	O(1)
lookup	O(1)
delete	O(n)
search	O(n)

Hash Table Animation



Examples of Hash Tables

- Map is a collection of keyed data items, just like an Object. But the main difference is that Map allows keys of any type.
- A Set is a special type collection – "set of values" (without keys), where each value may occur only once.
- Map and Set
- Array vs Set vs Map vs Object
- ES6 — Map vs Object — What and when?
- ES6 — Set vs Array — What and when?
- The Importance of Hash Tables

Real life examples

- Suppose I stay in a hotel for a few days, because I attend a congress on hashing. At the end of the day, when I return to the hotel, I ask the desk clerk if there are any messages for me. Behind his back is a dovecot-like cupboard, with 26 entries, labeled A to Z. Because he knows my last name, he goes to the slot labeled W, and takes out three letters. One is for Robby Williams, one is for Jimmy Webb, and one is for me.

The clerk only had to inspect three letters. How many letters would he have to inspect if there would have been only one letter box?

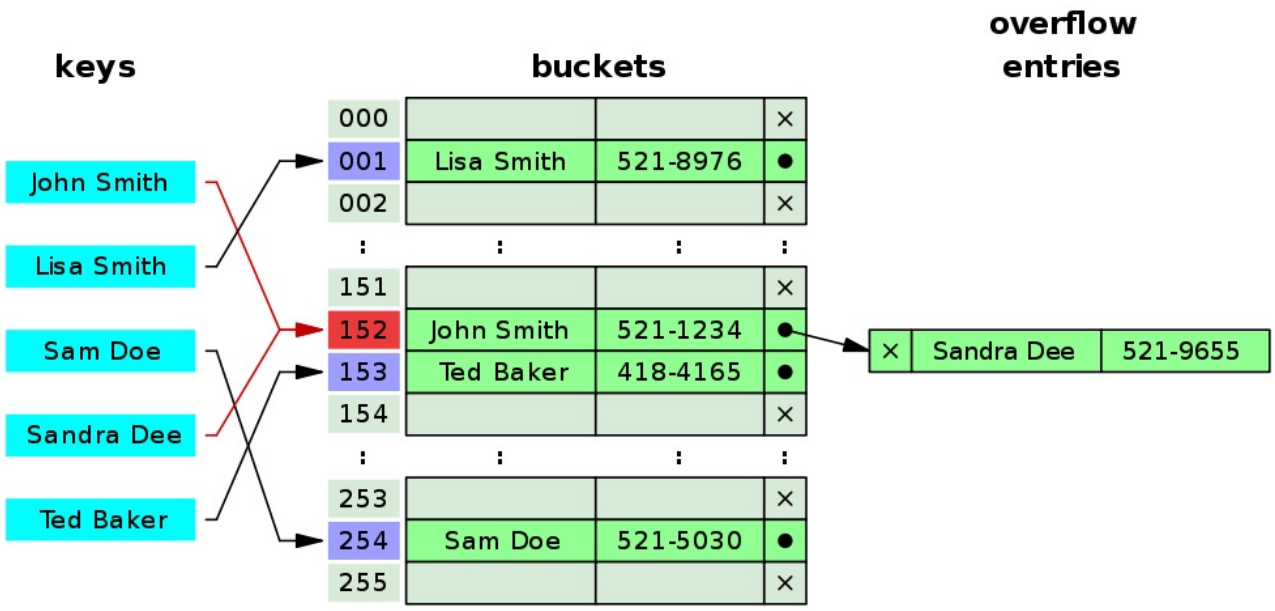
↑ back to top

Hash Function

md5 Hash Generator

↑ back to top

Hash Collisions



↑ back to top

Exercise: Implement A Hash Table

```
class HashTable {
  constructor(size){
    this.data = new Array(size);
    // this.data = [];
  }

  _hash(key) {
    let hash = 0;
    for (let i =0; i < key.length; i++){
        hash = (hash + key.charCodeAt(i) * i) % this.data.length
    }
    return hash;
  }
```

```
  set(key, value) {
    const address = this._hash(key);
    if (!this.data[address]) {
      this.data[address] = [];
    }
    this.data[address].push([key, value]);
    return this.data;
  }

  get(key) {
    const address = this._hash(key);
    const currentBucket = this.data[address]
    return currentBucket
      ? currentBucket.find(item => item[0] === key)[1]
      : undefined
  }

  keys() {
    return this.data.filter(item => !!item).map(item => item[0][0]);
  }
}

const myHashTable = new HashTable(50);
myHashTable.set('grapes', 10000)
myHashTable.get('grapes')
myHashTable.set('oranges', 54)
myHashTable.get('oranges')
myHashTable.set('apples', 9)
myHashTable.get('apples')
myHashTable.keys()
```

↑ back to top

Hash Tables VS Arrays

Arrays

Operation	Big O
search	O(n)
lookup	O(1)
push*	O(1)
insert	O(n)
delete	O(n)

Hash Tables

Operation	Big O
search	O(1)
insert	O(1)
lookup	O(n)
delete	O(n)

↑ back to top

Exercise: First Recurring Character

```
//Google Question
//Given an array = [2,5,1,2,3,5,1,2,4]:
//It should return 2

//Given an array = [2,1,1,2,3,5,1,2,4]:
//It should return 1

//Given an array = [2,3,4,5]:
//It should return undefined

//Bonus... What if we had this:
// [2,5,5,2,3,5,1,2,4]
// return 5 because the pairs are before 2,2

const firstRecurringCharacter1 = input => {
  const map = []
  for (let i = 0; i < input.length; i++) {
    const foundItem = map.find(item => item === input[i]);
    if(!!foundItem) return input[i];
    else map.push(input[i]);
  }
  return undefined;
}

firstRecurringCharacter1([2,5,1,2,3,5,1,2,4])
firstRecurringCharacter1([2,1,1,2,3,5,1,2,4])
firstRecurringCharacter1([2,3,4,5])
firstRecurringCharacter1([2,5,5,2,3,5,1,2,4])

const firstRecurringCharacter2 = input => {
  const mySet = new Set()
  for (let i = 0; i < input.length; i++) {
    const isFound = mySet.has(input[i]);
```

```
    if(isFound) return input[i];
    else mySet.add(input[i]);
  }
  return undefined;
}

firstRecurringCharacter2([2,5,1,2,3,5,1,2,4])
firstRecurringCharacter2([2,1,1,2,3,5,1,2,4])
firstRecurringCharacter2([2,3,4,5])
firstRecurringCharacter2([2,5,5,2,3,5,1,2,4])

const firstRecurringCharacter3 = input => {
  const myMap = new Map()
  for (let i = 0; i < input.length; i++) {
    const isFound = myMap.has(input[i])
    if (isFound) return input[i]
    else myMap.set(input[i], i)
  }
  return undefined;
}

firstRecurringCharacter3([2,5,1,2,3,5,1,2,4])
firstRecurringCharacter3([2,1,1,2,3,5,1,2,4])
firstRecurringCharacter3([2,3,4,5])
firstRecurringCharacter3([2,5,5,2,3,5,1,2,4])

const firstRecurringCharacter4 = input => {
  const hashtable = {}
  for (let item of input) {
    if(!hashtable[item]) hashtable[item] = true
    else return item;
  }
  return undefined;
}

firstRecurringCharacter4([2,5,1,2,3,5,1,2,4])
firstRecurringCharacter4([2,1,1,2,3,5,1,2,4])
firstRecurringCharacter4([2,3,4,5])
firstRecurringCharacter4([2,5,5,2,3,5,1,2,4])
```

⬆ back to top

Hash Tables Review

Completed JavaScript Data Structure Course, and Here is What I Learned About Hash Table.

Pros

- Fast lookups*

- Fast inserts
- Flexible Keys *Good collision resolution needed

Cons

- Unordered
- Slow key interation

↑ back to top

Section 8: Data Structures: Linked Lists

Linked Lists Introduction

Type

- Single Linked List
- Double Linked List

Real life examples

- Image viewer – Previous and next images are linked, hence can be accessed by next and previous button.
- Previous and next page in web browser – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
- Music Player – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

↑ back to top

What Is A Linked List?

- head -> apples
- tail -> pears

const basket = ['apples', 'grapes', 'pears']

linked list: apples --> grapes --> pears

apples 8947 --> grapes 8742 --> pears 372 --> null

↑ back to top

Exercise: Why Linked Lists?

VisuAlgo Linked List

Linked Lists

Operation	Big O
prepend	O(1)
append	O(1)
lookup	O(n)
insert	O(n)
delete	O(n)

↑ back to top

```
// Create the below linked list:
// myLinkedList = {
//   head: { value: 10,
//           next: { value: 5,
//                   next: { value: 16,
//                           next: null
//                         }
//                 }
//         }
// };
class Node {
  constructor(value) {
    this.value = value;
    this.next = null;
  }
}

class LinkedList {
  constructor(value) {
    this.head = new Node(value);
    this.tail = this.head;
    this.length = 1;
  }
  append(value) {
    const newNode = new Node(value);
    this.tail.next = newNode;
    this.tail = newNode;
    this.length++;
    return this;
```

```
  }
  prepend(value) {
    const newNode = new Node(value);
    newNode.next = this.head;
    this.head = newNode;
    this.length++;
    return this;
  }
  printList() {
    const array = [];
    let currentNode = this.head;
    while(currentNode !== null){
        array.push(currentNode.value)
        currentNode = currentNode.next
    }
    return array;
  }
  insert(index, value){
    if(index >= this.length) {
      return this.append(value);
    }
    const newNode = new Node(value);
    const leader = this.traverseToIndex(index-1);
    const holdingPointer = leader.next;
    leader.next = newNode;
    newNode.next = holdingPointer;
    this.length++;
    return this.printList();
  }
  traverseToIndex(index) {
    let counter = 0;
    let currentNode = this.head;
    while(counter !== index){
      currentNode = currentNode.next;
      counter++;
    }
    return currentNode;
  }
  remove(index) {
    const leader = this.traverseToIndex(index-1);
    const unwantedNode = leader.next;
    leader.next = unwantedNode.next;
    this.length--;
    return this.printList();
  }
}

const myLinkedList = new LinkedList(10);
myLinkedList
myLinkedList.append(5);
myLinkedList.append(16);
```

```
myLinkedList.prepend(1);
myLinkedList.insert(2, 99);
myLinkedList.insert(20, 88);
myLinkedList.remove(2);
```

↑ back to top

Doubly Linked Lists

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

class DoublyLinkedList {
  constructor(value) {
    this.head = new Node(value);
    this.tail = this.head;
    this.length = 1;
  }
  append(value) {
    const newNode = new Node(value);
    newNode.prev = this.tail
    this.tail.next = newNode;
    this.tail = newNode;
    this.length++;
    return this;
  }
  prepend(value) {
    const newNode = new Node(value);
    newNode.next = this.head;
    this.head.prev = newNode
    this.head = newNode;
    this.length++;
    return this;
  }
  printList() {
    const array = [];
    let currentNode = this.head;
    while(currentNode !== null){
      array.push(currentNode.value)
      currentNode = currentNode.next
    }
    return array;
  }
  insert(index, value){
```

```
    if(index >= this.length) {
      return this.append(value);
    }

    const newNode = new Node(value);
    const leader = this.traverseToIndex(index-1);
    const follower = leader.next;
    leader.next = newNode;
    newNode.prev = leader;
    newNode.next = follower;
    follower.prev = newNode;
    this.length++;
    console.log(this)
    return this.printList();
  }
  traverseToIndex(index) {
    //Check parameters
    let counter = 0;
    let currentNode = this.head;
    while(counter !== index){
      currentNode = currentNode.next;
      counter++;
    }
    return currentNode;
  }
  remove(index) {
    const leader = this.traverseToIndex(index-1);
    const unwantedNode = leader.next;
    leader.next = unwantedNode.next;
    this.length--;
    return this.printList();
  }
  reverse() {
    if (!this.head.next) {
      return this.head;
    }
    let first = this.head;
    this.tail = this.head;
    let second = first.next;

    while(second) {
      const temp = second.next;
      second.next = first;
      first = second;
      second = temp;
    }

    this.head.next = null;
    this.head = first;
    return this.printList();
  }
```

```
}

const myLinkedList = new DoublyLinkedList(10);
myLinkedList
myLinkedList.append(5)
myLinkedList.append(16)
myLinkedList.prepend(1)
myLinkedList.insert(2, 99)
myLinkedList.insert(20, 88)
myLinkedList.printList()
myLinkedList.remove(2)
myLinkedList.reverse()
```

⬆ back to top

Linked Lists Review

Pros

- Fast Insertion
- Fast Deletion
- Ordered
- Flexible Size

Cons

- Slow Lookup
- More Memory

⬆ back to top

Section 9: Data Structures: Stacks + Queues

Stacks + Queues Introduction

Stack and Queue Real World Examples

⬆ back to top

Stacks

Real life examples

- Browser history

- Store undo/redo operations in a word processor
- Maze

Stacks: LIFO

Operation	Big O
lookup	O(n)
pop	O(1)
push	O(1)
peek	O(1)

↑ back to top

Queues

Real life examples - Scheduling

- waitlist app to buy tickets for a concert
- restaurant app to see if you can get a table
- uber to grab a ride
- printer

Stacks: FIFO

Operation	Big O
lookup	O(n)
enqueue	O(1)
dequeue	O(1)
peek	O(1)

Stacks VS Queues

Stacks

- Implement with Arrays and Linked Lists

Queues

- Implement with Linked Lists

[↑ back to top]

Exercise: Stack Implementation (Linked Lists)

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

class Stack {
  constructor(){
    this.top = null;
    this.bottom = null;
    this.length = 0;
  }
  peek() {  // O(1)
    return this.top;
  }
  push(value){  // O(1)
    const newNode = new Node(value);
    if (this.length === 0) {
      this.top = newNode;
      this.bottom = newNode;
    } else {
      const holdingPointer = this.top;
      this.top = newNode;
      this.top.next = holdingPointer;
    }
    this.length++;
    return this;
  }
  pop(){  // O(1)
    if (!this.top) {
      return null;
    }
    if (this.top === this.bottom) {
      this.bottom = null;
    }
    const holdingPointer = this.top;
    this.top = this.top.next;
    this.length--;
    return this;
  }
}
```

```
const myStack = new Stack();
myStack.peek();
myStack.push('google');
myStack.push('udemy');
myStack.push('discord');
myStack.peek();
myStack.pop();
myStack.pop();
myStack.pop();
```

↑ back to top

Exercise: Stack Implementation (Array)

```
class Stack {
  constructor(){
    this.array = [];
  }
  peek() {  // O(1)
    return this.array[this.array.length-1];
  }
  push(value){
    this.array.push(value); // O(1)
    return this;
  }
  pop(){
    this.array.pop(); // O(1)
    return this;
  }
}

const myStack = new Stack();
myStack.peek();
myStack.push('google');
myStack.push('udemy');
myStack.push('discord');
myStack.peek();
myStack.pop();
myStack.pop();
myStack.pop();
```

↑ back to top

Exercise: Queue Implementation (Linked Lists)

```
class Node {
  constructor(value) {
```

```
    this.value = value;
    this.next = null;
  }
}

class Queue {
  constructor(){
    this.first = null;
    this.last = null;
    this.length = 0;
  }
  peek() {  // O(1)
    return this.first;
  }
  enqueue(value){ // O(1)
    const newNode = new Node(value);
    if (this.length === 0) {
      this.first = newNode;
      this.last = newNode;
    } else {
      this.last.next = newNode;
      this.last = newNode;
    }
    this.length++;
    return this;
  }
  dequeue(){  // O(1)
    if (!this.first) {
      return null;
    }
    if (this.first === this.last) {
      this.last = null;
    }
    const holdingPointer = this.first;
    this.first = this.first.next;
    this.length--;
    return this;
  }
}

const myQueue = new Queue();
myQueue.peek();
myQueue.enqueue('Joy');
myQueue.enqueue('Matt');
myQueue.enqueue('Pavel');
myQueue.peek();
myQueue.dequeue();
myQueue.dequeue();
myQueue.dequeue();
myQueue.peek();
```

back to top

Exercise: Queue Implementation (Array)

```
class Queue {
  constructor(){
    this.array = [];
  }
  peek() {
    return this.array[0]; // O(1)
  }
  enqueue(value){
    this.array.push(value); // O(1)
    return this;
  }
  dequeue(){
    this.array.shift(); // O(n)
    return this;
  }
}

const myQueue = new Queue();
myQueue.peek();
myQueue.enqueue('Joy');
myQueue.enqueue('Matt');
myQueue.enqueue('Pavel');
myQueue.peek();
myQueue.dequeue();
myQueue.dequeue();
myQueue.dequeue();
myQueue.peek();
```

back to top

Queues Using Stacks

```
class CrazyQueue {
  constructor() {
    this.first = [];
    this.last = [];
  }

  enqueue(value) {
    const length = this.first.length;
    for (let i = 0; i < length; i++) {
      this.last.push(this.first.pop());
    }
```

```
    this.last.push(value);
    return this;
  }

  dequeue() {
    const length = this.last.length;
    for (let i = 0; i < length; i++) {
      this.first.push(this.last.pop());
    }
    this.first.pop();
    return this;
  }
  peek() {
    if (this.last.length > 0) {
      return this.last[0];
    }
    return this.first[this.first.length - 1];
  }
}

const myQueue = new CrazyQueue();
myQueue.peek();
myQueue.enqueue('Joy');
myQueue.enqueue('Matt');
myQueue.enqueue('Pavel');
myQueue.peek();
myQueue.dequeue();
myQueue.dequeue();
myQueue.dequeue();
myQueue.peek();
```

[↑ back to top]

Stacks + Queues Review

Pros

- Fast Operations
- Fast Peek
- Ordered

Cons

- Slow Lookup

[↑ back to top]

Section 10: Data Structures: Trees

Trees Introduction

Real life examples

- DOM
- Chess Game use tree data structure to make decisions
- Facebook comments
- Family Tree
- Abstract Syntax Tree

↑ back to top

Binary Trees

Types of Binary Tree

- Perfect Binary Tree
- Full Binary Tree

Binary Search Tree

Operation	Big O
lookup	O(log n)
insert	O(log n)
delete	O(log n)

O(log n)

- Level 0: 2^0 = 1
- Level 1: 2^1 = 2
- Level 2: 2^2 = 4
- Level 3: 2^3 = 8

Nos. of nodes = 2 ^ h - 1 log nodes = steps

log 100 = 2 10^2 = 100

```
   101
   /  \
  33   105
 / \    / \
9  37  104 144
```

⬆ back to top

Balanced VS Unbalanced BST

Unbalanced BST

Operation	Big O
lookup	O(n)
insert	O(n)
delete	O(n)

⬆ back to top

BST Pros and Cons

Pros

- Better than O(n)
- Ordered
- Flexible Size

Cons

- No O(1) operations

⬆ back to top

Exercise: Binary Search Tree

```
class Node {
  constructor(value) {
    this.left = null;
    this.right = null;
    this.value = value;
  }
}
```

```
class BinarySearchTree {
  constructor() {
    this.root = null;
  }
  insert(value) {
    const newNode = new Node(value);
    if (this.root === null) {
      this.root = newNode;
      return this;
    } else {
      let currentNode = this.root;
      while (true) {
        if (value < currentNode.value) {
          //Left
          if (!currentNode.left) {
            currentNode.left = newNode;
            return this;
          }
          currentNode = currentNode.left;
        } else {
          //Right
          if (!currentNode.right) {
            currentNode.right = newNode;
            return this;
          }
          currentNode = currentNode.right;
        }
      }
    }
  }
  lookup(value) {
    if (!this.root) {
      return false;
    }
    let currentNode = this.root;
    while (currentNode) {
      if (value < currentNode.value) {
        currentNode = currentNode.left;
      } else if (value > currentNode.value) {
        currentNode = currentNode.right;
      } else if (currentNode.value === value) {
        return currentNode;
      }
    }
    return null;
  }
  remove(value) {
    if (!this.root) {
      return false;
    }
```

```
    let currentNode = this.root;
    let parentNode = null;
    while (currentNode) {
      if (value < currentNode.value) {
        parentNode = currentNode;
        currentNode = currentNode.left;
      } else if (value > currentNode.value) {
        parentNode = currentNode;
        currentNode = currentNode.right;
      } else if (currentNode.value === value) {
        //We have a match, get to work!

        //Option 1: No right child:
        if (currentNode.right === null) {
          if (parentNode === null) {
            this.root = currentNode.left;
          } else {
            //if parent > current value, make current left child a child of parent
            if (currentNode.value < parentNode.value) {
              parentNode.left = currentNode.left;

              //if parent < current value, make left child a right child of parent
            } else if (currentNode.value > parentNode.value) {
              parentNode.right = currentNode.left;
            }
          }

          //Option 2: Right child which doesnt have a left child
        } else if (currentNode.right.left === null) {
          currentNode.right.left = currentNode.left;
          if (parentNode === null) {
            this.root = currentNode.right;
          } else {
            //if parent > current, make right child of the left the parent
            if (currentNode.value < parentNode.value) {
              parentNode.left = currentNode.right;

              //if parent < current, make right child a right child of the parent
            } else if (currentNode.value > parentNode.value) {
              parentNode.right = currentNode.right;
            }
          }

          //Option 3: Right child that has a left child
        } else {
          //find the Right child's left most child
          let leftmost = currentNode.right.left;
          let leftmostParent = currentNode.right;
          while (leftmost.left !== null) {
            leftmostParent = leftmost;
            leftmost = leftmost.left;
```

```
          }

          //Parent's left subtree is now leftmost's right subtree
          leftmostParent.left = leftmost.right;
          leftmost.left = currentNode.left;
          leftmost.right = currentNode.right;

          if (parentNode === null) {
            this.root = leftmost;
          } else {
            if (currentNode.value < parentNode.value) {
              parentNode.left = leftmost;
            } else if (currentNode.value > parentNode.value) {
              parentNode.right = leftmost;
            }
          }
        }
        return true;
      }
    }
  }
}

const tree = new BinarySearchTree();
tree.insert(9);
tree.insert(4);
tree.insert(6);
tree.insert(20);
tree.insert(170);
tree.insert(15);
tree.insert(1);
tree.lookup(11);
tree.remove(170);

const traverse = (node) => {
  const tree = { value: node.value };
  tree.left = node.left === null ? null : traverse(node.left);
  tree.right = node.right === null ? null : traverse(node.right);
  return tree;
};

traverse(tree.root);

//     9
//  4     20
//1  6  15  170
```

back to top

AVL Trees vs Red Black Trees

AVL Trees

- Animation
- How it Works

Red Black Trees:

- Animation
- How it Works

⬆ back to top

Binary Heaps

Binary Heaps

Operation	Big O
lookup	O(n)
insert	O(log n)
delete	O(log n)

Real life example - Priority Queue

- A real-life example of a priority queue would be a hospital queue where the patient with the most critical situation would be the first in the queue. In this case, the priority order is the situation of each patient
- Implementation of Priority Queue in Javascript

Pros

- Better than O(n)
- Priority
- Flexible Size
- Fast Insert

Cons

- Slow Lookup

⬆ back to top

Trie

Real life example

- auto complete your text

[↑] back to top

Section 11: Data Structures: Graphs

Types Of Graphs

- Directed - Twitter
- Undirected - Facebook
- Weighted - Shortest path
- Unweighted
- Cyclic - Google map
- Acyclic
- Data Structures 101: Graphs—A Visual Introduction for Beginners

Real world examples

- The Internet map
- Facebook: Each user is represented as a vertex and two people are friends when there is an edge between two vertices. Similarly friend suggestion also uses graph theory concept.
- Google Maps: Various locations are represented as vertices and the roads are represented as edges and graph theory is used to find shortest path between two nodes.
- Recommendations on e-commerce websites: The “Recommendations for you” section on various e-commerce websites uses graph theory to recommend items of similar type to user’s choice.
- Graph theory is also used to study molecules in chemistry and physics.

[↑] back to top

Exercise: Graph Implementation

```
class Graph {
  constructor() {
    this.numberOfNodes = 0;
```

```
    this.adjacentList = {};
  }
  addVertex(node)  {
    this.adjacentList[node] = [];
    this.numberOfNodes++;
  }
  addEdge(node1, node2) {
    //uniderected Graph
    this.adjacentList[node1].push(node2);
    this.adjacentList[node2].push(node1);
  }
  showConnections() {
    const allNodes = Object.keys(this.adjacentList);
    for (let node of allNodes) {
      let nodeConnections = this.adjacentList[node];
      let connections = "";
      let vertex;
      for (vertex of nodeConnections) {
        connections += vertex + " ";
      }
      console.log(node + "-->" + connections);
    }
  }
}

const myGraph = new Graph();
myGraph.addVertex('0');
myGraph.addVertex('1');
myGraph.addVertex('2');
myGraph.addVertex('3');
myGraph.addVertex('4');
myGraph.addVertex('5');
myGraph.addVertex('6');
myGraph.addEdge('3', '1');
myGraph.addEdge('3', '4');
myGraph.addEdge('4', '2');
myGraph.addEdge('4', '5');
myGraph.addEdge('1', '2');
myGraph.addEdge('1', '0');
myGraph.addEdge('0', '2');
myGraph.addEdge('6', '5');
myGraph
myGraph.showConnections();
```

↑ back to top

Graphs Review

neo4j

Pro

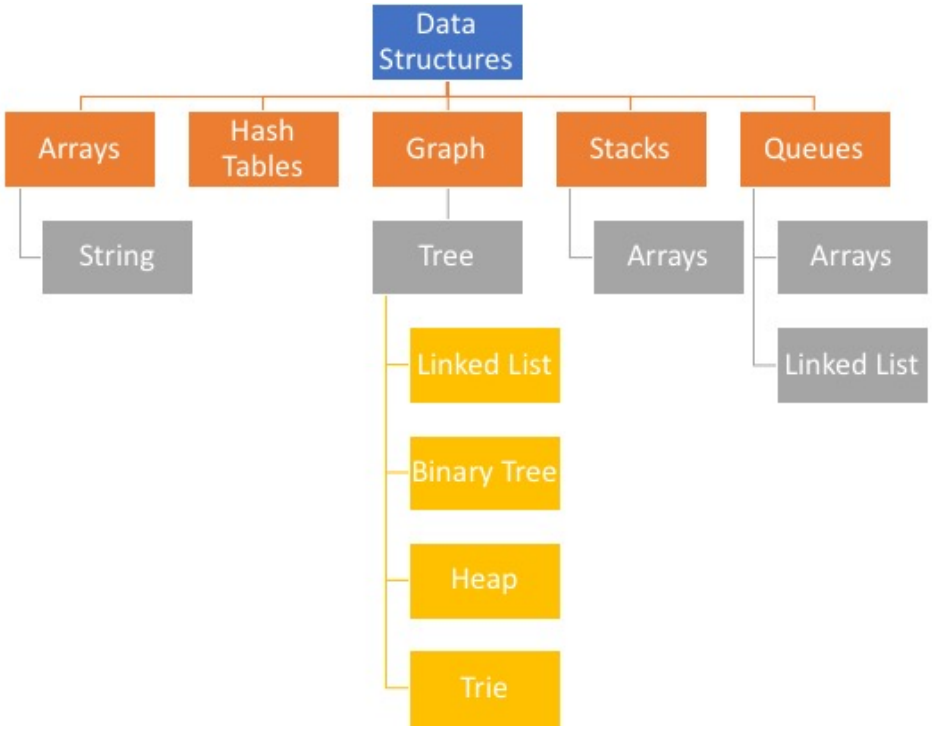
- Relationships

Con

- Scaling is hard

back to top

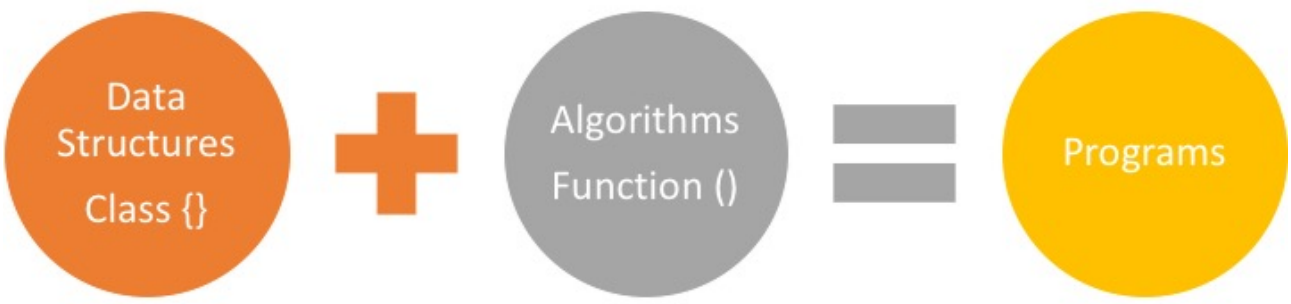
Data Structures Review



back to top

Section 12: Algorithms: Recursion

Introduction to Algorithms



Data Structures	Data Structures
Arrays	Queues
Hash tables	Trees
Linked Lists	Tries
Stacks	Graphs

Algorithms
Recursion
Sorting
BFS + DFS (Searching)
Dynamic Programming

↑ back to top

Stack Overflow

```
function inception() {
  debugger;
  inception();
}
```

↑ back to top

Anatomy Of Recursion

- Identify the base case
- Identify the recursive case
- Get closer and closer and return when needed. Usually you have 2 returns

```
let counter = 0;

const inception = () => {
  console.log(counter)
  // base case
  if(counter > 3) {
    return 'done!';
  }
  counter++;

  // recursive calls
  return inception();
}

inception();
```

↑ back to top

Exercise: Factorial

```
// Write two functions that finds the factorial of any number. One should use recurs

const findFactorialRecursive = number => {  // O(n)
  if(number <= 2) return number;
  return number * findFactorialRecursive(number - 1);
}

const findFactorialIterative = number => { // O(n)
  let answer = 1;
  for(let i = 2; i <= number; i++)
    answer = answer * i
  return answer;
}

findFactorialRecursive(5)
findFactorialIterative(5)
```

↑ back to top

Exercise: Fibonacci

```
// Given a number N return the index value of the Fibonacci sequence, where the sequ

// 0  1  2  3  4  5  6  7   8
// 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 ...
// the pattern of the sequence is that each value is the sum of the 2 previous value

//For example: fibonacciRecursive(6) should return 8

const fibonacciIterative = n => {
  const arr = [0, 1];
  for (let i = 2; i <= n; i++){
    arr.push(arr[i - 2] + arr[i -1]);
  }
 return arr[n];
}
fibonacciIterative(3);

const fibonacciRecursive = n => {
  if(n <= 1) return n;
  return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);
}
fibonacciRecursive(6)
```

↑ back to top

Recursive VS Iterative

Anything you do with a recursion can be done iteratively (loop)

Recursion

Pros

- DRY
- Readability

Cons

- Large Stack

Tail call optimization in ECMAScript 6

↑ back to top

When To Use Recursion

Real-world examples of recursion

Every time you are using a tree or converting something into a tree, consider recursion

- Divided into a number of subproblems that are smaller instances of the same problem
- Each instance of the subproblem is identical in nature
- The solutions of each subproblem can be combined to solve the problem at hand.

Divide and Conquer using Recursion

↑ back to top

Exercise: Reverse String With Recursion

```
//Implement a function that reverses a string using iteration...and then recursion!
const reverseString = str => str.split('').reverse().join('');
reverseString('yoyo mastery')

const reverseStringRecursive = str => {
  if(str.length === 1) return str;
  return reverseStringRecursive(str.substring(1)).concat(str[0]);
}
reverseStringRecursive('yoyo mastery')
```

↑ back to top

Recursion Review

- Merge Sort
- Quick Sort
- Tree Traversal
- Graph Traversal

↑ back to top

Section 13: Algorithms: Sorting

Sorting Introduction

- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort
- Quick Sort

⬆ back to top

The Issue With sort()

```
// default sort order is ascending
// converting the elements into strings
// then comparing their sequences of UTF-16 code units values
const months = ['March', 'Jan', 'Feb', 'Dec'];
months.sort();
console.log(months);
// expected output: Array ["Dec", "Feb", "Jan", "March"]

const array1 = [1, 30, 4, 21, 100000];
array1.sort();
console.log(array1);
// expected output: Array [1, 100000, 21, 30, 4]
```

⬆ back to top

Sorting Algorithms

Sorting Algorithms Animations

⬆ back to top

Exercise: Bubble Sort

```
const numbers = [99, 44, 6, 2, 1, 5, 63, 87, 283, 4, 0];

const bubbleSort = array => {
  const length = array.length;
  for (let i = 0; i < length; i++) {
    for (let j = 0; j < length; j++) {
      if(array[j] > array[j+1]) {
        [array[j], array[j+1]] = [array[j+1], array[j]];
      }
```

```
      console.log(array)
    }
  }
}

bubbleSort(numbers);
```

back to top

Exercise: Selection Sort

```
const numbers = [99, 44, 6, 2, 1, 5, 63, 87, 283, 4, 0];

const selectionSort = array => {
  const length = array.length;
  for (let i = 0; i < length; i++) {
    let min = i;
    for (let j = i + 1; j < length; j++) {
      if (array[min] > array[j]) {
          min = j;
      }
    }
    if (min !== i) {
      [array[i], array[min]] = [array[min], array[i]]
    }
  }
  return array;
}

selectionSort(numbers);
```

back to top

Dancing Algorithms

AlgoRythmics

back to top

Insertion Sort

```
const numbers = [99, 44, 6, 2, 1, 5, 63, 87, 283, 4, 0];

const insertionSort = array => {
  for (let i = 1; i < array.length; i++) {
    let j = i - 1
```

```
    let tmp = array[i]
    while (j >= 0 && array[j] > tmp) {
      array[j + 1] = array[j]
      j--
    }
    array[j+1] = tmp
  }
  return array
}

insertionSort(numbers);
```

⬆ back to top

O(n log n)

Divide & Conquer

- Merge Sort
- Quick Sort

⬆ back to top

Exercise: Merge Sort

```
const numbers = [99, 44, 6, 2, 1, 5, 63, 87, 283, 4, 0];

const mergeSort = array => {
  // base case
  if (array.length === 1) {
    return array
  }
  // Split Array in into right and left
  const length = array.length;
  const middle = Math.floor(length / 2)
  const left = array.slice(0, middle)
  const right = array.slice(middle)

  return merge(
    mergeSort(left),
    mergeSort(right)
  )
}

const merge = (left, right) => {
  const result = [];
  let leftIndex = 0;
  let rightIndex = 0;
```

```
  while(leftIndex < left.length && rightIndex < right.length){
    if(left[leftIndex] < right[rightIndex]){
      result.push(left[leftIndex]);
      leftIndex++;
    } else{
      result.push(right[rightIndex]);
      rightIndex++
    }
  }
  return result.concat(left.slice(leftIndex)).concat(right.slice(rightIndex));
}

const answer = mergeSort(numbers);
```

back to top

Stable VS Unstable Algorithms

What is stability in sorting algorithms and why is it important?

back to top

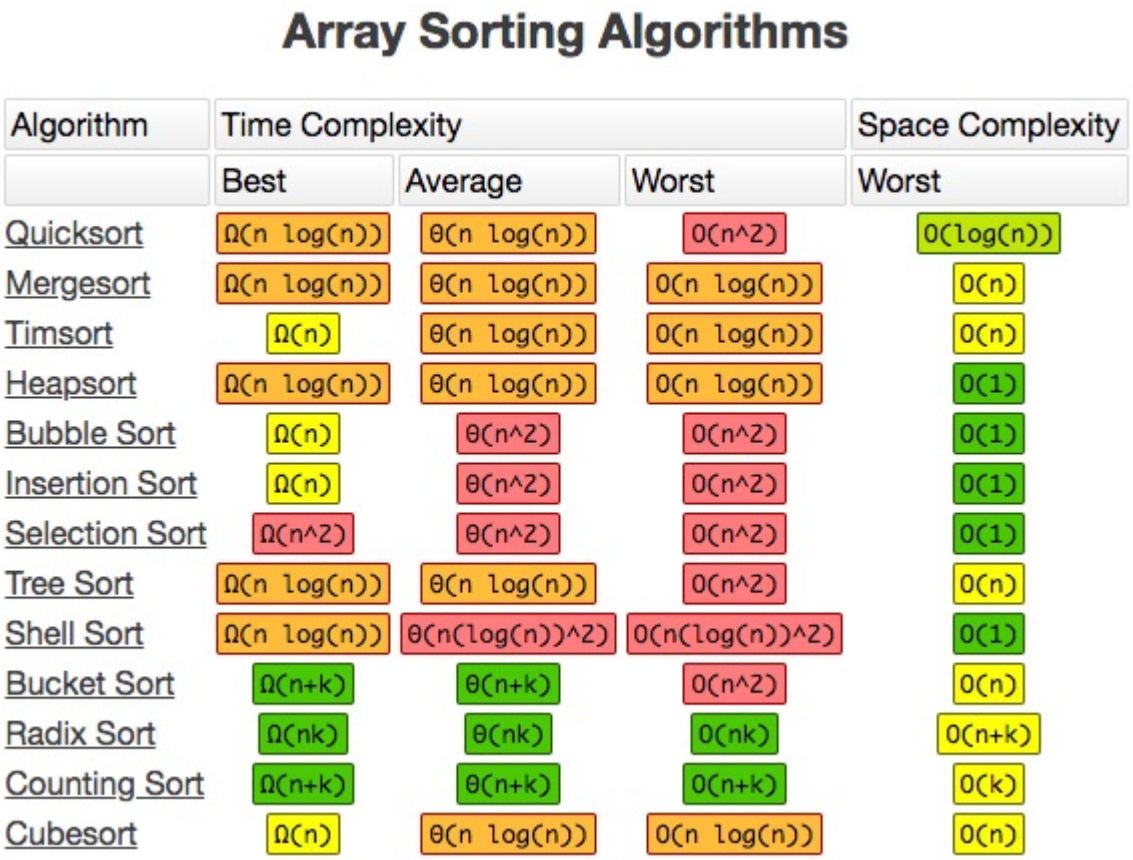
Quick Sort

```
const numbers = [99, 44, 6, 2, 1, 5, 63, 87, 283, 4, 0];

const swap = (array, a, b) => [array[a], array[b]] = [array[b], array[a]];
const partition = (array, pivot, left, right) => {
  let pivotValue = array[pivot];
  let partitionIndex = left;

  for(let i = left; i < right; i++) {
    if(array[i] < pivotValue){
      swap(array, i, partitionIndex);
      partitionIndex++;
    }
  }
  swap(array, right, partitionIndex);
  return partitionIndex;
}

const quickSort = (array, left = 0, right = array.length - 1) => {
  let pivot;
  let partitionIndex;

  if(left < right) {
    pivot = right;
    partitionIndex = partition(array, pivot, left, right);
```

```
    //sort left and right
    quickSort(array, left, partitionIndex - 1);
    quickSort(array, partitionIndex + 1, right);
  }
  return array;
}

quickSort(numbers);
```

↑ back to top

Which Sort Is Best?



Criteria for Choosing a Sorting Algorithm

Criteria	Sorting algorithm
Only a few items	Insertion Sort
Items are mostly sorted already	Insertion Sort
Concerned about worst-case scenarios	Heap Sort
Interested in a good average-case result	Quicksort

Criteria	Sorting algorithm
Items are drawn from a dense universe	Bucket Sort
Desire to write as little code as possible	Insertion Sort

↑ back to top

Heap Sort

```
const swap = (array, a, b) => [array[a], array[b]] = [array[b], array[a]];

const max_heapify = (array, i, length) => {
  while (true) {
    let left = i*2 + 1;
    let right = i*2 + 2;
    let largest = i;

    if (left < length && array[left] > array[largest]) {
        largest = left;
    }

    if (right < length && array[right] > array[largest]) {
        largest = right;
    }

    if (i == largest) {
        break;
    }

    swap(array, i, largest);
    i = largest;
  }
}

const heapify = (array, length) => {
  for (let i = Math.floor(length/2); i >= 0; i--) {
    max_heapify(array, i, length);
  }
}

const heapsort = array => {
  heapify(array, array.length);

  for (let i = array.length - 1; i > 0; i--) {
    swap(array, i, 0);
    max_heapify(array, 0, i-1);
  }
}
```

```
const numbers = [99, 44, 6, 2, 1, 5, 63, 87, 283, 4, 0];
heapsort(numbers);
```

↑ back to top

Radix Sort + Counting Sort

Can we beat O(nlogn)?

Comparison Sort	Non-Comparison Sort
Bubble Sort	Counting Sort
Insertion Sort	Radix Sort
Selection Sort	
Merge Sort	
Quick Sort	

- Radix Sort
- Radix Sort Animation
- Counting Sort
- Counting Sort Animation

↑ back to top

Sorting Interview

Question	Sorting algorithm
Sort 10 schools around your house by distance	Insertion Sort
eBay sorts listings by the current Bid amount	Radix or Counting sort
Sort scores on ESPN	Quick sort
Massive database (can't fit all into memory) needs to sort through past year's user data	Merge Sort
Almost sorted Udemy review data needs to update and add 2 new reviews	Insertion Sort

Question	Sorting algorithm
Temperature Records for the past 50 years in Canada	radix or counting Sort, Quick sort if decimal places
Large user name database needs to be sorted. Data is very random.	Quick sort
You want to teach sorting	Bubble sort

[↑ back to top]

Section 14: Algorithms: Searching + BFS + DFS

Searching + Traversal Introduction

- Linear Search
- Binary Search
- Depth First Search
- Breadth First Search

[↑ back to top]

Linear Search

```
const beasts = ['Centaur', 'Godzilla', 'Mosura', 'Minotaur', 'Hydra', 'Nessie'];

beasts.indexOf('Godzilla'); // O(n)
beasts.findIndex(item => item === 'Godzilla');  // O(n)
beasts.find(item => item === 'Godzilla'); // O(n)
beasts.includes('Godzilla'); // O(n)
```

[↑ back to top]

Binary Search

```
const binarySearch = (array, x, start = 0, end = array.length-1) => {

  // Base Condition
  if (start > end) return false;

  // Find the middle index
  let mid=Math.floor((start + end)/2);
```

```
  // Compare mid with given key x
  if (array[mid]===x) return true;

  // If element at mid is greater than x,
  // search in the left half of mid
  if(array[mid] > x)
    return binarySearch(array, x, start, mid-1);
  else
    // If element at mid is smaller than x,
    // search in the right half of mid
    return binarySearch(array, x, mid+1, end);
}

const numbers = [1, 3, 5, 7, 8, 9];
binarySearch(numbers, 5, 0, numbers.length-1)
binarySearch(numbers, 6, 0, numbers.length-1)
```

↑ back to top

BFS vs DFS

What is the time and space complexity of a breadth first and depth first tree traversal?

```
//     9
//  4     20
//1  6  15  170
// BFS - [9, 4, 20, 1, 6, 15, 170]
// DFS - [9, 4, 1, 6, 20, 15, 170]
```

BFS

Pros	Cons
Shortest Path	More Memory
Closer Nodes	

DFS

Pros	Cons
Less Memory	Does Path Exists?
Can Get Slow	

Question	BFS or DFS
If you know a solution is not far from the root of the tree	Does Path Exists?
If the tree is very deep and solutions are rare	BFS (DFS will take long time)
If the tree is very wide	DFS (BFS will need too much memory)
If solutions are frequent but located deep in the tree	DFS
determining whether a path exists between two nodes	DFS
Finding the shortest path	BFS

↑ back to top

breadthFirstSearch()

```
class Node {
  constructor(value){
    this.left = null;
    this.right = null;
    this.value = value;
  }
}

class BinarySearchTree {
  constructor(){
    this.root = null;
  }
  insert(value){
    const newNode = new Node(value);
    if (this.root === null) {
      this.root = newNode;
    } else {
      let currentNode = this.root;
      while(true){
        if(value < currentNode.value){
          //Left
          if(!currentNode.left){
            currentNode.left = newNode;
            return this;
          }
          currentNode = currentNode.left;
```

```
        } else {
          //Right
          if(!currentNode.right){
            currentNode.right = newNode;
            return this;
          }
          currentNode = currentNode.right;
        }
      }
    }
  }
  lookup(value){
    if (!this.root) {
      return false;
    }
    let currentNode = this.root;
    while(currentNode){
      if(value < currentNode.value){
        currentNode = currentNode.left;
      } else if(value > currentNode.value){
        currentNode = currentNode.right;
      } else if (currentNode.value === value) {
        return currentNode;
      }
    }
    return null
  }
  remove(value) {
    if (!this.root) {
      return false;
    }
    let currentNode = this.root;
    let parentNode = null;
    while(currentNode){
      if(value < currentNode.value){
        parentNode = currentNode;
        currentNode = currentNode.left;
      } else if(value > currentNode.value){
        parentNode = currentNode;
        currentNode = currentNode.right;
      } else if (currentNode.value === value) {
        //We have a match, get to work!

        //Option 1: No right child:
        if (currentNode.right === null) {
          if (parentNode === null) {
            this.root = currentNode.left;
          } else {

            //if parent > current value, make current left child a child of parent
            if(currentNode.value < parentNode.value) {
```

```
            parentNode.left = currentNode.left;

          //if parent < current value, make left child a right child of parent
          } else if(currentNode.value > parentNode.value) {
            parentNode.right = currentNode.left;
          }
        }

      //Option 2: Right child which doesnt have a left child
      } else if (currentNode.right.left === null) {
        if(parentNode === null) {
          this.root = currentNode.left;
        } else {
          currentNode.right.left = currentNode.left;

          //if parent > current, make right child of the left the parent
          if(currentNode.value < parentNode.value) {
            parentNode.left = currentNode.right;

          //if parent < current, make right child a right child of the parent
          } else if (currentNode.value > parentNode.value) {
            parentNode.right = currentNode.right;
          }
        }

      //Option 3: Right child that has a left child
      } else {

        //find the Right child's left most child
        let leftmost = currentNode.right.left;
        let leftmostParent = currentNode.right;
        while(leftmost.left !== null) {
          leftmostParent = leftmost;
          leftmost = leftmost.left;
        }

        //Parent's left subtree is now leftmost's right subtree
        leftmostParent.left = leftmost.right;
        leftmost.left = currentNode.left;
        leftmost.right = currentNode.right;

        if(parentNode === null) {
          this.root = leftmost;
        } else {
          if(currentNode.value < parentNode.value) {
            parentNode.left = leftmost;
          } else if(currentNode.value > parentNode.value) {
            parentNode.right = leftmost;
          }
        }
      }
```

```
        return true;
        }
      }
    }
    BreadthFirstSearch(){
      let currentNode = this.root;
      let list = [];
      let queue = [];
      queue.push(currentNode);

      while(queue.length > 0){
        currentNode = queue.shift();
        list.push(currentNode.value);
        if(currentNode.left) {
          queue.push(currentNode.left);
        }
        if(currentNode.right) {
          queue.push(currentNode.right);
        }
      }
      return list;
    }
    BreadthFirstSearchR(queue, list) {
      if (!queue.length) {
        return list;
      }
      const currentNode = queue.shift();
      list.push(currentNode.value);

      if (currentNode.left) {
        queue.push(currentNode.left);
      }
      if (currentNode.right) {
        queue.push(currentNode.right);
      }
      return this.BreadthFirstSearchR(queue, list);
    }
}

const tree = new BinarySearchTree();
tree.insert(9)
tree.insert(4)
tree.insert(6)
tree.insert(20)
tree.insert(170)
tree.insert(15)
tree.insert(1)

tree.BreadthFirstSearch()
tree.BreadthFirstSearchR([tree.root], [])
```

```
//     9
//  4     20
//1  6  15  170
// BFS - [9, 4, 20, 1, 6, 15, 170]
```

⬆ back to top

PreOrder, InOrder, PostOrder

```
In Order - left, root, right
In Order - [1, 4, 6, 9, 15, 20, 170]

Pre Order - root, left, right (use to re-create the tree)
Pre Order - [9, 4, 1, 6, 20, 15, 170]

Post Order - left, right, root
Post Order - [1, 6, 4, 15, 170, 20, 9]
```

⬆ back to top

depthFirstSearch()

```
class Node {
  constructor(value){
    this.left = null;
    this.right = null;
    this.value = value;
  }
}

class BinarySearchTree {
  constructor(){
    this.root = null;
  }
  insert(value){
    const newNode = new Node(value);
    if (this.root === null) {
      this.root = newNode;
    } else {
      let currentNode = this.root;
      while(true){
        if(value < currentNode.value){
          //Left
          if(!currentNode.left){
            currentNode.left = newNode;
            return this;
```

```
          }
          currentNode = currentNode.left;
        } else {
          //Right
          if(!currentNode.right){
            currentNode.right = newNode;
            return this;
          }
          currentNode = currentNode.right;
        }
      }
    }
  }
  lookup(value){
    if (!this.root) {
      return false;
    }
    let currentNode = this.root;
    while(currentNode){
      if(value < currentNode.value){
        currentNode = currentNode.left;
      } else if(value > currentNode.value){
        currentNode = currentNode.right;
      } else if (currentNode.value === value) {
        return currentNode;
      }
    }
    return null
  }
  remove(value) {
    if (!this.root) {
      return false;
    }
    let currentNode = this.root;
    let parentNode = null;
    while(currentNode){
      if(value < currentNode.value){
        parentNode = currentNode;
        currentNode = currentNode.left;
      } else if(value > currentNode.value){
        parentNode = currentNode;
        currentNode = currentNode.right;
      } else if (currentNode.value === value) {
        //We have a match, get to work!

        //Option 1: No right child:
        if (currentNode.right === null) {
          if (parentNode === null) {
            this.root = currentNode.left;
          } else {
```

```
            //if parent > current value, make current left child a child of parent
            if(currentNode.value < parentNode.value) {
              parentNode.left = currentNode.left;

            //if parent < current value, make left child a right child of parent
            } else if(currentNode.value > parentNode.value) {
              parentNode.right = currentNode.left;
            }
          }

        //Option 2: Right child which doesnt have a left child
        } else if (currentNode.right.left === null) {
          if(parentNode === null) {
            this.root = currentNode.left;
          } else {
            currentNode.right.left = currentNode.left;

            //if parent > current, make right child of the left the parent
            if(currentNode.value < parentNode.value) {
              parentNode.left = currentNode.right;

            //if parent < current, make right child a right child of the parent
            } else if (currentNode.value > parentNode.value) {
              parentNode.right = currentNode.right;
            }
          }

        //Option 3: Right child that has a left child
        } else {

          //find the Right child's left most child
          let leftmost = currentNode.right.left;
          let leftmostParent = currentNode.right;
          while(leftmost.left !== null) {
            leftmostParent = leftmost;
            leftmost = leftmost.left;
          }

          //Parent's left subtree is now leftmost's right subtree
          leftmostParent.left = leftmost.right;
          leftmost.left = currentNode.left;
          leftmost.right = currentNode.right;

          if(parentNode === null) {
            this.root = leftmost;
          } else {
            if(currentNode.value < parentNode.value) {
              parentNode.left = leftmost;
            } else if(currentNode.value > parentNode.value) {
              parentNode.right = leftmost;
            }
```

```
          }
        }
      return true;
      }
    }
  }
  DFTPreOrder(currentNode, list) {
    return traversePreOrder(this.root, []);
  }
  DFTPostOrder(){
    return traversePostOrder(this.root, []);
  }
  DFTInOrder(){
    return traverseInOrder(this.root, []);
  }
}

const traversePreOrder = (node, list) => {
  list.push(node.value);
  if(node.left) {
    traversePreOrder(node.left, list);
  }
  if(node.right) {
    traversePreOrder(node.right, list);
  }
  return list;
}

const traverseInOrder = (node, list) => {
  if(node.left) {
    traverseInOrder(node.left, list);
  }
  list.push(node.value);
  if(node.right) {
    traverseInOrder(node.right, list);
  }
  return list;
}

const traversePostOrder = (node, list) => {
  if(node.left) {
    traversePostOrder(node.left, list);
  }
  if(node.right) {
    traversePostOrder(node.right, list);
  }
  list.push(node.value);
  return list;
}

const tree = new BinarySearchTree();
```

```
tree.insert(9)
tree.insert(4)
tree.insert(6)
tree.insert(20)
tree.insert(170)
tree.insert(15)
tree.insert(1)

tree.DFTPreOrder();
tree.DFTInOrder();
tree.DFTPostOrder();

//     9
//  4     20
//1  6  15  170
// In Order - left, root, right
// In Order - [1, 4, 6, 9, 15, 20, 170]

// Pre Order - root, left, right (use to re-create the tree)
// Pre Order - [9, 4, 1, 6, 20, 15, 170]

// Post Order - left, right, root
// Post Order - [1, 6, 4, 15, 170, 20, 9]
```

⬆ back to top

Exercise: Validate A BST

Validate Binary Search Tree

⬆ back to top

Graph Traversals

- Breadth First Search - Shortest path
- Depth First Search - Check to see if it exists

BFS

Pros	Cons
Shortest Path	More Memory
Closer Nodes	

DFS

Pros	Cons
Less Memory	Does Path Exists?
Can Get Slow	

[↑ back to top]

Dijkstra + Bellman-Ford Algorithms

Finding The Shortest Path, With A Little Help From Dijkstra

[↑ back to top]

Section 15: Algorithms: Dynamic Programming

Dynamic Programming Introduction

- It is an optimization technique
- Do you have something you can cache? Dynamic Programming
- Dynamic Programming = Divide & Conquer + Memorization

When to use Dynamic Programming?

- Can be divided into subproblem
- Recursion Solution
- Are there repetitive subproblems?
- Memorize subproblems

[↑ back to top]

Memoization

Optimization - Caching

[↑ back to top]

Memoization

```
const addTo80 = n => n + 80;
addTo80(5)
addTo80(5)
addTo80(5)
```

```
//learn to cache
let cache = {};
const memoizeAddTo80 = n => {
  if (n in cache) {
    return cache[n];
  } else {
    console.log('long time');
    const answer = n + 80;
    cache[n] = answer;
    return answer;
  }
}
memoizeAddTo80(5)
memoizeAddTo80(5)
memoizeAddTo80(5)

// let's make that better with no global scope
// This is closure in javascript
const memoizeAddTo80Closure = n => {
  const cache = {};
  return n => {
    if (n in cache) {
      return cache[n];
    } else {
      console.log('long time');
      const answer = n + 80;
      cache[n] = answer;
      return answer;
    }
  }
}

const memoized = memoizeAddTo80Closure();
memoized(5)
memoized(5)
memoized(5)
```

↑ back to top

Fibonacci and Dynamic Programming

```
//0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233...
let calculations = 0;
const fibonacci = n => { //O(2^n)
  calculations++;
  if (n < 2) {
    return n
  }
```

```
  return fibonacci(n-1) + fibonacci(n-2);
}
fibonacci(10)
calculations

calculations = 0
const fibonacciMaster = () => { //O(n)
  let cache = {};
  return function fib(n) {
    calculations++;
    if (n in cache) {
      return cache[n];
    } else {
      if (n < 2) {
        return n;
      } else {
        cache[n] = fib(n-1) + fib(n-2);
        return cache[n];
      }
    }
  }
}
const fasterFib = fibonacciMaster();
fasterFib(10)
calculations

const fibonacciMaster2 = n => {
  let answer = [0, 1];
  for ( let i = 2; i <= n; i++) {
    answer.push(answer[i-2]+ answer[i-1]);
  }
  return answer.pop();
}
fibonacciMaster2(10)
```

↑ back to top

Interview Questions: Dynamic Programming

- House Robber
- Best Time to Buy and Sell Stock
- Climbing Stairs

↑ back to top

Section 16: Non Technical Interviews

Section Overview

- Tell me about a problem you solved?
- Why do you want to work for us?
- What to ask the Interviewer?
- Tell me about yourself?
- What is your biggest weaknes?
- Secret Weapon

↑ back to top

During The Interview

Mindset

- Treat everything as a learning experience.
- Walking in to meet a best friend and try to make the interviewers day

3 Questions

- Can you do the job?
- Can I work with you?
- Are you going to improve?

4 Heros

- Technical
- Success
- Leadership
- Challenge

↑ back to top

Tell Me About Yourself (1 min)

- Your triggers of success
- Mention things you want to get asked
- Skills should be relevant to job

My story

- Challenge

- Technical
- Success

[↑ back to top]

Why Us?

- Show you want to grow
- Demonstrate why you are the best

Why did you leave your job?

- No negativity

[↑ back to top]

Tell Me About A Problem You Have Solved

- Have this prepared
- Have metrics and numbers
- Scaling, performance, security

SAR method

- Situation
- Action
- Result

Tell me about an interesting project?

- Show how you are different
- Relate it to this job

[↑ back to top]

What Is Your Biggest Weakness

- Real answer
- Show how you improved it

[↑ back to top]

Any Questions For Us?

Reverse interview

- Focus on them, not company (YOU)
- Mention something they mentioned

↑ back to top

Secret Weapon

- Simplicity over complexity
- Premature Optimization is the root of all evil
- Focus on overall goal and not be myopic
- No complaining about client/code/etc
- No ego

↑ back to top

After The Interview

- Don't overuse "I"
- Talk about the interviewr
- Express how you are ideal candidate
- Don't brag

↑ back to top

Section Summary

- Have a good mindset
- Bring the energy and have a two way conversation with them
- And when you leave the interview in the final minutes also and on a strong note have a strong closing argument of why they should hire you
- 4 Heroes: Technical, Success, Leadership, Challenge
- If I hire you will you make my life easier.
    - Can you speak to people on your team?
    - Can you communicate with clients and customers and coworkers?
    - Are you able to focus on prioritizing tasks and not getting stuck solving a problem?
    - Do you need hand-holding or you're able to be independent and solve problems on your own?

↑ back to top

Section 17: Offer + Negotiation

Negotiation 101

- Don't end the conversation
- Give reason for everything
- Always negotiate
- Be positive
- Have stakes

↑ back to top

Handling An Offer

- Don't end the conversation
- Be positive
- Ask for time
- Let other companies know

To Do

- Find exact salary you want
  - Salary.com
  - Glassdoor
  - PayScale
- What value do you provide?
- Go higher

↑ back to top

Handling Multiple Offers

- Is there an offer that you feel you are under qualified for?
- Long term growth potential
- Will you respect people around you?
- Salary and benefits?
- Is your decision based on desperation?

back to top

Getting A Raise

- Ask for a raise
- Show. Don't tell

back to top

Negotiation Master

- StackOverflow Developer Salary
- How to Negotiate Your Salary

back to top

Section 19: Extras: Google, Amazon, Facebook Interview Questions

- Top Interview Questions
- Amazon Interview Questions
- Facebook Interview Questions
- Google Interview Questions
- Domain Specific Questions

back to top

Releases

No releases published

Packages

No packages published