LEETCODE PROBLEM	LEARNINGS
Problem 1: Two Sums Difficulty: Easy Topics: Array, Hash Table	Given an array of integers ***nums*** and an integer ***target***, return indices of the two numbers such that they add up to ***target.*** You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice. You can return the answer in any order. - First usage of Hashmaps instead of nested for loops to efficiently solve the problem with O(n) time complexity rather than O(n2) - Storing values in hashmap and then using the target to see whether it exists in the hashmap - The following is the first approach: `def twoSums(nums, target):` `  prevMap = {} # value:index` `  for i, num in enumerate(nums):` `  if target - num in prevMap:` `  return prevMap[target - num], i` `  else:` `  prevMap[num] = i` - The following is the second approach: `def twoSums(nums, target):` `  for i in range(len(nums)):` `  prevMap[nums[i]] = i  # value:index` `  for i in range(len(nums)):`

```
        y = target - nums[i]

        if y in prevMap and prevMap[y] != i:
            return prevMap[y], i
```

Problem 2: Palindrome
Difficulty: Easy
Topics: Math

Given an integer x, return true *if* x *is a* ***palindrome****, and* false *otherwise*.

- Brute force method was to make a forward and reverse string and compare the two (either using negative indices or the reverse function in python)
- More technical method involved not converting the integer to string and using mathematical operations like mod and division to solve the problem
- Not converting to string is beneficial because if the integer is a very big number, then converting it into string is too much of a burden on machine
- The following is the code:

```
def isPalindrome(x):
    If x < 0 or (x != 0 and x % 10 == 0)
        Return False
    Half = 0
    While half > x:
        Half  = (half * 10) + (x mod 10)
        x = x // 10
    Return half == x or half mod 10 == x
```

Problem 3: Roman to Numeral
Difficulty: Easy
Topics: Hash Table, Math, String

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

	For example, 2 is written as II in Roman numeral, just two ones added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II. . Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used: ● I can be placed before V (5) and X (10) to make 4 and 9. ● X can be placed before L (50) and C (100) to make 40 and 90. ● C can be placed before D (500) and M (1000) to make 400 and 900. Given a roman numeral, convert it to an integer. - Usage of Hashmap/Dictionaries to map values to unique characters - Usage of replace function for example Def Roman_to_numeral(s) s = s.replace(“IV”, “IIII”) s = s.replace(“IX”, VIIII”).... (for all the unique cases) Note: only make use of it when there are a few unique cases
Problem 4: Longest Common Prefix Difficulty: Easy Topic: String, Trie	Write a function to find the longest common prefix string amongst an array of strings.If there is no common prefix, return an empty string "". - Learnt how to operate vertical scanning on these problems to compare each index of the strings - A more quicker and faster way was to make use of a new Data Structure: Trie (a combination of tree and lookup table) - Just understood the concept of Trie, though did not use it in this problem - Trie is a good solution for such problems because it allows you to quickly insert, delete and search characters. One example is the auto completion of text on google search engine. `def longest_common_prefix(strs):`

```
    if len(strs) == 0:
        return ""

    base = strs[0]
    for i in range(len(base)):
        for word in strs[1:]:
            if (i == len(word) or word[i] != base[i]):
                return base[0:i]

    return base
```

Problem 5: Valid Parentheses
Difficulty: Easy
Topic: String, Stack

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

- In the brute attempt, made use of concepts from Two Sum problem and Roman to Numerical Problem. Used Hashmaps to assign each parenthesis a value (from Problem 3) and used the target variable (from Problem 1).
- The best attempt was to make use of the Last in First Out principle of Stacks. We would create a pair dictionary with the opening brackets as keys and their respective closing brackets as the values. If the brackets are in the dictionary pairs, then they are appended to the stack; otherwise, the length of the stack is checked and made sure its not zero as well as the value of popped bracket (key). If either of them are true, False is returned. At the end, if the length of the stack is zero, then return true.
- First application of stack as a data structure.

```
def isValid(s):

    stack = []
    pairs = { '(' : ')',
              '[' : ']',
              '{' : '}'}
```

```
    for brackets in s:
        if brackets in pairs: #keys
            stack.append(brackets)
        elif len(stack) == 0 or brackets != pairs[stack.pop()]:
            return False

    return len(stack) == 0
```

Problem 6: Merge Two Sorted Lists
Difficulty: Easy
Topic: Linked list, Recursion

You are given the heads of two sorted linked lists list1 and list2.

Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return *the head of the merged linked list*.

```
def mergeTwoLists(list1, list2):

    head = ListNode()
    current = head

    while list1 and list2:
        if  list1.val < list2.val:
            current.next = list1
            list1 = list1.next
        else:
            current.next = list2
            list2 = list2.next
        current = current.next

    if list1 != None:
        current.next = list1
    else:
        current.next = list2
    return head.next

-----------------------------------------------------------------------------------------------------
```

```python
def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode]) -> Optional[ListNode]:
        if not list1 or not list2:
            return list1 if list1 else list2

        # Swapping to make sure that list1 is the smallest value
        if list1.val > list2.val:
            list1, list2 = list2, list1

        list1.next = self.mergeTwoLists(list1.next, list2)
        return list1
```

- Very interesting problem I must say. Taught me how to deal with a single linked list and how we can manipulate the pointers to create a new linked list.
- Other methods of solving this problem included creating a whole new linked list based on the values given by the pointers
- Another method involved just linking the second lists nodes to the first's lists nodes by making use of, again, the pointers.
- So, for the recursive solution, this was the first time so I kind of didn't have an idea on how to work this out. I did end up getting the correct base cases again by following the example of taking the smallest possible inputs.

KEY TAKEAWAY

- Since the node class (not shown here) has public attributes, you can call them outside the class easily using variable.[attribute_name]. Always remember that!!!!
- From the recursive solution we can take a couple of very important things and concepts:
- We just again realised that if the recursive solution is looking complicated, it is most-definitely wrong
- First time using linkedlist in a recursive manner.

Problem 7: Remove Duplicates from Sorted Lists
Difficulty: Easy
Topic: Array, Two Pointers

Given an integer array nums sorted in **non-decreasing order**, remove the duplicates **in-place** such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**. Then return *the number of unique elements in* nums.

Consider the number of unique elements of nums to be k, to get accepted, you need to do the following things:

- Change the array nums such that the first k elements of nums contain the unique elements in the order they were present in nums initially. The remaining elements of nums are not important as well as the size of nums.

	• Return k. - The first method I used was when there was no restriction on creating a new list so i just made a new list and appended each unique item in the list - The second method was the usage of the pop method. In that method, I learnt to keep an eye on the length of the array as that was continuously changing so you had to be wary of as to when to increment the pointer's value - The reason pop method is not efficient is because once you pop an element, you move the whole list of next elements to the left by one and you do that for n times so that makes the whole operation $O(n^2)$ - The best method is the replace method which takes O(n). Its also the shortest and easiest to code. The following is the code: `def SortedArray(num):` `  replace = 1` `  for index in range(1, len(num)):` `  if num[index - 1] != num[index]:` `  num[replace] = num[index]` `  replace += 1` `  return replace` - Always remember to make use of two pointers and replace methods when you have to remove duplicates rather than the pop method.
Problem 8: Remove Element Difficulty: Easy Topic: Array, Two Pointers	Given an integer array nums and an integer val, remove all occurrences of val in nums **in-place**. The order of the elements may be changed. Then return *the number of elements in* nums *which are not equal to* val. Consider the number of elements in nums which are not equal to val be k, to get accepted, you need to do the following things: • Change the array nums such that the first k elements of nums contain the elements which are not equal to val. The remaining elements of nums are not important as well as the size of nums. • Return k. `def Remove_Element(nums, val):`

```
        replace = 0
        for index in range(0, len(nums)):
            if nums[index] == val:
                index += 1
            else:
                nums[replace] = nums[index]
                replace += 1

        return replace

print(Remove_Element([2,2,2,3,3,3,4,5,6], 2))
```

- Really similar to the previous question aka " Remove Duplicates from Sorted Arrays". Again, could have used another list or pop() method but those would have been inefficient and caused the code to run in the $O(n^2)$ time complexity.
- Similarly to the previous problem, made use of a replace pointer and an index pointer. The replace pointer is set to 0 in this case because the first element can be removed as well. The index is set to zero as well and incremented if the value at the particular index is equal to the value to be removed.
- If value at the particular index is not equal to the value, then the value in that particular index is replaced with the particular element where replace is pointing to.
- At the end, replace is just returned which gives the number of elements in the mutated list.

KEY TAKEAWAY
While removing an element from any list, always prefer this two pointers over pop()

Problem 9: Find the Index of the First Occurance in a String
Difficulty: Easy
Topic: Two Pointers, String, String matching

Given two strings needle and haystack, return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

- First immediate thought got me three methods:
- 1 - To use string splicing, leading to O(n)
- 2 - To use vertical scanning, which would lead to O(n) (idea from previous problems)
- 3- To use two pointers, which would lead to O(n) as well (idea from previous problems)

First method: String Splicing

```
def strStr(string_1, string_2):

    length_str_2 = len(string_2)
    first_char_str_2 = string_2[0]
```

```
    position = -1

    for index, char in enumerate(string_1):
        if char == first_char_str_2:
            spliced_str = string_1[index : index + length_str_2]
            if spliced_str == string_2:
                position = index
                return position

    return position
```

Second Method (NEW): Two Pointers and KMP Algorithm
Time Complexity : O(n) (even though it doesn't look it) (need more practice on Time Complexity Analysis)

```
def strstr(haystack, needle):
    # pre-work i-e creating LPS array (longest common prefix that is also a suffix)
    lps = [0] * len(needle)
    pre = 0
    for i in range(1, len(needle)):
        while (pre > 0 and needle[i] != needle[pre]):
            pre = lps[pre - 1]
        if needle[pre] == needle[i]:
            pre += 1
            lps[i] = pre

    n = 0 # needle index
    for h in range(len(haystack)):
        while (n > 0 and needle[n] != haystack[h]):
            n = lps[n - 1]
        if needle[n] == haystack[h]:
            n += 1
        if n == len(needle):
            return h - n + 1

    return -1
```

- Few things that I learnt from the KMP algorithm include the concept of Longest Proper Prefix that is also the Suffix (LPS). However,

don't get too fixated on this algorithm, as it is rarely used so not very useful and complicated for no reason. Though, I should know how it operates just in case.

Problem 10: Search Insert Position
Difficulty: Easy
Topic: Array, Binary Search

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with O(log n) runtime complexity.

- If we could write an algorithm of O(n) time complexity then the following is the code that I wrote:

```
def searchInsert(nums: [int], target: int):

    index = 0
    if target < nums[0]:
        return 0
    else:
        while index < len(nums) and index + 1 != len(nums):
            if nums[index] == target:
                return index
            if nums[index] < target and target < nums[index + 1]:
                return index + 1
            index += 1
    return len(nums)
```

- Indexing was definitely a big issue especially with the nums[index + 1]. If you ever face that error, try putting a condition in the loop of index + 1 not equally the length of the array.
- For O(logn) time complexity, the first algorithm that came to mind was binary search and indeed, that is what we do instead of the linear search model. Given its a sorted array, binary search would work perfectly fine.
- whenever you have to search something, and given its sorted, always use binary search over linear search!!!

```
def searchInsert(nums: [int], target: int):
    low = 0
    high = len(nums) - 1
    while low <= high:
        mid = (low + high) // 2
```

```
        if nums[mid] < target:
            low = mid + 1
        elif nums[mid] > target:
            high = mid - 1
        else:
            return mid
    return low
```

Problem 11: Length of Last Word
Difficulty: Easy
Topic:

Given a string s consisting of words and spaces, return *the length of the **last** word in the string.*
A **word** is a maximal
substring
consisting of non-space characters only.

```
def lenofLastWord(s):
    length = 0
    starting_index = len(s) - 1
    while s[starting_index] == " " and starting_index >= 0:
        starting_index -= 1
    while s[starting_index] != " " and starting_index >= 0:
        length += 1
        starting_index -= 1

    return length
```

- We could have also used the s.split() method which would have returned an array of all the characters in the string separated by the space character. Then we would just access the last element of the list using -1 index and then use the length method.
- That approach is expensive if the string is really big!!!

KEY TAKEAWAYS

- Be careful that space character is “ ” and not this “”
- Be on the lookout of what you are trying to find in the problem. If it includes the word last, try thinking of whether you can solve the problem from the end and likewise if its middle try doing the same. That could possibly help you from getting not into complexities.
- **Always always remember that the length of a string or an array or any sequence is one greater than the last index because the indexing starts from 0.**

Problem No 12: Plus One

You are given a **large integer** represented as an integer array digits, where each digits[i] is the i_{th} digit of the integer. The digits are ordered

Difficulty: Easy
Topics: Array, Math

from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's.

Increment the large integer by one and return *the resulting array of digits*.

```
def PlusOne(digits):

   current_pointer =  -1
   flag = False
   while current_pointer >= -len(digits) and flag == False:
       if digits[current_pointer] == 9:
           digits[current_pointer] = 0
           current_pointer = current_pointer - 1
       else:
           digits[current_pointer] = digits[current_pointer] + 1
           flag = True

   if digits[0] == 0:
       # digits.insert(0, 1) # to add something at some position other than the end
       digits = [1] + digits

   return digits
```

- The above code is my version of the code.
- Insert function was used for the problematic case when the first element in the digits array was 9.
- The other simpler method from an online video was:

```
def PlusOne(digits):

   for i in reversed(range(len(digits))):
       if digits[i] != 9:
           digits[i] += 1
           return digits
       digits[i] = 0

   return [1] + digits
```

- Time complexity wise it is the same. However, its better in terms of space and memory.

KEY TAKEAWAYS

- Reversed function can be used to iterate backward which is useful
- .insert is another useful function that can be used to insert any value at any index in a sequence.

Problem No 13: Add Binary
Difficulty: Easy
Topics: Math, String, Bit Manipulation, Simulation

Given two binary strings a and b, return *their sum as a binary string*.

- I had the same approach but was confused about the cases where the strings wouldn't be of equal length.

```
def addBinary(a, b):
    sumBinary = ""
    carry = 0

    a, b = a[::-1], b[::-1]
    print(a, b)
    for i in range(max(len(a), len(b))):
        digitA = int(a[i]) if i < len(a) else 0
        digitB = int(b[i]) if i < len(b) else 0

        sum = digitA + digitB + carry
        char = str(sum % 2)
        sumBinary = char + sumBinary
        carry = sum // 2
        print(carry)

    if carry:
        sumBinary = "1" + sumBinary
    return sumBinary
```

- My logic was 100% the same, I just lacked the proper coding syntax that needed to be known.
- Few very important key takeaways arose from this problem

KEY TAKEAWAYS

- If you need to run a loop whose iterations is equal to the length of the largest sequence, then use the max function
- ord() function takes a single char as an input and returns the ascii char for it.

	- If you need to reverse a sequence, instead of just using the reverse function, remember to make use of string[::-1] - This problem also taught me how to add zeros or any other value to a string when the iteration is going to be greater than the actual length of the string. i-e `digitA = int(a[i]) if i < len(a) else 0`
Problem No 14: Sqrt(x) Difficulty: Easy Topics: Math, Binary Search	Given a non-negative integer x, return *the square root of* x *rounded down to the nearest integer*. The returned integer should be **non-negative** as well. You **must not use** any built-in exponent function or operator. ● For example, do not use pow(x, 0.5) in c++ or x ** 0.5 in python. `upper = x` `lower = 1` `while lower <= upper:` `  mid = (upper + lower) // 2` `  square = mid * mid` `  if (square) == x:` `  return mid` `  elif square > x:` `  upper = mid - 1` `  else:` `  lower = mid + 1` `  return upper` - There were two main key realizations to solving this problem. The first one being that if you have to search through every integer from 1 to that number, do it using binary searching and the second realization was that in the case where the sqrt was a decimal, you just needed to spot that the answer would be the value at the upper variable. *KEY TAKEAWAYS* - If at the end of the binary search you need the last two values/indexes of the binary search, the lower value is stored in the upper variable and the higher value is stored in the lower variable.

Problem 15: Climbing Stairs
Difficulty: Easy
Topics: Dynamic Programming, Memoization, Math

You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

- The first method that I myself used was making use of math to solve the problem using permutation. The code runs in O(n) time complexity, though the space complexity is not that good.
- In each iteration, I combined 1s to form a 2 and then added the no of ways of the new space and data to the *ways* variable. I kept doing it until no more two's could be formed. However, a thing to think about is whether a question where one person could take 3 steps, could be solved using a similar strategy??

```
def climbStairs(n):

    repeats = 0
    ways = 0
    no_of_twos = n // 2

    while repeats <= no_of_twos:
        one_in_space = n - (2 * repeats)
        two_in_space = repeats
        space = n - repeats
        ways +=  int(math.factorial(space) / (math.factorial(one_in_space) * math.factorial(two_in_space)))
        repeats += 1

    return ways
```

- However, let's see how this problem is solved using recursion and dynamic programming.

```
# Recursive Way
    if n == 1:
        return 1
    if n == 2:
        return 2
    return climbStairs(n - 1) + climbStairs(n - 2)
```

- Since in the recursive case, we are calling the function with the same arguments over and over again (can be illustrated by a tree diagram), we need to be efficient and in order to be efficient, we need to make use of *Dynamic Programming*.
- Dynamic Programming is basically breaking the problem into smaller subproblems and storing the answers to each subproblem for them to be used for other subproblems. This technique is also referred to as memoization and makes the code efficient. Its basically remembering partial solutions to find the overall solution.

	- Time complexity for the Recursive way is $O(2^n)$, which is inefficient. - For dynamic programming, we have a Top-Down and Bottom-up approach. The following code is for the Bottom-up approach. `# Dynamic Programming: Bottom Up` `def climbStairs(n):` `  if n == 1:` `  return 1` `  one_before = 1` `  two_before = 1` `  total = 0` `  for i in range(2, n+1):` `  total = one_before + two_before` `  two_before = one_before` `  one_before = total` `  return total` - This is overall the best approach because not only in terms of time complexity it is O(n) but in terms of space complexity it is O(1). - Don't have to create an array to use this approach however, it's good for visualizing. *KEY TAKEAWAYS* - There is always more than one way to solve the problem. - Whenever you do recursion, see if dynamic programming can be used to make it more efficient. - This problem really portrayed that thinking and visualizing through pen and paper before implementing is the way to go.
Problem 16: Remove Duplicates from Sorted Lists Difficulty: Easy Topics: Linked List	Given the head of a sorted linked list, *delete all duplicates such that each element appears only once*. Return *the linked list **sorted** as well.* - This problem helped a lot in understanding how singly linked lists work, and how to make use of head and how a dummy node can be made when the head can be changed but was not needed for this particular problem. - There were multiple ways to solve the problem but I preferred using two pointers rather than one. For one pointer, we can watch the solution on NeetCode yt channel. `def deleteDuplicates(head):`

```
    if not head:
        return head

    previous = head
    current = head.next
    while current:
        if previous.val != current.val:
            previous = current
            current = current.next
        else:
            temp_next = current.next
            previous.next = temp_next
            current.next = None
            current = temp_next
    return head
```

- I need to learn how to call such a function since I had trouble doing that.
- Obviously there is a Node Class which is not shown here but it has val and next as its attributes.

KEY TAKEAWAYS

- If a function has nested loops doesn't mean that it is $O(n^2)$.
- Only create a dummy node if you think the value of the head is going to change.
- While current means that the function will run until the current is not equal to NONE.

Problem 17: Merge Sorted Array
Difficulty: Easy
Topics: Sorting, Two Pointers and Array

You are given two integer arrays nums1 and nums2, sorted in **non-decreasing order**, and two integers m and n, representing the number of elements in nums1 and nums2 respectively.

Merge nums1 and nums2 into a single array sorted in **non-decreasing order**.

The final sorted array should not be returned by the function, but instead be *stored inside the array* nums1. To accommodate this, nums1 has a length of m + n, where the first m elements denote the elements that should be merged, and the last n elements are set to 0 and should be ignored. nums2 has a length of n.

```
def merge(nums1, m, nums2, n):
```

```
    last = m + n - 1

    while m>0 and n>0:
        if nums1[m - 1] > nums2[n - 1]:
            nums1[last] = nums1[m - 1]
            m -= 1
        else:
            nums1[last] = nums2[n - 1]
            n -= 1
        last -= 1

    # fill num1 with leftover num2
    while n > 0:
        nums1[last] = nums2[n - 1]
        n, last = n - 1, last - 1

    return nums1
```

- Definitely was not easy given the approach I was going with. I was thinking of using two pointers, but iterating from the start rather than the end which made things really complicated.
- Need to review it again and solve the problem again and understand it properly.
- Though one thing is for certain that pointers are really imperative to arrays/lists/any sequence.

KEY TAKEAWAYS

- Making use of pointers and learning how to play with them is really important to nail lists/any sequence questions in good time complexity and good space complexity.
- Always look for the best route possible. Like in this case, it was way easier to go from the back, rather than the front.

Problem 18: Container With Most Water
Difficulty: Medium
Topics: Arrays, Two Pointers, Greedy

You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the i^{th} line are (i, 0) and (i, height[i]).

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return *the maximum amount of water a container can store*.

Notice that you may not slant the container.

```
def maxArea(self, height: List[int]) -> int:
        max_area = 0
        left_pointer = 0
        right_pointer = len(height) - 1

        while left_pointer <= right_pointer:
            area = base * min_height
            if area > max_area:
                max_area = area

            if min_height == height[left_pointer]:
            base = right_pointer - left_pointer
            min_height = min(height[left_pointer], height[right_pointer])

                left_pointer += 1
            else:
                right_pointer -= 1

        return max_area
```

- First medium problem solved without any assistance WOOHOO!!
- Did good in terms of thinking about the problem and trying to find an efficient solution
- There are few key takeaways from a problem like this. This is considered greedy because at each step, we are choosing the best option at the moment without considering whether it will lead to the best overall result. Initially, I thought of an $O(n^2)$ time complexity solution but immediately managed to find an O(n) solution which is nice.
- I really liked the way I particularly coded this problem.

KEY TAKEAWAYS

- If you encounter a runtime error saying “some [datatype] is not subscriptable, " you are probably reusing variable names for two different data_types in the question. Like in this case, I was using height, instead of min_height initially, which caused the error.
- Be careful of the indexes because I got them wrong in calculating the base first time (i-e I was adding one which is not necessary)

	- Apart from that, have trust in yourself and rock each and every single question.
Problem 19: Three Sum Difficulty: Medium Topics: Arrays, Two Pointers, Sorting	Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0. Notice that the solution set must not contain duplicate triplets. - Few Important things was the usage of *continue* inside a loop for python 3. - It is crucial that you do all the previous problems before you jump to the next ones because in this case if i would have done two sum

```python
def threeSum(self, nums: List[int]) -> List[List[int]]:

        result = []
        nums.sort()

        for i, a in enumerate(nums):
            if i > 0 and a == nums[i-1]:
                continue

            l = i + 1
            r = len(nums) - 1
            while l < r:
                ThreeSum = a + nums[l] + nums[r]
                if ThreeSum > 0:
                    r -= 1
                elif ThreeSum < 0:
                    l += 1
                else:
                    nums.append([a, nums[l], nums[r]])
                    l += 1
                    while nums[l] == nums[l - 1] and l < r:
                        l += 1

        return result
```

	ll problems, I would have been easily able to solve it. - Sometimes the problem can be solved by dividing it into smaller subproblems that you already know of. *KEY TAKEAWAYS* *- Do all the corresponding easy problems before jumping onto the medium version, because that makes things a lot simpler.* *- Problems can be solved by dividing a complex problem into subproblems and solving the subproblems using patterns/problems you have already solved. So, it is really imperative that you continuously go through this doc and revise what you have learnt so far.*
Problem 20: Combinations Difficulty: Medium Topics: Backtracking	Given two integers n and k, return *all possible combinations of* k *numbers chosen from the range* [1, n]. You may return the answer in **any order**. My failed attempt to solving it without backtracking: `def combine(self, n: int, k: int) -> List[List[int]]:` `  combinations = []` `  last_val = n` `  if last_val == 1 and k == 1:` `  combinations.append([1])` `  while last_val > 1:` `  for j in range(1, last_val):` `  first_combo = []` `  first_combo.append(last_val)` `  first_combo.append(last_val - j)` `  if len(first_combo) == k:` `  combinations.append(first_combo)` `  last_val -= 1` `  return combinations`

- I think it can be solved without backtracking, and I will try solving it later again. However, it is not bad to learn the new concept of backtracking.
- Following is the code for backtracking:

```
def combine(self, n: int, k: int) -> List[List[int]]:
        res = []

        def backtrack(start, comb):
            # base case
            if len(comb) == k:
                res.append(comb.copy())
                return

            # data tree
            for i in range(start, n + 1):
                comb.append(i)
                backtrack(i+1, comb)
                comb.pop()

        backtrack(1, [])
        return res
```

- The idea of backtracking is interesting. It involves a base case and then a data tree. It is definitely a brute force approach and the time complexity is $O(k*n^k)$ i-e $O(branches^{depth})$
- Also don't forget to call the function

KEY TAKEAWAYS

- Backtracking is a useful strategy to solve combinatorics problems.
- Backtracking is just recursion but where you make decisions and undo them. Also use them once you are being asked for “ALL” possible values/solutions.

Problem 21: Permutations
Difficulty: Medium
Topics: Backtracking

Given an array nums of distinct integers, return all the possible permutations. You can return the answer in **any order**.

```
def permute(self, nums: List[int]) -> List[List[int]]:
        permutations = []
```

```
        def backtrack(perm):
            # base case
            if len(perm) == len(nums):
                permutations.append(perm[:])
                return

            # data tree
            for i in range(len(nums)):
                if nums[i] in perm:
                    continue
                perm.append(nums[i])
                backtrack(perm)
                perm.pop()

        backtrack([])
        return permutations
```

- Literally same as combinations in many aspects. The concept of backtracking is very important and should be understood. I, now, think I understand it.
- There are many ways to solve this problem. The one I particularly liked was the one from the Neetcode. Following is the solution:

```
def permute(self, nums: List[int]) -> List[List[int]]:
        if len(nums) == 0:
            return [[]]

        perms = self.permute(nums[1:])
        res = []
        for p in perms:
            for i in range(len(p) + 1):
                p_copy = p.copy()
                p_copy.insert(i, nums[0])
                res.append(p_copy)
        return res
```

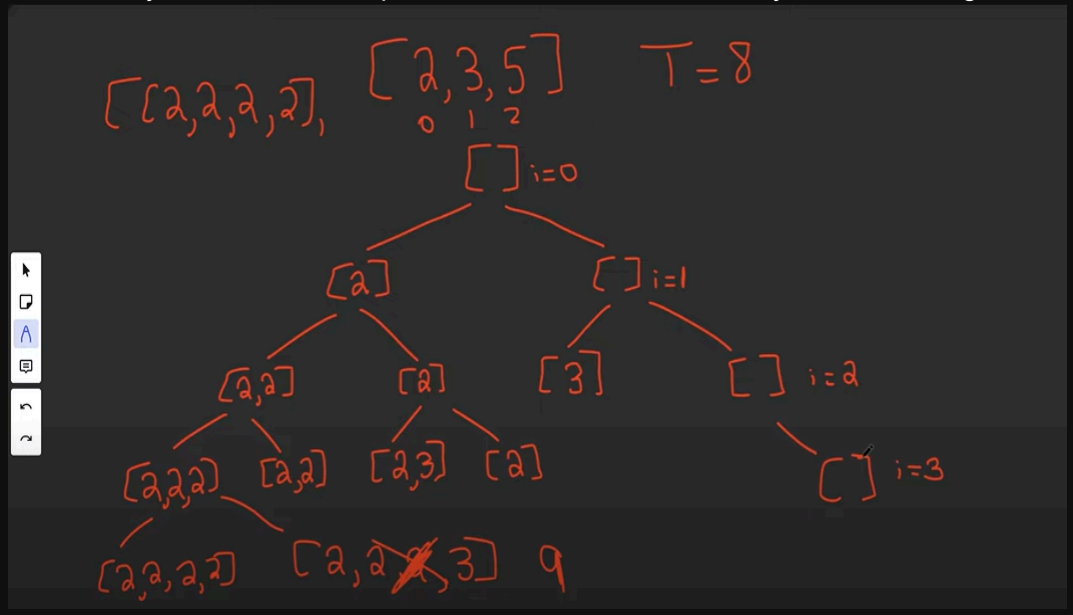
- Can be solved without recursion using the method in the above recursive solution.

	- Instead of creating a tree, you solve it using one branch. The time complexity and space complexity ends up being n! *KEY TAKEAWAYS* - *In python, [:] is the same as .copy()* - *There are many different ways of solving these combinatorial problems but it would be good if you just pick one and roll with it.* - *Backtracking and the other method described above seem to be good choices to be familiar with for permutation questions.*
Problem 22: Combinations Sum Difficulty: Medium Topics: Backtracking	Given an array of **distinct** integers candidates and a target integer target, return *a list of all **unique combinations** of* candidates *where the chosen numbers sum to* target. You may return the combinations in **any order**. The **same** number may be chosen from candidates an **unlimited number of times**. Two combinations are unique if the frequency of at least one of the chosen numbers is different. The test cases are generated such that the number of unique combinations that sum up to target is less than 150 combinations for the given input. `def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:` `  res = []` `  addition = 0` `  def backtrack(comb, start):` `  # base case` `  if sum(comb) == target:` `  res.append(comb[:])` `  return` `  if sum(comb) > target:` `  return` `  # data tree` `  for i in range(start, len(candidates)):` `  comb.append(candidates[i])` `  backtrack(comb, i)` `  comb.pop()`

```
        backtrack([], 0)
        return res
```

Time Complexity: $O(n * 2^T * T)$ where n is no of candidates and T is target, Space Complexity: $O(2^T * T)$

- I devised this solution using knowledge from the previous combination question. This is the same approach of making a decision tree.
- However, there are many duplicates involved in my code
- Another way to avoid those duplicates and do it more efficiently is the following:



- The following is the code for the following: (pretty easy to understand if the above picture is used as a reference)

```
def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        res, sol = [], []
        nums = candidates
        n = len(nums)

        def backtrack(i, curr_sum):
            if curr_sum == target:
```

```
                res.append(sol[:])
                return

            if curr_sum > target or i == n:
                return

            backtrack(i+1, curr_sum)

            sol.append(nums[i])
            backtrack(i, curr_sum + nums[i])
            sol.pop()

        backtrack(0, 0)
        return res
```

Time Complexity: $O(n^t)$, Space Complexity: O(n)

KEY TAKEAWAYS

- *So the two strategies I have liked have been used in both of the two problems to solve the problem. Make sure you keep on practicing these type of questions so that you become a master at these problems*

Problem 23: Triangle
Difficulty: Medium
Topics: Arrays, Recursion, Dynamic Programming

Given a triangle array, return *the minimum path sum from top to bottom*.

For each step, you may move to an adjacent number of the row below. More formally, if you are on index i on the current row, you may move to either index i or index i + 1 on the next row.

```
def minimumTotal(self, triangle: List[List[int]]) -> int:

        dp = [0] * (len(triangle) + 1)

        for row in triangle[::-1]:
            for i, n in enumerate(row):
                dp[i] = n + min(dp[i], dp[i + 1])

        return dp[0]
```

Time Complexity: $O(n^2)$, Space Complexity: O(n)

	- A really interesting, what initially seems like a dfs problem which it is, but in order to be more space efficient, it has to be done through dynamic programming, more specifically bottom up approach. - We start with an empty array and keep building it up as we move up the tree. *KEY TAKEAWAYS* - One thing that I am noticing is that if we want to be efficient while doing recursion, memoization and dynamic programming is usually the approach.
Problem 24: Permutation II Difficulty: Medium Topics: Backtracking	Given a collection of numbers, nums, that might contain duplicates, return *all possible unique permutations* ***in any order***. `def permuteUnique(self, nums: List[int]) -> List[List[int]]:` `permutations = []` `perm = []` `count = {n:0 for n in nums}` `for n in nums:` `count[n] += 1` `def backtrack():` `# base case` `if len(perm) == len(nums):` `permutations.append(perm[:])` `return` `# data tree` `for n in count:` `if count[n] > 0:` `perm.append(n)` `count[n] -= 1` `backtrack()` `count[n] += 1` `perm.pop()` `backtrack()`

```
        return permutations
```

- I must say I did not expect the usage of the hashmaps even though I knew what to do. The question one should ask here is why did we use hashmaps? How would you have identified using hashmap?
- To answer the above question, we identified that we have repeating numbers and we will be using each number once but the count will matter as we will need to keep updating that count. However, if we leave in form of arrays, we cannot manipulate the count in such an easy way while not even considering them at the same time.

KEY TAKEAWAYS

- *Don't get restricted by previous problems. If there is a need to make changes i-e add new data structures, do it. Have confidence in yourself!!*
- *Hashmaps are extremely useful if you have information in various pieces and you want to combine that information in a more ordered form.*
- *Do more problems on hashmaps to identify its usages.*

Problem 25: Best Time to Buy and Sell Stock
Difficulty: Easy
Topics: Arrays, Dynamic Programming

You are given an array of prices where prices[i] is the price of a given stock on the i^{th} day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

```
def maxProfit(self, prices: List[int]) -> int:
        max_profit = 0
        buy_pointer = 0
        sell_pointer = 1

        while sell_pointer < len(prices):
            if prices[sell_pointer] < prices[buy_pointer]:
                temp = sell_pointer
                buy_pointer = sell_pointer
                sell_pointer = temp + 1
            else:
                profit = prices[sell_pointer] - prices[buy_pointer]
                if profit > max_profit:
                    max_profit = profit
                sell_pointer += 1
```

```
        return max_profit
```

Time Complexity: O(n), Space Complexity: O(1)

- This is the first method that came to my mind and honestly it works really nice!. It makes use of two pointers and hence allows checking all possible conditions in one complete iteration.
- The method that I used is known famously as the “Sliding Window” approach.

KEY TAKEAWAYS

- *Always be on the lookout for two pointers in array problems.*
- *If two pointers don't seem to solve the problem, it is probably but not necesarily recursion or dp related.*

Problem 26: Path Sum
Difficulty: Easy
Topics: Trees, BFS, DFS

Given the root of a binary tree and an integer targetSum, return true if the tree has a **root-to-leaf** path such that adding up all the values along the path equals targetSum.

A **leaf** is a node with no children.

```
def hasPathSum(self, root: Optional[TreeNode], targetSum: int) -> bool:

        def dfs(node, sum):
            if not node:
                return False

            sum += node.val

            # Base Case
            if node.left == None and node.right == None:
                return sum == targetSum

            # Recursive Case
            return dfs(node.left, sum) or dfs(node.right, sum)

        return dfs(root, 0)
```

- The first ever tree problem solved!! Woohoo!!!. Getting a hang of recursion now and it does not seem that difficult now. Moreover, now have a basic understanding of how to deal with dfs.

	- Now I know how to do the traversal. Also the base case setup was nice. - Also just a side note but there was indeed a node class which is not shown here. - Also sum it before the recursive case because the last node's sum would be missed as the base case would be reached. *KEY TAKEAWAYS* - *Important to check whether there are nodes in the tree or not as the first condition before the base case.* - *Do all the three traversals which are very easy. I-e Inorder (LRootR), Pre-order(RootLR), Post order(LRRoot)*
Problem 27: Excel Sheet Column Title Difficulty: Easy (Cap) Topics: String, Math	Given an integer columnNumber, return *its corresponding column title as it appears in an Excel sheet.* `def convertToTitle(self, columnNumber: int) -> str:` `  res = ""` `  while columnNumber > 0:` `  offset = (columnNumber - 1) % 26` `  res += chr(ord('A') + offset)` `  columnNumber = (columnNumber - 1) // 26` `  return res[::-1]` Time Complexity: O(log(n)), Space Complexity: O(n) - Despite it being termed as easy and its small solution, it is a really challenging problem. - The main premise of the problem is to convert a base 10 system to a base 26 system. - The issue also comes because it starts off from 1 rather than 0. - Also, there was no reason to create a hashmap. *KEY TAKEAWAYS* - *Don't use stuff when not necessary.* - *Now we have an idea on how to deal with base 10 and any other base related questions.*
Problem 28: Inorder Root Transversal Difficulty: Easy Topics: Binary Tree, Recursion	Given the root of a binary tree, return *the inorder traversal of its nodes' values.* `def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:` `  path = []` `  def inorder(root):`

```
            if not root:
                return

            inorder(root.left)
            path.append(root.val)
            inorder(root.right)

        inorder(root)
        return path
```

Time Complexity: O(n), Space Complexity: O(n)

- Left Root Right. I think that is enough to say for this.

Problem 29: Word Search
Difficulty: Medium
Topics: Matrix, Backtracking, Recursion, Array, String, DFS

Given an m x n grid of characters board and a string word, return true *if the word exists in the grid*.

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

```
def exist(self, board: List[List[str]], word: str) -> bool:
        rows = len(board)
        columns = len(board[0])
        path = set()

        def backtrack(r, c, i):
            # Base Cases
            if i == len(word):
                return True
            if (r < 0 or c < 0 or
                r >= rows or c >= columns or
                word[i] != board[r][c] or
                (r, c) in path):
                return False

            # recursive case
            path.add((r,c))
            res = (backtrack(r+1, c, i+1) or
                   backtrack(r-1, c, i+1) or
```

```
                    backtrack(r, c+1, i+1) or
                    backtrack(r, c-1, i+1))
            path.remove((r,c))
            return res

        for r in range(rows):
            for c in range(columns):
                if backtrack(r, c, 0):
                    return True

        return False
```

Time Complexity: O(n * m * 4^len(word)) Space Complexity:

- A really really interesting and important problem. There are few patterns and templates we can learn from this problem and it also solidified my juvenile recursion problem solving.
- Remember the template of this problem because we will find many similar problems and identifying will help us to solve them quicker. Plus remember the base cases as for matrix problems few of the base cases especially the ones related to being inside the boundaries will remain the same.

KEY TAKEAWAYS

- Take recursion step by step!!! Don't hasten otherwise you will get lost in the abyss. First make a template by calling the function at the end and then add the base cases and the recursive cases
- Remember this template for matrix/dfs problems. It is going to be extremely helpful in a lot of problems and it's a useful thing to know of in general.
- First instance where we are calling the backtrack function on every single element of the matrix.
- Instead of the set, we could have set the used element with a special character like # as well but yes set is the preferred choice.

Problem 30: Longest Substring Without Repeating Characters.
Difficulty: Medium
Topics: String, Sliding Window/Two Pointers, Hashmap/Set

Given a string s, find the length of the **longest substring** without duplicate characters.

```
    def lengthOfLongestSubstring(self, s: str) -> int:
        longest_length = 0
        l = 0
        charSet = set()

        for r in range(len(s)):
            while s[r] in charSet:
```

```
                charSet.remove(s[l])
                l += 1
            charSet.add(s[r])
            longest_length = max(longest_length, r - l + 1)

        return longest_length
```

- I knew what to do in this problem but something that really caught me confused was the usage of the hashmap. If I would have tried my method using an array or a set, I would have been able to solve it myself.
- The key thing here to learn is that a sliding window is more than just what I initially thought it to be.
- Also just because the topic of the problem says one thing, don't be swayed by it and do what seems right to you.

Problem 31: Longest Palindrome Substring
Difficulty: Medium
Topics: String

Given a string s, return *the longest palindromic substring* in s.

```
def longestPalindrome(self, s: str) -> str:
        longest_substring = ""
        l = 0
        r = len(s) - 1
        f_char_idx = 0 #
        l_char_idx = 0 #
        move_left = True

        if len(s) == 1:
            return s[0]

        while l <= r:
            if s[l] == s[r]:
                if f_char_idx == 0 and l_char_idx == 0:
                    f_char_idx = l
                    l_char_idx = r
                l += 1
                r -= 1
            else:
                f_char_idx = 0
```

```
                l_char_idx = 0
                if move_left:
                    l += 1
                    move_left = False
                else:
                    r -= 1
                    move_left = True

        if f_char_idx != 0 or l_char_idx != 0:
            curr_substring = s[f_char_idx : l_char_idx + 1]
            if len(curr_substring) > len(longest_substring):
                longest_substring = curr_substring

        return longest_substring
```

- The above approach is the one I was going off with and managed to pass half of the test cases. The only problem was the cases where incrementing the left pointer first caused an issue.
- The second approach (below) is from neetcode and he starts at the middle for each string char and expands outwards.

```
def longestPalindrome(self, s: str) -> str:
        res = []
        resLen = len(res)

        for i in range(len(s)):
            l, r = i, i # for odd length
            while l >= 0 and r < len(s) and s[l] == s[r]:
                if (r - l + 1) > resLen:
                    res = s[l : r + 1]
                    resLen = (r - l + 1)
                l -= 1
                r += 1

            l, r = i, i + 1 # for even length
            while l >= 0 and r < len(s) and s[l] == s[r]:
                if (r - l + 1) > resLen:
                    res = s[l : r + 1]
                    resLen = (r - l + 1)
```

```
                l -= 1
                r += 1

        return res
```

KEY TAKEAWAYS

- *One other tactic leant. We can start off in the middle and expand both ways.*
- *Also learnt how to alternate switching between incrementing between two pointers for each iteration.*

Problem 32: Ransom Note
Difficulty: Easy
Topics: Hashmaps, String, Counting

Given two strings ransomNote and magazine, return true *if* ransomNote *can be constructed by using the letters from* magazine *and* false *otherwise*.

Each letter in the magazine can only be used once in ransomNote.

```
def canConstruct(self, ransomNote: str, magazine: str) -> bool:

        magazine_map = {} # value : count
        for s in magazine:
            if s in magazine_map:
                magazine_map[s] += 1
            else:
                magazine_map[s] = 1


        for r in ransomNote:
            if r not in magazine_map:
                return False
            else:
                magazine_map[r] -= 1
                if magazine_map[r] == 0:
                    del magazine_map[r]

        return True
```

	Time Complexity: O(n+m), Space Complexity: O(1) Since the maximum number of keys can be 26 *KEY TAKEAWAYS* - So, the dictionary making part can also be done using default dict and giving it the parameter “int”. That would make the code have less lines. Also, it can be written in one line by making use of Counter() which we can import, along with defaultdict, from collections.
Problem 33: Isomorphic Strings Difficulty: Easy Topics: Hashtable, String	Given two strings s and t, *determine if they are isomorphic*. Two strings s and t are isomorphic if the characters in s can be replaced to get t. All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself. `def isIsomorphic(self, s: str, t: str) -> bool:` `mappings = {} # str in s : str in t` `values = []` `keys = []` `if len(s) != len(t):` `return False` `else:` `for i in range(len(s)):` `if s[i] in mappings:` `if mappings[s[i]] != t[i]:` `return False` `else:` `mappings[s[i]] = t[i]` `for k, v in mappings.items():` `if k not in keys and v in values:` `return False` `else:` `keys.append(k)` `values.append(v)`

```
        return True
```

- Instead of going through the key value pairs using mappings.items(), we could have also used the following line:
- However, the time complexity and space complexity completely differs in both cases.

```
if t[i] in mappings.values() and s[i] not in mappings:

                    return False
```

KEY TAKEAWAYS

- *Whenever the question feels like a function, think about the mappings and hence hashmaps.*
- *The time complexity is way worse for mapping.values() than for the mappings.items().*
- *If you need both keys and values, using map.values() forces you to find corresponding keys manually, which is inefficient and hence map.items() should be used.*

Problem 34: Word Pattern
Difficulty: Easy
Topics: Hashtable, String

Given a pattern and a string s, find if s follows the same pattern.

Here follows means a full match, such that there is a bijection between a letter in pattern and a non-empty word in s. Specifically:

Each letter in pattern maps to exactly one unique word in s.

Each unique word in s maps to exactly one letter in pattern.

No two letters map to the same word, and no two words map to the same letter.

```
def wordPattern(self, pattern: str, s: str) -> bool:
        char_s = s.split(" ")
        mappings = {}
        keys = []
        values = []

        if len(pattern) != len(char_s):
            return False
```

```
        for i in range(len(pattern)):
            if pattern[i] in mappings and mappings[pattern[i]] != char_s[i]:
                return False
            else:
                mappings[pattern[i]] = char_s[i]

        for k, v in mappings.items():
            if k not in keys and v in values:
                return False
            else:
                keys.append(k)
                values.append(v)

        return True
```

Second approach using two different mappings (hashmaps):

```
def wordPattern(self, pattern: str, s: str) -> bool:
        char_s = s.split(" ")
        p_map_s = {}
        s_map_p = {}

        if len(pattern) != len(char_s):
            return False

        for i in range(len(pattern)):
            if ((pattern[i] in p_map_s and p_map_s[pattern[i]] != char_s[i])
                or (char_s[i] in s_map_p and s_map_p[char_s[i]] != pattern[i])):
                return False
            else:
                p_map_s[pattern[i]] = char_s[i]
                s_map_p[char_s[i]] = pattern[i]

        return True
```

- Pretty similar to the question above. The only difference is that the lengths can be different in this one and also instead of 2 continuous string sequences, one of the strings sequence is not continuous i-e it has spaces in between so .split() needs to be used

to form a list that can be iterated over properly.

Problem 35: Valid Anagrams
Difficulty: Easy
Topics: Hashmaps, String, Counting

Given two strings s and t, return true if t is an anagram of s, and false otherwise.

```
from collections import Counter
    def isAnagram(self, s: str, t: str) -> bool:
        # Adding all elements of s in a hashtable that acts as a counter
        counter = Counter(s)

        # Checking the possibility of an anagram
        if len(s) > len(t):
            return False

        for each_char in t:
            if each_char not in counter:
                return False
            else:
                if counter[each_char] == 1:
                    del counter[each_char]
                else:
                    counter[each_char] -= 1

        return True
```

- Ditto copy of the ransom question so no key takeways.

Problem 36: Group Anagrams
Difficulty: Medium
Topics:

Given an array of strings strs, group the anagrams together. You can return the answer in **any order**.

```
from collections import defaultdict

    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        res = defaultdict(list) # mappings charCount to list of anagrams

        for s in strs:
            count = [0] * 26 # a -> z
```

```
            for c in s:
                count[ord(c) - ord("a")] += 1

            res[tuple(count)].append(s)

        return list(res.values())
```

Time Complexity: O(m * n)

- Really interesting problem. What I was trying to do is reuse the previous problem, however that was me being just lazy.
- An important thing to take is the ord(c) – ord("a") and how it gives us the correct index of the char in the alphabets

KEY TAKEAWAYS

- *Hashmaps are very versatile and dont be scared to make values other than plain integers.*
- *ord() function returns the ascii value of the characters so be wary of that*
- *sort() could also be used to solve for anagrams and its time complexity is O(log(n))*
- *In python, keys have to be immutable so no lists. That's why we converted it to a tuple.*

Problem 37: Happy Number
Difficulty: Easy
Topics: Sets, Two Pointers, LinkedList Cycle

Write an algorithm to determine if a number n is happy.

A **happy number** is a number defined by the following process:

- Starting with any positive integer, replace the number by the sum of the squares of its digits.
- Repeat the process until the number equals 1 (where it will stay), or it **loops endlessly in a cycle** which does not include 1.
- Those numbers for which this process **ends in 1** are happy.

Return true *if* n *is a happy number, and* false *if not*.

```
def isHappy(self, n: int) -> bool:
        seen = set()

        while n not in seen:
            seen.add(n)
            n = self.sumofSquares(n)
```

```
            if n == 1:
                return True

        return False

def sumofSquares(self, n: int) -> int:
        sum = 0

        while n:
            digit = n % 10
            digit = digit ** 2
            sum +=  digit
            n = n // 10

        return sum
```

- Even Though an easy question, I learnt a lot of important stuff from this question.
- First thing was that don't get swayed by reading the topics of the question.
- Even Though this is termed as a hashset question, it is a linked list cycle (circular linked list) question by heart
- It showcase important skill of breaking down integer numbers into digits without having to convert them to strings
- Also, this is the first question where a helper function in leetcode is used so that was kinda fun.

KEY TAKEAWAYS

- If you have to break down an integer into its digit, mod by 10 will give you the last digit and then divide the number by 10 to get the next subsequent digit.

Problem 38: Contain Duplicates II
Difficulty: Easy
Topics: Array, Hashtable, Sliding Window

Given an integer array nums and an integer k, return true *if there are two* ***distinct indices*** i *and* j *in the array such that* nums[i] == nums[j] *and* abs(i - j) <= k.

```
def containsNearbyDuplicate(self, nums: List[int], k: int) -> bool:
        map = {} # val : index

        for i in range(len(nums)):
            if nums[i] in map:
                index = map[nums[i]]
```

```
                if abs(index - i) <= k:
                    return True
                else:
                    map[nums[i]] = i
            else:
                map[nums[i]] = i

        return False
```

- Pretty simple problem. Just realize that a sliding window was not the more efficient choice here because of various reasons.

Problem 39: Linked List Cycle
Difficulty: Easy
Topics: Linked List, Hashmap, Sets, Two Pointers

Given head, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. **Note that pos is not passed as a parameter**.

Return true *if there is a cycle in the linked list*. Otherwise, return false.

```
def hasCycle(self, head: Optional[ListNode]) -> bool:
        visited = set()
        current = head

        while current:
            if current in visited:
                return True
            else:
                visited.add(current)
                current = current.next

        return False
```

KEY TAKEAWAYS

- Whenever you think about checking whether something exists or not, make use of sets. We have indeed seen this in multiple

algorithms such as dfs, bfs, kruskal's, prims where we don't want to go to the nodes we have already seen/checked.

Problem 40: Reverse Linked List
Difficulty: Easy
Topics: Linked List, Recursion

Given the head of a singly linked list, reverse the list, and return *the reversed list*.

```
def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        curr = head
        prev = None

        while curr:
            temp = curr.next
            curr.next = prev
            prev = curr
            curr = temp

        return prev
```

```
def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        if not head or not head.next:
            return head

        # Reverse the rest of the list
        revHead = self.reverseList(head.next)

        # Make the current head the last node
        head.next.next = head

        # Update the next of current head to None
        head.next = None

        # Return the new head of the reversed list
        return revHead
```

- Pretty easy problem but visualizing on paper pen was really imperative.
- For the recursive solution, it was quite hard for me to get it and I actually didn't get it. The base case thought process worked out fine where I take the smallest possible input and see what the result would be in that case for the specific function i-e what would be the reverse of a empty list or a list with just one node?. However, the recursive case's first step I got but the rest I couldnt work out a way to shift the head around. Ig I just didn't think about head.next.next being a thing.

Problem 41: Best Time to Buy and Sell Stocks II
Difficulty: Medium
Topics: Arrays, Greedy, Dynamic Programming

You are given an integer array prices where prices[i] is the price of a given stock on the ith day.

On each day, you may decide to buy and/or sell the stock. You can only hold **at most one** share of the stock at any time. However, you can buy it then immediately sell it on the **same day**.

Find and return *the **maximum** profit you can achieve*.

```
def maxProfit(self, prices: List[int]) -> int:
        max_profit = 0
        s = 1
        b = 0

        while s < len(prices):
            if prices[s] < prices[b]:
                b = s
                s += 1
            else:
                profit = prices[s] - prices[b]
                max_profit += profit
                b = s
                s += 1

        return max_profit
```

TIme Complexity: O(n), Space Complexity: O(1)

	- Ngl pretty easy (as easy as the easy version of this problem i-e the first part Another slightly better solution in terms of time complexity and simpler solution: `def maxProfit(self, prices: List[int]) -> int:` `max_profit = 0` `for i in range(len(prices)-1):` `if prices[i+1]> prices[i]:` `profit = prices[i+1] - prices[i]` `max_profit += profit` `return max_profit` *KEY TAKEAWAYS* - *This is a prime example of a greedy algorithm where in each step, we choose the best possible outcome.*
Problem 42: Gas Station Difficulty: Medium Topics: Arrays, Greedy	There are n gas stations along a circular route, where the amount of gas at the ith station is gas[i]. You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from the ith station to its next (i + 1)th station. You begin the journey with an empty tank at one of the gas stations. Given two integer arrays gas and cost, return *the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return* -1. If there exists a solution, it is **guaranteed** to be **unique**. `def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:` `if sum(gas) < sum(cost):` `return -1` `total = 0`

```
        res = 0
        start = 0
        for i in range(len(gas)):
            total += (gas[i] - cost[i])

            if total < 0:
                total = 0
                start = i + 1

        return start
```

Time Complexity: O(n), Space Complexity: O(1)

KEY TAKEAWAYS

Problem 43: Summary Ranges
Difficulty: Easy
Topics: Intervals, Arrays

You are given a **sorted unique** integer array nums.

A **range** [a,b] is the set of all integers from a to b (inclusive).

Return *the **smallest sorted** list of ranges that **cover all the numbers in the array exactly***. That is, each element of nums is covered by exactly one of the ranges, and there is no integer x such that x is in one of the ranges but not in nums.

Each range [a,b] in the list should be output as:

- "a->b" if a != b
- "a" if a == b

```
def summaryRanges(self, nums: List[int]) -> List[str]:
        l = 0
        r = 1
        ranges = []

        while r < len(nums):
            if nums[r] != nums[r-1] + 1:
                interval = len(nums[l : r])
                if interval == 1:
                    ranges.append(str(nums[l]))
                    l = r
```

```
                else:
                    ranges.append(str(nums[l]) + "->" + str(nums[r-1]))
                    l = r
            r += 1

        if len(nums[l:r]) == 1:
            ranges.append(str(nums[l]))
        elif len(nums[l:r]) > 1:
            ranges.append(str(nums[l]) + "->" + str(nums[r-1]))

        return ranges
```

Problem 44: Maximum Depth of Binary Tree
Difficulty: Easy
Topics: Binary Tree, DFS, BFS

**** IMPORTANT ****

Given the root of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

Method1: Recursive DFS

```
def maxDepth(self, root: Optional[TreeNode]) -> int:
        if not root:
            return 0

        return 1 + max(self.maxDepth(root.left), self.maxDepth(root.right))
```

Method2: Iterative BFS

```
def maxDepth(self, root: Optional[TreeNode]) -> int:
        if not root:
            return 0

        level = 0
        q = deque([root])

        while q:
```

```
            for _ in range(len(q)):
                node = q.popleft()
                if node.left:
                    q.append(node.left)
                if node.right:
                    q.append(node.right)

            level += 1

        return level
```

Method 3: Iterative DFS

```
def maxDepth(self, root: Optional[TreeNode]) -> int:
        stack = [[root, 1]]
        res = 0

        while stack:
            node, depth = stack.pop()

            if node:
                res = max(res, depth)
                stack.append([node.left, depth + 1])
                stack.append([node.right, depth + 1])

        return res
```

Time Complexity: O(n) and Space Complexity: O(n) (for all three methods)

KEY TAKEAWAYS / PATTERNS:

- *Recursive DFS is the go to for binary tree problems as I have seen.*
- *THINK OF EACH NODE AS A BINARY TREE IN ITSELF*
- *Implementation using queue and stack is not bad*
- *Queue and stack can be implemented using a list. In python, there is not a specific inbuilt library for them.*

	- *Remember the base case and the recursive case always.* - *This problem is the core of binary trees! It must be understood to its depths.* - *If not node and If node == NONE is the same exie base case*
Problem 45: Same Tree Difficulty: Easy Topics: Binary Tree	Given the roots of two binary trees p and q, write a function to check if they are the same or not. Two binary trees are considered the same if they are structurally identical, and the nodes have the same value. `def isSameTree(self, p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:` `# Base case: both nodes are None (identical trees)` `if not p and not q:` `return True` `# If one is None and the other is not, or values are different, return False` `if not p or not q or p.val != q.val:` `return False` `# recursive case` `return self.isSameTree(p.left, q.left) (wrong)` `return self.isSameTree(p.right, q.right)(wrong)` `return self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right) (correct)` Time Complexity: O(p + q) , Space Complexity: - I was really close to solving it and I knew what to do. Again, I made mistakes in the implementation. I had the logic but did not know how to implement that logic properly. - Do a proper post mortem of this problem because these problems are the building blocks of tree problems and if I dont understand these fully, then there is no point in moving ahead. *KEY TAKEAWAYS / PATTERNS:* - *Recursive DFS again*

	- *There was one important misunderstanding that was cleared. You don't have to think of the right and left subtree as separate.*
Problem 46: Invert a Binary Tree Difficulty: Easy Topics: Binary Tree, Recursion, DFS	Given the root of a binary tree, invert the tree, and return *its root.* `def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:` `  if not root:` `  return None` `  root.left, root.right = root.right, root.left` `  self.invertTree(root.left)` `  self.invertTree(root.right)` `  return root` Time Complexity: O(n), Space Complexity: O(n) (worst case) and O(log(n)) (best case) *** Learn how to analyze these time complexities *** *KEY TAKEAWAYS / PATTERNS:* - Recursive DFS again works. - Getting a hang of the binary trees and recursion - Tip from Adam (the US one): If you are writing a recursion code and it is getting complicated, you are probably doing something wrong.
Problem 47: Symmetric Tree Difficulty: Easy Topics: Binary Tree, Recursion, DFS	Given the root of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center). `def isSymmetric(self, root: Optional[TreeNode]) -> bool:` `  def dfs(left, right):` `  if not left and not right:` `  return True` `  if not left or not right:` `  return False` `  if left.val != right.val:` `  return False` `  return dfs(left.left, right.right) and dfs(left.right, right.left)`

```
        return dfs(root.left, root.right)
```

Time Complexity: O(n), Space Complexity: O(h) where h is the depth of the binary tree

- There are few important things to notice here. Firstly, we don't need to compare the root as the root itself will be symmetric with itself.
- Secondly, we knew that we have to run dfs on the left and right side so there were two parameters needed and hence since our primary function was talking only one parameter, we needed to create a new helper function.
- As thought, we are running dfs on both the right and left subtrees and comparing the left.left with right.right as well as left.right with right.left.

KEY TAKEAWAYS / PATTERNS:

- Again DFS and Recursion
- Now we know what to do if we want to simultaneously compare the left and right subtrees of the binary tree.

Problem 48: Average of Levels in Binary Tree
Difficulty: Easy
Topics: Binary Tree, BFS

Given the root of a binary tree, return *the average value of the nodes on each level in the form of an array*. Answers within 10^{-5} of the actual answer will be accepted.

```
def averageOfLevels(self, root: Optional[TreeNode]) -> List[float]:

        if not root:
            return []

        average = []
        q = [root]
        sum = 0

        while q:
            length_q = len(q)
            for i in range(length_q):
                node = q.pop(0)
                sum += node.val
                if node.left:
                    q.append(node.left)
                if node.right:
```

```
                    q.append(node.right)

            average.append(sum/length_q)
            sum = 0

        return average
```

Time Complexity: O(n), Space Complexity: O(n)

- We could have used deque as well by importing it from collections
- A binary tree problem where it is almost kinda necessary to just use BFS rather than DFS
- Learned how to make use of the length of the queue to calculate the total number of nodes in a particular level.

KEY TAKEAWAYS / PATTERNS:

- Calculating the number of nodes in a particular level in BFS using length of the queue
- Where to prefer BFS over DFS

Problem 49: Minimum Absolute Difference in BST
Difficulty: Easy
Topics: Binary Tree Search, DFS, BFS

Given the root of a Binary Search Tree (BST), return *the minimum absolute difference between the values of any two different nodes in the tree*.

```
def getMinimumDifference(self, root: Optional[TreeNode]) -> int:
        min_dist = [float('inf')]
        prev = [None]

        def dfs(node):
            if not node:
                return

            dfs(node.left)
            if prev[0] is not None:
                min_dist[0] = min(min_dist[0], node.val - prev[0])

            prev[0] = node.val

            dfs(node.right)

        dfs(root)
```

```
        return min_dist[0]
```

KEY TAKEAWAYS / PATTERNS:

- The one main thing in BST is the inorder traversal (it gives a sorted list)
- In python, variables cannot be constituted as global so you have to make use of lists if you are making a helper function and you need to keep a variable keeping the values.

Problem 50: Convert Sorted List into Binary Search Tree
Difficulty: Easy
Topics: Binary Search Tree, Divide and Conquer

Given an integer array nums where the elements are sorted in **ascending order**, convert *it to a **height-balanced** binary search tree*.

```
def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:

        def helper(l,  r):
            if l > r:
                return None

            mid = (l + r) // 2
            root = TreeNode(nums[mid])
            root.left = helper(l, mid - 1)
            root.right = helper(mid + 1, r)
            return root

        return helper(0, len(nums)-1)
        # Time Complexity: O(n)
        # Space Complexity: O(logn)
```

KEY TAKEAWAYS / PATTERNS:

- I knew what to do and I knew it was divide and conquer, but I struggled implementing it.
- It's one of the first problems where we created a binary tree so it's important that I try to visualize and understand it.

Problem 51: Majority Element
Difficulty: Easy
Topics: Array, Divide and Conquer, HashTable, Sorting, Counting

*** Important ***

Given an array nums of size n, return *the majority element*.

The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

```
def majorityElement(self, nums: List[int]) -> int:
        # Hashmap Method
        from collections import Counter

        count = Counter(nums)
        majority = len(nums) // 2

        for k, v in count.items():
            if v > majority:
                return k
        # Time Complexity: O(n)
        # Space Complexity: O(n)
-------------------------------------------------------------------------------------------------------
        # Divide and Conquer Method
        def helper(l, r, list):
            # Base Case
            if l == r:
                return list[l]

            # Divide
            mid = (l + r) // 2
            # Conquer
            left_majority = helper(l, mid, list)
            right_majority = helper(mid + 1, r, list)

            if left_majority == right_majority:
                return left_majority
            else:.
                # Otherwise, count occurrences of both candidates in the full segment
                left_count = sum(1 for i in range(l, r + 1) if list[i] == left_majority)
                right_count = sum(1 for i in range(l, r + 1) if list[i] == right_majority)

            # Return the one that occurs more than n//2 times
```

```
            return left_majority if left_count > right_count else right_majority

        return helper(0, len(nums)-1, nums)
-------------------------------------------------------------------------------------------------------------
        # Most Optimal Method: Boyer-Moore Voting Algorithm
        res, count = 0, 0
        for n in nums:
            if count == 0:
                res = n
            count += (1 if n == res else -1)
        return res
        # Time Complexity: O(n)
        # Space Complexity: O(1)
```

KEY TAKEAWAYS / PATTERNS:

- This is a really important question and I learnt a lot of things from doing this question.
- Firstly, was easily able to come up with the hashmap solution and that is a nice thing
- For divide and conquer, I learnt how to divide the array into two subarrays and then how would you break the problem down and solve it. Though in terms of time and space complexity, it is the worst solution
- Learned the Boyer-Moore Voting Algorithm which takes advantage of the fact that there does exist a majority in the array. It traverses the whole array once and does not make use of extra memory, making it the most efficient solution.
- As always, there is more than one solution.
- Learned a few code techniques as well like the left_count and the right_count in the divide and conquer method. Also, practice list comprehensions and writing if statements in one line.

Problem 52: Fibonacci Number
Difficulty: Easy
Topics: Recursion, DP

The **Fibonacci numbers**, commonly denoted F(n) form a sequence, called the **Fibonacci sequence**, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

F(0) = 0, F(1) = 1

F(n) = F(n - 1) + F(n - 2), for n > 1.

Given n, calculate F(n).

```python
def fib(self, n: int) -> int:
        # DP (Memoization)
        dp = [0] * 31

        if n == 0 or n == 1:
            return n

        first = dp[n - 1] if dp[n - 1] != 0 else self.fib(n - 1)
        second = dp[n - 2] if dp[n - 2] != 0 else self.fib(n - 2)

        res = first + second
        dp[n] = res
        return dp[n]

        # DP (Tabular/Bottom-up)
        if n <= 1:
            return n
        dp = [0] * (n + 1)

        dp[0] = 0
        dp[1] = 1
        for i in range(2, n + 1):
            dp[i] = dp[i - 1] + dp[i - 2]

        return dp[n]
```

Problem 53: Removing an element from Linked List
Difficulty: Easy
Topics: LinkedList, Pointers

Given the head of a linked list and an integer val, remove all the nodes of the linked list that has Node.val == val, and return *the new head*.

```
def removeElements(self, head: Optional[ListNode], val: int) -> Optional[ListNode]:
        prev = 0
        current = head

        while current:
            if current.val == val:
                if current == head:
                    head = current.next
                    current = current.next
                else:
                    temp = current.next
                    prev.next = temp
                    current = temp
            else:
                prev = current
                current = current.next

        return head
```

```
def removeElements(self, head: Optional[ListNode], val: int) -> Optional[ListNode]:
        if not head:
            return head

        head.next = self.removeElements(head.next, val)
        if head.val == val:
```

```
            head = head.next

        return head
```

- The iterative method was pretty simple so I definitely don't have to talk about my thought process regarding that.
- However, the recursive method was quite interesting. It gave me a new framework to think about these recursion problems. After calling the function, think about the last thing that will be popped off the stack and then the next lines of the codes should do what you would want for that specific input. Now, also be careful about the recursive call itself and that it should always keep things moving towards the base case.

Problem 54: Palindrome Linked List
Difficulty: Easy
Topics: Linkedlist, Recursion, Two Pointers

*** IMPORTANT ***

Given the head of a singly linked list, return true *if it is a palindrome or* false *otherwise*.

```
def isPalindrome(self, head: Optional[ListNode]) -> bool:
        fast = head
        slow = head

        # find the middle using the slow pointer
        while fast and fast.next:
            fast = fast.next.next
            slow = slow.next

        # reverse the second half of the linkedlist
        prev = None
        while slow:
            temp = slow.next
            slow.next = prev
            prev = slow
            slow = temp
```

```python
        # now you check whether the left and the right half are palindromes
        left, right = head, prev
        while right:
            if left.val != right.val:
                return False
            left = left.next
            right = right.next
        return True
        # Time Complexity: O(n)
        # Space Complexity: O(1)
```

Java | Python3 | C++

```python
class Solution:
    def isPalindrome(self, head: Optional[ListNode]) -> bool:
        self.curr = head
        return self.solve(head)

    def solve(self, head: Optional[ListNode]) -> bool:
        if head is None:
            return True
        ans = self.solve(head.next) and head.val == self.curr.val
        self.curr = self.curr.next
        return ans
```

Interview : Can you come up with a solution with O(1) space ?
You : Sure. (I already knew that)

Approach#4

The main idea to check palindrome is , if the first and last elements are same or not and then check for second and second last

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def isPalindrome(self, head: Optional[ListNode]) -> bool:
        self.front = head  # Use self.front to track left-to-right traversal

        def helper(node: Optional[ListNode]) -> bool:
            if node is None:
                return True

            # Recurse to the end of the list
            if not helper(node.next):
                return False

            # Compare values from front and back
            if node.val != self.front.val:
                return False

            self.front = self.front.next
            return True

        return helper(head)
```

	- So, the first method which was O(n) space was the usage of a list to store all the values of the linked list and then making use of two pointers in the list to check whether it's a palindrome. - The second method, however, is super important because it gives us a few patterns to know and remember. Firstly, it taught me how to find the middle of the linkedlist. Then, it also technically made me understand how we can break the linked list into two halves and we can use this tactic to solve a bunch of problems. - The recursive method to this problem was hella important. IT WAS NICE TO SEE that I was very close to the solution and actually had the right idea. Now one thing I forgot was that the helper function was also the isPalindrome function and was gonna return Boolean and that messed me up. Apart from that, the self part was confusing. We do it so that it is not considered as local within the helper function and actually survives the recursive calls!!! (because we are calling the helper function and not the palindrome function :))
Problem 55: Find the Kth Character In String Game I Difficulty: Easy Topics: Math, Bit Manipulation, Recursion, Simulation	Alice and Bob are playing a game. Initially, Alice has a string word = "a". You are given a **positive** integer k. Now Bob will ask Alice to perform the following operation **forever**: • Generate a new string by **changing** each character in word to its **next** character in the English alphabet, and **append** it to the *original* word. For example, performing the operation on "c" generates "cd" and performing the operation on "zb" generates "zbac". Return the value of the k_{th} character in word, after enough operations have been done for word to have **at least** k characters. **Note** that the character 'z' can be changed to 'a' in the operation. `def kthCharacter(self, k: int) -> str:` `  # Iterative` `  temp = 'a'` `  while len(temp) < k:` `  curr = ''` `  for char in temp:` `  if char == 'z':`

```
                    curr += 'a'
                else:
                    curr += chr(ord(char) + 1)
            temp += curr

        return temp[k - 1]
```

```
        # Recursive
        def helper(word):

            if len(word) >= k:
                return word[k - 1]

            for char in word:
                new_char = 'a' if char is 'z' else chr(ord(char) + 1)
                word = word + new_char

            return helper(word)

        return helper('a')
```

- Few important things learnt from this question. One was the correct usage of ord() and chr() function. Remember that the ASCII value of “a” is 97 and “z” is 122.
- Managed to be able to do the recursive solution myself which was really nice and it was the easier approach.

Problem 56: Top K Frequent Elements
Difficulty: Medium
Topic: Bucket Sort, Array, Heaps, Hashmaps

Given an integer array nums and an integer k, return *the* k *most frequent elements*. You may return the answer in **any order**.

```
def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        # Bucket Sort (In Innovative Manner)
        counter = {}
        freq = [[] for i in range(len(nums) + 1)]
```

```
        for n in nums:
            counter[n] = 1 + counter.get(n, 0)
        for n, c in counter.items():
            freq[c].append(n)

        res = []
        for i in range(len(freq) - 1, 0, -1):
            for num in freq[i]:
                res.append(num)
                if len(res) == k:
                    return res
```

Time Complexity: O(n) (even though many for loops yes!!!), Space Complexity: O(n)

- Few Very Very Important things and patterns that I noticed in this question. So, to begin with, even I knew that a counter was essential to this question. Now, I was originally going with the same bucket sort technique but this question is more than just applying the simple bucket rule (basically the normal bucket rule is making use of indices of a list as the number and then the value at those indices as the number of times that elements appears) (this approach was wrong in many ways as even I identified the issue of an input being a very large number). Rather, it involves the usage of the bucket rule in an updated manner where you set the indices as the counter and the elements at those indices as the elements that have the number of counts.

Problem 57: Encode and Decode Strings
Difficulty: Medium
Topics: Arrays

Design an algorithm to encode a list of strings to a single string. The encoded string is then decoded back to the original list of strings.

Please implement encode and decode

```python
class Solution:

    def encode(self, strs: List[str]) -> str:
        # "3#leet4#code" --- way of encoding
        # Since we are told no additional space so no arrays or hashmaps
        res = ""
        for s in strs:
            res += str((len(s))) + "#" + s
        return res

    def decode(self, s: str) -> List[str]:

        res, i = [], 0

        while i < len(s):
            j = i
            while s[j] != "#":
                j += 1
            length = int(s[i : j])
            char = s[j + 1 : j + 1 + length]
            res.append(char)
            i = j + 1 + length

        return res
```

	- Must say a weird question and a first of its kind. The first initial thought was to add an operator but that would only work with the assumption that there are no characters in the string. The second thought was to make use of an array or hashmap to save the lengths of each word but that was not allowed. So, the third solution is to add two identifiers in the front of the word: first being the number of characters in the word and second being an delimiter. - It is essential in the sense that it was unique and also the decode code taught me a few things on how to deal with two delimiters at the front. This pattern can help me in future problems.
Problem 58: Product of Array Except Self Difficulty: Medium Topics: Arrays, Prefix Sum	Given an integer array nums, return *an array* answer *such that* answer[i] *is equal to the product of all the elements of* nums *except* nums[i]. The product of any prefix or suffix of nums is **guaranteed** to fit in a **32-bit** integer. You must write an algorithm that runs in O(n) time and without using the division operation. `def productExceptSelf(self, nums: List[int]) -> List[int]:` `res = [1] * len(nums)` `prefix = 1` `for i in range(len(nums)):` `res[i] = prefix` `prefix *= nums[i]` `postfix = 1` `for i in range(len(nums) - 1, -1, -1):` `res[i] *= postfix` `postfix *= nums[i]` `return res` `# Time Complexity: O(n), Space Complexity: O(1) (Since they are not counting te res array as extra space)` - Really interesting problem and a new pattern/algorithm to know yesss!!!. So basically, this algorithm involves calculating the prefix and the postfix of each number. This pattern will be helpful if we want to do something with all the numbers of the list except for the current number, especially in O(n) time. This particular solution is O(1) space complexity but if we were to make a separate list for

	each of the prefix and postfix, then the space complexity would bump upto O(n) as well.
Problem No 59: Longest Consecutive Sequence Difficulty: Medium Topics: Arrays, Hashtable, Union Find	Given an unsorted array of integers nums, return *the length of the longest consecutive elements sequence.* You must write an algorithm that runs in O(n) time. `def longestConsecutive(self, nums: List[int]) -> int:` `nums = set(nums)` `longest = 0` `for num in nums:` `if num - 1 not in nums:` `y = num + 1` `while y in nums:` `y += 1` `longest = max(longest, y - num)` `return longest` `# Time Complexity: O(n) Space Complexity: O(n)` - I was able to come up with the brute force solution and to some extent, I was able to sole the efficient solution as well. - For the efficient solution, I knew I had to make use of set but the main key to realize was where to start and find the starting points of the sequence and the tactic was very simple. After that it was common sense.
Problem No 60: Valid Sudoku Difficulty: Medium Topics:	Determine if a 9 x 9 Sudoku board is valid. Only the filled cells need to be validated **according to the following rules**: 1. Each row must contain the digits 1-9 without repetition. 2. Each column must contain the digits 1-9 without repetition. 3. Each of the nine 3 x 3 sub-boxes of the grid must contain the digits 1-9 without repetition. **Note:** • A Sudoku board (partially filled) could be valid but is not necessarily solvable.

- Only the filled cells need to be validated according to the mentioned rules.

```
def isValidSudoku(self, board: List[List[str]]) -> bool:
        rows = collections.defaultdict(set)
        cols = collections.defaultdict(set)
        squares = collections.defaultdict(set) # key = (r//3, c//3)

        for r in range(9):
            for c in range(9):
                if board[r][c] == ".":
                    continue

                if (board[r][c] in rows[r] or
                    board[r][c] in cols[c] or
                    board[r][c] in squares[r//3, c//3]):
                    return False

                rows[r].add(board[r][c])
                cols[c].add(board[r][c])
                squares[(r//3, c//3)].add(board[r][c])

        return True
```

Time complexity: $O(9^2)$ and Space Complexity(9^2)

- This question turned out to be a disappointment for me because I could have done it but gave up too early. However, there are many important things to note from this question. This question, specifically, highlights the power of default dicts and also the usage of dictionaries. Also, it makes us realize that keys can be used differently and really showcases the versatility of the data structure.

Problem 61: Min Stack

```
class MinStack:
```

Difficulty: Medium
Topics: Stacks, Arrays

```
    def __init__(self):
        self.stack = []
        self.minStack = []

    def push(self, val: int) -> None:
        self.stack.append(val)
        val = min(val, self.minStack[-1] if self.minStack else val)
        self.minStack.append(val)

    def pop(self) -> None:
        self.stack.pop()
        self.minStack.pop()

    def top(self) -> int:
        return self.stack[-1]

    def getMin(self) -> int:
        return self.minStack[-1]
```

- Second design question that I have done. I must say I would have gotten the O(n) solution but not the O(1) time complexity solution. The idea of creating a second stack who kept the minimum value of the stack at the point was the key. Once you know that and figure that out, it's actually very easy to code. Again, one important realization in this question was that if you wanna improve the time complexity, then you have to actually give away some space complexity.
- Also, the self part in this question confused me and I feel dumb honestly but its fine and let's just try to get better and better with each question.

Problem No 62: Daily Temperatures

Given an array of integers temperatures represents the daily temperatures, return *an array* answer *such that* answer[i] *is the number of days*

Difficulty: Medium
Topics: Stacks, Arrays, Monotonic Decreasing Stack

you have to wait after the ith *day to get a warmer temperature*. If there is no future day for which this is possible, keep answer[i] == 0 instead.

```
def dailyTemperatures(self, temperatures: List[int]) -> List[int]:
        # Two Pointers (Brute Force)
        # Time Complexity O(n2) in case of descending order temperatures.
        # i-e temperatures = [99,98,97,96]
        l = 0
        r = 1
        answer = [0] * len(temperatures)

        while l < len(temperatures) and r < len(temperatures):
            if temperatures[r] > temperatures[l]:
                days = r - l
                answer[l] = days
                l += 1
                r = l + 1
            else:
                if r == len(temperatures) - 1:
                    l += 1
                    r = l + 1
                else:
                    r += 1
        return answer

        # Stack Method
        # Time Complexity: O(n)
        answer = [0] * len(temperatures)
        stack = []  # store indices of temps
        for i, temp in enumerate(temperatures):
```

```
            while stack and temperatures[i] > temperatures[stack[-1]]:
                prev_day = stack.pop()
                answer[prev_day] = i - prev_day
            stack.append(i)
        return answer
```

- So basically, I would say definitely remember the pattern for this question. So, when you are trying to do two pointers and you run into an issue where you cannot use the sliding window and you have to make the right pointer come back again and again and that you are storing values that are monotonic and decreasing, remember to use a stack!!!

Problem No 63: Car Fleet
Difficulty: Medium
Topics: Stacks, Arrays, Monotonic Stacks, Sorting

**** Challenging ****

There are n cars at given miles away from the starting mile 0, traveling to reach the mile target.

You are given two integer array position and speed, both of length n, where position[i] is the starting mile of the i_{th} car and speed[i] is the speed of the i_{th} car in miles per hour.

A car cannot pass another car, but it can catch up and then travel next to it at the speed of the slower car.

A **car fleet** is a car or cars driving next to each other. The speed of the car fleet is the **minimum** speed of any car in the fleet.

If a car catches up to a car fleet at the mile target, it will still be considered as part of the car fleet.

Return the number of car fleets that will arrive at the destination.

```
def carFleet(self, target: int, position: List[int], speed: List[int]) -> int:
        pair = [[p, s] for p,s in zip(position, speed)] #List Comprehension

        stack = []
        for p, s in reversed(sorted(pair)):
            stack.append((target - p) / s)
            if len(stack) >= 2 and stack[-1] <= stack[-2]:
                stack.pop()
```

```
        return len(stack)
```

Time Complexity: O(nlog(n)) and Space Complexity: O(n)

- In my opinion, definitely, one of the hardest medium questions I have done. I could not for the love of god figure this one out. One problem that is definitely happening is that I am thinking too much on how to use stack since I know it will be used. I think if I just focus on solving the problem more, I will end up doing better.
- Anyways, this question taught me a shit ton of stuff. Firstly, it gave me a good reminder of list comprehension and zip(). Moreover, again the tactin of going backwards worked so don't be too fixated on going from start to end. I think visualizing was the key which I tried to do but failed. As I have realized, the code itself is not the hard part if you know what you are doing so don't worry about spending a lot of time trying to figure out the problem.

Problem No 64: Generate Parentheses
Difficulty: Medium
Topics: Stacks, DP, Backtracking

Given n pairs of parentheses, write a function to *generate all combinations of well-formed parentheses*

```python
def generateParenthesis(self, n: int) -> List[str]:
        # only add an open parenthesis if open < n
        # only add a closed parenthesis if close < open
        # valid IFF closed == open == n
        res = []
        stack = []

        def backtrack(openN, closedN):
            # Base Case
            if openN == closedN == n:
                res.append("".join(stack)) #"".join(stack) adds all the characters in stack to emptystring
                return                     # could do it without a stack and just a string

            # Recursive Case
            if openN < n:
                stack.append("(")
                backtrack(openN + 1, closedN)
```

```
                stack.pop()
            if closedN < openN:
                stack.append(")")
                backtrack(openN, closedN + 1)
                stack.pop()

        backtrack(0, 0)
        return res

def generateParenthesis(self, n: int) -> List[str]:
        # String Solution (have to pass as a parameter bcz it is immutable unlike stack)
        res = []

        def backtrack(openN: int, closedN: int, current: str):
            # Base Case
            if openN == closedN == n:
                res.append(current)
                return

            # Recursive Case
            if openN < n:
                backtrack(openN + 1, closedN, current + "(")
            if closedN < openN:
                backtrack(openN, closedN + 1, current + ")")

        backtrack(0, 0, "")
        return res
```

	- I knew what to do: which was that it could be solved using backtracking. But, I did not know how to implement the code, especially the stack one. I could have come with a string solution with hints though. - We can take a few important things from this question. How to join all elements of a stack to a string. Also how an immutable and mutable data type changes how we code backtracking. It was not as hard as I was expecting it to be. The good thing, however, was that I was able to identify most of the things about the problem. - One more thing, if I cannot figure out the solution and then look for the solution, only just watch the explanation part and then try to code for yourself before looking at the code (PLEASE).
Problem No 65: Evaluate Reverse Polish Notation Difficulty: Medium Topics:	You are given an array of strings tokens that represents an arithmetic expression in a Reverse Polish Notation. Evaluate the expression. Return *an integer that represents the value of the expression*. **Note** that: • The valid operators are '+', '-', '*', and '/'. • Each operand may be an integer or another expression. • The division between two integers always **truncates toward zero**. • There will not be any division by zero. • The input represents a valid arithmetic expression in a reverse polish notation. • The answer and all the intermediate calculations can be represented in a **32-bit** integer. `def evalRPN(self, tokens: List[str]) -> int:` `  stack = []` `  for s in tokens:` `  if s in {"+", "-", "*", "/"}:` `  b = int(stack.pop())` `  a = int(stack.pop())` `  if s == "+":` `  res = a + b` `  elif s == "-":` `  res = a - b`

```
                elif s == "*":
                    res = a * b
                elif s == "/":
                    res = int(a / b)  # Truncates toward zero
                stack.append(res)
            else:
                stack.append(int(s))

        return stack[0]
```

- Pretty chill problem. The only medium problem in the stack playlist that I completely did myself.

Problem No 66: Search a 2D Matrix
Difficulty: Medium
Topics: Binary Search, Matrix

You are given an m x n integer matrix matrix with the following two properties:

- Each row is sorted in non-decreasing order.
- The first integer of each row is greater than the last integer of the previous row.

Given an integer target, return true *if* target *is in* matrix *or* false *otherwise*.

You must write a solution in O(log(m * n)) time complexity.

```
def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        # My Code (Time Complexity: O(log(m) + log(n))
        l, r = 0, len(matrix[0]) - 1
        upper, lower = 0, len(matrix) - 1

        while upper <= lower:
            midR = (upper + lower) // 2
            if matrix[midR][r] < target:
                upper = midR + 1
            elif matrix[midR][l] > target:
```

```
                lower = midR - 1
            else:
                while l <= r:
                    midC = (l + r) // 2
                    if matrix[midR][midC] == target:
                        return True
                    elif matrix[midR][midC] > target:
                        r = midC - 1
                    else:
                        l = midC + 1
        return False

        # Time Complexity: O(log(m * n))
        if not matrix or not matrix[0]:
            return False

        m, n = len(matrix), len(matrix[0])
        left, right = 0, m * n - 1

        while left <= right:
            mid = (left + right) // 2
            row, col = divmod(mid, n) # Takes two numbers and return a tuple containing the quotient and
remainder of the division
            val = matrix[row][col]

            if val == target:
                return True
            elif val < target:
```

```
                left = mid + 1
            else:
                right = mid - 1

        return False
```

- This question is important because it tells us two methods on how to do binary search on a 2D matrix. First option is the one I did where you treat it as a 2D matrix, meanwhile the second is where you treat it as a 1D array.
- It is important that you actually remember the way to calculate mid and the row, column in the 2nd method because its hard and you are not gonna be able to figure it out on the spot.

Problem No 67: Koko Eating Bananas
Difficulty: Medium
Topics: Binary Search

Koko loves to eat bananas. There are n piles of bananas, the i^{th} pile has piles[i] bananas. The guards have gone and will come back in h hours.

Koko can decide her bananas-per-hour eating speed of k. Each hour, she chooses some pile of bananas and eats k bananas from that pile. If the pile has less than k bananas, she eats all of them instead and will not eat any more bananas during this hour.

Koko likes to eat slowly but still wants to finish eating all the bananas before the guards return.

Return *the minimum integer* k *such that she can eat all the bananas within* h *hours*.

```
def minEatingSpeed(self, piles: List[int], h: int) -> int:
        l, r = 1, max(piles)
        res = r

        while l <= r:
            k = (l + r) // 2
            hours = 0
            for p in piles:
                hours += math.ceil(p / k)

            if hours <= h:
```

```
                res = min(res, k)
                r = k - 1
            else:
                l = k + 1
        return res
```

- This was not a difficult problem in any way. I was able to deduce the solution but was unable to code it. Also, one major problem was that I was too fixated on using the list and didn't even think about just going from 1 to max(piles). That is why it is to important to come up with a brute force first so that you can easily transition into a more optimal solution afterwards.

Problem No 68: Find Minimum in Rotated Sorted Array
Difficulty: Medium
Topics: Binary Search

Suppose an array of length n sorted in ascending order is **rotated** between 1 and n times. For example, the array nums = [0,1,2,4,5,6,7] might become:

- [4,5,6,7,0,1,2] if it was rotated 4 times.
- [0,1,2,4,5,6,7] if it was rotated 7 times.

Notice that **rotating** an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums of **unique** elements, return *the minimum element of this array*.

You must write an algorithm that runs in O(log n) time.

```
def findMin(self, nums: List[int]) -> int:
        l, r = 0, len(nums) - 1

        while l < r:
            mid = (l + r) // 2
            if nums[mid] > nums[r]:
                l = mid + 1
            else:
```

```
                r = mid
        return nums[l]
```

- Very dumb moment ngl. This was the easiest medium problem I have encountered and yet somehow I complicated it.
- Draw a graph to visualize it and it would become very easy!!!
- Add graphs as a toolkit to look problems from a new perspective!!!!

Problem No 69: Search in Rotated Sorted Array
Difficulty: Medium
Topics: Binary Search

There is an integer array nums sorted in ascending order (with **distinct** values).

Prior to being passed to your function, nums is **possibly rotated** at an unknown pivot index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (**0-indexed**). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].

Given the array nums **after** the possible rotation and an integer target, return *the index of* target *if it is in* nums, *or* -1 *if it is not in* nums.

You must write an algorithm with O(log n) runtime complexity.

```
def search(self, nums: List[int], target: int) -> int:
        l, r = 0, len(nums) - 1

        while l <= r:
            mid = (l + r) // 2
            if nums[mid] == target:
                return mid

            if nums[l] <= nums[mid]: # Checking where our middle value belongs (left sorted part or right
                                                                                    sorted part)
                # In this case we are in left sorted portion
                # Drawing and visualizing with a graph will make a big difference
                if target < nums[l] or target > nums[mid]:
                    l = mid + 1
```

```
                else:
                    r = mid - 1
            else:
                if target > nums[r] or  target < nums[mid]:
                    r = mid - 1
                else:
                    l = mid + 1
        return -1
```

- The most important thing I learnt from this problem is the power of visualizing through graphs. Make sure from now ahead, you draw graphs especially for complicated binary search problems. It will make your life so much easier!!!! Also, for rotated arrays if there is something to remember, make sure you remember that to check if the middle value is in the left sorted portion, you check whether it is greater than or equal to the value at the left pointer.

Problem No 70: Time Based Key Value Store
Difficulty: Medium
Topics: Binary Search

Design a time-based key-value data structure that can store multiple values for the same key at different time stamps and retrieve the key's value at a certain timestamp.

Implement the TimeMap class:

- TimeMap() Initializes the object of the data structure.
- void set(String key, String value, int timestamp) Stores the key key with the value value at the given time timestamp.
- String get(String key, int timestamp) Returns a value such that set was called previously, with timestamp_prev <= timestamp. If there are multiple such values, it returns the value associated with the largest timestamp_prev. If there are no values, it returns "".

```
class TimeMap:
    from collections import defaultdict

    def __init__(self):
        self.data = defaultdict(list)  # key : val => key : [[]]

    def set(self, key: str, value: str, timestamp: int) -> None:
        self.data[key].append([value, timestamp])
```

```
    def get(self, key: str, timestamp: int) -> str:
        valueMatrix = self.data[key]
        if valueMatrix == []:
            return ""
        else:
            l, r = 0, len(valueMatrix) - 1
            if timestamp < valueMatrix[l][1]:
                return ""
            while l <= r:
                mid = (l + r) // 2
                if valueMatrix[mid][1] > timestamp:
                    r = mid - 1
                elif valueMatrix[mid][1] < timestamp:
                    l = mid + 1
                else:
                    return valueMatrix[mid][0]

            return valueMatrix[r][0]
```

- Managed to do it all by myself which is very nice!! This was an important question because it really tests out the understanding of data structures.

Problem No 71: Longest Repeating Character Replacement
Difficulty: Medium
Topics: Sliding Window, Arrays, Two Pointers,

You are given a string s and an integer k. You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most k times.

Return *the length of the longest substring containing the same letter you can get after performing the above operations*.

```
def characterReplacement(self, s: str, k: int) -> int:
```

Hashmaps

```
        count = {}
        res = 0
        max_freq = 0

        l = 0
        for r in range(len(s)):
            count[s[r]] = 1 + count.get(s[r], 0)
            max_freq = max(max_freq, count[s[r]])

            while (r - l + 1) - max_freq > k:
                count[s[l]] -= 1
                l += 1

            res = max(res, r - l + 1)

        return res
```

- I must say this is a really important question. Few things that I realized was the fact that I am not very good at solving sliding window problems as of now. Now one more thing to realize is that using a counter isn't always O(n) space bcz in this particular example, it is going to be O(1) since there are only 26 characters that are gonna be stored in the hashmap at max.
- The main thing that was needed to be realized was the while loop condition and that was the part tbh.

Problem No 72: Permutation in String
Difficulty: Medium
Topics: Sliding Window, Hashmaps

*** HARD ***

Given two strings s1 and s2, return true if s2 contains a permutation of s1, or false otherwise.

In other words, return true if one of s1's permutations is the substring of s2.

```
from collections import Counter
    # MY SOLUTION (WITH HELP FROM GPT) (O(26* n))
    def checkInclusion(self, s1: str, s2: str) -> bool:
        if len(s1) > len(s2):
```

```
            return False

        s1_count = Counter(s1)
        window_count = Counter()

        for i in range(len(s2)):
            window_count[s2[i]] += 1

            # Shrink window to size of s1
            if i >= len(s1):
                left_char = s2[i - len(s1)]
                window_count[left_char] -= 1
                if window_count[left_char] == 0:
                    del window_count[left_char]

            # Compare counters
            if window_count == s1_count:
                return True

        return False

    # NEETCODE SOLUTION (O(n))
    def checkInclusion(self, s1: str, s2: str) -> bool:
        if len(s1) > len(s2):
            return False

        s1Count, s2Count = [0] * 26, [0] * 26
        for i in range(len(s1)):
```

```
            s1Count[ord(s1[i]) - ord('a')] += 1
            s2Count[ord(s2[i]) - ord('a')] += 1

        matches = 0
        for i in range(26):
            matches += (1 if s1Count[i] == s2Count[i] else 0)

        l = 0
        for r in range(len(s1), len(s2)):
            if matches == 26:
                return True

            index = ord(s2[r]) - ord('a')
            s2Count[index] += 1
            if s1Count[index] == s2Count[index]:
                matches += 1
            elif s1Count[index] + 1 == s2Count[index]:
                matches -= 1

            index = ord(s2[l]) - ord('a')
            s2Count[index] -= 1
            if s1Count[index] == s2Count[index]:
                matches += 1
            elif s1Count[index] - 1 == s2Count[index]:
                matches -= 1
            l += 1
        return matches == 26
```

- Wow, Both of the approaches were hella smart. I am going to write each of the next paragraph to dedicate what was happening in both of the solutions

	- So, in the first solution, the time complexity is O(26*n) (basically O(n)). We are making use of two counters and comparing the counter of s1 with that of the window of s2 and as we are incrementing the left pointer, if it is not zero count, we are decrementing by 1 and if zero, we are removing it from the counter. (i Honestly did not know you could compare hashmaps like this). The order in which you are doing shit inside the loop is so important so be wary of that. And the reason we are not checking the length condition only once is because after the first time, as soon as you move the right pointer by one, the length of the window will exceed the length of the s1 string. - So, in the second solution, the time complexity is O(n) to the dot (basically O(26 + n) because we do go through the whole hashmap once. So, in the solution instead of using hashmap counters, we are using arrays to store the count (we can definitely use hashmaps but arrays are more complex and fun). So, we go through s1 and fill the array counters of both s1 and s2. Now, we need to see if the matched variable is 26 and if so we immediately return True which we do the first thing inside of the loop. Now, we go into the sliding window part of this problem where we just keep on moving and updating the matched, the counter of s1 and the counter of s2.
Problem No 73: Reorder List Difficulty: Medium Topics: LinkedList	You are given the head of a singly linked-list. The list can be represented as: $L_0 \rightarrow L_1 \rightarrow \ldots \rightarrow L_{n-1} \rightarrow L_n$ *Reorder the list to be on the following form:* $L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \ldots$ You may not modify the values in the list's nodes. Only nodes themselves may be changed. `def reorderList(self, head: Optional[ListNode]) -> None:` `"""` `Do not return anything, modify head in-place instead.` `"""` `if not head or not head.next:` `return` `# Step 1: Find the middle` `slow = head`

```
        fast = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        # Step 2: Reverse the second half
        prev = None
        curr = slow.next
        slow.next = None  # Cut the list into two halves
        while curr:
            temp = curr.next
            curr.next = prev
            prev = curr
            curr = temp

        # Step 3: Merge the two halves
        first = head
        second = prev
        while second:
            temp1 = first.next
            temp2 = second.next

            first.next = second
            second.next = temp1

            first = temp1
            second = temp2
```

- Not a hard question at all. This was just basically the combination of three questions I had previously done. One important thing I learned is that it might be really helpful sometimes to just divide the linked list into two using the fast and slow pointers. Also trying to

do this question made me realize, we can use stack to store each element as we traverse through it to store previous elements.

- This question made me realize that I have to make a new google sheet or doc and sort the questions based on the topics.
- The brute force solution is indeed what I was trying: Basically storing all the nodes in a list and then making use of that list to change pointers.

Problem No 74: Remove Nth Node from End of List
Difficulty: Medium
Topics: LinkedList

Given the head of a linked list, remove the n_{th} node from the end of the list and return its head.

```python
def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:
        # My Method
        dummy = ListNode(0, head)
        slow = fast = dummy

        for _ in range(n):
            fast = fast.next
        while fast:
            fast = fast.next
            prev = slow
            slow = slow.next

        temp = slow.next
        prev.next = temp
        slow.next = None
        return dummy.next

        # Neetcode Method (Preferable)
        dummy = ListNode(0, head)
        left = dummy
        right = head
```

```
        while n > 0 and right:
            right = right.next
            n -= 1

        while right:
            right = right.next
            left = left.next

        left.next = left.next.next
        return dummy.next
```

- So basically, the brute force solution is to reverse the list and then it becomes super easy to remove the nth element. The second method is more important and a pattern that must be remembered.
- Basically, since I did not use the fact that left.next = left.next.next so I had to make use of another pointer aka prev, which is not really necessary and that is why i prefer the neetcode solution.

Problem No 75: Copy List with Random Pointer
Difficulty: Medium
Topics: LinkedList

A linked list of length n is given such that each node contains an additional random pointer, which could point to any node in the list, or null.

Construct a **deep copy** of the list. The deep copy should consist of exactly n **brand new** nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. **None of the pointers in the new list should point to nodes in the original list**.

For example, if there are two nodes X and Y in the original list, where X.random --> Y, then for the corresponding two nodes x and y in the copied list, x.random --> y.

Return *the head of the copied linked list*.

The linked list is represented in the input/output as a list of n nodes. Each node is represented as a pair of [val, random_index] where:

- val: an integer representing Node.val

- random_index: the index of the node (range from 0 to n-1) that the random pointer points to, or null if it does not point to any node.

Your code will **only** be given the head of the original linked list.

```
def copyRandomList(self, head: 'Optional[Node]') -> 'Optional[Node]':
        link = { None : None }  # original --> copy
        curr = head

        while curr:
            copy = Node(curr.val)
            link[curr] = copy
            curr = curr.next

        curr = head
        while curr:
            copy = link[curr]
            copy.next = link[curr.next]
            copy.random = link[curr.random]
            curr = curr.next

        return link[head]

        # Dummy method: The one that I was trying to implement
        if not head:
            return None

        old_to_new = {None: None}
        curr = head
        dummy = Node(0)
```

```
        copy_curr = dummy

        # First pass: create new nodes and build mapping
        while curr:
            copy = Node(curr.val)
            old_to_new[curr] = copy
            copy_curr.next = copy
            copy_curr = copy_curr.next
            curr = curr.next

        # Second pass: assign random pointers
        curr = head
        while curr:
            copy = old_to_new[curr]
            copy.random = old_to_new[curr.random]
            curr = curr.next

        return dummy.next
```

- A really solid question. Gave me a new pattern to recognize how we can use hashmaps to basically make new linked lists and also create copies.
- Also wtf do you get scared when u know it requires two passes, like that was dumb of me to stop. No stopping before 1 hr spent on a question.

Problem No 76: Add Two Numbers
Difficulty: Medium
Topics: LinkedList, Recursion

You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in **reverse order**, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

```
def addTwoNumbers(self, l1: Optional[ListNode], l2: Optional[ListNode]) -> Optional[ListNode]:
```

```
        dummy = ListNode()
        curr = dummy

        carry = 0
        while l1 or l2 or carry:
            v1 = l1.val if l1 else 0
            v2 = l2.val if l2 else 0

            # new digit
            val = v1 + v2 + carry
            carry = val // 10
            val = val % 10
            curr.next = ListNode(val)

            # update the pointers
            l1 = l1.next if l1 else None
            l2 = l2.next if l2 else None
            curr = curr.next

        return dummy.next
```

- This was an easy question and an interesting one. There were few edge cases to deal with and that was the main issue else the logic itself was pretty chill. Whenever you need to make a new linked list, think about creating a dummy node.

Problem No 77: Find the Duplicate Number
Difficulty: Hard
Topics: LinkedList

Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive.

There is only **one repeated number** in nums, return *this repeated number*.

You must solve the problem **without** modifying the array nums and using only constant extra space.

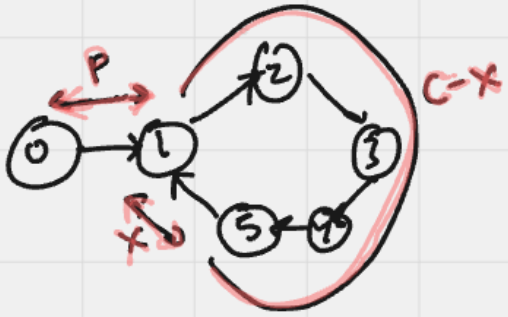
```python
def findDuplicate(self, nums: List[int]) -> int:
        slow, fast = 0, 0
        while True:
            slow = nums[slow]
            fast = nums[nums[fast]]
            if slow == fast:
                break

        slow2 = 0
        while True:
            slow = nums[slow]
            slow2 = nums[slow2]
            if slow == slow2:
                return slow
```

- Find the Duplicate Number Explaination.

We needed to realise two things:

1. Its a linked list cycle problem
2. We need to make use of Floyd's algorithm to find the start of the cycle.



$2 \cdot slow = fast$

$2(p+c-x) = p+c-x+c$

$2p+2c-2x = p+2c-x$

$p = x$

- so basically, you make use of a slow & fast pointer & find the first intersection point.
- Then, you run another slow pointer from the beginning & where the two slow pointers intersect is where you find the answer (duplicate in this specific example)

Problem No 78: LRU Cache
Difficulty: Medium
Topics: LinkedList, Hashtable, Design, Doubly Linked List

Design a data structure that follows the constraints of a **Least Recently Used (LRU) cache**.

Implement the LRUCache class:

- LRUCache(int capacity) Initialize the LRU cache with **positive** size capacity.
- int get(int key) Return the value of the key if the key exists, otherwise return -1.
- void put(int key, int value) Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, **evict** the least recently used key.

The functions get and put must each run in O(1) average time complexity.

```
class Node:
    def __init__(self, key, val):
        self.key, self.val = key, val
        self.prev = self.next = None

class LRUCache:

    def __init__(self, capacity: int):
        self.cap = capacity
        self.cache = {} # key --> Node

        # Left = LRU, Right = most recent used
        self.left, self.right = Node(0, 0), Node(0, 0)
        self.left.next, self.right.prev = self.right, self.left

    def remove(self, node):
        prev, nxt = node.prev, node.next
        prev.next, nxt.prev = nxt, prev
```

```
    def insert(self, node):
        prev, nxt = self.right.prev, self.right
        prev.next = nxt.prev = node
        node.next, node.prev = nxt, prev

    def get(self, key: int) -> int:
        if key in self.cache:
            self.remove(self.cache[key])
            self.insert(self.cache[key])
            return self.cache[key].val
        else:
            return -1

    def put(self, key: int, value: int) -> None:
        if key in self.cache:
            self.remove(self.cache[key])
        self.cache[key] = Node(key, value)
        self.insert(self.cache[key])

        if len(self.cache) > self.cap:
            # remove the LRU
            lru = self.left.next
            self.remove(lru)
            del self.cache[lru.key]
```

- So, i would classify this question as one of my favourite ones so far because it's so mazedar!! In the question, we learned how to tackle Node objects as values in the key-val pairs of a dictionary (hashmap). In this question, I also really liked the way we used the left and right dummy nodes to know what is most recently used and what is least recently used. Also, the usage of helper functions

	was made well in this question. Overall, there was a lot to learn from this question and should be done again for revision 🙂
Problem 79: Diameter of a Binary Tree Difficulty: Easy Topics: Trees, DFS	Given the root of a binary tree, return *the length of the **diameter** of the tree*. The **diameter** of a binary tree is the **length** of the longest path between any two nodes in a tree. This path may or may not pass through the root. The **length** of a path between two nodes is represented by the number of edges between them. `def diameterOfBinaryTree(self, root: Optional[TreeNode]) -> int:` `  self.max_diameter = 0` `  def dfs(node):` `  if not node:` `  return 0` `  left_height = dfs(node.left)` `  right_height = dfs(node.right)` `  self.max_diameter = max(self.max_diameter, left_height + right_height)` `  return 1 + max(left_height, right_height)` `  dfs(root)` `  return self.max_diameter` - One of the first things to realize is that since the variables in the python are considered global, you have to use self for them to access them; meanwhile that is not the case with lists. - It is an interesting problem I must say and makes you think about recursion in a better way. As I have seen, the tactic for base case works perfectly. The recursive case tactic seems nice to me as well now. It is just a matter of practice I think. I think now I should be able to do most of the easy recursion/binary tree problems.

Problem 80: Balanced Binary Tree
Difficulty: Easy
Topics: Trees, DFS

Given a binary tree, determine if it is **height-balanced**.

```python
def isBalanced(self, root: Optional[TreeNode]) -> bool:
        # Solution with me and ChatGPT
        def depth(node):
            if not node:
                return 0

            left = depth(node.left)
            right = depth(node.right)

            # If either subtree is unbalanced, propagate -1 up
            if left == -1 or right == -1:
                return -1

            if abs(left - right) > 1:
                return -1

            return 1 + max(left, right)
        return depth(root) != -1

        # NEETCODE SOLUTION (VERY IMPORTANT)
        def dfs(root):
            if not root: return [True, 0]

            left, right = dfs(root.left), dfs(root.right)
            balanced = (left[0] and right[0] and
                        abs(left[1] - right[1]) <= 1)
```

```
            return [balanced, 1 + max(left[1], right[1])]


        return dfs(root)[0]
```

- I would say that this is one of the most important binary tree questions I have solved so far since I was able to learn a lot of stuff. So, the thought process of solving this question was the same as Neetcode for me but I didn't know how to incorporate the height part without making a new function. So, one of the most important things I learnt was how we can return just more than one thing to have more information about the return calls. In this case, it was the height of the subtrees and how they were being stored in an array.
- Moreover, I just realized that in the future questions, when I am trying to solve for the recursive case, I need to not just consider the most simplest binary tree apart from the single node and the empty tree, I should also think about how there are more trees beneath the right and left subtree and how should I tackle with that as I did with my solution and seeing if any returned -1 so we could immediately return False.

Problem 81: SubTree of Another Tree
Difficulty: Easy
Topics: Trees, DFS

Given the roots of two binary trees root and subRoot, return true if there is a subtree of root with the same structure and node values of subRoot and false otherwise.

A subtree of a binary tree tree is a tree that consists of a node in tree and all of this node's descendants. The tree tree could also be considered as a subtree of itself.

```
def isSubtree(self, root: Optional[TreeNode], subRoot: Optional[TreeNode]) -> bool:
    # CHATGPT CODE
        def isIdentical(r, s):
            if not r and not s:
                return True  # Both are None -> identical
            if not r or not s:
                return False  # One is None, the other isn't -> not identical
            if r.val != s.val:
                return False  # Mismatched values

            # Recursively check left and right subtrees
            return isIdentical(r.left, s.left) and isIdentical(r.right, s.right)
```

```
        # Main check: traverse the root tree and check for subtree match
        if not root:
            return False  # If main tree is empty, can't contain subRoot
        if isIdentical(root, subRoot):
            return True  # Found a matching subtree
        # Recur down left and right subtrees
        return self.isSubtree(root.left, subRoot) or self.isSubtree(root.right, subRoot)

    # NEETCODE
    def isSubtree(self, root: Optional[TreeNode], subRoot: Optional[TreeNode]) -> bool:
        # Base Cases
        if not subRoot: return True
        if not root: return False
        if self.isSameTree(root, subRoot):
            return True

        return (self.isSubtree(root.left, subRoot) or
                self.isSubtree(root.right, subRoot))

    def isSameTree(self, p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:
        # Base case: both nodes are None (identical trees)
        if not p and not q:
            return True
        # If one is None and the other is not, or values are different, return False
        if not p or not q or p.val != q.val:
            return False

        # recursive case
```

```
        return self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)
```

- One of the most important things that I have realized is to keep revising the old problems as we might use them in future problems. You should definitely think whether you could make use of another problem that you have solved prior to a current problem. Also, for every topic, you must have a list of all the easy problems in your mind and should be able to dictate them out easily.
- Again, understanding the problem was not an issue, rather it was the coding part. However, we are getting better and better with each repetition.
- Both the chat gpt and the neetcode solutions are exactly the same but differently formatted.
- As for the time complexity, in the worst case, it would be O(r * s) as we would have to go through every single node of r and s.

Problem 82: Lowest Common Ancestor of a Binary Search Tree
Difficulty: Medium
Topics: DFS, Recursion

Given a binary search tree (BST), find the lowest common ancestor (LCA) node of two given nodes in the BST.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

```
def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':


        def dfs(r, p, q):
            if not r:
                return None
            if p.val < r.val and q.val < r.val:
                return dfs(r.left, p, q)
            elif p.val > r.val and q.val > r.val:
                return dfs(r.right, p, q)
            else:
                return r


        return dfs(root, p, q)
```

- So basically the stupidity was that I was not using the binary search tree aspect of the problem WOW lol. I need to be careful about that from now on. Apart from that, it was actually pretty simple.
- The time complexity is O(logn) (which is the height of the tree) and the space complexity is O(h) bcz of the stack call. However, we

	can solve it iteratively pretty easily and that would make the space complexity into O(1).
Problem 83: Binary Tree Level Order Traversal Difficulty: Medium Topics: BFS	Given the root of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level). `from collections import deque` `    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:` `        if not root:` `            return []` `        res, level = [], []` `        queue = deque([root])` `        while queue:` `            for _ in range(len(queue)):` `                node = queue.popleft()` `                level.append(node.val)` `                if node.left: queue.append(node.left)` `                if node.right: queue.append(node.right)` `            res.append(level)` `            level = []` `        return res` - Pretty self-explanatory. Just make sure to use deque instead of a list as a queue because that makes the time complexity better as the whole array is not moved when the first item is removed/popped. - The time complexity of this algorithm is O(n) and the space complexity is also O(n).

Problem 84: Binary Tree Right Side View
Difficulty: Medium
Topics: BFS, DFS

Given the root of a binary tree, imagine yourself standing on the **right side** of it, return *the values of the nodes you can see ordered from top to bottom*.

```python
from collections import deque
   def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
       # BFS SOLUTION
       if not root:
           return []

       res, level = [], []
       queue = deque([root])

       while queue:
           for _ in range(len(queue)):
               node = queue.popleft()
               if len(level) < 1:
                   res.append(node.val)
                   level.append(node.val)
                   if node.right: queue.append(node.right)
                   if node.left: queue.append(node.left)
               else:
                   if node.right: queue.append(node.right)
                   if node.left: queue.append(node.left)
                   continue

           level = []

       return res
```

```
        # DFS SOLUTION
        res = []

        def dfs(node, depth):
            if not node:
                return []

            # If visiting this depth for the first time, add it
            if depth == len(res):
                res.append(node.val)

            # Right-first traversal
            dfs(node.right, depth + 1)
            dfs(node.left, depth + 1)

        dfs(root, 0)
        return res
```

- So basically, I had the right initial thought for dfs where I knew we had to maintain the depths/levels of the binary tree
- Then, I was able to code the bfs version myself
- Though, I needed help for the recursive solution. My logic was correct but the base case not node.left and not node.right wasnt right. Also, I was just traversing the right side and not the left side. Now, we know how we can just right traversal and also as a consequence, the left traversal of the binary tree.

Problem 85: Count Good Nodes in Binary Tree
Difficulty: Medium
Topics: Binary Trees, DFS, BFS

Given a binary tree root, a node *X* in the tree is named **good** if in the path from root to *X* there are no nodes with a value *greater than* X.

Return the number of **good** nodes in the binary tree.

```
def goodNodes(self, root: TreeNode) -> int:
        # global variable method
        self.count = 0
```

```
        def dfs(node, curr_max):
            if not node:
                return 0

            if node.val >= curr_max:
                self.count += 1
                curr_max = max(curr_max, node.val)

            dfs(node.right, curr_max)
            dfs(node.left, curr_max)

        dfs(root, root.val)
        return self.count

        # return method
        def dfs(node, curr_max):
            if not node:
                return 0

            good = 1 if node.val >= curr_max else 0
            curr_max = max(curr_max, node.val)

            left = dfs(node.left, curr_max)
            right = dfs(node.right, curr_max)

            return good + left + right
```

```
        return dfs(root, root.val)
```

- Surprisingly, these medium questions are not as hard as I was expecting them to be or am I just getting better!!! I was able to solve the global variable version with just a hint. However, for the return version, I needed help.
- I think we need to revisit and draw these problems out and we will get them!!!

Problem 86: Validate Binary Search Tree
Difficulty: Medium
Topics: BST, DFS, Recursion

Given the root of a binary tree, *determine if it is a valid binary search tree (BST)*.

A **valid BST** is defined as follows:

- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

```
def isValidBST(self, root: Optional[TreeNode]) -> bool:

        def valid(node, left, right):
            if not node:
                return True
            if not (node.val > left and node.val < right):
                return False

            return (valid(node.left, left, node.val) and
                    valid(node.right, node.val, right))

        return valid(root, float("-inf"), float("inf"))
```

- I am so glad I just did this question because my whole understanding of a BST was wrong (CRAZY IK but BETTER LATE THAN NEVER) Every node on the right side of the root node should be greater and every node on the left side should be smaller
- Apart from that, I think I have got RECURSION!!!
- Time Complexity of this code is O(n) and the space is O(n) as well. Brute Force would have been O(n^2) since we would have went to every single node and for that node, we would have searched the whole left and right side.

Problem 87: Kth Smallest Element in a BST
Difficulty: Medium
Topics: DFS, Recursion, BST

Given the root of a binary search tree, and an integer k, return *the* k^{th} *smallest value (**1-indexed**) of all the values of the nodes in the tree*.

```python
# Simplest Solulu: TC : O(n), SC: O(n)
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        inorder = self.inOrderTraversal(root)
        return inorder[k - 1]

    def inOrderTraversal(self, root):
        res = []
        def inorder(node):
            if not node:
                return

            inorder(node.left)
            res.append(node.val)
            inorder(node.right)

        inorder(root)
        return res

    # Another Efficient Solution
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        self.count = 0
        self.result = None

        def dfs(node):
            if not node or self.result is not None:
                return
```

```
            dfs(node.left)

            self.count += 1
            if self.count == k:
                self.result = node.val
                return

            dfs(node.right)

        dfs(root)
        return self.result

    # Iterative Solution AKA Iterative Inorder Traversal of BST
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        n = 0
        stack = []
        curr = root

        while curr and stack:
            while curr:
                stack.append(curr)
                curr = curr.left

            curr = stack.pop()
            n += 1
            if n == k:
                return curr.val
            curr = curr.right
```

```python
# MORRIS TRAVERSAL
   def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
       curr = root

       while curr:
           if not curr.left:
               k -= 1
               if k == 0:
                   return curr.val
               curr = curr.right
           else:
               # Find InOrder Predecessor (IP)
               pred = curr.left
               while pred.right and pred.right != curr:
                   pred = pred.right

               if not pred.right:
                   pred.right = curr # create thread
                   curr = curr.left
               else:
                   pred.right = None # remove thread
                   k -= 1
                   if k == 0:
                       return curr.val
                   curr = curr.right

       return -1
```

	- The first method was pretty simple and I was easily able to solve it. - The second method is a bit more efficient because it stops as soon it finds the correct k value. The time complexity goes from nlogn to n. - The third method is an iterative method. It is important because it uses iteration and hence we learn how to iteratively traverse the Binary Search Tree in Order. - Also, I learnt about Morris Inorder Traversal which is the most optimal in terms of time and space complexity. Its basically involves forming and deleting threads to go back to the root node after traversing the left subtree. Also, use the concept of IP. It has a clear template and it would be best to just memorize it just in case an interviewer asks for a follow-up question.
Problem 87: Construct Binary Tree from Preorder and Inorder Traversal Difficulty: Medium Topics: Trees, Binary Trees, Hashmaps, Arrays, Recursion	Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return *the binary tree*. `def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:` `        if not preorder or not inorder:` `            return None` `        root = TreeNode(preorder[0]) # Making use of the fact that the first preorder is always the rootnode` `        mid = inorder.index(preorder[0]) # Finding the index of root in inorder to divide left and right subtrees` `        root.left = self.buildTree(preorder[1 : mid + 1], inorder[:mid])` `        root.right = self.buildTree(preorder[mid + 1:], inorder[mid + 1:])` `        return root` - I must say I figured out the root part of the preorder but couldn't figure out the division part. This is really interesting because firstly it made me truly understand the traversals. Moreover, now I can make Binary Trees using any valid combination of traversals. - This problem also highlights the power of RECURSION!!! Made the code look so easy.

BINARY TREES PROBLEMS QUICK REVISION NOTES

Fundamental Problems that I have Solved for BinaryTrees

- Is SameTree → fundamental/trivial
- Invert a Binary Tree → fundamental/trivial
- Is Subtree → Just make use of SameTree problem.
- Diameter of Binary Tree → learnt about nonlocal variables in python.
  → checks your understanding of the problem mar depth of binary trees.
- Max Depth of Binary Tree → Really easy & imp & fundamental problem.
  → 1 + max(Depth(root.right), Depth(root.left))
- Balanced Binary Tree → Actually the hardest easy problem.
  → really interesting as you incorporate a list as a return so that you can have more data. Need to ask ChatGPT why the helper function couldn't be used for that extra info.
- Lowest common ancestor of a Binary Search Tree → interesting problem.
- Binary Tree Level order Traversal → pretty simple BFS problem but important. Should be memorized.
- Binary Tree Right Side View → How to do right side traversal.
  → Can do it using BFS
  
  → For DFS, you have to keep track of depth/levels.
- Count Good Nodes in Binary Tree → How to make use of accumax to see if there are any nodes with larger values along the path from root to that particular node.
- Valid BinarySearch Tree → True definition of a BST & how we can validate one.
- K^{th} Smallest Element in a BST → inorder Traversal iteratively for a BST.
- Construct Binary Tree from pre-order and inorder Traversal → learned how we can build Binary Trees using any valid traversals. Also made me completely understand inorder and preorder traversals.

Problem 88: Subsets
Difficulty: Medium
Topics: Backtracking

Given an integer array nums of **unique** elements, return *all possible subsets (the power set)*.

The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

```
def subsets(self, nums: List[int]) -> List[List[int]]:
        res = []

        def dfs(start, subset):
            res.append(subset[:])  # Copy the current subset
            for i in range(start, len(nums)):
                subset.append(nums[i])
                dfs(i + 1, subset)
                subset.pop()

        dfs(0, [])
        return res
        # Time Complexity: O(n*2^n)

        # NEETCODE SOL
        res = []
        subset = []
        def dfs(i):
            if i >= len(nums):
                res.append(subset.copy())
                return

            # decision to include nums[i]
            subset.append(nums[i])
            dfs(i + 1)
```

```
            # decision NOT to include nums[i]
            subset.pop()
            dfs(i + 1)

        dfs(0)
        return nums
```

- Pretty nice problem. I think I have now gotten how backtracking (DFS) works. The code is pretty similar to that of a combination question.
- Always remember that when you try the code, and see empty lists, check whether you have made a copy of the list or not.

Problem 89: Combination Sum II
Difficulty: Medium
Topics: Backtracking

Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target.

Each number in candidates may only be used **once** in the combination.

Note: The solution set must not contain duplicate combinations.

```
def combinationSum2(self, candidates: List[int], target: int) -> List[List[int]]:
        res = []
        candidates.sort()

        def dfs(i, cur, total):
            if total == target:
                res.append(cur.copy())
                return
            if i >= len(candidates) or total > target:
                return

            # include candidates[i]
```

```
            cur.append(candidates[i])
            dfs(i + 1, cur, total + candidates[i])
            cur.pop()

            # skip candidates[i]
            while i + 1 < len(candidates) and candidates[i] == candidates[i + 1]:
                i += 1
            dfs(i + 1, cur, total)

        dfs(0, [], 0)
        return res
```

- Similar to combination sum 1 except for the fact that now we needed to sort and also skip the repeats.
- Why did we wanna skip the repeats? Because they would result in duplicates.

Problem 90: Subsets II
Difficulty: Medium
Topics: Backtracking

Given an integer array nums that may contain duplicates, return *all possible subsets (the power set)*.

The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

```
def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        # Method One
        nums.sort()
        res = []
        subset = []
        def dfs(i):
            if i >= len(nums):
                res.append(subset.copy())
                return

            # decision to include nums[i]
```

```
            subset.append(nums[i])
            dfs(i + 1)
            subset.pop()
            # decision NOT to include nums[i]
            while i + 1 < len(nums) and nums[i] == nums[i + 1]:
                i += 1
            dfs(i + 1)

        dfs(0)
        return res

        # Method Two
        nums.sort()
        res = []

        def dfs(start, subset):
            res.append(subset[:])  # Copy the current subset
            for i in range(start, len(nums)):
                if i > start and nums[i] == nums[i - 1]:
                    continue
                subset.append(nums[i])
                dfs(i + 1, subset)
                subset.pop()

        dfs(0, [])
        return res
```

- This question needed the same amendment that combination sum 2 needed from combination sum1 . We just had to tackle duplicates and the way to do that was to just sort the list and keep i moving until a new element is found.

	- Time complexity of this question is: O(n * 2^n)
Problem 91: Palindrome Partitioning Difficulty: Medium Topics: Backtracking	Given a string s, partition s such that every substring of the partition is a **palindrome**. Return *all possible palindrome partitioning of* s.

```
def partition(self, s: str) -> List[List[str]]:
        res, path = [], []

        def dfs(i):
            # Base Case
            if i >= len(s):
                res.append(path.copy())
                return

            for j in range(i, len(s)):
                if self.isPalindrome(s, i, j):
                    path.append(s[i : j+1])
                    dfs(j + 1)
                    path.pop()

        dfs(0)
        return res


    def isPalindrome(self, s, l, r):
        while l < r:
            if s[l] != s[r]:
                return False
            l, r = l + 1, r - 1
```

```
        return True
```

- So, the tough part about this question was making the decision tree. Once the decision tree is made, and you understand what is going on, the coding part is not hard.
- Though, this was definitely on the harder side in terms of coding.

Problem 92: Letter Combinations of a Phone Number
Difficulty: Medium
Topics: Backtracking

Given a string containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent. Return the answer in **any order**.

A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

```
def letterCombinations(self, digits: str) -> List[str]:
        digit_to_letters = {
                            "2": "abc",
                            "3": "def",
                            "4": "ghi",
                            "5": "jkl",
                            "6": "mno",
                            "7": "pqrs",
                            "8": "tuv",
                            "9": "wxyz"
                                    }

        res = []

        if not digits:
            return res
```

```
        def dfs(i, comb):
            if len(comb) == len(digits):
                res.append(comb)
                return
            if i >= len(digits):
                return

            for letter in digit_to_letters[digits[i]]:
                dfs(i + 1, comb + letter)

        dfs(0, "")
        return res
```

- Today is a bad day because I was making so many stupid mistakes while coding this. I had the solution in mind and it was right but the coding aspect ruined it. Firstly, I was treating it as a list i-e appending and stuff (quite dumb of me). Secondly, I introduced a new loop which wasn't really required.
- Few questions that I need to ask is why didn't we pop() the elements that we added. How is it different from the rest of the questions?
- We didn't need pop() since we are using strings and they are immutable. So, for each recursive call, a new string is created.
- As for the time complexity, it will be O(4*4^n). Basically, the length of each substring would be at least n and there would be at most 4^n substrings in the worst case.

Problem No 93: Kth Largest Element in a Stream
Difficulty: Easy
Topics: Heap, Design, Data Stream, Binary Tree

You are part of a university admissions office and need to keep track of the kth highest test score from applicants in real-time. This helps to determine cut-off marks for interviews and admissions dynamically as new applicants submit their scores.

You are tasked to implement a class which, for a given integer k, maintains a stream of test scores and continuously returns the kth highest test score **after** a new score has been submitted. More specifically, we are looking for the kth highest score in the sorted list of all scores.

Implement the KthLargest class:

- KthLargest(int k, int[] nums) Initializes the object with the integer k and the stream of test scores nums.
- int add(int val) Adds a new test score val to the stream and returns the element representing the k_{th} largest element in the pool of test scores so far.

```python
class KthLargest:
    import heapq

    def __init__(self, k: int, nums: List[int]):
        self.k = k
        self.min_heap = nums
        heapq.heapify(self.min_heap)

        # Reduce heap to size k by removing smallest elements
        while len(self.min_heap) > k:
            heapq.heappop(self.min_heap)

    def add(self, val: int) -> int:
        heapq.heappush(self.min_heap, val)

        # If heap size exceeds k, pop smallest element
        if len(self.min_heap) > self.k:
            heapq.heappop(self.min_heap)

        # The root of the heap is the kth largest element
        return self.min_heap[0]
```

Time Complexity: O(logK), Space Complexity: O(K)

- I couldn't solve it but it was the first ever heap question that I was solving so it's fine. I did, however, take many lessons from this question.
- So, if i was to do brute force, it would involve sorting and hence the time complexity would be O(nlogn) which is wayyy worse than the solution we devised. Always remember that if it involves sorting and then it involves constant removal and addition of elements, its way better to just use heaps. Also, it was better to heapify in the initialization part because otherwise the add function would be O(n).

Problem No 94: Last Stone Weight
Difficulty: Easy
Topics: Heaps, Arrays

You are given an array of integers stones where stones[i] is the weight of the i_{th} stone.

We are playing a game with the stones. On each turn, we choose the **heaviest two stones** and smash them together. Suppose the heaviest two stones have weights x and y with x <= y. The result of this smash is:

- If x == y, both stones are destroyed, and
- If x != y, the stone of weight x is destroyed, and the stone of weight y has new weight y - x.

At the end of the game, there is **at most one** stone left.

Return *the weight of the last remaining stone*. If there are no stones left, return 0.

```
    import heapq
    def lastStoneWeight(self, stones: List[int]) -> int:
        if not stones: return 0

        max_heap = stones.copy()
        # Creating a max heap
        for i in range(len(max_heap)):
            max_heap[i] = -max_heap[i]
        heapq.heapify(max_heap)

        # Actual Problem
        while len(max_heap) > 1:
            stone1 = -heapq.heappop(max_heap)
            stone2 = -heapq.heappop(max_heap)
            if stone1 == stone2:
                continue
            else:
                new_stone = stone1 - stone2
                heapq.heappush(max_heap, -new_stone) #adding the new positive stone weight
```

```
        return -heapq.heappop(max_heap) if len(max_heap) == 1 else 0
```

Time Complexity: O(nlogn), Space Complexity: O(n)

- I managed to do the problem myself only because I first taught myself the concept of heaps in detail (in detail i mean i watched a video of 23 minutes). This was an easy problem in terms of understanding what to do. The main part was the coding aspect.

Problem No 95: K Closest Points to Origin
Difficulty: Medium
Topics: Heaps

Given an array of points where points[i] = [x_i, y_i] represents a point on the **X-Y** plane and an integer k, return the k closest points to the origin (0, 0).

The distance between two points on the **X-Y** plane is the Euclidean distance (i.e., $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$).

You may return the answer in **any order**. The answer is **guaranteed** to be **unique** (except for the order that it is in).

```
import heapq
    def kClosest(self, points: List[List[int]], k: int) -> List[List[int]]:
        res = []
        heap = []

        for i in range(len(points)):
            distance = (points[i][0])**2 + (points[i][1])**2
            heapq.heappush(heap, (distance, points[i]))

        for _ in range(k):
            min_dist = heapq.heappop(heap)
            res.append(min_dist[1])

        return res
```

Time Complexity: O(klogn), Space Complexity: O(n)

- We could have also used heapify(just like Neetcode) but this seemed better to me.
- Apart from that, we did not need to create a dictionary to make the tuple so make sure you are not creating extra space for your code

for no reason.

Problem No 96: Kth Largest Element in an Array
Difficulty: Medium
Topics: Heaps

Given an integer array nums and an integer k, return the kth largest element in the array.

Note that it is the kth largest element in the sorted order, not the kth distinct element.

Can you solve it without sorting?

```
import heapq
    def findKthLargest(self, nums: List[int], k: int) -> int:
        # SOLUTION ONE: MAX HEAP (Time Complexity: O(klogn))
        maxHeap = nums.copy()
        # for i in range(len(nums)):
        #     maxHeap[i] = -maxHeap[i]
        maxHeap = [-x for x in maxHeap] #list comprehension
        heapq.heapify(maxHeap)

        while k > 0:
            maxVal = -heapq.heappop(maxHeap)
            k -= 1

        return maxVal

        # SOLUTION TWO: QUICK SELECT (Average Case Time Complexity: O(n)
                                    # Worst Case Time Complexity: O(n2))

        k = len(nums) - k
        def quickSelect(l, r):
            pivot, p = nums[r], l
```

```
            for i in range(l, r):
                if nums[i] <= pivot:
                    nums[p], nums[i] = nums[i], nums[p]
                    p += 1

            nums[p], nums[r] = nums[r], nums[p]

            if p > k:
                return quickSelect(l, p - 1)
            elif p < k:
                return quickSelect(p + 1, r)
            else:
                return nums[p]

        return quickSelect(0, len(nums) - 1)
```

- Without sorting I would have definitely gone with the max heap method.
- However, it is important to know the quicksort / quickselect algorithm as it can be helpful like in this case because for the average case, the time complexity is O(n).
- So, basically, the main learning from this question is the quicksort algorithm and the quickselect variation of that algorithm. Apart from that, learnt how to do list comprehension on the swapping part of maxHeap.

Problem No 97: Task Scheduler (Fav Question So Far)
Difficulty: Medium
Topics: Heaps

You are given an array of CPU tasks, each labeled with a letter from A to Z, and a number n. Each CPU interval can be idle or allow the completion of one task. Tasks can be completed in any order, but there's a constraint: there has to be a gap of at least n intervals between two tasks with the same label.

Return the minimum number of CPU intervals required to complete all tasks.

```
def leastInterval(self, tasks: List[str], n: int) -> int:
        # Time Complexity: O(n)
        count = Counter(tasks)
        maxHeap = [-cnt for cnt in count.values()]
        heapq.heapify(maxHeap)

        time = 0
        q = deque() # [-cnt, idletime]

        while maxHeap or q:
            time += 1
            if maxHeap:
                cnt = 1 + heapq.heappop(maxHeap)
                if cnt:
                    q.append([cnt, time + n])
            if q and q[0][1] == time:
                heapq.heappush(maxHeap, q.popleft()[0])
        return time
```

- This just might be one of my favourite questions in leetcode. It is so interesting and makes you understand such an important computer science concept trivially: Scheduling.
- I knew what to do in this question i-e making use of Counter and A MaxHeap and then keeping track of idletime. However, I didn't think about using queue data structure.
- I must say my problem solving skills have drastically improved and given I had spent more time on this question, I would have figured it out.
- So, basically, what we are doing is that we are constantly scheduling the most occurring task first and then appending them onto the queue. If the time is equal to the idle time of the first element of the queue, we send it back to the maxHeap. We keep on doing it until the MaxHeap, which is storing the count, is empty.

Problem No 98: Design Twitter
Difficulty: Hard
Topics: Heaps

Design a simplified version of Twitter where users can post tweets, follow/unfollow another user, and is able to see the 10 most recent tweets in the user's news feed.

Implement the Twitter class:

- Twitter() Initializes your twitter object.
- void postTweet(int userId, int tweetId) Composes a new tweet with ID tweetId by the user userId. Each call to this function will be made with a unique tweetId.
- List<Integer> getNewsFeed(int userId) Retrieves the 10 most recent tweet IDs in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user themself. Tweets must be **ordered from most recent to least recent**.
- void follow(int followerId, int followeeId) The user with ID followerId started following the user with ID followeeId.
- void unfollow(int followerId, int followeeId) The user with ID followerId started unfollowing the user with ID followeeId.

```python
class Twitter:
    # My Solution: IT works but is inefficient
    def __init__(self):
        self.userTofollowers = defaultdict(set)
        self.tweetHeap = []
        self.priority = -1

    def postTweet(self, userId: int, tweetId: int) -> None:
        heapq.heappush(self.tweetHeap, (self.priority, tweetId, userId))
        self.priority -= 1

    def getNewsFeed(self, userId: int) -> List[int]:
        res = []
        tempHeap = self.tweetHeap.copy()  # shallow copy of the heap
        count = 0

        while tempHeap and count < 10:
            priority, tweetId, authorId = heapq.heappop(tempHeap)
```

```
            if authorId == userId or authorId in self.userTofollowers[userId]:
                res.append(tweetId)
                count += 1

        return res

    def follow(self, followerId: int, followeeId: int) -> None:
        self.userTofollowers[followerId].add(followeeId)

    def unfollow(self, followerId: int, followeeId: int) -> None:
        self.userTofollowers[followerId].discard(followeeId)


class Twitter:
    # NEETCODE Solution: Efficient
    def __init__(self):
        self.count = 0
        self.tweetMap = defaultdict(list)  # userId -> list of [count, tweetIds]
        self.followMap = defaultdict(set)  # userId -> set of followeeId

    def postTweet(self, userId: int, tweetId: int) -> None:
        self.tweetMap[userId].append([self.count, tweetId])
        self.count -= 1

    def getNewsFeed(self, userId: int) -> List[int]:
        res = []
        minHeap = []
        self.followMap[userId].add(userId)
```

```
        for followeeId in self.followMap[userId]:
            if followeeId in self.tweetMap:
                index = len(self.tweetMap[followeeId]) - 1
                count, tweetId = self.tweetMap[followeeId][index]
                heapq.heappush(minHeap, [count, tweetId, followeeId, index - 1])

        while minHeap and len(res) < 10:
            count, tweetId, followeeId, index = heapq.heappop(minHeap)
            res.append(tweetId)
            if index >= 0:
                count, tweetId = self.tweetMap[followeeId][index]
                heapq.heappush(minHeap, [count, tweetId, followeeId, index - 1])
        return res

    def follow(self, followerId: int, followeeId: int) -> None:
        self.followMap[followerId].add(followeeId)

    def unfollow(self, followerId: int, followeeId: int) -> None:
        if followeeId in self.followMap[followerId]:
            self.followMap[followerId].remove(followeeId)
```

- Wow, first of all, props to me for solving this hard question, without taking any assistance, (even if it was inefficient). Showcases that I know understand how and when to use what data structures and their tradeoffs.
- So, my approach works but there is a main problem with it in terms of efficiency. What I am doing wrong is storing all the tweets in a single heap and then popping all the elements of the heap to check for the relevant tweets and then reading them back. This makes it very inefficient for large scale because in a large scale setting there could be millions of tweets.
- The solution to this problem was making another map which dedicated each user to its tweets. This can then be employed to look for

	the tweets of only those people who you follow and you don't have to go through all the tweets ever posted on the internet.
Problem No 99: Number of Islands Difficulty: Medium Topics: Graphs, BFS, DFS	Given an m x n 2D binary grid grid which represents a map of '1's (land) and '0's (water), return the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

```
def numIslands(self, grid: List[List[str]]) -> int:
        # Time Complexity: O(m * n), Space Complexity: O(m * n)
        # Recursive DFS
        R, C = len(grid), len(grid[0])
        seen = set()
        self.numIslands = 0

        def dfs(r, c):
            # Base Cases
            if (r < 0 or c < 0 or
                r >= R or c >= C or
                (r, c) in seen or
                grid[r][c] == "0"):
                return

            seen.add((r, c))
            dfs(r+1, c)
            dfs(r, c+1)
            dfs(r-1, c)
            dfs(r, c-1)

        for r in range(R):
```

```
            for c in range(C):
                if grid[r][c] == "1" and (r, c) not in seen:
                    self.numIslands += 1
                    dfs(r, c)

        return self.numIslands

    # Iterative BFS
    def numIslands(self, grid: List[List[str]]) -> int:
        R, C = len(grid), len(grid[0])
        visited = set()
        numIslands = 0

        def bfs(r, c):
            q = collections.deque()
            visited.add((r, c))
            q.append((r, c))
            while q:

                row, col = q.popleft()
                directions = [[0, 1], [1, 0], [-1, 0], [0, -1]]

                for dr, dc in directions:
                    r, c = row + dr, col + dc
                    if (r >= 0 and c >= 0 and
                        r < R and c < C and
                        grid[r][c] == "1" and
                        (r, c) not in visited):
```

```
                    q.append((r, c))
                    visited.add((r, c))

        for r in range(R):
            for c in range(C):
                if grid[r][c] == "1" and (r, c) not in visited:
                    bfs(r, c)
                    numIslands += 1

        return numIslands
```

- This is an important question because it allows us to know a template for 2D matrix graph questions and how to run not only Recursive DFS (which we also did in Word Search) but also Iterative DFS and BFS on these types of questions.

Problem No 100: Max Area of Island
Difficulty: Medium
Topics: Graphs, BFS, DFS

You are given an m x n binary matrix grid. An island is a group of 1's (representing land) connected **4-directionally** (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

The **area** of an island is the number of cells with a value 1 in the island.

Return *the maximum **area** of an island in the grid*. If there is no island, return 0.

```
def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        if not grid: return 0

        visited = set()
        max_area = 0
        R, C = len(grid), len(grid[0])
        # self.area = 0
```

```
        def dfs(r, c):
            # Base Cases
            if (r < 0 or c < 0 or
                r >= R or c >= C or
                (r, c) in visited or
                grid[r][c] == 0):
                return 0

            # self.area += 1
            area = 1
            visited.add((r, c))
            area += dfs(r+1, c)
            area += dfs(r, c+1)
            area += dfs(r-1, c)
            area += dfs(r, c-1)
            return area

        for r in range(R):
            for c in range(C):
                if grid[r][c] == 1 and (r, c) not in visited:
                    area = dfs(r, c)
                    max_area = max(area, max_area)
                    # dfs(r, c)
                    # max_area = max(self.area, max_area)
                    # self.area = 0

        return max_area
```

- Really similar to the number of islands in question. It was the same algorithm with just a few changes. I did both the member variable method (commented) and then the return method. Honestly, the return method is more interesting and looks nice but the member variable shows greater knowledge. However, both methods work and one should be able to do both.
- YAYYY!!! HAPPY 100 REPITITIONS!!!!!

Problem No 101: Clone Graph
Difficulty: Medium
Topics: Graphs

Given a reference of a node in a **connected** undirected graph.

Return a **deep copy** (clone) of the graph.

Each node in the graph contains a value (int) and a list (List[Node]) of its neighbors.

```
class Node {

    public int val;

    public List<Node> neighbors;

}
```

Test case format:

For simplicity, each node's value is the same as the node's index (1-indexed). For example, the first node with val == 1, the second node with val == 2, and so on. The graph is represented in the test case using an adjacency list.

An adjacency list is a collection of unordered **lists** used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.

The given node will always be the first node with val = 1. You must return the **copy of the given node** as a reference to the cloned graph.

```
def cloneGraph(self, node: Optional['Node']) -> Optional['Node']:
        # Iterative DFS (1)
        if not node:
            return None
```

```
        # Map original nodes to their clones
        old_to_new = {}

        # Initialize stack with the first node
        stack = [node]

        # Create the clone for the first node
        old_to_new[node] = Node(node.val)

        while stack:
            current = stack.pop()

            for neighbor in current.neighbors:
                if neighbor not in old_to_new:
                    # Clone the neighbor
                    old_to_new[neighbor] = Node(neighbor.val)
                    # Add to stack to continue traversal
                    stack.append(neighbor)

                # Link the cloned current node to the cloned neighbor
                old_to_new[current].neighbors.append(old_to_new[neighbor])

        # Return the clone of the starting node
        return old_to_new[node]

        # Recursive DFS (2)
        old_to_new = {}
```

```
        def dfs(node):
            if node in old_to_new:
                return old_to_new[node]

            copy = Node(node.val)
            old_to_new[node] = copy
            for nei in node.neighbors:
                copy.neighbors.append(dfs(nei))
            return copy

        return dfs(node) if node else None
```

- Pretty simple problem. The only and main realization was that you have to use a map.
- It was stupid of me to not realize that a map is required given that we had done a similar question in linkedlists.
- Tip: Whenever you are asked for a deep copy, think of creating a map between the old and the new.

Problem No 102: Islands and Treasures (Gates and Walls)
Difficulty: Medium
Topics: Graphs, Multisource BFS, DFS

You are given a

m×n

m×*n* 2D grid initialized with these three possible values:

1. -1 - A water cell that *can not* be traversed.
2. 0 - A treasure chest.
3. INF - A land cell that *can* be traversed. We use the integer 2^31 - 1 = 2147483647 to represent INF.

Fill each land cell with the distance to its nearest treasure chest. If a land cell cannot reach a treasure chest then the value should remain INF.

Assume the grid can only be traversed up, down, left, or right.

Modify the grid **in-place**.

```
def islandsAndTreasure(self, grid: List[List[int]]) -> None:
        # Less Optimal My Way (Time Complexity: O(k * m * n))
        if not grid: return None

        R, C = len(grid), len(grid[0])
        directions = [[0, 1], [0, -1], [-1, 0], [1, 0]]
        visited = set()

        def bfs(r, c):
            q = collections.deque()
            q.append((r, c, 0))
            visited.add((r, c))

            while q:
                row, col, dist = q.popleft()
                for dr, dc in directions:
                    new_row, new_col = row + dr, col + dc
                    if (0 <= new_row < R and 0 <= new_col < C and
                        grid[new_row][new_col] != -1 and
```

```
                    (new_row, new_col) not in visited):

                    if dist + 1 < grid[new_row][new_col]:
                        grid[new_row][new_col] = dist + 1
                    visited.add((new_row, new_col))
                    q.append((new_row, new_col, dist + 1))

    for row in range(R):
        for col in range(C):
            if grid[row][col] == 0:
                bfs(row, col)
                visited.clear()

    # MULTISOURCE BFS (OPTIMAL: TIME COMPLEXITY: O(m*n))

    ROWS, COLS = len(grid), len(grid[0])
    visit = set()
    q = deque()

    def addCell(r, c):
        if (min(r, c) < 0 or r == ROWS or c == COLS or
            (r, c) in visit or grid[r][c] == -1):
```

```
                return
            visit.add((r, c))
            q.append((r, c))

        for r in range(ROWS):
            for c in range(COLS):
                if grid[r][c] == 0:
                    q.append((r, c))
                    visit.add((r, c))

        dist = 0
        while q:
            for i in range(len(q)):
                r, c = q.popleft()
                grid[r][c] = dist
                addCell(r+1, c)
                addCell(r-1, c)
                addCell(r, c+1)
                addCell(r, c-1)
            dist += 1
```

- This is a really nice problem for multiple reasons and I learnt a few things: First I learnt how to multisource BFS and how it is better

	than calling BFS on every single gate or 0. Then, it learnt the different approaches of actually carrying out the BFS. We could have also done DFS but it was not very optimal (i.e the time complexity would have been O(m*n)^2)
Problem No 103: Rotting Oranges Difficulty: Medium Topics: Graphs, Multisource BFS	You are given an m x n grid where each cell can have one of three values: • 0 representing an empty cell, • 1 representing a fresh orange, or • 2 representing a rotten orange. Every minute, any fresh orange that is **4-directionally adjacent** to a rotten orange becomes rotten. Return *the minimum number of minutes that must elapse until no cell has a fresh orange*. If *this is impossible, return* -1.

```python
def orangesRotting(self, grid: List[List[int]]) -> int:
        # Algorithm: Multisource BFS
        # Time Complexity: O(m*n), Space Complexity: O(m*n)
        R, C = len(grid), len(grid[0])
        visit = set()
        directions = [[0, 1], [0, -1], [1, 0], [-1, 0]]
        q = collections.deque()

        for r in range(R):
            for c in range(C):
                if grid[r][c] == 2:
                    q.append((r, c))
                    visit.add((r, c))

        time = -1 # Start at -1 so the first level (already rotten) counts as 0
        while q:
            for i in range(len(q)):
```

```
                row, col = q.popleft()
                for drr, drc in directions:
                    new_row, new_col = row + drr, col + drc
                    if (0 <= new_row < R and 0 <= new_col < C and
                        grid[new_row][new_col] != 0 and
                        (new_row, new_col) not in visit):

                        grid[new_row][new_col] = 2
                        visit.add((new_row, new_col))
                        q.append((new_row, new_col))
            time += 1

        for r in range(R):
            for c in range(C):
                if grid[r][c] == 1:
                    return -1

        return max(time, 0) #basically in the case that no oranges were ever rotten
```

- Did the problem myself completely. Pretty amazing and I am actually loving these problems since I am able to do them.
- Its almost the same as the previous problem and basically you have to use a multisource bfs algorithm. I deliberately did without using the helper function and I actually prefer this approach.

Problem No 104: Pacific Atlantic Water Flow
Difficulty: Medium
Topics: Graphs, Two Sets DFS, BFS

There is an m x n rectangular island that borders both the **Pacific Ocean** and **Atlantic Ocean**. The **Pacific Ocean** touches the island's left and top edges, and the **Atlantic Ocean** touches the island's right and bottom edges.

The island is partitioned into a grid of square cells. You are given an m x n integer matrix heights where heights[r][c] represents the **height above sea level** of the cell at coordinate (r, c).

The island receives a lot of rain, and the rain water can flow to neighboring cells directly north, south, east, and west if the neighboring cell's

height is **less than or equal to** the current cell's height. Water can flow from any cell adjacent to an ocean into the ocean.

Return *a **2D list** of grid coordinates* result *where* result[i] = [r_i, c_i] *denotes that rain water can flow from cell* (r_i, c_i) *to **both** the Pacific and Atlantic oceans*.

```
def pacificAtlantic(self, heights: List[List[int]]) -> List[List[int]]:
        # Time Complexity and Space Complexity: O(M*N)
        R, C = len(heights), len(heights[0])
        pacific_reacheable = set()
        atlantic_reacheable = set()

        def dfs(r, c, visited, prevHeight):
            # Base Cases
            if (r < 0 or c < 0 or
                r >= R or c >= C or
                (r, c) in visited or
                heights[r][c] < prevHeight):
                return

            visited.add((r, c))
            dfs(r+1, c, visited, heights[r][c])
            dfs(r-1, c, visited, heights[r][c])
            dfs(r, c+1, visited, heights[r][c])
            dfs(r, c-1, visited, heights[r][c])

        # (0 row and 0 col is Pacific and R-1 row and C-1 col is Atlantic)
        for c in range(C):
            dfs(0, c, pacific_reacheable, heights[0][c]) # Top Row --> Pacific
            dfs(R-1, c, atlantic_reacheable, heights[R-1][c]) # Bottom Row --> Atlantic
```

```
        for r in range(R):
            dfs(r, 0, pacific_reacheable, heights[r][0]) # Left Col --> Pacific
            dfs(r, C-1, atlantic_reacheable, heights[r][C-1]) # Right Col --> Atlantic

        # looking for intersection between the two sets
        result = []
        for r in range(R):
            for c in range(C):
                if (r, c) in pacific_reacheable and (r, c) in atlantic_reacheable:
                    result.append((r, c))

        return result
```

- This has to be one of my most favorite questions I have solved (Not really but I was really close)
- The idea of making two sets is fascinating and how we are finding the intersection between two sets is so cool
- What I was trying to do was to run the dfs on every cell and then see if the right coordinates exist in the visited cell.

Problem No 105:
Surrounded Regions
Difficulty: Medium
Topics: Graphs, Border DFS, BFS

You are given an m x n matrix board containing **letters** 'X' and 'O', **capture regions** that are **surrounded**:

- **Connect**: A cell is connected to adjacent cells horizontally or vertically.
- **Region**: To form a region **connect every** 'O' cell.
- **Surround**: The region is surrounded with 'X' cells if you can **connect the region** with 'X' cells and none of the region cells are on the edge of the board.

To capture a **surrounded region**, replace all 'O's with 'X's **in-place** within the original board. You do not need to return anything.

```
def solve(self, board: List[List[str]]) -> None:
        # Time Complexity, Space Complexity: O(R * C)
        # Approach One: My approach
        """
        Do not return anything, modify board in-place instead.
```

```
        """
        if not board or not board[0]:
            return

        R, C = len(board), len(board[0])
        visited = set()

        def dfs(r, c):
            nonlocal on_edge
            if (r < 0 or r >= R or c < 0 or c >= C or
                (r, c) in visited or board[r][c] != "O"):
                return

            visited.add((r, c))
            just_visited.add((r, c))
            if r == 0 or r == R - 1 or c == 0 or c == C - 1:
                (can also write if r in [0, R-1] or c in [0, C-1])
                on_edge = True

            dfs(r + 1, c)
            dfs(r - 1, c)
            dfs(r, c + 1)
            dfs(r, c - 1)

        for r in range(R):
            for c in range(C):
                if board[r][c] == "O" and (r, c) not in visited:
                    just_visited = set()
```

```
                on_edge = False
                dfs(r, c)
                if not on_edge:
                    for row, col in just_visited:
                        board[row][col] = "X"

# Approach 2: More Efficient Approach → Reverse Thinking
def solve(self, board: List[List[str]]) -> None:
    if not board or not board[0]:
        return

    R, C = len(board), len(board[0])

    def dfs(r, c):
        if r < 0 or r >= R or c < 0 or c >= C:
            return
        if board[r][c] != 'O':
            return

        board[r][c] = 'S'  # Mark as safe
        dfs(r + 1, c)
        dfs(r - 1, c)
        dfs(r, c + 1)
        dfs(r, c - 1)

    # 1. Start DFS from all border 'O's
    for r in range(R):
        if board[r][0] == 'O':
```

```
                dfs(r, 0)
            if board[r][C - 1] == 'O':
                dfs(r, C - 1)
        for c in range(C):
            if board[0][c] == 'O':
                dfs(0, c)
            if board[R - 1][c] == 'O':
                dfs(R - 1, c)

        # 2. Flip interior 'O's to 'X' and restore 'S' to 'O'
        for r in range(R):
            for c in range(C):
                if board[r][c] == 'O':
                    board[r][c] = 'X'
                elif board[r][c] == 'S':
                    board[r][c] = 'O'
```

- So, there are two approaches to solving this problem and the second one is slightly more efficient.
- In the first approach (my approach), you are basically going to each zero (i.e. region) and running dfs on it and checking whether the row and column isn't that of the boundary and if so setting on_edge Boolean True.
- The second approach is more efficient because we just run the dfs on the 0's at the edges and mark them as Safe("S"). Then we are going through every coordinate in the board and replacing every region with X that is not safe.

Problem No 106: Course Schedule
Difficulty: Medium
Topics: Graphs, Topological Sorts, Adjacency Lists, DFS, BFS

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [a_i, b_i] indicates that you **must** take course b_i first if you want to take course a_i.

- For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return true if you can finish all courses. Otherwise, return false.

```python
def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
    # Time Complexity, Space Complexity: O(N)
    adjList = defaultdict(list)

    for course, prereq in prerequisites:
        adjList[course].append(prereq)

    visited = set()
    def dfs(course):
        if course in visited:
            return False
        if adjList[course] == []:
            return True

        visited.add(course)
        for neighbors in adjList[course]:
            if not dfs(neighbors): return False

        visited.remove(course)
        adjList[course] = []
        return True

    for course in range(numCourses):
        if not dfs(course): return False
    return True
```

- Definitely not a hard problem. It was stupid of me to not read the question properly.
- I knew I had to make the adjacency list but was unsure how to carry out dfs.
- First of its kind: Topological Sort.

Problem No 107: Course Schedule II
Difficulty: Medium
Topics: Graphs, Topological Sort, DFS, Two Sets

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array of prerequisites where prerequisites[i] = [a_i, b_i] indicates that you **must** take course b_i first if you want to take course a_i.

- For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return *the ordering of courses you should take to finish all courses*. If there are many valid answers, return **any** of them. If it is impossible to finish all courses, return **an empty array**.

```
def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[int]:
        adjList = defaultdict(list)

        for course, prereq in prerequisites:
            adjList[course].append(prereq)

        visit = set()
        cycle = set()
        res = []

        def dfs(crs):
            if crs in cycle:
                return False
            if crs in visit:
                return True

            cycle.add(crs)
            for prereq in adjList[crs]:
                if not dfs(prereq): return False
            cycle.remove(crs)
            visit.add(crs)
            res.append(crs)
```

```
            return True

        for crs in range(numCourses):
            if not dfs(crs): return []
        return res
```

- The important thing to learn from this question is how we are using the visit and cycle set. This question is an epitome of a topological sort problem. The visit set is useful in the sense that if we have already visited one course, we don't have to add it again to the res variable and just return True.

Problem No 108: Graph Valid Tree
Difficulty: Medium
Topics: Graphs, DFS, Trees

Given n nodes labeled from 0 to n - 1 and a list of **undirected** edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

```
from collections import defaultdict

    def validTree(self, n: int, edges: List[List[int]]) -> bool:
        connections = defaultdict(list)

        for source, destination in edges:
            connections[source].append(destination)
            connections[destination].append(source)

        visited = set()
        def dfs(node, parent):
            if node in visited: return False

            visited.add(node)
            for nei in connections[node]:
```

```
                if nei == parent: continue
                if not dfs(nei, node): return False

            return True

        if not dfs(0,-1) or len(visited) != n: return False

        return True
```

- One important thing to realize was that you needed to check two conditions: Connectivity and Cycle. In an undirected graph, you make sure that you add both edges in the adjacency list. Then in the dfs you need to make sure that if you get to a parent, you need to skip that iteration. These are the only main important things to learn from this question.

Problem No 109: Number of Connected Components in an Undirected Graph
Difficulty: Medium
Topics: Graphs, Union Find, DFS

There is an undirected graph with n nodes. There is also an `edges` array, where `edges[i] = [a, b]` means that there is an edge between node a and node b in the graph.

The nodes are numbered from `0` to `n - 1`.

Return the total number of connected components in that graph.

```
def countComponents(self, n: int, edges: List[List[int]]) -> int:
        Method 1: DFS: Time and Space Complexity: O(V+E)
        visited = set()
        no_of_CC = 0

        adj = defaultdict(list)
        for source, destination in edges:
            adj[source].append(destination)
```

```
            adj[destination].append(source)

        def dfs(node, parent):
            if node in visited:
                return

            visited.add(node)
            for nei in adj[node]:
                if nei == parent: continue
                dfs(nei, node)

        for nodes in range(n):
            if nodes in visited:
                continue
            dfs(nodes, -1)
            no_of_CC += 1
            if len(visited) == n:
                return no_of_CC

        # Method 2: UNION FIND (IMP ALGORITHM TO KNOW)
        # AS EFFICIENT AS IT GETS
    def countComponents(self, n: int, edges: List[List[int]]) -> int:
        parent = [i for i in range(n)]
        rank = [1] * n

        def find(n1):
```

```
            res = n1

            while res != parent[res]:
                parent[res] = parent[parent[res]] # Optimization basically...
                res = parent[res]
            return res

        def union(n1, n2):
            p1, p2 = find(n1), find(n2)

            if p1 == p2:
                return 0

            if rank[p1] > rank[p2]:
                parent[p2] = p1
                rank[p1] += rank[p2]
            else:
                parent[p1] = p2
                rank[p2] += rank[p1]
            return 1

        no_of_CC = n
        for n1, n2 in edges:
            no_of_CC -= union(n1, n2)
        return no_of_CC
```

	- Learnt a new algorithm from this question: UNION FIND - Basically in union find, we set the number of connected components to the number of nodes initially and as we go through the edges, we decrement if we make a new connection. We also maintain the parents and rank (size) of the connected components so that we can make appropriate connections and decrement when it is actually a new connection rather than a connection which would increase the rank(size) of an existing connected component. - The DFS method was pretty chill. Coded it in one go. Just going through every node and skipping those that are already visited. If I do find a node, I just return. After each dfs, we basically increase the number of connected components. Once, the length of the visited is equal to the number of nodes, we return the no of connected components.
Problem No 110: Redundant Connection Difficulty: Medium Topics: Graphs	In this problem, a tree is an **undirected graph** that is connected and has no cycles. You are given a graph that started as a tree with n nodes labeled from 1 to n, with one additional edge added. The added edge has two **different** vertices chosen from 1 to n, and was not an edge that already existed. The graph is represented as an array of edges of length n where edges[i] = [a_i, b_i] indicates that there is an edge between nodes a_i and b_i in the graph. Return *an edge that can be removed so that the resulting graph is a tree of* n *nodes*. If there are multiple answers, return the answer that occurs last in the input. `def findRedundantConnection(self, edges: List[List[int]]) -> List[int]:` `N = len(edges)` `par = [i for i in range(N + 1)]` `rank = [1] * (N + 1)` `def find(n):` `res = n` `while res != par[res]:` `par[res] = par[par[res]]` `res = par[res]` `return res`

```
        def union(n1, n2):
            p1, p2 = find(n1), find(n2)
            if p1 == p2:
                return False

            if rank[p1] > rank[p2]:
                par[p2] = p1
                rank[p1] += rank[p2]
            else:
                par[p1] = p2
                rank[p2] += rank[p1]
            return True

        for n1, n2 in edges:
            if not union(n1, n2):
                return [n1, n2]
```

- The most important thing that I learnt from this question is that the Union Find can be used to detect cycles.
- The other approach where you would store the visited edges and then go reverse order from edges to see if the edge exists is a valid solution as well and that's what I was trying to implement and almost implemented.

Problem No 111: Word Ladder Difficulty: Hard Topics: Graphs	
Problem No 112: Min Cost Climbing Stairs Difficulty: Easy Topics: 1D Dynamic Programming, Recursion	You are given an integer array cost where cost[i] is the cost of i_{th} step on a staircase. Once you pay the cost, you can either climb one or two steps. You can either start from the step with index 0, or the step with index 1.

Return *the minimum cost to reach the top of the floor*.

```
def minCostClimbingStairs(self, cost: List[int]) -> int:
        [10, 15, 20][0]
        cost.append(0)

        for i in range(len(cost) - 3, -1, -1):
            cost[i] = min(cost[i] + cost[i + 1], cost[i] + cost[i + 2])

        return min(cost[0], cost[1])
        # Time: O(n), Space: O(1)
```

Problem No 113: House Robber
Difficulty: Medium
Topics: 1D Dynamic Programming, Recursion

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array nums representing the amount of money of each house, return *the maximum amount of money you can rob tonight **without alerting the police***.

```
def rob(self, nums: List[int]) -> int:
        # Top Down --> Memoization --> Time: O(n), Space: O(n)
        memo = [-1] * len(nums)

        def dfs(i):
            if i >= len(nums):
                return 0
            if memo[i] != -1:
                return memo[i]
            else:
```

```
            memo[i] = max(dfs(i + 1), nums[i] + dfs(i + 2))
            return memo[i]

        return dfs(0)

        # Bottom-up --> Tabulation
        if not nums:
            return 0
        if len(nums) == 1:
            return nums[0]

        dp = [0] * len(nums)
        dp[0] = nums[0]
        dp[1] = max(nums[0], nums[1])

        for i in range(2, len(nums)):
            dp[i] = max(dp[i - 2] + nums[i], dp[i - 1])

        return dp[-1]

        # Variables Bottom-up --> Constant Space
        rob1, rob2 = 0, 0

        for num in nums:
            temp = max(num + rob1, rob2)
            rob1 = rob2
            rob2 = temp
        return rob2
```

	- I have done all three methods and the answer is pretty self explanatory. - Just remember to use SRTBOT if you get stuck. - First try greedy for most of these problems but if that doesnt work then look for DP. - Just remember DP is nothing but local brute force. - Common subproblems for one dimension dp include: prefixes and suffixes.
Problem No 114: House Robber II Difficulty: Medium Topics: 1D Dynamic Programming, Recursion	You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have a security system connected, and **it will automatically contact the police if two adjacent houses were broken into on the same night**. Given an integer array nums representing the amount of money of each house, return *the maximum amount of money you can rob tonight **without alerting the police***. `def rob(self, nums: List[int]) -> int:` `  # Time: O(n), Space: O(1)` `  def helper_rob(nums):` `  rob1, rob2 = 0, 0` `  for num in nums:` `  temp = max(num + rob1, rob2)` `  rob1 = rob2` `  rob2 = temp` `  return rob2` `  sub_num1 = helper_rob(nums[0 : len(nums) - 1])` `  sub_num2 = helper_rob(nums[1 :])` `  return max(sub_num1, sub_num2, nums[0])` - Very similar to house robber 1. You just had to realize that you have to run it on the two subarrays and pick the max out of those two.

Problem No 115: Palindrome Substrings
Difficulty: Medium
Topics: 1D Dynamic Programming, Recursion

Given a string s, return *the number of **palindromic substrings** in it*.

A string is a **palindrome** when it reads the same backward as forward.

A **substring** is a contiguous sequence of characters within the string.

```python
def countSubstrings(self, s: str) -> int:
        res = 0

        for i in range(len(s)):
            l, r = i, i # for odd length
            while l >= 0 and r < len(s) and s[l] == s[r]:
                l -= 1
                r += 1
                res += 1

            l, r = i, i + 1 # for even length
            while l >= 0 and r < len(s) and s[l] == s[r]:
                l -= 1
                r += 1
                res += 1

        return res
```

- The same code as the longest palindrome substring. You just had to tweak one or two things.
- This code is better than the DP one since its O(n^2) time (same as dp) but O(1) space (dp is O(n^2)).

Problem No 116: Decode Ways
Difficulty: Medium
Topics: 1D Dynamic

You have intercepted a secret message encoded as a string of numbers. The message is **decoded** via the following mapping:
"1" -> 'A'

"2" -> 'B'

Programming, Recursion

...

"25" -> 'Y'

"26" -> 'Z'
However, while decoding the message, you realize that there are many different ways you can decode the message because some codes are contained in other codes ("2" and "5" vs "25").
For example, "11106" can be decoded into:

- "AAJF" with the grouping (1, 1, 10, 6)
- "KJF" with the grouping (11, 10, 6)
- The grouping (1, 11, 06) is invalid because "06" is not a valid code (only "6" is valid).

Note: there may be strings that are impossible to decode.

Given a string s containing only digits, return the **number of ways** to **decode** it. If the entire string cannot be decoded in any valid way, return 0.
The test cases are generated so that the answer fits in a **32-bit** integer.

```
def numDecodings(self, s: str) -> int:
        # Time and Space: O(N)
        dp = { len(s) : 1 }

        def dfs(i):
            if i in dp:
                return dp[i]
            if s[i] == "0":
                return 0

            res = dfs(i + 1)
            if (i + 1 < len(s) and (s[i] == "1" or
                s[i] == "2" and s[i + 1] in "0123456")):
```

```
                res += dfs(i + 2)

            dp[i] = res
            return res

        return dfs(0)
```

- Okay so lets be real, I wouldn't have been able to solve this question. I did, however, figure out the conditions and the decision tree. It was the coding aspect that I couldn't get and also couldn't make the recurrence relation.
- The if condition part is really interesting.
- Also, the first initialization of the dp cache is something to notice as well.

Problem No 117: Coin Change
Difficulty: Medium
Topics: 1D Dynamic Programming, Recursion

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

```
def coinChange(self, coins: List[int], amount: int) -> int:
        memo = {} # amount : no of min coins needed

        def dfs(amount):
            if amount == 0:
                return 0
            if amount in memo:
                return memo[amount]

            res = 1e9
            for coin in coins:
```

```
                if amount - coin >= 0:
                    res = min(res, 1 + dfs(amount - coin))

            memo[amount] = res
            return res

        minCoins = dfs(amount)
        return -1 if minCoins >= 1e9 else minCoins
```

- So, the main thing is that I need to treat DP questions firstly as recursive. I need to form the recursive solutions and then worry about the DP aspect.
- DECISION TREES are as important as we can. If you form a decision tree, then the questions become substantially easy.
- The first thing that I tried was greedy and then disproved it quickly. The best way to find if an algorithm is greedy is to assume greedy and then find a counterexample.

Problem No 118: Maximum Product Subarray
Difficulty: Medium
Topics: 1D Dynamic Programming, Recursion

Given an integer array nums, find a subarray that has the largest product, and return *the product*.

The test cases are generated so that the answer will fit in a **32-bit** integer.

```
def maxProduct(self, nums: List[int]) -> int:
        # Time, Space = O(n), O(1) (DP SOLUTION)
        if not nums: return 0

        max_so_far = nums[0]
        min_so_far = nums[0]
        global_max = nums[0]

        for i in range(1, len(nums)):
            prev_max = max_so_far
            curr = nums[i]
```

```
            max_so_far = max(curr, curr * prev_max, curr * min_so_far)
            min_so_far = min(curr, curr * prev_max, curr * min_so_far)

            global_max = max(global_max, max_so_far)

        return global_max
```

- A really interesting solution I must say. You could have also used the prefix and suffix sums and that would have given us the same time and space complexity.
- So, we don't have to deal with 0 separately bcz choosing the maximum between all the values already clears the zero and hence we don't end up losing our streak.
- Dp problems doesn't mean always having a cache memo or a dp array. In this problem, we just needed variables.

Problem No 119: Word Break
Difficulty: Medium
Topics: 1D Dynamic Programming, Recursion

Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

```
def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        # Greedy - doesn't work for inputs requiring backtracking
        letterToWord = defaultdict(list)

        for word in wordDict:
            letterToWord[word[0]].append(word)

        curr = 0
        while curr < len(s):
            if s[curr] not in letterToWord:
                return False

            found = False
```

```
            for eword in letterToWord[s[curr]]:
                length = len(eword)
                if s[curr : curr + length] == eword:
                    curr += length
                    found = True
                    break

            if not found:
                return False

        return True

        # DP --> Most Efficient
        dp = [False] * (len(s) + 1)
        dp[len(s)] = True

        for i in range(len(s) - 1, -1, -1):
            for w in wordDict:
                if (i + len(w)) <= len(s) and s[i : i + len(w)] == w:
                    dp[i] = dp[i + len(w)]
                if dp[i]:
                    break

        return dp[0]
```

- A really interesting problem. I did not think we could make a decision tree off of this but I was surely wrong.
- The solution I came up with worked with some inputs but not all (those involving backtracking)
- The brute force I could do as well.
- This solution, however, I couldn't figure out. Clearly, I would need a lot more practice with dynamic programming questions. Lets try

	and treat it the way neetcode does and first come with a decision tree and if that is not possible then use SRTBOT.
Problem No 120: Longest Increasing Subsequence (LIS) Difficulty: Medium Topics: 1D Dynamic Programming, Recursion	Given an integer array nums, return *the length of the longest* ***strictly increasing subsequence***. `def lengthOfLIS(self, nums: List[int]) -> int:` `  # Recursive Top Down DP` `  memo = {}` `  def LIS(i):` `  if i in memo:` `  return memo[i]` `  max_len = 1  # at least the element itself` `  for j in range(i + 1, len(nums)):` `  if nums[i] < nums[j]:` `  max_len = max(max_len, 1 + LIS(j))` `  memo[i] = max_len` `  return max_len` `  return max(LIS(i) for i in range(len(nums)))` `  # Bottom Up DP` `  LIS = [1] * len(nums)` `  for i in range(len(nums) - 1, -1, -1):` `  for j in range(i + 1, len(nums)):` `  if nums[j] > nums[i]:`

```
                    LIS[i] = max(LIS[i], 1 + LIS[j])


        return max(LIS)
```

- This is one of the classical dynamic programming questions. It tells us how sometimes we have to use the concept of subproblems constraint. Basically, with the basic subproblem: Let L(i) be the max length of the longest increasing subsequence i onwards, we would be making a lot of assumptions.
- On the other hand, by adding the following constraint: Let L(i) be the max length of LIS at that particular i, we basically run L(i) for every single i and return the max of that.

Problem No 121: Partition Equal Subset Sum
Difficulty: Medium
Topics: 1D Dynamic Programming, Recursion

Given an integer array nums, return true *if you can partition the array into two subsets such that the sum of the elements in both subsets is equal or* false *otherwise.*

```
def canPartition(self, nums: List[int]) -> bool:
        # # Top-Down Memoization
        # Time, Space = O(n * sum(nums)), O(n * sum(nums))

        memo = {} # (index, curr_sum) --> True/False
        max_sum = sum(nums)
        if max_sum % 2 != 0:
            return False

        target = max_sum / 2

        def dfs(i, curr_sum):
            if curr_sum == target:
                return True
            if curr_sum > target or i >= len(nums):
                return False
            if (i, curr_sum) in memo:
```

```
            return memo[(i, curr_sum)]

        if dfs(i + 1, curr_sum + nums[i]):
            memo[(i, curr_sum)] = True
            return True

        if dfs(i + 1, curr_sum):
            memo[(i, curr_sum)] = True
            return True

        memo[(i, curr_sum)] = False
        return False

    return dfs(0, 0)

    # Bottom-up Approach
    # Time, Space = O(n * sum(nums))

    if sum(nums) % 2:
        return False

    dp = set()
    dp.add(0)
    target = sum(nums) // 2

    for i in range(len(nums) - 1, -1, -1):
        nextDP = set()
        for t in dp:
```

```
                nextDP.add(t + nums[i])
                nextDP.add(t)
            dp = nextDP

        return True if target in dp else False
```

- I clearly saw a difference when I tried backtracking and the decision tree concept instead of SRTBOT.
- First, always see the brute force using the decision tree and then optimize it using cache. Then try and figure out a way for a bottom up solution using DP.

Problem No 122: Insert Intervals
Difficulty: Medium
Topics: Intervals, Arrays

You are given an array of non-overlapping intervals intervals where intervals[i] = [$start_i$, end_i] represent the start and the end of the i^{th} interval and intervals is sorted in ascending order by $start_i$. You are also given an interval newInterval = [start, end] that represents the start and end of another interval.

Insert newInterval into intervals such that intervals is still sorted in ascending order by $start_i$ and intervals still does not have any overlapping intervals (merge overlapping intervals if necessary).

Return intervals *after the insertion*.

Note that you don't need to modify intervals in-place. You can make a new array and return it.

```
def insert(self, intervals: List[List[int]], newInterval: List[int]) -> List[List[int]]:
        nst, net = newInterval
        res = []
        placed = False

        for st, et in intervals:
            if et < nst:  # No overlap, interval comes before newInterval
                res.append([st, et])
            elif st > net:  # No overlap, interval comes after newInterval
                if not placed:
                    res.append([nst, net])
```

```
                    placed = True
                res.append([st, et])
            else:  # Overlap case
                nst = min(nst, st)
                net = max(net, et)

        if not placed:
            res.append([nst, net])

        return res
```

- This was not a hard problem. Visualization was really important and apart from that, you just had to deal with 3 cases:
- No overlap and NewInterval comes before the interval; No overlap and NewInterval comes after the interval; Overlap case.

Problem No 123: Merge Intervals
Difficulty: Medium
Topics: Intervals, Arrays, Sorting

Given an array of intervals where intervals[i] = [$start_i$, end_i], merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input*.

```
def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        # Time, Space = O(nlogn), O(n)
        intervals.sort()
        nst, net = intervals[0]
        res = []

        for i in range(1, len(intervals)):
            st, et = intervals[i]
            if st > net:
                res.append([nst, net])
                nst, net = st, et
            else: # overlap
                nst = min(nst, st)
```

```
                net = max(net, et)

        res.append([nst, net])
        return res
```

- This is a pretty chill question. The two main realizations were that you had to sort the intervals in non-decreasing order based on the start time and that you had to keep the process of overlapping going.

Problem No 124: Non Overlapping Intervals
Difficulty: Medium
Topics: Intervals, Greedy, Arrays, Sorting

Given an array of intervals intervals where intervals[i] = [start$_i$, end$_i$], return *the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping*.

Note that intervals which only touch at a point are **non-overlapping**. For example, [1, 2] and [2, 3] are non-overlapping.

```
def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        # Sort by start time
        intervals.sort()

        # Initialize with first interval
        prev_start, prev_end = intervals[0]
        removals = 0

        for i in range(1, len(intervals)):
            curr_start, curr_end = intervals[i]

            if curr_start < prev_end:  # Overlap
                removals += 1
                # Keep the interval with the smaller end time (greedy choice)
                prev_end = min(prev_end, curr_end)
            else:
                prev_end = curr_end  # No overlap, move to next
```

```
        return removals
```

- The most important thing is visualization. Apart from that pretty chill problem.
- Honestly, I would recommend that you keep in mind this template for any interval problem.
- Time, Space = O(nlogn), O(n)

Problem No 125: Meeting Rooms
Difficulty: Easy
Topics: Intervals

Given an array of meeting time interval objects consisting of start and end times [[start_1,end_1],[start_2,end_2],...] (start_i < end_i), determine if a person could add all meetings to their schedule without any conflicts.

```
Definition of Interval:
class Interval(object):
    def __init__(self, start, end):
        self.start = start
        self.end = end
"""

class Solution:
    def canAttendMeetings(self, intervals: List[Interval]) -> bool:
        if not intervals: return True
        intervals = sorted(intervals, key = lambda x : x.start)
        prev_st, prev_et = intervals[0].start, intervals[0].end

        for i in range(1, len(intervals)):
            curr_st, curr_et = intervals[i].start, intervals[i].end

            if curr_st < prev_et:
                # OVERLAP
                return False
            else:
```

```
                prev_st, prev_et = curr_st, curr_et

        return True
```

- Learned two important things: How to deal when intervals are given as a class and how to sort them using lambda.

Problem No 126: Meeting Rooms II
Difficulty: Medium
Topics: Intervals

Given an array of meeting time interval objects consisting of start and end times [[start_1,end_1],[start_2,end_2],...] (start_i < end_i), find the minimum number of days required to schedule all meetings without any conflicts.

Note: (0,8),(8,10) is not considered a conflict at 8.

```
def minMeetingRooms(self, intervals: List[Interval]) -> int:
        start = sorted([i.start for i in intervals])
        end = sorted([i.end for i in intervals])

        s, e = 0, 0
        res, count = 0, 0

        while s < len(intervals):
            if start[s] < end[e]:
                s += 1
                count += 1
            else:
                e += 1
                count -= 1
            res = max(res, count)
        return res
```

- A really interesting problem. Again, the main thing is to visualize. The idea of making two lists, start and end, was intriguing.

Problem No 127: Single Number
Difficulty: Easy
Topics: Bit Manipulation

Given a **non-empty** array of integers nums, every element appears *twice* except for one. Find that single one.

You must implement a solution with a linear runtime complexity and use only constant extra space.

```
def singleNumber(self, nums: List[int]) -> int:
        res = 0
        for num in nums:
            res = res ^ num
        return res
```

- You just had to use the XOR operator that's all. The base value is 0 because XOR of 0 with any value is the value itself.

Problem No 128: Number of One Bits
Difficulty: Easy
Topics: Bit Manipulation

Given a positive integer n, write a function that returns the number of set bits in its binary representation (also known as the Hamming weight).

```
def hammingWeight(self, n: int) -> int:
        # Solution 1: TC, SC, O(32), O(1)
        res = 0
        while n:
            res += n % 2
            n = n >> 1
        return res

        # Solution 2: TC, SC = O(1), O(1)
        res = 0
        while n:
            n = n & (n - 1)
            res += 1
        return res
```

- Few things learnt from this question:
- You don't have to convert integers into bits to do logical operations on them

	- The technique of anding n with n-1 removes each 1 in the original n one by one. (remember it since it is a trick)
Problem No 129: Counting Bits Difficulty: Easy Topics: Bit Manipulation	Given an integer n, return *an array* ans *of length* n + 1 *such that for each* i (0 <= i <= n), ans[i] *is the* ***number of*** 1***'s*** *in the binary representation of* i.

```python
def countBits(self, n: int) -> List[int]:
        # Solution1: TC, SC = O(nlogn), O(n)
        res = [0] * (n + 1)
        for i in range(n, -1, -1):
            hamming_no = self.hammingWeight(i)
            res[i] = hamming_no
        return res

    def hammingWeight(self, n: int) -> int:
        # Solution 1: TC, SC, O(32), O(1)
        res = 0
        while n:
            n = n & (n - 1)
            res += 1
        return res

        # Solution2: DP --> TC, SC = O(n), O(n)
        dp = [0] * (n + 1)
        offset =  1

        for i in range(1, n + 1):
            if offset * 2 == i:
                offset = i
            dp[i] = 1 + dp[i - offset]
```

	`` return dp ``` - The DP solution is very interesting but not very intuitive. One wouldn't be able to figure it out unless they draw it out and try and figure out the pattern. - However, from now on, do remember that this pattern exists.
Problem No 130: Reverse Bits Difficulty: Easy Topics: Bit Manipulation	Reverse bits of a given 32 bits unsigned integer. **Note:** • Note that in some languages, such as Java, there is no unsigned integer type. In this case, both input and output will be given as a signed integer type. They should not affect your implementation, as the integer's internal binary representation is the same, whether it is signed or unsigned. • In Java, the compiler represents the signed integers using 2's complement notation. ``` def reverseBits(self, n: int) -> int: # Time, Space = O(1) res = 0 for i in range(32): bit = (n >> i) & 1 res = res | (bit << (31 - i)) return res ``` - Interesting problem and a problem that should be memorized. - The bit shifting part is really interesting.
Problem No 131: Missing Number Difficulty: Easy Topics: Bit Manipulation	Given an array nums containing n distinct numbers in the range [0, n], return *the only number in the range that is missing from the array.* ``` def missingNumber(self, nums: List[int]) -> int: # Adding and Subtracting: O(n) TC, O(1) SC ```

```
        res = len(nums)
        for i in range(len(nums)):
            res += (i - nums[i])
        return res

        # XOR Method
        n = len(nums)
        xor = 0
        for i in range(n):
            xor = xor ^ i ^ nums[i]
        return xor
```

- XOR method and the sum method.
- Both are O(1) space
- You could have implemented a hash set but that's an O(n) space.

Problem No 132: Sum of Two Integers
Difficulty: Medium
Topics: Bit Manipulation

- IMP

Given two integers a and b, return *the sum of the two integers without using the operators + and -.*

```
def getSum(self, a: int, b: int) -> int:
        # T, S = O(1)
        mask = 0xFFFFFFFF
        max_int = 0x7FFFFFFF

        while b != 0:
            carry = (a & b) << 1
            a = (a ^ b) & mask
            b = carry & mask

        return a if a <= max_int else ~(a ^ mask)
```

- Few important things to know:

	- 0xFFFFFFFF is a 32 bit mask to ensure we simulate 32 bit integer overflow (important for Python because python can grow more than 32 bits) - basically used to cut off bits. - max_int is the largest positive signed 32 bit integer - helps us decide if the final integer is positive or negative. - A ^ mask flips the bits and ~ converts the integer to a negative number in python.
Problem No 133: Reverse Integer Difficulty: Medium Topics: Bit Manipulation	Given a signed 32-bit integer x, return x *with its digits reversed*. If reversing x causes the value to go outside the signed 32-bit integer range $[-2^{31}, 2^{31} - 1]$, then return 0. **Assume the environment does not allow you to store 64-bit integers (signed or unsigned).** `def reverse(self, x: int) -> int:` `  # T, S = O(1)` `  MIN = -2147483648  # -2^31,` `  MAX = 2147483647  #  2^31 - 1` `  res = 0` `  while x:` `  digit = int(math.fmod(x, 10)) # have to do since python is dumb` `  x = int(x / 10) # truncate towards zero since python is dumb` `  # Checking if res in the range` `  if res > MAX // 10 or (res == MAX // 10 and digit > MAX % 10):` `  return 0` `  if res < MIN // 10 or (res == MIN // 10 and digit < MIN % 10):` `  return 0` `  res = (res * 10) + digit` `  return res` - Everything is almost mentioned in the comments. The hard part was just making sure that the res was within the range. - Also make sure to remember the helper function and truncate towards zero once dealing with negative integers as well since python

	is dumb.
Problem No 134: Maximum Subarray Difficulty: Medium Topics: Greedy	
Problem No 135: Jump Game Difficulty: Medium Topics: Greedy	
Problem No 136: Jump Game II Difficulty: Medium Topics: Greedy	
Problem No 137: Hand of Straights Difficulty: Medium Topics: Greedy	
Problem No 138: Merge Triplets to Form Target Triplets Difficulty: Medium Topics: Greedy	
Problem No 139: Partition Labels Difficulty: Medium Topics: Greedy	

Problem No 140: Valid Parenthesis String Difficulty: Medium Topics: Greedy	
Problem No 141: Implement Trie Prefix Tree Difficulty: Medium Topics: Trie	
Problem No 142: Design Add and Search Words Data Structure Difficulty: Medium Topics: Trie	
Problem No 143: Word Search II Difficulty: Hard Topics: Trie	
Problem No 144: Difficulty: Topics:	
Problem No 145: Difficulty: Topics:	
Problem No 146: Difficulty: Topics:	
Problem No 147:	

Difficulty: Topics:	
Problem No 148: Difficulty: Topics:	
Problem No 149: Difficulty: Topics:	
Problem No 150: Difficulty: Topics:	
Problem No 151: Difficulty: Topics:	

