**Data Structures and Algorithms Master Class**

**Section 1 - Introduction**

* **What are Data Structures?**
  + Data structures are a way of organizing data so that it can be used effectively. I.e. Take a bag full of different fruits and putting them in multiple bags with one type of fruit in them to make it easier to choose a fruit one likes.
* **What is an algorithm?**
  + An algorithm is a set of instructions to complete a task step by step. I.e Following a recipe to cook a vegetable soup. Steps would include washing the vegetables. Chopping vegetables. Preheating the saucepan. Adding oil to the pan. Sauteing the onions. Adding the rest of the vegetables. Etc
  + Algorithms either work well or not. The measure of an algorithms performance rely on Correctness and Efficiency.
* **Why are Data Structures and Algorithms important?**
  + When manipulating data in computers, we attempt to do so as quickly and effectively as possible. To do so we need to have structures that organize the data so that it is workable. And then we need steps to work the data. Those steps must work fast and effectively. Data organization requires data structures. Data manipulation requires great algorithms.
* **Why are Data Structures and Algorithms important in interviews?**
  + One’s ability to work with these concepts assesses one’s problem-solving skills. One’s ability to find optimal ways to achieve tasks as well as doing so without leaving gaps in the solution. I.e. Creating a program that works but has memory leaks.
  + By assessing a candidate’s ability to use the existing constructs in data structures and algorithms (the fundamentals of programming), their ability to work with or learn other programming languages is also assessed in this way.
* **Types of Data Structures**
  + Primitive: These data structures tend to be readily available as part of the programming language. I.e Integer, Float, Character or Boolean.
  + Non-Primitive: These are user defined data structures.These are separated into two types.
    - Linear
      * Static: Arrays
      * Dynamic: Linked list, Queue, Stack
    - Non-Linear
      * Graph
      * Tree
* **Types of Algorithms**
  + Simple recursive: An algorithm that calls itself repeatedly depending on some ending/closing condition.
  + Divide and conquer: When a problem is divided into sub problems that are solved separately and then combined to form the solution.
  + Dynamic programming: They work based on memorization. I.e. They keep track of some value(s) and find the best solution.
  + Greedy algorithms:
  + Brute force algorithm
  + Randomized algorithm

**Section 2 - Recursion**

* What is Recursion?
* Why do we need recursion?
* How Recursion works?
* Recursive vs Iterative Solutions
* When to use/avoid Recursion?
* How to write Recursion in 3 steps?
* How to find Fibonacci numbers using Recursion?

**Section 3 - Cracking Recursion Interview Questions**

* Question 1 - Sum of Digits
* Question 2 - Power
* Question 3 - Greatest Common Divisor
* Question 4 - Decimal To Binary

**Section 4 - Bonus CHALLENGING Recursion Problems (Exercises)**

* power
* factorial
* productofArray
* recursiveRange
* fib
* reverse
* isPalindrome
* someRecursive
* flatten
* captalizeFirst
* nestedEvenSum
* capitalizeWords
* stringifyNumbers
* collectStrings

**Section 5 - Big O Notation**

**What is Big O Notation?**

**Big O notation is the language and metric we use to determine the efficiency of algorithms. We need to mathematically figure out which code works efficiently.**

* Time Complexity: A way of showing how the runtime of a function increases as the input increases. If you have to process very little data vs a lot of data. How does your function perform as that metric changes. I.e. A large file sent over the internet takes longer than a small file. The time taken to send the file linearly increases as the file gets larger. Big O measures the number of the operations required to perform a task.
* Big O: Worse case (Industry tends to use Big O to determine performance)
* Big Theta: Average case
* Big Omega: Best case
* O(1) Constant time complexity.
  + In this case whatever the parameters of a function, the number of operations required to perform a task is one! The time taken to perform a task therefore never changes.
    - Complexity graph will display a straight line along the x axis.
* O(n) Linear time complexity.
  + In this case the time taken to perform a task depends on the number of parameters. The number of operations required increases linearly with the number of parameters.
    - Complexity graph will display a straight diagonal line equidistant from the x and y axis (right down the middle)
* Drop Constants
  + We drop constants bexause they ultimately do not effect the outcome.
  + O(2n) becomes O(n).
* O(n2) Quadratic time complexity
  + In this case the number of operations required to complete a task increase by the value of the quadratic. Usually one illustrates this by having for loops within for loops. One for loop in another illustrates O(n2). A third for loop within the second for loop illustrates O(n3). The time taken to complete a function increases very quickly and is very inefficient.
    - Complexity graph will show a line that quickly veers away from the x axis and bends acutely towards the y axis and then travels almost in parallel to the y axis.
* Non dominant terms O(n2 + n)
  + In this case we don’t consider the single “n” as relevant. It is non dominant and won’t affect the speed of processing in a major way so we remove it.
* O(log n) Logarithmic O
  + Refers to when we solve a problem using the divide and conquer approach. I.e. Searching for a number in an array by dividing and ordered array based on comparisons with the number we’re searching for. This decreases the search time because we don’t need to visit each number. Time taken can be calculated mathematically log28 = 3 or log21048576 = 20 (which is much faster than checking each number)
    - Complexity graph would show a line that quickly veers towards the x axis and then travels almost parallel to the x axis.

* Add vs Multiply
  + This represents an exception case for O(n) situations. If you have two non-nested for loops following each other in a method but using two different variables as parameters for their max size then use O(a + b).
  + However if the loops are nested then use O(a \* b)
* Space Complexity: Refers to the amount of memory used to perform a task.
  + O(n) space complexity
    - In the case of a recursive method, each iteration or recursion is saved to memory is saved in memory until the exit condition is met. Then the program takes the value return from each recursion and adds it to the other recursions. I.e. increases by one n times and then decreases by one n times.
  + O(1) space complexity.
    - In this case there is no recursion. One method may call another method from a for loop. I.e. the called method will be called as many times as the calling method loops. However, the result of the called method will be saved to memory once and then removed for each loop of the calling method. Therefore, there isn’t really an accumulation in memory exceeding one.
* How to measure the codes using Big O?

Perform some calculations to practice determining Big O.

* How to find time complexity for Recursive calls?
* How to measure Recursive Algorithms that make multiple calls?

**Section 6 - Top 10 Big O Interview Questions (Amazon, Facebook, Apple and Microsoft)**

* Product and Sum
* Print Pairs
* Print Unordered Pairs
* Print Unordered Pairs 2 Arrays
* Print Unordered Pairs 2 Arrays 100000 Units
* Reverse
* O(N)  Equivalents
* Factorial Complexity
* Fibonacci Complexity
* Powers of 2

**Section 7 - Arrays**

* What is an Array?
* Types of Array
* Arrays in Memory
* Create an Array
* Insertion Operation
* Traversal Operation
* Accessing an element of Array
* Searching for an element in Array
* Deleting an element from Array
* Time and Space complexity of One Dimensional Array
* One Dimensional Array Practice
* Create Two Dimensional Array
* Insertion - Two Dimensional Array
* Accessing an element of Two Dimensional Array
* Traversal - Two Dimensional Array
* Searching for an element in Two Dimensional Array
* Deletion - Two Dimensional Array
* Time and Space complexity of Two Dimensional Array
* When to use/avoid array

**Section 8 - Cracking Array Interview Questions (Amazon, Facebook, Apple and Microsoft)**

* Question 1 - Missing Number
* Question 2 - Pairs
* Question 3 - Finding a number in an Array
* Question 4 - Max product of two int
* Question 5 - Is Unique
* Question 6 - Permutation
* Question 7 - Rotate Matrix

**Section 9 - CHALLENGING Array Problems (Exercises)**

* Middle Function
* 2D Lists
* Best Score
* Missing Number
* Duplicate Number
* Pairs

**Section 10 - Linked List**

* **What is a Linked List?**
* **Linked List vs Arrays**
  + Elements of Linked List are independent objects.
  + The size of a Linked List is not predefined.
  + Arrays allow random access. Therefore accessing elements is more efficient in arrays while Linked Lists require the traversal of the Linked List to get to a particular node.
* **Types of Linked List**
  + Singly Linked List
    - Each node in the list stores the reference to the next node. There is no reference to the previous node. The lust node has a null reference instead of a reference to the next node.
  + Circular Linked List
    - Similarly, to singly linked list, each node stores the reference to the next node. However, in this case the last node points to the first node. Circular linked list is used when you need to traverse a list and then go back to the beginning when you reach the last list item.
  + Doubly Linked List
    - This linked list includes a reference to the next node as well as the previous node. Note that the previous reference of the first node is null and the reference of the next reference in the last node is null. In this case, if you had an app that plays music, you would want to go to next song or previous sound. So, this datatype would allow this action. (backward and forward traversal)
  + Circular Doubly Linked List
    - Like the doubly linked list with two changes. The previous reference of the first node points to the last node. And the next reference of the last node points to the first node. This allows forward and reverse traversal and or continue from the last node to the first and vice versa.
* **Linked List in the Memory**
  + Linked List nodes or elements are not located next to each(contiguously) other in memory like arrays. The reference to the next node is used to find the next node. Therefore, we cannot access any element directly.
* **Creation of Singly Linked List**
  + Create both the head and the tail and initialize their pointers/references to null.
  + Create a blank node and assign a value to it but the reference should point to null since it is the only node so far.
  + Now make both the head and tail point/refer to this one node
* Insertion in Singly Linked List in Memory
  + We can insert a node at the beginning, middle or end of a singly linked list.
  + Inserting at the beginning
    - Create a node.
    - Point new node to the first node.
    - Update the head node to point to the new node thus making it the first node.
  + Inserting in the middle
    - Create a new node in memory.
    - Traverse from head node to the desired location/current node.
    - Point the reference of the current node to the new node.
    - Point the reference of the new node to the next node.
  + Inserting at the end
    - Create a new node in memory.
    - Traverse the Linked List to the last node
    - Point the reference of the current/last node to the new node.
    - Point the reference of the new node to the tail node.
* Insertion in Singly Linked List Algorithm
  + Review algorithm
* Insertion Method in Singly Linked List
* Traversal of Singly Linked List
* Search for a value in Single Linked List
* Deletion of node from Singly Linked List
* Deletion Method in Singly Linked List
* Deletion of entire Singly Linked List
* Time and Space Complexity of Singly Linked List

**Section 11 - Circular Singly Linked List**

* Creation of Circular Singly Linked List
* Insertion in Circular Singly Linked List
* Insertion Algorithm in Circular Singly Linked List
* Insertion method in Circular Singly Linked List
* Traversal of Circular Singly Linked List
* Searching a node in Circular Singly Linked List
* Deletion of a node from Circular Singly Linked List
* Deletion Algorithm in Circular Singly Linked List
* Method in Circular Singly Linked List
* Deletion of entire Circular Singly Linked List
* Time and Space Complexity of Circular Singly Linked List

**Section 12 - Doubly Linked List**

* Creation of Doubly Linked List
* Insertion in Doubly Linked List
* Insertion Algorithm in Doubly Linked List
* Insertion Method in Doubly Linked List
* Traversal of Doubly Linked List
* Reverse Traversal of Doubly Linked List
* Searching for a node in Doubly Linked List
* Deletion of a node in Doubly Linked List
* Deletion Algorithm in Doubly Linked List
* Deletion Method in Doubly Linked List
* Deletion of entire Doubly Linked List
* Time and Space Complexity of Doubly Linked List

**Section 13 - Circular Doubly Linked List**

* Creation of Circular Doubly Linked List
* Insertion in Circular Doubly Linked List
* Insertion Algorithm in Circular Doubly Linked List
* Insertion Method in Circular Doubly Linked List
* Traversal of Circular Doubly Linked List
* Reverse Traversal of Circular Doubly Linked List
* Search for a node in Circular Doubly Linked List
* Delete a node from Circular Doubly Linked List
* Deletion Algorithm in Circular Doubly Linked List
* Deletion Method in Circular Doubly Linked List
* Entire Circular Doubly Linked List
* Time and Space Complexity of Circular Doubly Linked List
* Time Complexity of Linked List vs Arrays

**Section 14 - Cracking Linked List Interview Questions (Amazon, Facebook, Apple and Microsoft)**

* Linked List Class
* Question 1 - Remove Dups
* Question 2 - Return Kth to Last
* Question 3 - Partition
* Question 4 - Sum Linked Lists
* Question 5 - Intersection

**Section 15 - Stack**

* What is a Stack?
* What and Why of Stack?
* Stack Operations
* Stack using Array vs Linked List
* Stack Operations using Array (Create, isEmpty, isFull)
* Stack Operations using Array (Push, Pop, Peek, Delete)
* Time and Space Complexity of Stack using Array
* Stack Operations using Linked List
* Stack methods - Push , Pop, Peek, Delete and isEmpty using Linked List
* Time and Space Complexity of Stack using Linked List
* When to Use/Avoid Stack
* Stack Quiz

**Section 16 - Queue**

* What is a Queue?
* Linear Queue Operations using Array
* Create, isFull, isEmpty and enQueue methods using Linear Queue Array
* Dequeue, Peek and Delete Methods using Linear Queue Array
* Time and Space Complexity of Linear Queue using Array
* Why Circular Queue?
* Circular Queue Operations using Array
* Create, Enqueue, isFull and isEmpty Methods in Circular Queue using Array
* Dequeue, Peek and Delete Methods in Circular Queue using Array
* Time and Space Complexity of Circular Queue using Array
* Queue Operations using Linked List
* Create, Enqueue and isEmpty Methods in Queue using Linked List
* Dequeue, Peek and Delete Methods in Queue using Linked List
* Time and Space Complexity of Queue using Linked List
* Array vs Linked List Implementation
* When to Use/Avoid Queue?

**Section 17 - Cracking Stack and Queue Interview Questions (Amazon,Facebook, Apple, Microsoft)**

* Question 1 - Three in One
* Question 2 - Stack Minimum
* Question 3 - Stack of Plates
* Question 4 - Queue via Stacks
* Question 5 - Animal Shelter

**Section 18 - Tree / Binary Tree**

* What is a Tree?
* Why Tree?
* Tree Terminology
* How to create a basic tree in Java?
* Binary Tree
* Types of Binary Tree
* Binary Tree Representation
* Create Binary Tree (Linked List)
* PreOrder Traversal Binary Tree (Linked List)
* InOrder Traversal Binary Tree (Linked List)
* PostOrder Traversal Binary Tree (Linked List)
* LevelOrder Traversal Binary Tree (Linked List)
* Searching for a node in Binary Tree (Linked List)
* Inserting a node in Binary Tree (Linked List)
* Delete a node from Binary Tree (Linked List)
* Delete entire Binary Tree (Linked List)
* Create Binary Tree (Array)
* Insert a value Binary Tree (Array)
* Search for a node in Binary Tree (Array)
* PreOrder Traversal Binary Tree (Array)
* InOrder Traversal Binary Tree (Array)
* PostOrder Traversal Binary Tree (Array)
* Level Order Traversal Binary Tree (Array)
* Delete a node from Binary Tree (Array)
* Entire Binary Tree (Array)
* Linked List vs Python List Binary Tree

**Section 19 - Binary Search Tree**

* What is a Binary Search Tree? Why do we need it?
* Create a Binary Search Tree
* Insert a node to BST
* Traverse BST
* Search in BST
* Delete a node from BST
* Delete entire BST
* Time and Space complexity of BST

**Section 20 - AVL Tree**

* What is an AVL Tree?
* Why AVL Tree?
* Common Operations on AVL Trees
* Insert a node in AVL (Left Left Condition)
* Insert a node in AVL (Left Right Condition)
* Insert a node in AVL (Right Right Condition)
* Insert a node in AVL (Right Left Condition)
* Insert a node in AVL (all together)
* Insert a node in AVL (method)
* Delete a node from AVL (LL, LR, RR, RL)
* Delete a node from AVL (all together)
* Delete a node from AVL (method)
* Delete entire AVL
* Time and Space complexity of AVL Tree

**Section 21 - Binary Heap**

* What is Binary Heap? Why do we need it?
* Common operations (Creation, Peek, sizeofheap) on Binary Heap
* Insert a node in Binary Heap
* Extract a node from Binary Heap
* Delete entire Binary Heap
* Time and space complexity of Binary Heap

**Section 22 - Trie**

* What is a Trie? Why do we need it?
* Common Operations on Trie (Creation)
* Insert a string in Trie
* Search for a string in Trie
* Delete a string from Trie
* Practical use of Trie

**Section 23 - Hashing**

* What is Hashing? Why do we need it?
* Hashing Terminology
* Hash Functions
* Types of Collision Resolution Techniques
* Hash Table is Full
* Pros and Cons of Resolution Techniques
* Practical Use of Hashing
* Hashing vs Other Data structures

**Section 24 - Sort Algorithms**

* What is Sorting?
* Types of Sorting
* Sorting Terminologies
* Bubble Sort
* Selection Sort
* Insertion Sort
* Bucket Sort
* Merge Sort
* Quick Sort
* Heap Sort
* Comparison of Sorting Algorithms

**Section 25 - Searching Algorithms**

* Introduction to Searching Algorithms
* Linear Search
* Linear Search in Python
* Binary Search
* Binary Search in Python
* Time Complexity of Binary Search

**Section 26 - Graph Algorithms**

* What is a Graph? Why Graph?
* Graph Terminology
* Types of Graph
* Graph Representation
* Graph in Java using Adjacency Matrix
* Graph in Java using Adjacency List

**Section 27 - Graph Traversal**

* Breadth First Search Algorithm (BFS)
* Breadth First Search Algorithm (BFS) in Java - Adjacency Matrix
* Breadth First Search Algorithm (BFS) in Java - Adjacency List
* Time Complexity of Breadth First Search (BFS) Algorithm
* Depth First Search (DFS) Algorithm
* Depth First Search (DFS) Algorithm in Java - Adjacency List
* Depth First Search (DFS) Algorithm in Java - Adjacency Matrix
* Time Complexity of Depth First Search (DFS) Algorithm
* BFS Traversal vs DFS Traversal

**Section 28 - Topological Sort**

* What is Topological Sort?
* Topological Sort Algorithm
* Topological Sort using Adjacency List
* Topological Sort using Adjacency Matrix
* Time and Space Complexity of Topological Sort

**Section 29 - Single Source Shortest Path Problem**

* SWhat is Single Source Shortest Path Problem?
* Breadth First Search (BFS) for Single Source Shortest Path Problem (SSSPP)
* BFS for SSSPP in Java using Adjacency List
* BFS for SSSPP in Java using Adjacency Matrix
* Time and Space Complexity of BFS for SSSPP
* Why does BFS not work with Weighted Graph?
* Why does DFS not work for SSSP?

**Section 30 - Dijkstra's Algorithm**

* Dijkstra's Algorithm for SSSPP
* Dijkstra's Algorithm in Java - 1
* Dijkstra's Algorithm in Java - 2
* Dijkstra's Algorithm with Negative Cycle

**Section 31 - Bellman Ford Algorithm**

* Bellman Ford Algorithm
* Bellman Ford Algorithm with negative cycle
* Why does Bellman Ford run V-1 times?
* Bellman Ford in Python
* BFS vs Dijkstra vs Bellman Ford

**Section 32 - All Pairs Shortest Path Problem**

* All pairs shortest path problem
* Dry run for All pair shortest path

**Section 33 - Floyd Warshall**

* Floyd Warshall Algorithm
* Why Floyd Warshall?
* Floyd Warshall with negative cycle,
* Floyd Warshall in Java,
* BFS vs Dijkstra vs Bellman Ford vs Floyd Warshall,

**Section 34 - Minimum Spanning Tree**

* Minimum Spanning Tree,
* Disjoint Set,
* Disjoint Set in Java,

**Section 35 - Kruskal's and Prim's Algorithms**

* Kruskal Algorithm,
* Kruskal Algorithm in Python,
* Prim's Algorithm,
* Prim's Algorithm in Python,
* Prim's vs Kruskal

**Section 36 - Cracking Graph and Tree Interview Questions (Amazon,Facebook, Apple, Microsoft)**

**Section 37 - Greedy Algorithms**

* What is Greedy Algorithm?
* Well known Greedy Algorithms
* Activity Selection Problem
* Activity Selection Problem in Python
* Coin Change Problem
* Coin Change Problem in Python
* Fractional Knapsack Problem
* Fractional Knapsack Problem in Python

**Section 38 - Divide and Conquer Algorithms**

* What is a Divide and Conquer Algorithm?
* Common Divide and Conquer algorithms
* How to solve Fibonacci series using Divide and Conquer approach?
* Number Factor
* Number Factor in Java
* House Robber
* House Robber Problem in Java
* Convert one string to another
* Convert One String to another in Java
* Zero One Knapsack problem
* Zero One Knapsack problem in Java
* Longest Common Sequence Problem
* Longest Common Subsequence in Java
* Longest Palindromic Subsequence Problem
* Longest Palindromic Subsequence in Java
* Minimum cost to reach the Last cell problem
* Minimum Cost to reach the Last Cell in 2D array using Java
* Number of Ways to reach the Last Cell with given Cost
* Number of Ways to reach the Last Cell with given Cost in Java

**Section 39 - Dynamic Programming**

* What is Dynamic Programming? (Overlapping property)
* Where does the name of DC come from?
* Top Down with Memoization
* Bottom Up with Tabulation
* Top Down vs Bottom Up
* Is Merge Sort Dynamic Programming?
* Number Factor Problem using Dynamic Programming
* Number Factor : Top Down and Bottom Up
* House Robber Problem using Dynamic Programming
* House Robber : Top Down and Bottom Up
* Convert one string to another using Dynamic Programming
* Convert String using Bottom Up
* Zero One Knapsack using Dynamic Programming
* Zero One Knapsack - Top Down
* Zero One Knapsack - Bottom Up

**Section 40 - CHALLENGING Dynamic Programming Problems**

* Longest repeated Subsequence Length problem
* Longest Common Subsequence Length problem
* Longest Common Subsequence  problem
* Diff Utility
* Shortest Common Subsequence  problem
* Length of Longest Palindromic Subsequence
* Subset Sum Problem
* Egg Dropping Puzzle
* Maximum Length Chain of Pairs

**Section 41 - A Recipe for Problem Solving**

* Introduction
* Step 1 - Understand the problem
* Step 2 - Examples
* Step 3 - Break it Down
* Step 4 - Solve or Simplify
* Step 5 - Look Back and Refactor

**Section 41 - Wild West**