Chapter 1: Arrays and Strings:

1. Implement an algorithm to determine if a string has all unique characters. What if you cannot use additional data structures?

- T: Have an eye for detail
- T: Demonstrate a solid fundamental of Computer Science
- T: Clarify your assumptions with the interviewer
- T: Discuss alternative ways to solve the problem after you developed your code
- Q: How can I reduce the space complexity of my algorithm?
- DS: Boolean array of 128 characters (Unicode)
- Q: Is the string an ASCII string or Unicode string?
- S: Create an array of boolean values, where the flag at index i indicates whether character i in the Unicode is contained in the string. Return false when you immediately see the same value for the second time.

2. Given two strings, write a method to decide if one is a permutation of the other

- T: Draw and describe a simple and representative model
- T: Reverse engineer the problem
- Permutation must be of the same length
- Q/T: Describe a permutation
- Q: Is whitespace significant?
- S1: If two strings are permutations, then we know that they have the same characters in different orders. Sorting the string will put the characters from two permutations in the same order. Then compared sorted versions of the string.
- S2: Check if two words have the same character counts
- P: If any position in the array of letters is less than zero, return false, else return true

3. Write a method to replace all space in a string with '%20'. You may assume that the string has sufficient space at the end to hold the additional characters, and that you are given the "true" length of the string.

- S: Common approach in string manipulation problems is to edit the string starting from the end and working backwards. This is useful because we have an extra buffer at the end, which allows us to change characters without worrying about what we're overwriting.
- P: Be careful of edge cases with for loops
- Algo: Function 1: Count the number of spaces, then Function 2:

4. Given a string, write a function to check if it is a permutation of a palindrome. A palindrome is a word or phrase that is the same forwards and backwards. A permutation is an rearrangement of letters, does not need to be limited to just dictionary words.

- Ex: Tact Coa
- T: Figure out what it means for a string to be a permutation of a palindrome
- Palindrome string is the same forwards as it is backwards
- T: Understand and describe how the math breaks down to the interviewer
- S1: Count how many times each character appears, then iterate through the hash table and ensure that no more than one character has an odd count
- S2: Make smaller incremental improvements -- instead of checking the number of odd counts at the end, we can check as we go along, then when we get to the end, we have our answer
- T: Consider the given counting techniques

5. Given two strings, write a function to check if they are one edit or zero edits away, given: insert a character, remove a character, or replace a character.

- T: Start by first thinking about the "brute force" algorithm to solve the problem
- T: Ask yourself what this problem means, think about the meaning of each operation
- T: Understand what each method means
- T: Understand what exactly it is that you are doing
- T: Explain the design of your logica
- T: Functionalize and modularize your program
- T: Do not be distracted by what is in the example
- Q: Are these correct mechanics of the algorithm?
- S1: Check for one character replacement away, one insertion away, and one removal (inverse of insertion) away

<u>Chapter 2: Linked Lists:</u>

1. Write code to remove duplicates from an unsorted linked list

- St: Must be able to track duplicates, using a simple hash table
- T: Explain how much and why the solution takes O(x) time
- P: Use a hash table and map current values into it

2. Implement an algorithm to find the kth to last element of a singly linked list

- T: Approach the problem both recursively and non-recursively
- T: Articulately explain the algorithm to the interviewer
- K = 0 would return the last element, etc.
- S1: If the size of the linked list is known, then the kth to last element is the (length-k)th element. Iterate through the linked list to find this element.
- S2: Algorithm that recurses through the entire linked list. When it hits the end, the method passes back a counter set to 0. Each parent call adds 1 to this counter. When the counter equals k, we know we have reached the kth to last element of the linked list.

- S3: More optimal but less straightforward way is to implement this iteratively. Can use two pointers, p1 and p2. Place them k nodes apart in the linked list by putting p2 at the beginning and move p1 k nodes into the list. Then, when we move them at the same pace, p1 will hit the end of the linked list after length - k steps. At that point, p2 will be k nodes from the end.
- T: Draw out an example for the interviewer
- T: Stay away from solutions that are way too trivial

3. Implement an algorithm to delete a node in the middle (any node but the first and last node, not necessarily the exact middle) of a singly linked list, given only access to that node

- Ex: Node c from a b c d e f -> a b d e f
- Edge case: Too small
- S: Simply copy the data from the next node over to the current node and then delete the next node
- T: Point out when the problem can not be solved

5. You have two numbers represented by a linked, where each node contains a single digit. The digits are stored in reverse order, such that the 1's digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list.

- Ex: 7 1 6 = 6 1 7
- T: Explain how the mathematical operation works in the given scenario (addition)
- T: Mimic mathematical processes recursively
- T: Take careful considerations when implementing your code
- T: Understand the parameters that are passed into the function
- T: Want to avoid stack overflow and or accessing invalid memories
- T: Understand the concept behind the code that you are implementing
- T: Highlight how a typical case would break down
- T: Make the code easy and clean to read
- T: Talk about various manners to solve the problem
- T: Can this solution apply to the vast majority of solutions
- S: Reversely add and carry over the sum of two digits and then reverse the digits

6. Implement a function to check if a linked list is a palindrome

- T: Explain what a palindrome is (same forward as it is backward)
- S1: Reverse and compare -- reverse the linked list and compare the reversed list to the original list. If they are the same, then they are identical. Only need to compare the first half of the list.
- T: Modularize and functionalize your code

- S2: Iterative approach -- We want to detect linked lists where the front half of the list is the reverse of the second half. Do that by reversing the front half of the list.
- T: Use the most efficient and appropriate data structure
- M: Push the first half of the elements onto a stack
- T: Be careful of caveats and edge cases
- T: For half/midpoints, use the slow/fast runner technique
- T: Understand the specific data type (singly v. doubly linked list)
- T: Understand the problem diagrammatically
- T: Iterate and pass through list with recursion
- S3: Recursive Approach
- T: Must understand where you are in the recursive process
- T: Programming Conception
- T: Map and plan out your functions before implementing them

Chapter 3: Stacks and Queues

1. Describe how you could use a single array to implement three stacks
    - S1: Fixed Division -- Divide the array in three equal parts and allow the individual stack to grow in that limited space.
    - T: Can you bundle the code into a stack?
    - T: Functionize and modularize the problem
    - S2: Allow the stacks to be flexible in size. When one stack exceeds its initial capacity, we grow the allowable capacity and shift elements as necessary. Design array to be circular, such that the final stack may start at the end of the array and wrap around to the beginning

2. How would you design a stack which, in addition to push and pop, has a function min which returns the minimum element? Push, pop, and min should all operate in O(1) time.
    - T: Brainstorm solutions, refine your first solution
    - T: Walk through a short example
    - T: Keep track of specific data at each state
    - T: Always seek to refine your solution

3. Imagine a literal stack of plates. If the stack gets too high, it might topple. We would likely start a new stack when the previous stack exceeds some threshold. Implement a data structure SetOfStacks that mimics this. SetOfStacks should be composed of several stacks and should create a new stack once the previous one exceeds capacity. SetOfStacks.push() and SetOfStacks.pop() should behave identically to a single stack (pop() should return the same values as it would if there were just a single stack). Implement a function popAt(int index) which performs a pop operation on a specific sub-stack.

- T: Draw/diagram the problem
- T: Follow the set of directions spelled out in the problem
- T: Understand the data structure you are given
- T: Understand the methods available on a certain data structure
- T: Be able to implement the methods of a given data structure

4. Queues via Stacks: Implement a MyQueue class which implements a queue using two stacks.
- DS: Queue is FIFO and stack is LIFO, modify peek and pop to go in reverse order
- S1: stackNewest has the newest elements on top and stackOldest has the oldest elements on top. When dequeuing an element, we want to remove the oldest element first, and so we dequeue from stackOldest. If stackOldest is empty, then we want to transfer all elements from stackNewest into this stack in reverse order. To insert an element, we push onto stackNewest, since it has the newest elements on top
- T: Understand and describe how data structures work
- P: Properly use conditional statements
- Interviewers are more interested in your understanding of the big picture, it is okay to ask them big picture questions

5. Sort Stack: Write a program to sort a stack such that the smallest items are on the top. You can use an additional temporary stack, but you may not copy the elements into any other data structure (such as an array). The stack supports the following operations: push, pop, seek, and isEmpty.
- One approach is to implement a rudimentary sorting algorithm
- T: Understand how counting work and how to efficiently count data
- T: Make use of multiple of the same or different data structures
- T: Properly implement the appropriate methods for the given data structure
- Methods: pop, push, peek
- T: Explain to the interviewer the time and space complexity at some point at the end of the algorithm

<u>Chapter 4. Trees and Graphs</u>

1. Route between nodes: Given a directed graph, design an algorithm to find out whether there is a route between two nodes.
- S: Simple graph traversal, such as DFS or BFS. Check with one of the two nodes and during a traversal, check if the other node is found. Mark any node found in the course of the algorithm as “already visited” to avoid cycles and repetition of the nodes
- T: Initiate data structure
- T: Talk about the tradeoffs between BFS and DFS for this and other problems

- DS: BFS can be useful to find the shortest path, whereas DFS may traverse one adjacent node very deeply before ever going onto the immediate neighbors

2. Minimal tree: Given a sorted (increasing order) array with unique integer elements, write an algorithm to create a binary search tree with minimal height.

- S: To create a tree of minimal height, we need to match the number of nodes in the left subtree to the number of nodes in the right subtree as much as possible.
- T: Draw out a diagram/example of the solution
- T: Understand the methods available on the given data structure
- T: Explain the algorithm time analysis
- T: Highlight the methods of the algorithm and explain what they do
- T: Use recursion if possible
- T: Pass in correct parameters to the function
- T: Avoid little off-by-one errors
- T: Test the sensitive areas/edge cases very thoroughly -- they can ruin everything

3. List of Depths: Given a binary tree, design an algorithm which creates a linked list of all the nodes at each depth

- S: We can traverse the graph any way that we would like, provided we know which level we're on as we do so. We can implement a simple modification of the pre-order traversal algorithm, where we pass in level+1 to the next recursive call. The code below provides an implementation using depth first search.
- T: Implement algorithm using depth-first search
- T: Understand the efficiencies between other algorithms and explain them to the interviewer

4. Check Balanced: Implement a function to check if a binary tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that the heights of the two subtrees of any node never differ by more than one.

- Balanced means that for each node, the two subtrees differ in height by no more than one
- S: Implement a solution based on this definition. Simply recurse through the entire tree, and for each node, compute the heights of each subtree
- T: Do two things at once with your algorithm
- T: Return an error code if the algorithm does not work
- Return two calls/options when the algorithm is finished
- Break down the space and time complexities for the interviewer

5. Validate BST: Implement a function to check if a binary tree is a binary search tree

- Can implement the solution in two different ways

- First strategy leverages in-order traversal, second builds off the property that left <= current < right
- S1: In-order traversal: Do an in-order traversal, copy the elements to an array, then check to see if the array is sorted. Takes up a lot of extra memory, but mostly works. Can't handle duplicate values in the tree properly.
- T: Point out possible problems with the algorithm
- T: Explain the functionality of the algorithm
- T: Keep track of the end of a data structure and edge cases
- Check and recurse various parts of the BST
- T: Explain what exactly the problem means and draw it out. Explain a right and wrong scenario
- T: Understand the conditions necessary for the problem to be solved as true
- T: Update the conditions as you work through your algorithm
- T: Handle null cases correctly as well

Chapter 5: Bit Manipulation

1. Insertion: You are given two 32-bit numbers, N and M, and two bit positions, i and j. Write a method to insert M into N such that M start at big j and end at bit i. You can assume that the bits j through i have enough space to fit all of M. That is, if M = 10011, you can assume that there are at least 5 bits between j and i.

- Ex. Input N = 10000000000. Output N = 10001001100. M = 10011, i=2, j=6
- S: Problem can be approached in three key steps
- T: Make sure to thoroughly test your code.
- T: Outline your technique and methodology

2. Binary to String: Given a real number between 0 and 1 that is passed in as a double, print the binary representation. If the number cannot be represented accurately in binary with at most 32 characters, print "ERROR"

- T: Thoroughly follow and listen to instructions
- Q: Ask yourself what a non-integer number in binary looks like
- S: To print the decimal part, we can multiply by 2 and check if 2n is greater than or equal to 1. This is essentially "shifting" the fractional sum.
- T: Implement error condition and or base case
- T: Set limits to what your code can do
- T: Prepare thorough test cases for this problem and actually run through them in your interview

3. Flip Bit to Win: You have an integer and you can flip exactly one bit from a 0 to a 1. Write code to find the length of the longest sequence of 1's you could create.

- Ex: 1775 -> 8 (11011101111)
- Think about each integer as being an alternating sequence of 0s and 1s
- Brute Force: Convert an integer into an array that reflects the lengths of the 0s and 1s sequences.
- T: Detail how you describe the runtime -- what does N mean
- What is the best conceivable runtime?

4. Next Number: Given a positive integer, print the next smallest and the next largest number that have the same number of 1 bits in their binary representation

- Number of ways to approach problem: brute force, bit manipulation, and clever arithmetic
- Understand and describe the approach before implementing it
- S1: Easy approach is brute force: count the number of 1s in n and then increment (or decrement) until you find a number with the same number of 1s
- Understand the methods required to solve the problem with the given approach
- Have a comprehensible understanding of the problem
- Walk through an example with the interviewer and clarify the problem
- Implement the correct variables
- S2: Bit manipulation approach to get previous number. 1. Compute number of trailing ones and zeros to the left of trailing ones 2. Flip the rightmost non-trailing one to a zero. 3. Clear all bits to the right of bit p. 4. Insert number of trailing ones + 1 immediately to the right of position p.
- Understand what each variable stands for and how it can be used
- Use binary conditions and assessments in your code
- Solutions start with a strong understanding of the problem and a feasible strategy at every step of the problem

5. Debugger: Explain what the following code does: ((n & (n-1)) == 0).

- Work backwards to solve the problem
- Understand what each part of the problem is
- Solve the problem in a manner that is parallel with the question
- Diligently break down and understand the problem
- All problems can be logically deduced

Chapter 6: