Brian Morales

75 LeetCode

**Arrays & Hashing**:

1. Two Sum:

* A HashMap can be used to optimize the runtime to O(n) instead of O(n^2) by using a for loop to iterate each value. We can find the specific value by subtracting the iterating value with the target to find the second target needed within the hash which is an O(1) lookup.

1. Contains Duplicate:

* This contains duplicate algorithm can be approached in a few ways. One for the most common ways would be to have a nested for loop iterate over each element in the array to check if there is a common value, the only issue would be that the run time complexity would run of O(n^2) which would be inefficient. The more optimized approach would be to use an O(n) space to get a better run time which in this case would be O(n) when using a set. This would allows us to just store each element in the array, causing the iteration of the array to occur once.

1. Product of Array Except Self:

* This algorithm is a bit trickier; it requires that we traverse through the array and multiply with every other element except itself in place of the index of the array. Example [1,2,3,4] = [24,12,8,6]. We would need to traverse through the array forward and then back to obtain the in-place index product of every other element except itself. We also need a temporary variable that will hold each of the new product value per index position. The approach is the obtain all values before and after initial index value. If it is out of bounds than we assume its 1.

1. **Three Sum: \*\*\*\*\*\* : Work on a little more to understand.**

* Best way to approach the combination of 3 integers to sum up to 0 will be the sort the array.
* We want to iterate through each index besides the last 2 of an array since we need 3 integers, it wouldn’t make sense at that point. In the case when we need to find combinations, it would be best to iterate and then have pointers from the end and +1 after the initial start index of the iteration. During the pointer movement, we skip duplicates since we essentially would have tried that combination already. Ex: -4 is our first iteration number, sum = 0- -4 = 4, and if there were two -1 next to each other, we would skip the iteration since we would have attempted that combination already.
* Essentially, we want two pointers: One after the first index ex: i+1 and one at the end ex: array length -1.

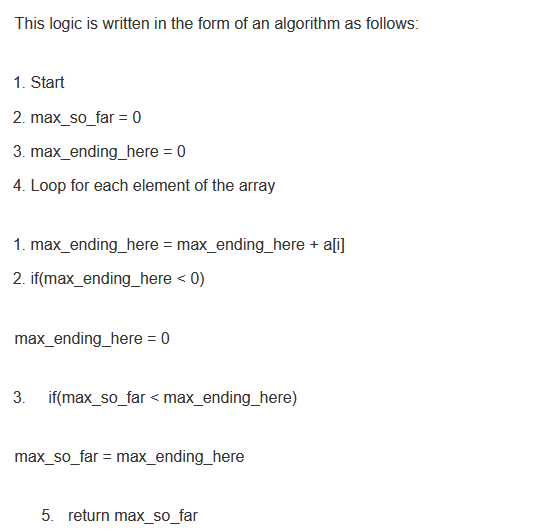
Sliding Window: is a computational technique which aims to reduce the use of nested loop and replace it with a single loop, thereby reducing the time complexity.

1. Best Time to Buy and Sell Stock:

* One Approach : The runtime complexity in this problem is O(n^2). We are reviewing each element and the one after and comparing the difference to obtain the max profit from a buy and sell standpoint. This requires a nested for loop from one element to the adjacent one. **Second Approach:** A more optimal approach would be to use left and right pointers to achieve this and receive the max profit. We can double check to see if the left pointer is always less than the right pointer so we can view the max profit and if that isn’t the case then we would update the pointers but the right would always update.

Greedy: paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solution are best fit for Greedy.

1. Maximum Subarray:

* Dynamic Programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions using a memory-based data structure (array, map, etc.). So the next time the same sub-problem occurs,
* Instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time.
* We can use Kadane algorithm so solve this, just keep the previous result of whichever operation it may be and see if the previous index or result adds to the final result needed at hand.
* 

1-D Dynamic Programming:

1. Maximum Product Subarray:

* This subarray problem is similar to the subarray problem above, a comparison is needed using Kadane’s algorithm to be able identify if a product of a contiguous sub array would be less then 0 or negative, otherwise multiply with the index number and compare for the largest product value. We need to watch out for negative number in succession as the contiguous subarray can equal to a max positive number. Use MATH.MIN and MATH.MAX. What we want to do is have a temp variable for the max as the max will change constantly. Only the max product will store the actual max value.

Binary Search:

1. Find Minimum in Rotated Sorted Array:

* In this case what we need to do is divide and conquer the array. A binary search is need for the approach. If the middle/pivot of the array is greater than the beginning index and the beginning index is less than the last index then we search for the left to the middle, otherwise we search from the middle index +1 to the end. This is essentially the binary search algorithm.

1. Search in Rotated Sorted Array:

* In this problem, we need to divide and conquer as well. It is best to use a binary search algorithm and break down the array by half every time per loop. We need two pointers to keep track of the start and end while also changing the middle each cycle essentially breaking down the array even further. We are also comparing the number of index start and middle in the array to determine where to start/increment. We are constantly moving the pointer until the start pointer lands on the value index.

Two-Pointers:

1. Container With Most Water:

* Similar to the problem above. REMEMBER: because we are searching for the best outcome or in this cast the container with the most water, we have to iterate through each combination. Here we can use pointer starting from the beginning and end to bring them closer as we compare it to our initial max value, whatever the assumption of that value may be.
* We want to compare both start and end of array values, if the start is less than the end we move the start and vice versa. This is because we are looking for the max , so any low number doesn’t help the approach. We continue to do this until we find the max area.

Heap:

1. Top K Frequent Elements:\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* For this problem we need to do a few steps. 1. Is to use a hashmap to keep track of the amount of reoccurring values. 2. Is to iterate through the keyset and create a listarray and using the index of the array will store the amount of times a value has occurred. For example: listarray[value the same as from key] = map[get value from key].add(key). So if the number “2” showed up 3 times then the index of 3 will store the key “2” as it showed up three times. 3. This we will need to iterate through the array list in reverse and keep a counter for both the array that will store the k elements and for the value within the array list in case there are multiple numbers that showed up the same amount of times, but we only want k elements.
* The runtime complexity for this is O(n) based on the amount of values in the hash map and array.
* The space complexity is also O(n).

Strings:

1. Group Anagrams:

* This problems requires converting a char array to a string such that the string value of is from the array itself. What we want to do is count each character and store them in an array and use the string value to convert that array and store that into a HashMap along with a list of corresponding words that match the letters with their number of occurrences.
* The time complexity is O(m\*n) with m being the amount or average characters per word that we need to store in the array, and n is the number of words stores in the list that we have to traverse through. The space complexity is O(n) as we have to store each by their character in the array to store in the HashMap.

Graphs:

1. Longest Consecutive Sequence:

* For this problem the brute force would be to check every combination of every number within the array but that would have a very slow runtime of O(n^2). The approach here is to make it O(n) where we traverse once in the array.
* We want to keep track of each number we traverse with its +1 and -1. We want to first store the array in a set and as we use a while loop to check for +1 and -1 separately, we track the amount of corresponding numbers in the set and remove them so we do not have to visit them when we get to those same numbers in the array. We continue to evaluate the max number until we visited every number and the set is essentially empty.
* This runs in a time complexity of O(n) with a space of O(n) for the set that we need to create.

**Binary / Bit Manipulation:**

1. Sum of Two Integers:
2. Number of 1 Bits:

* Using Brian Kernighan Algorithm, we will not check/compare or loop through all the 32 bits present but only count the set bits which is way better than checking all the 32 bits
* Suppose we have a number 00000000000000000000000000010110 (32 bits), now using this algorithm we will skip the 0's bit and directly jump to set bit(1's bit) and we don't have to go through each bit to count set bits i.e. the loop will be executed only for 3 times for the mentioned example and not for 32 times.
* Assume we are working for 8 bits for better understanding, but the same logic apply for 32 bits
* So, we will take a number having 3 set bits.
* n = 00010110
* n - 1 = 00010101(by substracting 1 from the number, all the bits gets flipped/toggled after the \*\*rightmost set bit including the rightmost set bit itself. \*\*
* After applying &(bitwise AND) operator on n and n - 1 i.e. (n & n - 1), the righmost set bit will be turned off/toggled/flipped
* Let's understand step by step
* 1st Iteration
* 00010110 --> (22(n) in decimal)
* & 00010101 --> (21(n - 1) in decimal i.e. flipping all the bits of n(22) after rightmost set bit including the rightmost set bit itself )
* 00010100 --> (20(n & n - 1) in decimal i.e after applying bitwise AND(&), the rightmost set bit will be turned off )
* After applying bitwise AND(&) ,assign this number to n i.e. n = n & n - 1
* n = 00010100 (20 in decimal)
* and increase the count
* bitCount++ (Initial bitCount = 0. By incrementing it, the bitCount = 1)
* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
* 2nd Iteration
* 00010100 --> (20(n) in decimal)
* & 00010011 --> (19(n - 1) in decimal i.e. flipping all the bits of n(20) after rightmost set bit including the rightmost set bit itself )
* 00010000 --> (16(n & n - 1) in decimal i.e after applying bitwise AND(&), the rightmost set bit will be turned off )
* After applying bitwise AND(&) ,assign this number to n i.e. n = n & n - 1
* n = 00010000 (16 in decimal)
* and increase the count
* bitCount++ (previous bitCount = 1. By incrementing it, the bitCount = 2)
* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
* 3rd Iteration
* 00010000 --> (16(n) in decimal)
* & 00001111 --> (15(n - 1) in decimal i.e. flipping all the bits of n(16) after rightmost set bit including the rightmost set bit itself )
* 00000000 --> (0(n & n - 1) in decimal i.e after applying bitwise AND(&), the rightmost set bit will be turned off )
* After applying bitwise AND(&) ,assign this number to n i.e. n = n & n - 1
* n = 00000000 (0 in decimal)
* and increase the count
* bitCount++ (previous bitCount = 2. By incrementing it, the bitCount = 3)
* Now, since the n = 0, there will be no further iteration as the condition becomes false, so it will come out of the loop and return bitCount which is 3 which is desired output.
* COMPLEXITY
* Time: O(log2n), where n is the given number
* Space: O(1), in-place

1. Counting Bits:
2. Missing Number:
3. Reverse Bits:

**Dynamic Programming:**

The basic idea of dynamic programming is to store the result of a problem after solving it. So when we get the need to use the solution of the problem, then we don't have to solve the problem again and just use the stored solution.

Imagine you are given a box of coins and you have to count the total number of coins in it. Once you have done this, you are provided with another box and now you have to calculate the total number of coins in both boxes. Obviously, you are not going to count the number of coins in the first box again. Instead, you would just count the total number of coins in the second box and add it to the number of coins in the first box you have already counted and stored in your mind. This is the exact idea behind dynamic programming.

Recording the result of a problem is only going to be helpful when we are going to use the result later i.e., the problem appears again. This means that dynamic programming is useful when a problem breaks into subproblems, the same subproblem appears more than once.

**Two Approaches of Dynamic Programming**

There are two approaches of the dynamic programming. The first one is the top-down approach and the second is the bottom-up approach. Let's take a closer look at both the approaches.

**Top-Down Approach**

The way we solved the Fibonacci series was the top-down approach. We just start by solving the problem in a natural manner and stored the solutions of the subproblems along the way. We also use the term **memoization**, a word derived from memo for this.

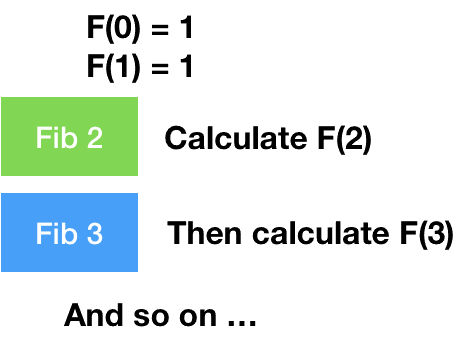
In other terms, it can also be said that we just hit the problem in a natural manner and hope that the solutions for the subproblem are already calculated and if they are not calculated, then we calculate them on the way.

**Bottom-Up Approach**

The other way we could have solved the Fibonacci problem was by starting from the bottom i.e., start by calculating the 2nd

term and then 3rd

and so on and finally calculating the higher terms on the top of these i.e., by using these values.



We use a term **tabulation** for this process because it is like filling up a table from the start.

Let's again write the code for the Fibonacci series using bottom-up approach.

Dynamic programming gives the power of speed. It can be faster than doing recursion. The only trade off would be the uses of memory as it uses more memory than recursive functions would.

1-D Dynamic Programming :

1. Climbing Stairs:

* This problem, do not overthink it. It is just the **Fibonacci sequence**, each step either takes 1 or 2 steps to get to the target step goal. We use the bottom-up approach and recursively subtract the number of steps by one and add the previous one step and two step into one variable and take the two-step variable and have that equal to what was originally in the one variable. Each time we go up it continues until we reach the base case. Memoization is needed in order to get the previous calculation to find the distinct steps.
* This would allow us to store the previous sum in a temp variable and the update but variables that either go by one step or two step and once the steps reach zero, we return the sum of both those variables. This is a bottom-up dynamic programming approach.

1. Coin Change:

* This problem can be very confusing if not approached a certain way. The way that we implemented it was that we created an array with n+1 length, this is so that we can compare each cell with every coin in the array so that we have a reference of that value if it enters the cell. The target value created by the for loop needs to be greater than or equal to the value in the coin array so that we can compare the minimum with that and the n+1 length.
* Just remember that this is a dynamic programming approach and that this can also be achieved with recursion. This is O(n) time complexity.

1. Longest Increasing Subsequence:

* Since we have to find the max subsequence, we need to iterate through the array. This needs to be done in a dynamic approach in which we will store the previous array value/cell/node subsequence value up to that point. So, the first array value will be 1 since it will automatically be considered a subsequence array.
* The more we continue down the array the more we can compare based on the values that were already stored in a different array keep the max values of subsequence steps up to those points.
* Two pointers in a for loop basically. The target being the 0+1 and the previous value being the first value of the array, with that we plus 1 for every time one is lower than the other.

1. Wordbreak:

* This operation requires a dynamic programming approach that will go through each letter using two pointers. One from the start and another from start+1; This is a runtime of O(n^2) since we have to run through each letter and ongoing until it reaches the second pointer. The words will essentially resize each time as both pointers get bigger going through each word and storing their Boolean value in a dp array. The last index of the dp array will contain the result.

1. House Robber:

* The Approach for this problem is O(n) linear time complexity.
* We essentially start from the assumed third index of the array using a pointer and from the index ongoing we calculate the non-adjacent maximum value for every iteration until we reach the last index.

1. House Robber 2:

* We can approach this problem using a few ways. One way is to use two pointers to starting at 0 and keep track of each value for the iteration of the array. We use one pointer to get the max of the two pointers and use the other pointer to equate to the temp variable that will add up the value in the array with the first pointer. Essentially, we want two pointers with a temp value to hold the addition of one pointer with the value of the iteration through the array.
* The time complexity is O(n) with a space of O(1) unless we go with a dp approach.
* We need to iterate forward and then backwards so we can avoid the first index and last index from touching each other.

1. Decode Ways:

* This approach uses a dynamic programming approach. This is a bottom up using memorization. We create a dp array of size +1 where it hold one for the inevitable times a single digit would count as a way. From there we work in reverse of the string.
* If the character is 0 we can skip an conditional statements as it is most likely part of a double digit. For a single digit the max is 1-9 , for double it is 0-9 but that only applies to where the first digit is 1. If the first digit is 2, the second digit can only go from 0-6 since there are 26 letters in the alphabet and that is part of the problem that we are trying to solve here.

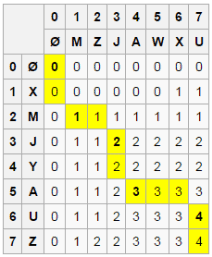
1. Longest Palindrome Substring:
2. Palindrome Substrings:

2-D Dynamic Programming:

1. Unique Paths:

* For unique paths, for this problem we need to use a dp approach. We need to store every value based on the number of different ways a position in the matrix can reach the end. For this we start at the end which is ultimately 1. We us a dp matrix that is +1 bigger and continuously add up the value stored in the right and bottom of the initial index position. We continue to do this in a for loop until we reach the first index of the matrix in which it adds up the right and bottom to get the full value.
* The runtime complexity is O(r+c) or O(n) since we only have to traverse through each column per array once. The space complexity is O(n) as well to store the values in a dp array.

1. Longest Common Subsequence:

* Find LCS;  
  Let X be “XMJYAUZ” and Y be “MZJAWXU”. The longest common subsequence between X and Y is “MJAU”. The following table shows the lengths of the longest common subsequences between prefixes of X and Y. The ith row and jth column shows the length of the LCS between X\_{1..i} and Y\_{1..j}.  
    
  you can refer to [here](https://en.m.wikipedia.org/wiki/Longest_common_subsequence_problem) for more details.
* What ends up happening here is that by doing top down, we can get a reference to the previous letter if the letters equal we can add a number going diagonally if not we continue with the max of the previous letter.

**Greedy:**

1. Jump Game:

* The jump game approach is very straight forward. We want to be able to reach the ending based on their indices and values within them. The max jump an index can jump is the value within that initial index. If the size of the array is 5 then the last index is 4 and that is the number we want to reach onwards based on the values that come before it. We can use two pointers one for the last index and another that will continuously change as we iterate through the array. The more the previous index can reach the end the more we know that we just need to reach to the second pointer to really get to the index. It is a dp approach s we already know that the index after can reach the end so we just need to know if we can reach the second pointer that is constantly changing. The jump is possible if the value constantly changing is the first index value of the whole array or 0. This approach is an O(n) time complexity while the space is O(1).

**Graphs:**

1. Number of Islands:

* Wee need to use DFS to recursively check each position in the matrix, only being able to move up, down, left and right. With this information we use recursion to check each position for every index in the matrix. To avoid revisiting the same position, we change the character from ‘1’ to ‘0’ and that in itself is our base case along with if the position goes out of bounds.
* The time complexity is O(V+E) or O(V). V being the vertex and E being the edges of the graph.

1. Clone Graph:

* This problem, we need to hard clone a graph. We need to use dfs to go through each neighbor of the nodes based on the node provided. We use a Hashmap of Node key and values to keep track of already visited nodes. With this we create brand new node object based on the value and insert the incoming nodes as neighbors after they have been visited.
* The base condition is if the nodes have been visited.

1. Pacific Atlantic Water Flow:

* The approach to this problem is to visit each of the values that can be reached from the Atlantic Ocean first which would be the top to left, afterwards do the same for the pacific with right to bottom. We use DFS to traverse through each and keep a list for both oceans to check per ocean if a row and column have been visited. This along with out of bounds and if the next value is smaller than a previously tracked value in a variable will be the base condition for the recursion. We can only move up, down, left, and right. This is a time complexity of O(A X P).
* For this problem we essentially want to move up meaning that the neighboring cell would essentially meet to either the pacific/Atlantic. Kinda working backwards.

1. Course Schedule:

* This approach is tricky, We need to visit each row, within them we need to use a hash map to store if we visited the prereq along with the class it needs to take, once we visited it and check its neighbors we remove that and use the hash map as part of the base condition.
* We use DFS to continue to check that there is essentially NO LOOP in this problem.
* It’s a LOOP DETECTION problem.

1. Number of Connected Components in Graph:

* WE use DFS using a Hash Set to check if a node value has been visited or not, if not then we go into the neighboring node and continue until we hit the base condition and that is if we visited the node already and if there isn’t then it will go through each connection until there isn’t and it will add the node to the visited set.

1. Graph Valid Tree:

* Using DFS and a set, we need to check if a matrix along with n number of nodes can make up a valid graph. This will require to have a set to keep track of visited and especially checking if a valid tree can be made up. What makes a valid graph is that there is no cycle and that the graph is fully connected so there can be a separate node. They all have to be connected.

**Intervals:**

1. Insert Interval:

* Using a list of arrays to keep track of intervals along with a Boolean value to ensure whether the interval has been inserted or not. The idea to this problem is to check for each interval using a while loop and pointer to check if the new interval overlaps with an incoming interval in the established matrix. You change the new interval until conditions aren’t met and then insert into the list changing the Boolean value.
* This assuming the arrays are sorted runs in linear time, O(n). The space is also O(n) worst case since we need a list of arrays to insert all the value, we know are valid and non-overlapping.

1. Merge Interval:

* Using a pointer that will only move to traverse the array when there isn’t any overlapping intervals. We start at the beginning index +1 and move forward as our pointer variable will merge based on conditions, once there isn’t any overlapping interval then we insert it into a new matrix object with all thee non overlapping intervals.
* The runtime complexity is O(n).

1. Non-Overlapping Interval:

* The approach to this problem is almost like the merge interval problem. You use two pointers, one fast and one slow. We use the fast to determine if it overlaps with the slow and if it does then we determine where the end of the fast array lies and if its greater than the slow end we continue to move the fast pointer. With this we have to sort the arrays to get the most optimal solution.
* The runtime for this is O(n Log n) Linearithimic time complexity.
* The space is O(1).

1. Meeting Rooms:

* This questions is fairly straight forward. We need to sort the array and the just iterate to see check if the ending of each incoming index is before the start of every interval after it. If it isn’t then it is false.
* The runtime complexity of this is O(n Logn) and this is because we have to sort the intervals.

1. Meeting Rooms 2:

* With this problem we want to keep track of the number of rooms that are needed and get the minimum. We need two pointers, a max room count and a room count to keep track since it will be fluctuating to gaining or losing a room based on whether the intervals are overlapping.
* We use two arrays one to store all the start and one to store all then ends. If they overlap in anyway then we increment the room count and move the pointer for the start up and if they don’t then we decrement and move the end pointer up. We return the max room count as we find the max after every iteration.
* The runtime complexity is O(n Logn)

**Linked List:**

1. Reverse a Linked List:

* In this problem we use a while loop to iterate over the list. We use three variables, current, next, and a temp node variable so that the next can equal to the temp as some connections will be cut off as we attempt to reverse the list.
* The time complexity of this problem is based on the number of nodes we need to iterate over in the list so this is a runtime of O(n).

1. Detect Cycle in a Linked List:

* Use two pointers one slow and one fast. The fast will move two steps ahead of the slow in the list. If the fast cycle ends that means right away there isn’t a loop but if it continues that means at some point the fast and slow pointers will meet and will then mean there is a cycle.
* The runtime complexity of this is O(n).

1. Merge Two Sorted Lists:

* The approach to this problem is to have a brand new merged list object along with a head object referencing the beginning of the new merged list as it will be moving. The first value is empty so anything afterwards will be the combination of the values in both list 1 and 2. A while loop is used to iterate as long as one of them still contain values as we move towards the end of each list.
* The runtime complexity of this is O(n) with O(n) space due to the new merged list object.

1. Merge K Sorted Lists:

* The idea is to go through each linked list one by one and then create a new ll to traverse and add numbers as see fit if they are less than or equal to. We use a for loop to traverse the array of ll and a while loop to traverse the ll and new ll itself.
* The time complexity isn’t that great as my solution run in O(n^2) so try to optimize this.

1. Remove Nth Node from End of List:

* For this problem we need to traverse through the list to find the number of nodes in the list with a counter. After this we want to iterate through the list using two pointers. One that start at the head and the other that is the next. As we iterate, we want to sub the counter with the nth node value from the end and because we want the fast pointer to land on the node that need to be removed, we subtract 1 to the value as well for the loop.
* The runtime complexity is O(n).

1. Reorder List:

* The idea to this problem is to find the middle of the list. Once we find the middle, we reorder that list in reverse from the middle to the end. We contain references to both the middle and the head and reorder them using references and temp variables and iterate using a while loop for the middle till the end since the middle will remain unchanged for the head pointer. Ex. – 1-2-3-4-5-6 -> 1-2-3-6-5-4 -> 1-6-2-5-3-4. Makes sure the middle next is null afterwards. There are three parts. 1. Find the middle. 2. Reverse the middle to the end in reverse order. 3. Iterate and using temp variables add starting from both head and middle and reorder the list as the middle pointer is going towards the end for the whole to become as the example above.
* The runtime complexity is O(n). The space is O(1) since we are using the same list and not creating another.

**Strings:**

Sliding Window:

1. Longest Substring Without Repeating Characters:

* In this problem we use a hash map to keep track of the characters and their indices. We use two pointer one slow and one fast that will iterate through the string. We have to keep track of the index of both slow and fast. We have to see if they are the same character in different indices or different characters but the hash map already contains the character. We also need to keep track of the max number substring in every iteration. We also need to empty the hashmap as well after every iteration if there is a character already in the map.
* The time complexity is O(n). The space is O(n) worst case since we possibly might need to store each character from the string if there isn’t any repeating characters.

Hashing:

1. Valid Anagram:

* This problem we need to use a hashmap to keep track of the characters and the amount of it used. We iterate through one string and count its characters. We then iterate through the second string and if the same amount of characters are in the hash, then the count from the hash will decrement to 0 the ensure that there is the same amount of characters used in both strings to produce an anagram. For the second string if there is an additional character then we will add nonetheless and iterate through the hash and verify if each character count is 0 for the anagram to be true.
* The runtime complexity is O(n) for each string. The Space is O(n) since we need to keep track of each character and their count.

Stack:

1. Valid Parenthesis:

* We use a stack to store all the right facing brackets and if we get to a left facing bracket, we peek the stack and if they are equal then we pop the stack and continue iterating through the string, if they are not then it is automatically false. We keep doing this until the end and see if the stack is empty or not, it should be for it to be a valid parenthesis.
* The runtime is O(n) as we iterate through the string. The space complexity is O(n) for the stack.

Two Pointers:

1. Valid Palindrome:

* The palindrome is a word that is the same in reverse without any non alphabetical characters. For this problem we use a stack and a string builder to iterate over the string and check if there is any non-alphabetical letters, numbers count as well. We then change all the characters to lowercase and then slowly iterate through the string and add them to the string builder and push to the stack. Afterwards, we iterate through both by popping and using a pointer and check if the letters match. If they don’t its automatically false.
* The runtime complexity is O(n). The space is also O(n) since we need space for the builder and stack which will empty out anyways.

Array & Hashing:

1. Encode and Decode Strings:

* To encode For this problem we need to create a string using the length of each word in the list along with a delimiter that will identify that the number before is the number that is used to verify the length of each incoming word. We iterate though the list with lengthofword+delimiter+word + …
* To decode we need to separate the length and the delimiter, once we do that we start from the delimiter and use the length of the word we decrement and add each character to a string builder and then add that to a list. We continue through the length of the whole word that was encoded.
* The runtime complexity is O(n) for both encoding and decoding.

1. Minimum Window Substring:

* For This problem, we need to keep track of the amount of characters we have and need. While the need will hold the amount of characters that are needed to solve the problem. We need hashmaps for both words and keep track and store the minimum word with its length. We only need to traverse both words once so this will be an O(n) time complexity with an O(n) space complexity due to the two hashmaps.

**Trees:**

1. Maximum Depth of Binary Tree:

* For this problem we just need to do DFS(preorder – Root-Left-Right). We recursively go to the left until it hits null and that’s our base condition, and we add one once the condition is hit and we continue to do that and always check for the max count. We do this for both left and right that we compare.
* The runtime complexity is O(|V|) where v us the number of nodes that we will need to traverse at least once. The max depth of a tree is the umber of nodes along the longest path from the root. The space is O(h) where h is the height since recursion uses a stack.

1. Same Tree:

* We use DFS to recursively check if both the values of the nodes from both trees equal. If they don’t or either are null then false is returned. If both reach null then it is true and those two are out base condition.
* The runtime complexity is O(|V|) for the amount of nodes we need to traverse in both trees.

1. Invert Binary Tree:

* For this problem we use BFS. With a queue we add the root node and switch its left and right, then go to the left and switch its left and right etc.. and this is based on what is in the queue as long as it isn’t empty. We want to use temp variables to store both the new left and right which are going contain their opposite values. This is through iterating the queue.
* The runtime complexity is O(|V| + |E|) for the vertex and edges of the tree as we need to iterate and visit each node at least once to invert them.

1. Subtree of another Tree:

* For this problem we use a stack to store each node from the root and then left to right. As we pop we want to find the node that has the same value as the sub root. Once we find that we just use the same algorithm as the same tree from above to check if both are the same and that would confirm the sub root is a subtree of the whole tree. We have a bool value that will change to true if the sub root is part of the whole tree.
* The runtime complexity is O(|v| +|e|) . The space complexity is also O(V) for the stack that we need to use to store the nodes.

1. Binary Tree Level Order Traversal:

* The approach to this problem is to use a HashMap to store the height values of the tree with a list of integers and store values accordingly as we traverse through the tree via DFS. We recursively traverse through the tree in pre-order to store results as they come through the HashMap to verify.
* The time complexity is an O(n) as we traverse through the tree once. The space complexity is an O(n) as we store in a hashmap and return a list of those values.

1. Valid Binary Search Tree:

* For this problem of course we use DFS. We then traverse through each root passing a Long min and max value as arguments to determine if the root is between both the min and max and as we go to each children we change the min and max based on the previous value of the previous root based on whether we went left or right.
* The time complexity is O(n) as we traverse through the tree once. The space is O(1).

1. Lowest Common Ancestor of a Binary Search Tree:

* For this problem use DFS and create a parameter that stores the lca incase the requirements of the two nodes to meet as we only use less and greater than. If this isn’t the case than we pass that root since it has to equal to one of the two nodes. We search as we go left for the less and right for the greater than. We traverse through this tree once.
* The time complexity is O(log n) logarithmic as we just need to compare the root with the two numbers if its less than or greater than. The space complexity is O(1).

1. Kth Smallest Element in a BST:

* We traverse through all of the root and its left children and store them into a stack. As we pop each out we decrement the kth number that is stated in the problem to obtain our value. If the the decremented number isn’t 0 yet then we use the current value and go into its right value and check if there are any left nodes from that, and if not we just continue with popping from the stack.
* The time complexity is logarithmic O(log n) and the space complexity is O(1) just using stack and popping each value out until we have our kth value.

1. Construct Binary Tree from Preorder and In order Traversal:

* For this problem we use DFS and search via the preorder and inorder array. We use the index of the preorder to traverse each value once and we use the inorder array to find its left and right children. We pass a new root to add the values along with the arrays and preorder index start and inorder index start/end to keep track.
* The most important part of this problem is to add a leftover value by subtracting the index value in which the preordeer value and inroder value meet based on the inorder and subtract that by the inorder start.
* The time complexity is linear time O(n) with the space com,plexity of O(n) to store the tree nodes.

1. Binary Tree Maximum Path Sum:

* DFS for this problem. We go through the left and right child. As we traverse through them we want to find out the max of the root + left, root +right and root iself and that will be the new value of the root value in question and the overall max will be based on the current overall max with the root+ left+right as that can be considered a path in itself.
* This run in a linear time complexity of O(n). The space complexity is O(1).