1. Two Sum

```
# Duplicates in input are allowed

# search complement before inserting into hash table
# if this is done before checking for complement, case [3,3] and target = 6 fails
# (duplicate) or case [3,4,2] and target = 6 fails (complement must not be number i
tself)

class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:

        result = []
        d ={}

        for index,element in enumerate(nums):
            complement = target - element
            if d.get(complement) != None:
                result.append(d[complement])
                result.append(index)
                break   # question says only one unique solution hence break when f
ound

            d[element] = index
        return result

# Time Complexity = O(n) - we traverse the input once
# Space Complexity = O(n) - space required to create a hash table
```

2. Add Two Numbers

```
# Cases: When one list is longer than the other, extra carry at end

class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
            # Create a dummy node without any address since it needs to be filled l
ater
                # put a ptr pointing to it
        dummy = ListNode(0)
        carry = 0
        curr = dummy

        while l1 or l2:
            # One List can be shorter than other
            if l1:
                x = l1.val
                l1 = l1.next # advance l1 only when l1 is not null
            else:
                x = 0

            if l2:
                y = l2.val
                l2 = l2.next # advance l2 only when l2 is not null
            else:
                y = 0

           sum = (x + y + carry) % 10
           carry = (x+y+carry)// 10 # this carry will be used in next iteration hen
ce after sum

            # dummy node was created to facilitate below
            curr.next = ListNode(sum) # create link to next node, then advance poin
ter
            curr = curr.next


        # The sum could have an extra carry of one at the end,
        if carry != 0:
            curr.next = ListNode(carry)

        return dummy.next

# Time and Space Complexity = O(max(m,n)) where m,n = length of list l1 and l2
# Assume that m and n represents the length of l1 and l2 respectively, the algorith
m above iterates at most max(m, n) times.
```

3. Longest Substring Without Repeating Characters

```
# Note: Plain hashmap + keeping track of maxlen will not solve this e.g. tmmzuxt, a
bba, (how/from where to start again when duplicate is found)

# Sliding window

# Define a hashmap of the characters: last seen index (since we need to know where
to move left ptr)
# Expand window until duplicate is seen using right ptr
# Once duplicate is seen, move left ptr to duplicate char's last seen index + 1 and
update the duplicate char's with the most recent index seen
# Note: left ptr needs to move forward only i.e. use max() fn when updating left pt
r

# keep track of max len

class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        char_to_index = {}

        left = 0
        max_len = 0

        for right in range(len(s)):
            if char_to_index.get(s[right]) == None:
                char_to_index[s[right]] = right

            else:
                left = max(char_to_index[s[right]] + 1, left) # e.g. abba-> at last
'a', left should not move backwards
                char_to_index[s[right]] = right

            max_len = max(max_len, right - left + 1)

        return max_len

# Time Complexity = O(n)
# Space Complexity = O(min (m,n)), where m is the number of unique chars of input t
o store in dict and n is max size of string
# If it was only letters then space complexity = O(1) as num of unique letters is f
ixed (here chars could be letters, spaces, digits, symbols)
```

```
# Note: Plain hashmap + keeping track of maxlen will not solve this e.g. tmmzuxt, a
bba, (how/from where to start again when duplicate is found)
# Need to have a sliding window to satisfy all test cases with 2 ptrs (slow and fas
t)


# define a hashmap of the characters: last seen index (since we need to know where
to move slow ptr)
# fast and slow ptr
# 2 cases when duplicate is found
# Since the hashmap keeps record of all chars seen until now, not just the chars in
current window, hence 2 cases
# Case1: Duplicate within current window
# At any time duplicate is seen by fast ptr, slow ptr moves +1 to fast ptr's prev i
ndex where this char was seen + hashmap is updated for the newly seen index of this
duplicate char
# Case 2: Duplicate not in current window
#  Increment fast ptr + update hashmap to the newly seen index of this duplicate ch
ar

# All along the way keep track of length of longest substring seen

class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        char_to_index = {}
        left = 0
        output = 0
        for right in range(len(s)):
            # If s[right] not in char_to_index, we can keep increasing the window s
ize by moving right pointer
            if s[right] not in char_to_index:
                char_to_index[s[right]] = right
                output = max(output,right-left+1)

            #There are two cases if s[r] in char_to_index:
            #case1: s[right] is not inside the current window,  keep track of max l
en and update hashmap with last index seen for this char
            #case2: s[right] is inside the current window, update slow ptr and upda
te hashmap with last index seen for this char

            else:
                if char_to_index[s[right]] < left:
                    output = max(output,right-left+1)
                    char_to_index[s[right]] = right
                else:
                    left = char_to_index[s[right]] + 1
                    char_to_index[s[right]] = right

        return output

# Time Complexity = O(n)
```

```
# Space Complexity = O(m), where m is the number of unique chars of input to store
in dict
# chars could be letters, digits, symbols and spaces
# If it was only letters then space complexity = O(1) as num of unique letters is f
ixed
```

4. Median of Two Sorted Arrays

https://www.youtube.com/watch?v=LPFhl65R7ww (https://www.youtube.com/watch?v=LPFhl65R7ww)

https://medium.com/@dimko1/median-of-two-sorted-arrays-b7f0c4284159
(https://medium.com/@dimko1/median-of-two-sorted-arrays-b7f0c4284159)

```
# Brute force: Merge Sort ( m+n log m+n) , where m,n = length of arrays
# Min Heap Method to merge sorted arrays: O ( m+n log 2)
# Below method: O(log min(m,n)) even when two arrays are of different size

# Binary search on smaller array
# partition the two arrays in such a way so that maxLeftX <= minRightY and minRight
X >= maxLeftY
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:
        if len(nums1) > len(nums2):
            # we want first part to be smaller then second
             nums1, nums2 = nums2, nums1


        x = len(nums1)
        y = len(nums2)

        low = 0
        high = x

        while low <= high:
            # basic selection of the partitioner
            partitionX = (low + high) // 2
            # partitioning of the second array
            partitionY = (x + y + 1) // 2- partitionX

            # getting values. sometimes maxLeftX index can be 0
            maxLeftX = nums1[partitionX - 1] if partitionX > 0 else float('-inf')
            minRightX = float('inf') if partitionX == x else nums1[partitionX]

            maxLeftY = nums2[partitionY - 1] if partitionY > 0 else float('-inf')
            minRightY = float('inf') if partitionY == y else nums2[partitionY]

            # Core of this algo: let's check if partition was selected correctly
            if maxLeftX <= minRightY and minRightX >= maxLeftY:
                if (x + y) % 2 == 0:
                    return (max(maxLeftX, maxLeftY) + min(minRightX, minRightY)) /
2.0
                else:
                    return max(maxLeftX, maxLeftY)
            # if not - move partitioning region
            elif maxLeftX > minRightY:
                high = partitionX -1
            else:
                low = partitionX + 1

# Time Complexity = O(log(min(len(nums1), len(nums2))))
# Space Complexity = O(1)
```

5. Longest Palindromic Substring

https://www.techiedelight.com/longest-palindromic-substring-non-dp-space-optimized-solution/ (https://www.techiedelight.com/longest-palindromic-substring-non-dp-space-optimized-solution/)

https://leetcode.com/problems/longest-palindromic-substring/discuss/2954/Python-easy-to-understand-solution-with-comments-(from-middle-to-two-ends) (https://leetcode.com/problems/longest-palindromic-substring/discuss/2954/Python-easy-to-understand-solution-with-comments-(from-middle-to-two-ends)).

```
# For every char in string, treat it as center of odd length and even length palind
rome
# For odd length palindrome, expand from itself (i,i) and check how far the expansi
on can be palindrome
# For even length plaindrome, expand from itself and next char (i,i+1) and check ho
w far the expansion can be palindrome

class Solution:
    def longestPalindrome(self, s: str) -> str:
        def expand(low, high):
            while(low >= 0 and high<len(s) and s[low]==s[high]):
                low = low -1
                high = high + 1

            return s[low+1:high]

        max_pal =""
        max_len = 0

        for i in range(len(s)):
            curr_odd_pal = expand(i,i)
            curr_even_pal = expand(i, i+1)
            max_len = max(len(curr_odd_pal), len(curr_even_pal), max_len)

            if max_len == len(curr_odd_pal):
                max_pal = curr_odd_pal
            elif max_len == len(curr_even_pal): #" else:" alone will not work since
max_len could be empty string
                max_pal = curr_even_pal

        return max_pal

# Time Complexity = O(n^2), at each position in n, expand till ends of string of le
n n
# Space Complexity = O(1)
```

6. Zigzag Conversion

```
# Step 1: Determine the size of matrix to be filled (i.e. num of columns in terms o
f given num of rows)
# num of columns = num of cols in each section * num of sections
# num of cols in each section = num of rows - 1
# num of sections = ceil (total no. of chars/ num of chars in each section)
# num of chars in each section = 2 * num of rows - 2 (numRow  in one column and num
Row - 2 in the diagonal)


# Step 2: Fill the matrix in zig-zag manner
# Traverse matrix in zig-zag manner (using curr_row and curr_col) - 1. Top-down 2.
Diagonal
# While moving from top to bottom in a column, currCol will remain the same but cur
rRow will go from 0 to numRows
# While moving diagonally up, we move one cell up and one cell right, thus incremen
t currCol by 1 and decrease currRow by 1 before it reaches the top (currRow>0)

# Step 3: Read the matrix line by line


class Solution:
    def convert(self, s: str, numRows: int) -> str:
        if numRows == 1:
             return s

        # Step 1
        num_columns_per_section = numRows - 1

        num_chars = len(s)
        num_sections = ceil(num_chars /((2 * numRows) - 2)) # ceil and not floor

        num_columns = num_columns_per_section * num_sections

        matrix = [[''] * num_columns for row in range(numRows)]

        # Step 2
        curr_row = 0
        curr_col = 0
        curr_index_str = 0

        while curr_index_str < len(s):

            # Move Down
            while curr_row < numRows and curr_index_str < len(s):
                matrix[curr_row][curr_col] = s[curr_index_str]
                curr_index_str = curr_index_str + 1
                curr_row = curr_row + 1

            # Adjust curr_row and curr_col
            curr_row = curr_row - 2
            curr_col = curr_col + 1
```

```
                        # Move diagonal
            while curr_col < num_columns and curr_row > 0 and curr_index_str < len
(s):
                matrix[curr_row][curr_col] = s[curr_index_str]
                curr_row = curr_row - 1
                curr_col = curr_col + 1
                curr_index_str = curr_index_str + 1


        # Step 3

        result = ""
        for row in range(len(matrix)):
            for col in range(len(matrix[0])):
                result = result + "".join(matrix[row][col])     # empty string + em
pty string = empty string
        return result

# Time Complexity = O(size of matrix) = O(num of rows * no. of columns) where no. o
f columns = O(n) where n is length of input string
# Space Complexity =  O(size of matrix) = O(num of rows * no. of columns)
# Here number of rows is part of input hence complexity can be expressed in terms o
f no. of rows
```

7. Reverse Integer

```
# extract sign separately
# take abs(x) and reverse
# check bounds : input is always in range but output could be out of range

class Solution:
    def reverse(self, x: int) -> int:
        if x > 0:
            sign = 1
        elif x < 0:
            sign = -1
        else:
            return 0

                # most imp snippet
        rev = 0
        x = abs(x)
        while x:
            x, remainder = divmod(x,10)
            rev = rev*10 + remainder

        if -pow(2,31) <= sign*rev <= pow(2,31)-1 :
            return sign*rev
        else:
            return 0

# Time Complexity = O(log (number of digits)). There are roughly log(x) to the base
10 digits in x.
# Space Complexity = O(1)
```

8. String to Integer (atoi)

```
# invalid inputs -> only'-', only'+', only'-+' or if start is not digit -> return 0
class Solution:
    def myAtoi(self, str: str) -> int:
        # case 1: empty string
        if len(str) == 0 :
            return 0

        # case 2: after stripping whitespace , empty string
        ls = list(str.strip(' '))
        if len(ls) == 0:
            return 0

        # case 3: collect sign and if only '-' -> return 0
        sign = 1 # see how sign is used
        two_strike = 0
        if ls[0] == '-' :
            sign = -1
            two_strike = two_strike + 1
            ls = ls[1:]
        if len(ls) == 0:
            return 0

        # case 4: collect sign and if only '+' -> return 0
        if ls[0] == '+':
            two_strike = two_strike + 1
            ls = ls[1:]
        if len(ls) == 0:
            return 0

        # case 5 : '-+' -> return 0
        if two_strike == 2:
            return 0

        # case 6: doesn't start with digit
        if ls[0].isdigit() == False:
            return 0

        # Main case : extract number before words begin
        ret = 0 # Important logic
        for index,element in enumerate(ls):
            if element.isdigit() == True:
                ret = ret*10 + ord(element) - ord('0')
            else:
                break

        ret = ret * sign  # Imp

        # Boundary conditions
        max_value = (2 ** 31) - 1
        if ret > max_value:
            return max_value
```

```
        if ret < - (max_value) - 1:
            return -(max_value) - 1
        return ret

# Time Complexity = O(n) for slicing and iterating thru each element of string
# Space Complexity = O(n) for slicing
```

9. Palindrome Number

https://leetcode.com/problems/reverse-integer/ (https://leetcode.com/problems/reverse-integer/)

```
# Initialize rev = 0
# 4 var: rev, x, quotient, remainder

class Solution:
    def isPalindrome(self, x: int) -> bool:
        if x < 0:
             return False

        if x == 0:
            return True

        rev = 0
        q = x
                # Imp code
        while q:
            q, remainder = divmod(q,10)
            rev = rev * 10 + remainder

        if rev == x:
            return True
        else:
            return False

# Time Complexity = O(log n) base 10. There are roughly log(x) to the base 10 digit
s in x.
# Space Complexity = O(1)
```

11. Container With Most Water

https://leetcode.com/problems/container-with-most-water/solution/ (https://leetcode.com/problems/container-with-most-water/solution/)

```
# 2 pointer approach

# Two pointers at opposite ends => maximizing base
# height of whichever pointer is smaller moves forward, in case of tie move begin p
tr
# keep track of max area


class Solution:
    def maxArea(self, height: List[int]) -> int:
        begin = 0
        end = len(height) - 1

        maxarea = 0

        while begin < end:
            if height[begin] <= height[end]:
                area = height[begin] * (end-begin)
                                maxarea = max(area, maxarea)
                begin = begin +1
            else:
                area = height[end] * (end-begin)
                                maxarea = max(area, maxarea)
                end = end -1

        return maxarea

# Time Complexity = O(n)
# Space Complexity = O(1)
```

12. Integer to Roman

Look at solution animation:

Go from left to right collecting quotient

```
# Create a list of tuple of value and symbol in descending order of value
# Make sure it contains representation of 900, 400, ...  etc
# keep dividing the num by value collecting digits
# Ex. 29 -> XXIX

class Solution:
    def intToRoman(self, num: int) -> str:
        value_symbol = [(1000, "M"), (900, "CM"), (500, "D"), (400, "CD"), (100,
"C"), (90, "XC"),
          (50, "L"), (40, "XL"), (10, "X"), (9, "IX"), (5, "V"), (4, "IV"), (1,
"I")]

        roman = []
        for value,symbol in value_symbol:
            if num == 0: # Stopping condition
                break
            count, num = divmod(num,value)
            roman.append(symbol*count)
        return "".join(roman)

# Time and Space Complexity = O(1)
# As there is a finite set of roman numerals, there is a hard upper limit on how ma
ny times the loop can iterate.
# The amount of memory used does not change with the size of the input integer, and
is therefore constant
```

13. Roman to Integer

```
# create a dict of char: val
# left to right pass -> if value[i] < value[i+1], take two chars and use substracti
on
# take example III and IV

class Solution:
    def romanToInt(self, s: str) -> int:
        values = {"I": 1,"V": 5,"X": 10,"L": 50,"C": 100,"D": 500,"M": 1000}

        i = 0
        result = 0
        while i < len(s): # if i+1 check done here then III -> gives wrong answer
            if i+1 < len(s) and values[s[i]] < values[s[i+1]]: # See how i+1 -> avo
ids overflow
                result = result + (values[s[i+1]] - values[s[i]])
                i = i+2 # increment by 2
            else:
                result = result + values[s[i]]
                i = i+1
        return result

# Time and Space Complexity = O(1)
```

There are just 7 Roman numerals: I, V, X, L, C, D and M. Unlike most other number systems, the numerals can only be used in particular sequences. Another rule is that apart from M, you can only have a maximum of 3 of any given numeral in a row. The highest number that can be expressed in Roman numerals is actually 3,999. This is written as MMMCMXCIX. This is because the number 4,000 would have to be written as MMMM, which goes against the principle of not having four consecutive letters of the same type together

14. Longest Common Prefix

```
# l = ['hello', 'hel', 'hell'] -> if you just compare first and last, common prefix
length = 4 but ans = 3
# l.sort()
# O/P -> ['hel', 'hell', 'hello'] -> now if you compare first and last, length of c
ommon prefix = 3

class Solution:
    def longestCommonPrefix(self, strs: List[str]) -> str:
        if not strs: return ""
        if len(strs) == 1: return strs[0]

        strs.sort() # this will sort all strings (by prefix)
        r = ""
        for x,y in zip(strs[0], strs[-1]): # check first and last string char by ch
ar
            if x ==y:
                r = r + x
            else: #only prefix matching
                break
        return r

# Time Complexity = O(nk log nk) where n = no. of words and k = max length of any w
ord for sorting. There is O(nk) comparison when comparing strs[0] and strs[-1] char
by char but sorting complexity dominates
# Time complexity can be reduced by not using sorting since we only need the max le
ngth and min length word =>  O(nk)

# Space Complexity = O(nk)
# Python's built in sort method is a spin off of merge sort called Timsort
# It's essentially no better or worse than merge sort, which means that its run tim
e on average is O(n log # n) and its space complexity is Ω(n)
# However, in-place sorting like quicksort can reduce space complexity to O(1)
```

15. 3Sum

```
# Two pointer approach
# Using hash table approach is trickier here to handle duplicates in input hence no
t used

# result should not contain duplicate triplets = result is a set with tuples (not l
ist of list) of triplets since set elements should be hashable but list of list is
not hashable, set cannot have list as an element,
# Though a single list can be converted to set by set(list_name)

class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        nums.sort()  # sort
        result = set()  # takes care of duplicate triplets

        for i in range(len(nums)-2): # take care, upto second last index

            l = i+1
            r = len(nums) - 1 # last index

            while l<r:  # < and not <= as i,l,r all have to be different
                total = nums[i] + nums[l] + nums[r]
                if total < 0:
                    l = l+1
                elif total > 0:
                    r = r -1
                elif total == 0:
                    result.add((nums[i], nums[l], nums[r])) # tuple is added hence
extra (), cannot add list
                    # can find multiple l and r for the same i
                    l = l+1
                    r = r-1
        return list(map(list,result))     # can also just return result, accepted b
y leetcode

# Time Complexity = O(n^2)
# Space Complexity = O(n)  due to sort in the beginning, if input is already sorted
then O(1)

# Python's built in sort method is a spin off of merge sort called Timsort with tim
e complexity O(n log n) and space complexity O(n) (uses temp array)
```

16. 3Sum Closest

```
# Compare abs diff between current total and target to closest sum seen so far and
target
# If abs diff between current total and target < abs diff between closest sum seen
so far and target
# Then current total becomes the new closest sum

class Solution:
    def threeSumClosest(self, nums: List[int], target: int) -> int:
        nums.sort()
        closest_sum = float('inf')

        for i in range(len(nums)-2):
            left = i+1
            right = len(nums) - 1
            while left < right:
                total = nums[i] + nums[left] + nums[right]

                if abs(target - total) < abs(target - closest_sum):  # use absolute
value
                    closest_sum = total
                if total < target:
                    left = left + 1
                else:
                    right = right - 1

        return closest_sum

# Time Complexity = O(n^2)
# Space Complexity = O(n)
```

17. Letter Combinations of a Phone Number

Backtracking is not considered an optimized technique to solve a problem. Its a brute force approach. It finds its application when the solution needed for a problem is not time-bounded.

```
class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        result = []
        d = {'2': ['a', 'b', 'c'],
             '3': ['d', 'e', 'f'],
             '4': ['g', 'h', 'i'],
             '5': ['j', 'k', 'l'],
             '6': ['m', 'n', 'o'],
             '7': ['p', 'q', 'r', 's'],
             '8': ['t', 'u', 'v'],
             '9': ['w', 'x', 'y', 'z']}

        def backtrack(output, index):
            if len(output) == len(digits):
                result.append(output)  # [:] not needed since output is a string an
d immutable, [:] needed for mutable types e.g. list
                return

            # most import code snippet
            for char in d[digits[index]]:   #  # digits is a string and can be acce
ssed by index
                output = output + char
                backtrack(output, index + 1)
                output = output[:-1] # only list has pop(), for string: take all el
ements except last

        if digits:
            backtrack("",0)
        return result

# Time and Space Complexity: O(3^N * 4^M) where N is the number of digits in the in
put that maps to 3 letters and M is the number of digits in the input that maps to
4 letters where N+M is total no. of digits in the number
# E.g. "23" ->  _ _  = 1st position 3 possibilities, 2nd position 3 possibilities =
3*3 = 3 ^2 = 3 ^ (no. of digits)
```

18. 4Sum

```
# See 3 sum approaches

class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        nums.sort()
        result = set()

        for i in range(len(nums)-3):
            for j in range(i+1, len(nums)-2): # start from i+1
                left = j+1
                right = len(nums)-1

                while left < right:
                    total = nums[i] + nums[j] + nums[left] + nums[right]

                    if total > target:
                        right = right - 1

                    elif total < target:
                        left = left + 1

                    else:
                        result.add((nums[i], nums[j], nums[left], nums[right]))
                        # Multiple solutions possible
                        right = right -1
                        left = left + 1

        return result

# Time Complexity = O(n^3)
# Space Complexity = O(n) due to sort
```

19. Remove Nth Node From End of List

Adding a dummy node at the head and /or tail might help to handle many edge cases where operations have to be performed at the head or the tail. The presence of dummy nodes ensures that operations will never have be executed on the head or the tail. Dummy nodes remove the headache of writing conditional checks to deal with null pointers. Be sure to remove them at the end of the operation.

Also if you start with slow = head and fast = head => both slow, fast and head are essentially the same var since they map to one memory location meaning any operations performed on one (e.g. slow) will be reflected in others (e.g. fast and head)

```
# Edge cases: [1,2] 2 -> [2] and [1,2] 1 -> [1] and only one node case

# dummy node simplifies edge cases when removing head and cases where only one node
is present
# Note that before deletion, slow points to one node before


class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
            # Create a dummy node that points to head node
                # Point slow and fast ptrs to dummy node
        dummy = ListNode(0)
        dummy.next = head

        fast = dummy
        slow = dummy

        # Advance fast pointer so that the gap between fast and slow is n nodes apa
rt
        i = 0
        while i <= n:
            fast = fast.next
            i = i+1

        # Move fast to the end, maintaining the gap
        while fast:
            fast = fast.next
            slow = slow.next

        # Adjust pointers to drop nth node from last
        slow.next = slow.next.next

        return dummy.next

# Time Complexity = O(n)
# Space Complexity = O(1)
```

20. Valid Parentheses

```
# Put all opening parenthesis in stack, and when closing parenthesis comes, pop + c
heck whether it is corresponding opening parenthesis
# Edge cases: 1) extra opening parenthesis 2) extra closing parenthesis

class Solution:
    def isValid(self, s: str) -> bool:
        d = {')':'(', '}':'{', ']':'['}
        stack = []

        for c in s:
            if c == '[' or c == '{' or c == '(':
                stack.append(c)

            if c == ']' or c == '}' or c == ')':
                # Edge case: closing parenthesis should have a complement in stack
e.g. ()]
                if len(stack) == 0:
                    return False



                # if stack is not empty, popped element should be the complement pa
renthesis
                if d[c] != stack.pop():
                    return False

        if len(stack) != 0: # Edge case: Stack should be empty at last e.g. (((
            return False
        else:
            return True

# Time Complexity = O(n) traverse through the string and push(), pop() on stack is
O(1)
# Space Complexity = O(n)
```

21. Merge Two Sorted Lists

```
class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
            # create a dummy node without any address since it needs to be filled l
ater
                # put a ptr pointing to it for navigating the list and re-adjusting
addresses (including dummy node's address)

        dummy = ListNode(0)
        prev = dummy

        while l1 and l2:
            if l1.val <= l2.val:
                prev.next = l1
                l1 = l1.next
            else:
                prev.next = l2
                l2 = l2.next

                        # imp: move prev to point to added node
            prev = prev.next

                # if either of the lists are smaller than the other, append the rem
aining portion

        if l1:
            prev.next = l1

        if l2:
            prev.next = l2

        return dummy.next

# Time Complexity = O(n) where n = combined length of both lists
# Space Complexity = O(1)
```

22. Generate Parentheses

```
# Backtrack from empty string and opening and closing bracket count = 0
# Can only start with opening bracket
# We can start an opening bracket if we still have one (of n) left to place.
# And we can start a closing bracket if it would not exceed the number of opening b
rackets.

# Understand with n = 2
# ["(())","()()"]

class Solution(object):
    def generateParenthesis(self, N):
        ans = []
        def backtrack(S, open, close):

            if len(S) == 2 * N:
                ans.append(S) # string is immutable hence not [:] or copy
                return

            if open < N:
                backtrack(S+'(', open+1, close)

            if close < open:
                backtrack(S+')', open, close+1)

        backtrack('', 0,0)
        return ans

# One way of describing Time and Space Complexity = We are generating all possible
strings of length 2n. At each character, we have two choices: choosing ( or ), whic
h means there are a total of 2^{2n} unique strings. O(n * 2^2n)
# Time and Space Complexity = O(n * 2^2n)
```

```
# Another way of coding

class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        result = []

        def backtrack(S, open, close):
            if len(S) == 2 * n:
                result.append(S)
                return

            if open < n:
                S = S+'('
                backtrack(S, open+1, close)
                S = S[:-1]

            if close < open:
                S = S+')'
                backtrack(S, open, close+1)
                S = S[:-1]

        backtrack('', 0, 0)
        return result

# Time Complexity = O(n * 2 ^ 2n)
# We are generating all possible strings of length 2n. At each character, we have t
wo choices: choosing ( or ), which means there are a total of
2^ 2n unique strings
# Space Complexity = O(n * 2^2n)
```

23. Merge k Sorted Lists

```
# Same as merge 2 lists, k times
# Idea: merge 2 consecutive lists at a time and repeat
# i) merge list 1 and 2 (list12), merge list 3 and 4 Z(list34), and so on
# ii) now merge list12 and list34, and so on

class Solution:
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        if not lists or len(lists) == 0:
            return None

        while len(lists) > 1:

            mergedLists = []

            # merge 2 consecutive lists at a time
            for i in range(0, len(lists), 2):
                l1 = lists[i]
                l2 = lists[i + 1] if (i + 1) < len(lists) else None
                mergedLists.append(self.mergeList(l1, l2))

                        # repeat again for this merged list
            lists = mergedLists

        return lists[0]

    def mergeList(self, l1, l2):
        dummy = ListNode(0)
        prev = dummy

        while l1 and l2:
            if l1.val < l2.val:
                prev.next = l1
                l1 = l1.next
            else:
                prev.next = l2
                l2 = l2.next
            prev = prev.next

        if l1:
            prev.next = l1
        if l2:
            prev.next = l2

        return dummy.next

# Time Complexity = O(N log k), we are reducing the merge by a factor of 2
# Space Complexity = O(1)
```

```
# Merge sort on k arrays = we need to keep track of which array element we took in
each step and increment the ptr for that array
# For k arrays, k ptrs need to be maintained, but with linked lists we can utilize
the address component of each node to navigate

# Since the lists are in sorted order, put the first element of k lists in a min he
ap in a tuple (node value, tie var, node address)
# whenever push happens on a heap, to avoid ties between node values, we also pad a
tie var (count) which is unique for every node value
# pop from the heap (get the min element) and use the address component to push the
next element onto heap (with tie var)
# create a dummy var that points to head, use its address component to traverse the
popped elements

class Solution:
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        curr = head = ListNode(0)
        count = 0
        heap = []

        for l in lists: # lists contain the head ptr to all lists, will iterate onl
y k (num of lists) times
            if l:
                count = count + 1
                heapq.heappush(heap, (l.val, count, l)) # This pushes only the firs
t elements of each list

        while len(heap) > 0:
            _,_, curr.next = heapq.heappop(heap)
            curr = curr.next
            if curr.next is not None:
                count = count + 1
                heapq.heappush(heap, (curr.next.val, count, curr.next))

        return head.next

# count var - it handles the case of a "tie" when two list nodes have the same valu
e. When that
# happens, Python will look at the next value in the tuple (in this case, count), a
nd sort based
# on that value. Without count, a "tie" would error out if the next value in the tu
ple were a
# ListNode (which can't be compared).

# Space Complexity: O ( k ), Our heap will need to hold k elements for most of this
process and cannot worsen past this.

# Time Complexity: O (n * 2 * log k), Extracting and adding to the min heap will bo
th take log(k) time  as max number of items in heap at a time is k (2 -> add + dele
te min), we do it for all elements in all lists which we  denote by n
```

26. Remove Duplicates from Sorted Array

https://www.youtube.com/watch?v=DEJAZBq0FDA&t=242s (https://www.youtube.com/watch?v=DEJAZBq0FDA&t=242s)

```
# sorted array

# 2 ptr approach: 1 ptr is responsible for writing unique values in our input arra
y, while 2nd ptr will read the input array and pass all the distinct elements to 1s
t ptr

# 2 pointers which start at index 1 (i and insert_index)
# i checks prev element with current element and insert_index keeps track of insert
ion position
# if i finds a unique element (keep nums[i]) i.e. i and i-1 elements are not same,
nums[i] is stored in insert_index and insert_index is incremented
# if i finds a duplicate element, it simply moves ahead


class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:

        insert_index = 1

        for i in range(1,len(nums)):
            if nums[i-1] != nums[i]:
                nums[insert_index] = nums[i]
                insert_index = insert_index + 1


        return insert_index



# Time complexity = O (n)
# Space complexity = O (1) (in-place)
```

27. Remove Element

```
# Two pointers begin and end
# If begin pointer encounters the value to be removed, it swaps with end pointer an
d decrements end pointer
# else begin pointer moves ahead

class Solution:
    def removeElement(self, nums: List[int], val: int) -> int:

        start = 0
        end = len(nums) - 1

        while start <= end:
            if nums[start] == val:
                nums[end], nums[start] = nums[start], nums[end]
                end = end - 1
            else:
                start = start + 1

        return start


# Time Complexity = O(n)
# Space Complexity = O(1)
```

28. Find the Index of the First Occurrence in a String

```
# Loop within loop (outer loop: window_start and inner loop: len of substring)
# window_start goes until starting index of last substring possible in string
# For each window_start, check whether substring is found. If not found, increment
window_start by 1



class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        window_start = 0

        while window_start <= len(haystack) - len(needle):
            for i in range(len(needle)):
                if needle[i] != haystack[window_start + i]:
                    break

                if i == len(needle)-1:
                    return window_start

            window_start = window_start + 1
        return -1

# Time Complexity = O(len of substring * len of string)
# Space Complexity = O(1)

# Standard algorithm like
# Rabin-Karp (hashing) has linear time complexity of O(length of string) with space
complexity = O(1)
# KMP algorithm has linear time complexity of O(length of string) linear with space
complexity = O(length of
# substring)
```

---

31. Next Permutation

https://www.youtube.com/watch?v=hPd4MFdg8VU (https://www.youtube.com/watch?v=hPd4MFdg8VU)

Approach 2 : Single Pass https://leetcode.com/problems/next-permutation/solution/ (https://leetcode.com/problems/next-permutation/solution/)

```
# The replacement must be in-place and use only constant extra memory.
# if digits are in descending order -> no higher permutation possible, reverse the
digits

# Find the first valley from right since  numbers to the right of this peak are in
descending order i.e. no larger permutation is possible
# swap this valley digit with the first digit from right which is higher than this
valley digit
# reverse the digits which are right of this valley digit  (excluding the valley di
git)

# e.g. 1243 -> 1342 -> 1324

class Solution:
    def nextPermutation(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """

        i = len(nums)-1
        while i > 0 and nums[i-1] >= nums[i]:
            i -= 1

        if i == 0:   # nums are in descending order
            nums.reverse()
            return


        # find the first element from right which is greater than valley digit
        j = len(nums)-1
        while nums[j] <= nums[i-1]: # <= and j decremented below => next higher dig
it to nums[i-1]
            j -= 1

        # swap
        nums[i-1], nums[j] = nums[j], nums[i-1]


        # reverse the digits after valley digit ( excluding valley digit)
        l, r = i, len(nums)-1
        while l <= r:
            nums[l], nums[r] = nums[r], nums[l]
            l +=1 ; r -= 1

# Time Complexity = O(n)
# Space Complexity = O(1)
```

33. Search in Rotated Sorted Array

```
# Find which side rotation happened by comparing mid number with end number
# Since the other side is not rotated and sorted, compare with first and last eleme
nt of the other side, adjust end pointer or begin pointer depending on where target
lies

# It is guaranteed that the array is rotated
# No duplicates

class Solution:
    def search(self, nums: List[int], target: int) -> int:
        begin = 0
        end = len(nums) - 1

        while begin <= end:
            mid = begin + (end-begin) // 2

            if nums[mid] == target:
                return  mid

            elif nums[mid] > nums[end]:
                if nums[mid] > target and nums[begin] <= target: # <=, since nums[m
id] == target is checked
                    end = mid -1
                else:
                    begin = mid +1

            else: # nums[mid] < nums[end]
                if nums[mid] < target and nums[end] >= target: # >=
                    begin = mid + 1
                else:
                    end = mid - 1

        return -1

# Time Complexity = O(log n)
# Space Complexity = O(1)
```

34. Find First and Last Position of Element in Sorted Array

For all binary Search problems, check for i) if array is empty ii) target not found iii) duplicates iv) leftmost match or rightmost match

```
# array is sorted and has duplicates

# 2 usual binary searches - one for leftmost position and one for rightmost positio
n
# when target is found
# i) for starting/leftmostmost index adjust high (mid-1) and keep track of min inde
x
# i) for ending/rightmost index adjust low (mid+1) and keep track of max index

class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        result = [-1, -1]

        result[0] = self.findStartingIndex(nums, target)
        result[1] = self.findEndingIndex(nums, target)

        return result

    def findStartingIndex(self, nums, target):

        begin = 0
        end = len(nums) - 1


        index = len(nums) # need to find min index so initialize accordingly
        while begin <= end:
            mid = begin+(end - begin) // 2

            if nums[mid] == target:
                index = min(index, mid)
                end = mid - 1

            if nums[mid] < target:
                begin = mid + 1
            elif nums[mid] > target:
                end = mid - 1

        return index if index != len(nums) else -1 # check if min index was found

    def findEndingIndex(self, nums, target):

        begin = 0
        end = len(nums) - 1

        index = -1 # need to find max index so initialize accordingly
        while begin <= end:
            mid = begin+(end-begin) // 2

            if nums[mid] == target:
                index = max(index, mid)
                begin = mid + 1
```

```
            if nums[mid] < target:
                begin = mid + 1
            elif nums[mid] > target:
                end = mid - 1

        return index # check if max index was found, anyways it was initialized to
default value

# Time Complexity = O(log n) -> binary search 2 times
# Space complexity = O(1), in-place
```

35. Search Insert Position

```
# Usual binary search except return left (in case only 1 element that does not matc
h the target, both cases: > or < target)

class Solution:
    def searchInsert(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums) - 1
        while left <= right:
            pivot = (left + right) // 2
            if nums[pivot] == target:
                return pivot
            if target < nums[pivot]:
                right = pivot - 1
            else:
                left = pivot + 1
        return left
```

36. Valid Sudoku

```
# 9 sets to keep track of which numbers have been visited for "each" row, col and b
ox (list of sets)
# box index = row // 3 * 3 + col // 3

# Initialize multiple empty sets as [set() for _ in range(9)]

class Solution:
    def isValidSudoku(self, board: List[List[str]]) -> bool:
        row_set = [set() for _ in range(9)]
        col_set = [set() for _ in range(9)]
        box_set = [set() for _ in range(9)]

        for row in range(9):
            for col in range(9):
                val = board[row][col]

                if val == ".":
                    continue


                if val in row_set[row]:
                    return False
                else:
                    row_set[row].add(val)


                if val in col_set[col]:
                    return False
                else:
                    col_set[col].add(val)


                idx = (row // 3) * 3 + col // 3
                if val in box_set[idx]:
                   return False
                else:
                    box_set[idx].add(val)

        return True

# Time Complexity = O(size of board)
# Space Complexity = O(3 * length of row or col), hashset for each row, col and box
```

38. Count and Say

```
# recursive sol
# keep appending count of unique digit and unique digit together
# base case: n=1

class Solution:
    def countAndSay(self, n: int) -> str:
        if n == 1:
            return "1"

        s = self.countAndSay(n - 1)

        res = ""
        counter = 0

        for i in range(len(s)):
            counter += 1
            if i == len(s) - 1 or s[i] != s[i + 1]: # compare i == len(s) - 1 if la
st digit is unique
                res = res + str(counter) + s[i]
                counter = 0

        return res
```

39. Combination Sum

https://leetcode.com/problems/combination-sum/discuss/16502/A-general-approach-to-backtracking-questions-in-Java-(Subsets-Permutations-Combination-Sum-Palindrome-Partitioning) (https://leetcode.com/problems/combination-sum/discuss/16502/A-general-approach-to-backtracking-questions-in-Java-(Subsets-Permutations-Combination-Sum-Palindrome-Partitioning))

complexity : https://leetcode.com/problems/combination-sum-iii/discuss/427713 (https://leetcode.com/problems/combination-sum-iii/discuss/427713)
https://algorithmsandme.com/tag/leetcode-combination-sum/ (https://algorithmsandme.com/tag/leetcode-combination-sum/)

Backtracking is not considered an optimized technique to solve a problem. Its a brute force approach. It finds its application when the solution needed for a problem is not time-bounded.

```
# Input has no duplicates
# number can be chosen from candidates unlimited number of times

# imp: start next search from current position and not from beginning as duplicate
combinations are not allowed
# base conditions
# 1) remainder == 0
# 2) remainder < 0

class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[in
t]]:
        def backtrack(temp_list, remain, start):
            if remain == 0: # base cond
                result.append(temp_list[:])
                                return

            if remain < 0: # base cond
                return

            for i in range(start,len(candidates)):  # Solution set should not conta
in duplicates
                temp_list.append(candidates[i])
                backtrack(temp_list, remain - candidates[i], i) # not i + 1 because
we can reuse
                temp_list.pop()

        result = []
        backtrack([], target, 0)
        return result

# Time Complexity if numbers cannot be reused =  C(n,1) + C(n,2) + … + C(n,n) = 2^n
- C(n,0) = O(2^n)
# Now every number can be reused, the max number of times a number can be used = ho
w many times the smallest number goes into target e.g. if target was 10 and smalles
t number was 2 then max number of times 2 can be used 10/2 = 5. hence time complexi
ty = O((target/smallest number) * 2^n)

# Space Complexity if numbers cannot be reused = O(n) ie. all numbers together sum
to target (worst case)
# However numbers can be repeated so again max number of times a number can be used
= how many times the smallest number goes into target => hence space complexity = O
((target/smallest number) * n)
```

40. Combination Sum II

Backtracking is not considered an optimized technique to solve a problem. Its a brute force approach. It finds its application when the solution needed for a problem is not time-bounded.

```
# The input has duplicates -> combinations can be duplicate but we want unique
# hence if char at i is same as char at  i-1 , skip it
# But we cannot reuse same element hence backtrack/dfs from next position
class Solution:
    def combinationSum2(self, candidates: List[int], target: int) -> List[List[in
t]]:
        def dfs(start, remain, combo):
            if remain == 0:
                result.append(combo[:])

            if remain < 0:
                return

            for i in range(start, len(candidates)):
                if i > start and candidates[i] == candidates[i-1]: # skip i if i an
d i-1 are same
                    continue

                combo.append(candidates[i])
                dfs(i+1, remain - candidates[i], combo) # i+1: cannot reuse same el
ement
                combo.pop()

        result = []
        candidates.sort() # Imp: if sorted then if i and i-1 are same elements, ski
p i
        dfs(0, target, [])
        return result

# Time Complexity if numbers cannot be reused =  C(n,1) + C(n,2) + … + C(n,n) = 2^n
- C(n,0) = O(2^n)
# Space Complexity if numbers cannot be reused = O(n) ie. all numbers together sum
to target (worst case)
```

41. First Missing Positive

https://github.com/Chanda-Abdul/Several-Coding-Patterns-for-Solving-Data-Structures-and-Algorithms-Problems-during-Interviews/blob/main/%E2%9C%85%20%20Pattern%2005%3A%20Cyclic%20Sort.md (https://github.com/Chanda-Abdul/Several-Coding-Patterns-for-Solving-Data-Structures-and-Algorithms-Problems-during-Interviews/blob/main/%E2%9C%85%20%20Pattern%2005%3A%20Cyclic%20Sort.md)

https://www.youtube.com/watch?v=TLiWieBQwUs (https://www.youtube.com/watch?v=TLiWieBQwUs)

```
# Input array is not sorted so we cannot apply the technique directly from 1539. Kt
h Missing Positive Number (need to sort first)

# One solution could be if we keep numbers [1,n] in hashset and traverse the input
array and keep checking in hashset. Space complexity = O(n)
# we need, space complexity = O(1)

# cyclic sort + extra check for n+1 as missing number

class Solution:
    def firstMissingPositive(self, nums: List[int]) -> int:
        i = 0

        while i < len(nums):
            # find current num that needs to be placed in its correct position (ind
ex is 1 less than the number)
            curr_num = nums[i] - 1

            # Swap if i) current num is in range [1, last_index] and ii) not in its
correct position
            if 0 <= curr_num < len(nums) and nums[i] != nums[curr_num]:
                nums[i], nums[curr_num] = nums[curr_num], nums[i]

            else:
                i = i + 1


        # If num could be placed in its correct position than it is placed
        # only num that cannot be placed in its correct position, remains

        for i in range(len(nums)):
            if nums[i] != i+1: # index is 1 less than the number
                return i + 1

        # If we reach here, num after the range [1,n] i.e. n+1 is the first missing
+ve
        return len(nums) + 1

# Time Complexity = O(n)
# Space Complexity = O(1)
```

42. Trapping Rain Water

See Approach 4 in solutions

```
# For each element in the array, find minimum of next maximum height of bars on bot
h the sides minus its own height.

# For a given index, find next greater to the right and left, take min of them and
substract height of index to get trapped water for that index

class Solution:
    def trap(self, height: List[int]) -> int:
        max_seen = float('-inf')

        L = [0] * len(height)
        for i in range(len(height)):
            if height[i] > max_seen:
                L[i] = height[i]
                                max_seen = height[i]
            else:
                L[i] = max_seen

        max_seen = float('-inf')
        R = [0] * len(height)
        for i in reversed(range(len(height))):
            if height[i] > max_seen:
                R[i] = height[i]
                                max_seen = height[i]
            else:
                R[i] = max_seen

        result = 0
        for i in range(len(height)):
            result = result + (min(L[i], R[i]) - height[i])

        return result


# Time Complexity = O(n)
# Space Complexity = O(n)
```

45. Jump Game II

```
# It is guaranteed to have a solution i.e. you will definitely reach last element

# Iterate thru the list (except last index since it is destination) and keep track
of max index that can be reached (max_reach)
# if this index was the max_reach possible until now deduced via jump_end_index
# then this max_reach becomes the new jump_end_index and jump count increments


class Solution:
    def jump(self, nums: List[int]) -> int:
        max_reach = 0
        jump_end_index = 0
        total_jump = 0

        for i, jump in enumerate(nums[:-1]):
            # nums[:-1] is not used because at that point we have reached the targe
t
            max_reach = max(max_reach, i + jump)

            # if this index was the max_reach possible until now (deduced via jump_
end_index)
            # then this max_reach becomes the end index of this portion of jump and
jump count increments
            if i == jump_end_index:
                total_jump = total_jump + 1
                jump_end_index = max_reach

        return total_jump


# Time Complexity = O(n)
# Space Complexity = O(1)
```

```
# BFS sol

from collections import deque
class Solution:
    def jump(self, nums: list[int]) -> int:

        # Find the number of nodes
        n = len(nums)

        # Initialize the queue and a set to keep track of visited nodes
        queue, visited = deque([0]), set([0])

        # Initialize the number of step
        step = 0

        # Iterate until the queue is empty
        while queue:

            # Find the number of nodes visitable at this step
            k = len(queue)

            # Process all nodes at this step
            for _ in range(k):

                # Pop a node
                node = queue.popleft()

                # If we reached the final node, return the number of step
                if node == n - 1:
                    return step

                # Add all unvisited next nodes into the queue
                for nextNode in range(node + 1, node + nums[node] + 1):

                    if nextNode in visited:
                        continue

                    queue.append(nextNode)
                    visited.add(nextNode)

            # Increment the number of step
            step += 1
```

46. Permutations

[1,2,3]

nums = [1, 2, 3] permutation = [] i = 0 nums = [2, 3] permutation = [1] i = 0 nums = [3] permutation = [1, 2] i = 0 nums = [] permutation = [1, 2, 3] unwinding unwinding i = 1 nums = [2] permutation = [1, 3] i = 0 nums = [] permutation = [1, 3, 2] unwinding unwinding unwinding i = 1 nums = [1, 3] permutation = [2] i = 0 nums = [3] permutation = [2, 1] i = 0 nums = [] permutation = [2, 1, 3] unwinding unwinding i = 1 nums = [1] permutation = [2, 3] i = 0 nums = [] permutation = [2, 3, 1] unwinding unwinding unwinding i = 2 nums = [1, 2] permutation = [3] i = 0 nums = [2] permutation = [3, 1] i = 0 nums = [] permutation = [3, 1, 2] unwinding unwinding i = 1 nums = [1] permutation = [3, 2] i = 0 nums = [] permutation = [3, 2, 1] unwinding unwinding unwinding

https://www.youtube.com/watch?v=KukNnoN-SoY (https://www.youtube.com/watch?v=KukNnoN-SoY)

https://leetcode.com/problems/combination-sum/discuss/16502/A-general-approach-to-backtracking-questions-in-Java-(Subsets-Permutations-Combination-Sum-Palindrome-Partitioning) (https://leetcode.com/problems/combination-sum/discuss/16502/A-general-approach-to-backtracking-questions-in-Java-(Subsets-Permutations-Combination-Sum-Palindrome-Partitioning))

https://leetcode.com/problems/permutations/discuss/360280/Python3-backtracking (https://leetcode.com/problems/permutations/discuss/360280/Python3-backtracking)

https://leetcode.com/problems/permutations-ii/discuss/309479/Simple-Python-DFS-solutions-for-8-backtrack-problems (https://leetcode.com/problems/permutations-ii/discuss/309479/Simple-Python-DFS-solutions-for-8-backtrack-problems)

https://www.geeksforgeeks.org/time-complexity-permutations-string/ (https://www.geeksforgeeks.org/time-complexity-permutations-string/)

```
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:

        result = []

        def backtrack(nums, output):
            if len(nums) == 0:
                result.append(output)
                return

            for i in range(len(nums)):
                backtrack(nums[:i] + nums[i+1:], output + [nums[i]])

        backtrack(nums, [])
        return result

# Time Complexity = O(N * N!) (Upper bound) + Slicing operation = O(N)
# Space complexity : O(N * N!) since one has to keep N! solutions.
```

47. Permutations II

https://leetcode.com/problems/combination-sum/discuss/16502/A-general-approach-to-backtracking-questions-in-Java-(Subsets-Permutations-Combination-Sum-Palindrome-Partitioning) (https://leetcode.com/problems/combination-sum/discuss/16502/A-general-approach-to-backtracking-questions-in-Java-(Subsets-Permutations-Combination-Sum-Palindrome-Partitioning))

https://leetcode.com/problems/permutations-ii/discuss/309479/Simple-Python-DFS-solutions-for-8-backtrack-problems (https://leetcode.com/problems/permutations-ii/discuss/309479/Simple-Python-DFS-solutions-for-8-backtrack-problems)

https://www.youtube.com/watch?v=KukNnoN-SoY (https://www.youtube.com/watch?v=KukNnoN-SoY)

```
class Solution:
    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        def dfs(nums, permutation, result):
            if nums == []:
                result.append(permutation)

            for i,x in enumerate(nums):
                dfs(nums[:i] + nums[i+1:], permutation + [nums[i]], result)

        result = []
        dfs(nums, [], result)

        # to find uniques in list of list, simply doing set() won't cut it!
        # also set => hash, list is mutable hence cannot be the key, need to conver
t to tuple
        s = set()
        for l in result:
            s.add(tuple(l))
        return list(s)

# Note instead of deduping at last using set, we can use result as set instead of l
ist

# Time Complexity = O(N * N!) (Upper bound) + Slicing operation = O(N)
# Space complexity : O(N!) since one has to keep N! solutions.
```

48. Rotate Image

```
# Transpose + reverse each row since it is a square matrix

# Transpose of square matrix = diagonal remains same, elements on one side of diago
nal get swapped
# need access to only side of diagonal to do the swap

class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place instead.
        """
        # rememeber this is square matrix so: # of rows = # of cols
        # also, that's why transpose can be done in below way i.e. col start from r
ow+1
        for i in range(len(matrix)):
            for j in range(i+1,len(matrix[0])):
                matrix[j][i], matrix[i][j] = matrix[i][j], matrix[j][i]

        for i in range(len(matrix)):
            matrix[i].reverse()

# Time Complexity = O(n^2)
# Space Complexity = O(1)
```

49. Group Anagrams

```
# key - sorted string, value - input string
# dict key = tuple, value = list (key needs to be tuple: immutable since sorted(st
r) returns list which is mutable )
# One -pass solution

from collections import defaultdict
class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        result = defaultdict(list)

        for s in strs:
            result[tuple(sorted(s))].append(s) # append function
        return result.values()

# The builtin list type should not be used as a dictionary key. Note that since tup
les are
# immutable, they do not run into the troubles of lists - they can be hashed by the
ir contents
# without worries about modification.

# Time Complexity: O(n k log k), where n is the length of strs, and k is the maximu
m length of a string in strs. Outer loop = O(n) and sort each string O(k log k)

# Space Complexity = O(n k) # max hash table size with n keys and max k values to e
ach key
```

50. Pow(x, n)

```
#  n is int
# n can be -ve, 0 and +ve (odd and even) : total 4 cases

class Solution:
    def myPow(self, x: float, n: int) -> float:
        if n < 0:
            x = 1/x
            n = -n

        if n == 0:
            return 1

        answer = 1
        current_product = x

        i = n
        while i > 0:
            if i % 2 == 1: # this will be hit twice, once when i gets odd and at la
st when i reaches 1, hence return variable is 'answer'
                answer = answer * current_product
                i = i-1

            else:
                current_product = current_product * current_product
                i = i/2

        return answer



# Time Complexity = O(log n)
# Space Complexity = O(1)
```

51. N-Queens

Checkout neetcode

53. Maximum Subarray

```
# Kadane's Algo

# Initialize first element as max sum and cum sum
# for every element from 1 to n, either it is part of running continuous sum or sta
rt of new continuous sum depending on whether continuous running sum is greater tha
n current num or not
# keep track of max sum

class Solution:
    def maxSubArray(self, nums: 'List[int]') -> 'int':

        cum_sum = max_sum = nums[0]
        for i in range(1, len(nums)):
                cum_sum = max(nums[i], cum_sum + nums[i])
                max_sum = max(cum_sum, max_sum)

        return max_sum

# Time Complexity = O(n)
# Space Complexity = O(1)
```

---

54. Spiral Matrix

http://theoryofprogramming.com/2017/12/31/print-matrix-in-spiral-order/
(http://theoryofprogramming.com/2017/12/31/print-matrix-in-spiral-order/)

```
# top, left, right and bottom pointers
# dir var
# condition to check before traversing any dir: while (top<=bottom) and (left<=righ
t)


class Solution:
    def spiralOrder(self, matrix: List[List[int]]) -> List[int]:
        if len(matrix) == 0 or len(matrix[0]) == 0:
            return []

        top = 0
        bottom = len(matrix) - 1
        left = 0
        right = len(matrix[0]) - 1


        dir = 1
        result = []
        while (top<=bottom) and (left<=right):
            if dir == 1:
                for i in range(left,right+1):
                    result.append(matrix[top][i])
                top=top+1
                dir = dir + 1

            elif dir == 2:
                for i in range(top,bottom+1):
                    result.append(matrix[i][right])
                right = right - 1
                dir = dir + 1

            elif dir == 3:
                for i in range(right,left-1,-1):
                    result.append(matrix[bottom][i])
                bottom = bottom -1
                dir = dir + 1

            elif dir == 4:
                for i in range(bottom,top-1,-1):
                    result.append(matrix[i][left])
                left = left + 1
                dir = 1

        return result

# Time Complexity = O(n)
# Space Complexity = O(1), if result array space is not included
```

55. Jump Game

```
# Iterate thru the list and keep track of max index that can be reached
# At any point while iterating thru the list, if max index that can be reached up u
ntil this point is less than current index, return False

class Solution:
    def canJump(self, nums: List[int]) -> bool:
        # keep track of the maximum distance that can be reached from the current p
osition
        max_reach = 0

        # iterate through the nums list
        for i, jump in enumerate(nums):
            # if the current position is greater than the max distance that can be
reached, return False
            if max_reach < i:
                return False
            # update the max distance that can be reached from the current position
            max_reach = max(max_reach, i + jump)

        # if the loop is finished, return True as the last index can be reached
        return True




# Time Complexity = O(n)
# Space Complexity = O(1)
```

56. Merge Intervals

https://www.youtube.com/watch?v=5rFZIPNH0Yw&list=PLxQ8cCJ6LyOYCas1Ln-L8kCBquxw20ljC&index=10 (https://www.youtube.com/watch?v=5rFZIPNH0Yw&list=PLxQ8cCJ6LyOYCas1Ln-L8kCBquxw20ljC&index=10)

```
# Sort each interval by start time
# If end time of prev interval > begin time of next interval -> merge with max end time of last interval and current interval
# To accomplish this, keep the prev interval in the output list for comparison and modify it if merge happens


class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals.sort(key=lambda pair: pair[0])
        output = [intervals[0]]

        for start, end in intervals[1:]:
            lastEnd = output[-1][1]

            if start <= lastEnd:
                # merge
                output[-1][1] = max(lastEnd, end)
            else:
                output.append([start, end])
        return output

# Time Complexity = O(n log n) for sorting
# Space Complxity = O(n) for sorting
```

57. Insert Interval

https://www.youtube.com/watch?v=A8NUOmlwOlM&t=1s (https://www.youtube.com/watch?v=A8NUOmlwOlM&t=1s)

```
# Intervals are already sorted and non-overlaping
# new interval needs to be inserted in sorted order and may need merging if overlap
happens
# one simple approach is to insert in sorted order by begin time and then merge (2
pass thru interval list but still O(n))
# Better approach is to insert and merge in one pass thru interval list (O(n))


# Iterate thru interval list and check where each interval lies w.r.t new interval

# 3 cases:
# interval is "clearly" before new interval i.e. interval[end] < newInterval[begi
n], add interval to left list
# interval is "clearly" after new interval i.e. interval[begin] > newInterval[end],
add interval to right list
# new interval is overlapping other intervals in the interval list i.e. merge needs
to happen.  Take min of begin and max of end to merge to form new interval
# concatenate left, merged interval and right list

class Solution:
 def insert(self, intervals: List[List[int]], newInterval: List[int]) -> List[List
[int]]:
        l, r = [], []
        end = 1
        begin = 0

        for interval in intervals:
            if interval[end] < newInterval[begin]:
                l.append(interval)
            elif interval[begin] > newInterval[end]:
                r.append(interval)
            else:
                newInterval = (min(interval[begin], newInterval[begin]), \
                            max(interval[end], newInterval[end]))

        return l + [newInterval] + r

# Time Complexity = O(n), array is already sorted and one-pass
# Space Complexity = O(1) excluding returned list (i.e. l and r)
```

59. Spiral Matrix II

```
class Solution:
    def generateMatrix(self, n: int) -> List[List[int]]:
        matrix = [[0 for _ in range(n)] for _ in range(n)]

        top = 0
        bottom = len(matrix) - 1
        left = 0
        right = len(matrix[0]) - 1


        dir = 1
        result = []
        counter = 0
        while (top<=bottom) and (left<=right):
            if dir == 1:
                for i in range(left,right+1):
                    counter  = counter + 1
                    matrix[top][i] = counter
                top=top+1
                dir = dir + 1

            elif dir == 2:
                for i in range(top,bottom+1):
                    counter  = counter + 1
                    matrix[i][right] = counter
                right = right - 1
                dir = dir + 1

            elif dir == 3:
                for i in range(right,left-1,-1):
                    counter  = counter + 1
                    matrix[bottom][i] = counter
                bottom = bottom -1
                dir = dir + 1

            elif dir == 4:
                for i in range(bottom,top-1,-1):
                    counter  = counter + 1
                    matrix[i][left] = counter
                left = left + 1
                dir = 1

        return matrix

# Time Complexity = O(n)
# Space Complexity = O(n), matrix with n elements is populated
```

62. Unique Paths

https://www.youtube.com/watch?v=GO5QHC_BmvM (https://www.youtube.com/watch?v=GO5QHC_BmvM)

```
# Base case: 0th row and 0th column = 1 (# of unique paths to traverse 0th row or 0
th column )
class Solution:
    def uniquePaths(self, m: int, n: int):
        if m == 0 or n == 0: # edge case
            return 0

                # Initialize with 1s (especially for top row and leftmost column)
        path_sum =[[1 for _ in range(n)] for _ in range(m)]

        for i in range(1,m):
            for j in range(1,n):
                path_sum[i][j] = path_sum[i-1][j] + path_sum[i][j-1]

        return path_sum[m-1][n-1]

# Time and Space complexity = O(m*n)
```

63. Unique Paths II

```
class Solution:
    def uniquePathsWithObstacles(self, obstacleGrid: List[List[int]]) -> int:

        # Flip 1 to 0 and vice-versa
        ROWS, COLS = len(obstacleGrid), len(obstacleGrid[0])
        for row in range(ROWS):
            for col in range(COLS):
                if obstacleGrid[row][col] == 0:
                    obstacleGrid[row][col] = 1
                else:
                    obstacleGrid[row][col] = 0

        # Fill 1st col if prev row value is not obstacle
        for row in range(1, ROWS):
            obstacleGrid[row][0] = obstacleGrid[row][0] * obstacleGrid[row-1][0]

                # Fill 1st row if prev col value is not obstacle
        for col in range(1, COLS):
            obstacleGrid[0][col] = obstacleGrid[0][col] * obstacleGrid[0][col-1]

       # Fill remaining rows and cols if cell value is not obstacle
             for row in range(1, ROWS):
            for col in range(1, COLS):
                if obstacleGrid[row][col] == 0:
                    continue
                else: # "+" since no of possible unique paths
                    obstacleGrid[row][col] = obstacleGrid[row-1][col] + obstacleGri
d[row][col-1]

        return obstacleGrid[ROWS-1][COLS-1]

# Time Complexity = O(m*n)
# Space Complexity = O(1) , no extra space used
```

64. Minimum Path Sum

```
# For the first row and first column, the minimum path sum can only be reached by m
oving right or down from the previous cell.
# Base case: 0th row and 0th column = cumulative sum

# min for ith row and jth column = Take min from left,top and add to current value

class Solution:
    def minPathSum(self, grid: List[List[int]]) -> int:
        ROWS = len(grid)
        COLS = len(grid[0])

        for row in range(1,ROWS):
            grid[row][0] += grid[row-1][0]

        for col in range(1, COLS):
            grid[0][col] += grid[0][col-1]

        for row in range(1, ROWS):
            for col in range(1, COLS):
                grid[row][col] += min(grid[row-1][col], grid[row][col-1])

        return grid[ROWS-1][COLS-1]

# Time Complexity = O(m*n)
# Space Complexity = O(1)
```

---

66. Plus One

```
# 3 cases: (scan from last to first)
# last digit < 9
# all digits != 9,  e.g. 499
# all digits 9, e.g. 999

class Solution:
    def plusOne(self, digits: List[int]) -> List[int]:

        for i in range(len(digits)-1, -1, -1):
            if digits[i] < 9:
                digits[i] = digits[i] + 1
                return digits  #imp to return here e.g. 499, 123
            else:
                digits[i] = 0

        digits.append(0)
        digits[0] = 1
        return digits

# Time Complexity = O(n), one-pass
# Space Complexity  = O(1), using the same array in-place
```

67. Add Binary

Samas Add Strings problem, cannot use the modulus and quotient logic as add strings problem

69. Sqrt(x)

```
# Square root of non-negative int => returns floor of result

# for x >= 2, square root is always between 2 and floor(x/2) -> binary search
# return right since need to return floor value if exact integer sqrt cannot be fou
nd
# edge case: if x < 2

class Solution:
    def mySqrt(self, x: int) -> int:
        if x < 2: # edge case 0,1
            return x

        left = 2 # start from 2
        right = x // 2 # floor int division

        while left <= right:
            # in binary search use this formula to find middle, works when left = 0
and when left = any other number (2 in this case)
            sqrt = left + (right - left) //2
            sqr = sqrt * sqrt

            if sqr == x:
                return sqrt

            if sqr > x:
                right = sqrt -1
            elif sqr < x:
                left = sqrt + 1


        return right  # for x = 3 etc. (if exact sqrt int is not found, return the
floor (sqrt))

# Time Complexity = O(log n)
# Space Complexity = O(1)
```

70. Climbing Stairs

```
# for n=1 -> 1, n=2 -> 2, n=3 -> 3, n=4-> 5, n=5 -> 8 i.e. f(x) = f(x-1) + f(x-2) e
xcept for n= 1 and 2
# fibonacci series

class Solution:
    def climbStairs(self, n: int) -> int:
        if n==1:
            return 1
        if n==2:
            return 2

        first = 1
        second = 2

        for i in range(3, n+1):  # n+1 since range function excludes end
            third = first + second
            first = second
            second = third

        return third

# Time Complexity = O(n)
# Space Complexity = O(1)
```

---

71. Simplify Path

```
# Split by / -> that means that whatever we have between two / characters is either
a directory name or a special character and we have to process them accordingly (di
rectory name -> append to stack)
# 3 special chars: ., .., //
# . or empty string (between / /)-> no op
# .. -> pop an entry from stack if stack is not empty  [e.g. /a/b/c/..  -> /a/b]
# not special char -> add it to stack
# print by joining all elements in stack with / and beginning with / as well

class Solution:
    def simplifyPath(self, path: str) -> str:

        # Initialize a stack
        stack = []

        # Split the input string on "/" as the delimiter and process each portion o
ne by one

        for portion in path.split("/"):

            # If the current component is a "..", then
            # we pop an entry from the stack if it's non-empty
            if portion == "..":
                if stack:
                    stack.pop()
            elif portion == "." or not portion:
                # A no-op for a "." or an empty string
                continue
            else:
                # Finally, a legitimate directory name, so we add it
                # to our stack
                stack.append(portion)

        # Stich together all the directory names together
        final_str = "/" + "/".join(stack)
        return final_str

# Time Complexity = O(n) where n = total chars in input
# Space Complexity = O(n)
```

72. Edit Distance

https://www.youtube.com/watch?v=XYi2-LPrwm4&t=1216s (https://www.youtube.com/watch?v=XYi2-LPrwm4&t=1216s)

```
# Bottom up DP

# base case:
# i) If len of any word is 0, then min num of operations required to convert one st
ring to another = len(non-empty string)
# ii) if both words are same or empty then min num of operations required to conver
t one string to another = 0

# Rationale: Example: word1 = "abd" word2 = "acd"
# Take 2 ptrs, i and j, for each char in word1 and word2
# if char[i] == char[j] = > 0 operation, i and j both increment by 1 (i.e. in dp ta
ble we take value from diagonal)
# else => 1 operation, either insert, delete or replace
# insert: i, j+1
# delete: i+1, j
# replace: i+1, j+1
# i.e. if chars are not same then while filing dp table we take 1 + min value in ((
i,j+1), (i,j+1) and (i+1, j+1) direction

class Solution:
    def minDistance(self, word1: str, word2: str) -> int:

                # initialize with inf since min
        dp = [[float("inf")] * (len(word2)+1) for _ in range(len(word1)+1)]

        # Base case: last row (empty string)
        for col in range(len(word2), -1, -1):
            dp[len(word1)][col] = len(word2) - col

       # Base case: last col (empty string)
        for row in range((len(word1)), -1, -1):
            dp[row][len(word2)] = len(word1) - row

                # Fill remaining rows and cols
                # if word1[char] == word2[char], dp[i][j] = dp[i+1][j+1], 0 operati
on
                # else: 1 operation
                # insert: i, j+1
                # delete: i+1, j
                # replace: i+1, j+1

        for row in range(len(word1)-1, -1,-1):
            for col in range(len(word2)-1, -1, -1):
                if word1[row] == word2[col]:
                    dp[row][col] = dp[row+1][col+1]
                else:
                    dp[row][col] = 1+ min(dp[row][col+1], dp[row+1][col], dp[row+1]
[col+1])

        return dp[0][0]
```

```
# Time and Space Complexity = O( len(word1) + len(word2) )
```

73. Set Matrix Zeroes

```
# Brute force: Iterate thru the entire matrix and if an element is zero, store its
row and column in a set and then again iterate over the entire matrix and if its ro
w and column matches whatever was stored in first pass, make it all zeros. Space Co
mplexity = O(M+N) as need to store all rows and cols which are zero

# Need space complexity: O(1)

# Use the first row and first column to track rows and columns where 0 is seen
# Now since first row and first col are used for tracking, first find out whether a
ny element in first row or  # col is zero, since their values will be over written
# matrix[0][0] also needs extra var to keep track of whether it is zero or not



class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place instead.
        """
        col0_make_zero = False
        row0_make_zero = False
        row0_col0_make_zero = False

        if matrix[0][0] == 0:
            row0_col0_make_zero = True

        for col in range(len(matrix[0])):
            if matrix[0][col] == 0:
                row0_make_zero = True

        for row in range(len(matrix)):
            if matrix[row][0] == 0:
                col0_make_zero = True

        for row in range(1, len(matrix)):
            for col in range(1, len(matrix[0])):
                if matrix[row][col] == 0:
                    matrix[row][0] = 0
                    matrix[0][col] = 0

        for row in range(1, len(matrix)):
            for col in range(1, len(matrix[0])):
                if matrix[row][0] == 0 or matrix[0][col] == 0:
                    matrix[row][col] = 0

        if row0_col0_make_zero:
            for col in range(len(matrix[0])):
                matrix[0][col] = 0
            for row in range(len(matrix)):
                matrix[row][0] = 0
```

```
        if col0_make_zero:
            for row in range(len(matrix)):
                matrix[row][0] = 0

        if row0_make_zero:
            for col in range(len(matrix[0])):
                matrix[0][col] = 0


# Time Complexity = O(M*N)
# Space Complexity = O(1)
```

74. Search a 2D Matrix

```
# Unroll the matrix into an array from index 0 to index m*n-1
# Now given an index, row = index // n (quotient) and col = index %n (remainder) wh
ere n = # of columns
# Usual Binary search

class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        m = len(matrix)
        if m == 0:
            return False
        n = len(matrix[0])

        low = 0
        high = m*n-1

        while low<=high:
            middle = low + (high-low) // 2
            middle_element = matrix[middle //n][middle % n]

            if target == middle_element:
                return True

            if target > middle_element:
                low = middle+1
            elif target < middle_element:
                high = middle-1

        return False

# Time Complexity = O(log mn)
# Space Complexity = O(1)
```

75. Sort Colors

https://www.youtube.com/watch?v=4xbWSRZHqac&t=688s (https://www.youtube.com/watch?v=4xbWSRZHqac&t=688s)

```
# p0 ptr is used to track rightmost boundary 0s + position where next 0 will land
# p2 ptr is used to track leftmost boundary of 2s + position where next 2 will land
# To traverse the input array another ptr curr is used, traversal happens upto and
including p2 (since p2 will be one index less than all 2s encountered)


class Solution:
    def sortColors(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        p0 = 0  # rightmost boundary 0s + position where next 0 will land
        p2 = len(nums) - 1 # leftmost boundary of 2s + position where next 2 will l
and

        curr = 0
        while curr <= p2:  # <= since p2 is one index less than all 2s and needs to
be checked as well

                    # swap, increment p0, increment curr
            if nums[curr] == 0:
                nums[curr], nums[p0] = nums[p0], nums[curr]
                curr = curr + 1
                p0 = p0 + 1

                        # swap, decrement p2, DO NOT increment curr since it could
be 0 after swap which means in next iteration it needs to go towards left
            elif nums[curr] == 2:
                nums[curr], nums[p2] = nums[p2], nums[curr]
                p2 = p2 -1
                # not needed curr = curr + 1

                        # increment curr
            else: # for 1s in the middle
                curr = curr + 1


# Time Complexity = O(n)
# Space Complexity = O(1)
```

https://www.interviewbit.com/tutorial/insertion-sort-algorithm/ (https://www.interviewbit.com/tutorial/insertion-sort-algorithm/) Time Complexity in worst case O(n^2) but Space Complexity is O(1) example - > [10,12,14,11]

```
def insertionSort(arr):
    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):
        key = arr[i]
        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >=0 and key < arr[j] :
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
```

https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheMergeSort.html (https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheMergeSort.html)

```
def mergeSort(alist):
    print("Splitting ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)

        i=0
        j=0
        k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] <= righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1

        while i < len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j < len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
    print("Merging ",alist)

alist = [54,26,93,17,77,31,44,55,20]
mergeSort(alist)
print(alist)
```

https://www.interviewbit.com/tutorial/quicksort-algorithm/ (https://www.interviewbit.com/tutorial/quicksort-algorithm/) https://www.youtube.com/watch?v=uXBnyYuwPe8 (https://www.youtube.com/watch?v=uXBnyYuwPe8)

```
# This function takes last element as pivot, places
# the pivot element at its correct position in sorted
# array, and places all smaller (smaller than pivot)
# to left of pivot and all greater elements to right
# of pivot

# i keeps track of the element just smller than pivot
# j keeps moving forward if element at j is greater than pivot
def partition(arr,low,high):
    i = ( low-1 )         # index of smaller element
    pivot = arr[high]     # pivot

    for j in range(low , high):

        # If current element is smaller than or
        # equal to pivot
        if arr[j] <= pivot:

            # increment index of smaller element
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]

    arr[i+1],arr[high] = arr[high],arr[i+1]
    return ( i+1 )

# Function to do Quick sort
def quickSort(arr,low,high):
    if low < high:

        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr,low,high)

        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)
```

76. Minimum Window Substring

substring of s and not of t

https://leetcode.com/problems/minimum-window-substring/discuss/26808/here-is-a-10-line-template-that-can-solve-most-substring-problems (https://leetcode.com/problems/minimum-window-substring/discuss/26808/here-is-a-10-line-template-that-can-solve-most-substring-problems)

https://www.youtube.com/watch?v=MK-NZ4hN7rs (https://www.youtube.com/watch?v=MK-NZ4hN7rs) https://medium.com/outco/how-to-solve-sliding-window-problems-28d67601a66 (https://medium.com/outco/how-to-solve-sliding-window-problems-28d67601a66)

https://www.techiedelight.com/sliding-window-problems/ (https://www.techiedelight.com/sliding-window-problems/)

```
# 1. counter var
# 2. how hash table of t is manipulated
from collections import Counter
class Solution:
    def minWindow(self, s: str, t: str) -> str:
        if len(t) > len(s):
            return ""

        if t =="" or s=="":
            return ""



        t_hash = Counter(t) #default dict with zero initialization for new keys as
encountered
        start = 0
        end = 0
        counter = len(t)
        min_len = float('inf')
        min_start = 0
        # move end to find a valid window.
        while end < len(s):
                        if t_hash.get(s[end], None) != None:
                                    if t_hash[s[end]] > 0:
                                            counter = counter -1

            # Need to decrement t_hash due to duplicates e.g. 'bba' and 'ab'
            t_hash[s[end]] = t_hash[s[end]] - 1
            end = end + 1
            # move start to find smaller window
            while counter == 0:
                if (end - start < min_len):
                    min_start = start
                    min_len = end - start
                # if the char at start position is part of t
                t_hash[s[start]] = t_hash[s[start]] + 1
                if t_hash[s[start]] > 0:
                    counter = counter + 1
                start = start + 1

        if min_len != float('inf'):
            return s[min_start: min_start+min_len]
        return ""
    # The key part is t_hash[s[end]] = t_hash[s[end]] -1  ; We decrease count for e
ach char in s. If it does not exist in t, the count will be negative.

# Time Complexity = O(len(s) + len(t))
# Space Complexity = O(len(s) + len(t)) -> when window size = entire string s lengt
h, and t has all unique chars
```

77. Combinations

https://www.youtube.com/watch?v=q0s6m7AiM7o&t=613s (https://www.youtube.com/watch?v=q0s6m7AiM7o&t=613s)

https://www.youtube.com/watch?v=7IQHYbmuoVU (https://www.youtube.com/watch?v=7IQHYbmuoVU)

Another solution similar to permutations: https://leetcode.com/problems/combinations/discuss/26990/Easy-to-understand-Python-solution-with-comments (https://leetcode.com/problems/combinations/discuss/26990/Easy-to-understand-Python-solution-with-comments).

https://v4.software-carpentry.org/python/alias.html (https://v4.software-carpentry.org/python/alias.html) #What does [:] do in Python? It's a slicing, and what it does depends on the type of population . If population is a list, this line will create a deep copy of the list. For an object of type tuple or a str , it will do nothing (the line will do the same without [:])

Because the list nums is being modified during the function calls. If you just append it to the output you append a reference (pointer) to nums not the actual list which means that after nums is modified from some other recursive function it will be "changed" in the output list that stores the reference to nums. In the end, output will contain pointers that will point to the same result (whatever was the last change in nums). So you need to make a deep copy of nums. I suggest you to look over list aliasing in Python.

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

        result = []
        def backtrack(start, output):
            if len(output) == k:  # Base condition
                result.append(output[:]) # Imp: List Aliasing in python since list
is mutable
                return

            # most imp piece of code, frequently re-used in multiple problems
            for i in range(start, n+1):
                output.append(i)
                backtrack(i+1, output) # recurse from i+1
                output.pop() # After base condition is met - backtrack


        backtrack(1, [])
        return result

# Time Complexity = O(k * nCk)
# Space Complexity = O( k * nCk)

# each time you run the backtracking method it will place 1 number in the correct p
osition.
# so how many numbers are there in the final result?
# that is = No. of numbers in one possible combination * No. of possible combinatio
ns. right?
# No. of numbers in one possible combination = k
# No. possible combinations. = nCk
```

78. Subsets

```
# Find all combinations of size 1,2...len(nums)

class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        def dfs(start, subset,k):
            if len(subset) == k:
                result.append(subset[:])
                                return
            for i in range(start, len(nums)):
                subset.append(nums[i])
                dfs(i+1, subset,k)
                subset.pop()

        result = []
        for k in range(len(nums)+1): # Find all combinations of size 0 (empty) ,1,
2...len(nums)
            dfs(0, [], k)
        return result

# Time complexity: O(N * 2^N) to generate all subsets and then copy them into outpu
t list
# Space Complexity: O(N * 2^N) to generate all subsets since each of N elements cou
ld be present or absent.
```

79. Word Search

https://www.youtube.com/watch?v=pfiQ_PS1g8E (https://www.youtube.com/watch?v=pfiQ_PS1g8E)

Basic idea is DFS with either a set to keep track of which characters were explored or directly mark expored character as * (invalid) and unmark after dfs backtracks.

https://cs.stackexchange.com/questions/96626/whats-the-big-o-runtime-of-a-dfs-word-search-through-a-matrix (https://cs.stackexchange.com/questions/96626/whats-the-big-o-runtime-of-a-dfs-word-search-through-a-matrix) Time Complexity: The complexity will be $O(m*n*4$ ^s) where m is the no. of rows and n is the no. of columns in the 2D matrix and s is the length of the input string.

When we start searching from a character we have 4 choices of neighbors for the first character and subsequent characters have only 3 or less than 3 choices but we can take it as 4 (permissible slopiness in upper bound). This slopiness would be fine in large matrices. So for each character we have 4 choices. Total no. of characters are s where s is the length of the input string. So one invocation of search function of your implementation would take $O($4^s) time.

Also in worst case the search is invoked for $m*n$ times. So an upper bound would be $O(m*n*4\hat{}s)$.

Space Complexity: O(length of board) + O(length of word) -> size of visited matrix + maximum length of recursive call stack

```
# Example 1: Target word = SEE
# ABCE
# SFCS
# ADEE
# First S will not yield match, second S will

# Example 2: Target word = ABCESEEEFS
# ABCE
# SFES
# ADEE
# Two ways to go after C to E, one way no match, another way match, hence backtrack
and remove visited letters until backtrack


# for each new DFS search, keep track of nodes visited
# if at a certain point, two ways to go and you pick one and it does not lead to ma
tch then backtrack and remove visited letters from path until backtrack
# cannot re-visit the same cell again in one DFS path


class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        ROWS, COLS = len(board), len(board[0])


        def dfs(r, c, i):
            if i == len(word):
                return True

            if (
                r < 0
                or c < 0
                or r >= ROWS
                or c >= COLS
                or word[i] != board[r][c]
                or (r, c) in path
            ):
                return False

            path.add((r, c))
            res = (
                dfs(r, c+1, i + 1)
                or dfs(r - 1, c, i + 1)
                or dfs(r+1, c, i + 1)
                or dfs(r, c - 1, i + 1)
            )
            # if at a certain point, two ways to go and you pick one and it does no
t lead to match then backtrack and remove visited letters until backtrack
            path.remove((r, c))
            return res
```

```
        for r in range(ROWS):
            for c in range(COLS):

                if board[r][c] == word[0]:
                    path = set() # resets for every dfs call
                    # if current search did not find the complete word, then try se
arch again with matching first letter, if any one search returns True, return True
                    if dfs(r, c, 0):
                        return True
        return False


 # Time Complexity : O(m*n*4^s) where m = # of rows, n = # of columns and s = lengt
h of word
# Space Complexity: O(length of board) + O(length of word) -> size of visited matri
x + maximum length of recursive call stack
```

80. Remove Duplicates from Sorted Array II

```
# sorted array
# 2 ptr approach: 1 ptr is responsible for writing unique values in our input arra
y, while 2nd ptr will read the input array and pass all the distinct elements to 1s
t ptr

# initialize two pointers i and insert_index at 1
# if element i and i - 1 are same, increment count else reset count

# if count < 2 (keep array[i]) -> store array[i] at insert_index and increment inse
rt_index
# otherwise, just keep moving forward i.


class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:

        insert_index = 1
        count = 0

        for i in range(1,len(nums)):
            if nums[i-1] == nums[i]:
                count = count + 1
            else:
                count = 0

            if count < 2:
                nums[insert_index] = nums[i]
                insert_index = insert_index + 1

        return insert_index


# Time Complexity = O(n)
# Space Complexity = O(1)
```

81. Search in Rotated Sorted Array II

```
# It is guaranteed that the array is rotated
# duplicates allowed

# Same as search in rotated sorted array except way to avoid duplicates

# while (begin + 1 <= len(nums)-1 and nums[begin] == nums[begin + 1]):
#    begin = begin + 1
# while (end -1 >= 0 and nums[end] == nums[end - 1]):
#     end = end - 1

# The above while loop can increase the complexity from O(log n) to O(n) in worst c
ase where all elements are same


class Solution:
    def search(self, nums: List[int], target: int) -> int:
        begin = 0
        end = len(nums) - 1

        while begin <= end:
            # To avoid duplicates (check bounds)
            while (begin + 1 <= len(nums)-1 and nums[begin] == nums[begin + 1]):
                begin = begin + 1
            while (end -1 >= 0 and nums[end] == nums[end - 1]):
                end = end - 1


            mid = begin + (end-begin) // 2

            if nums[mid] == target:
                return  True

            elif nums[mid] > nums[end]:
                if nums[mid] > target and nums[begin] <= target: # <=, since nums[m
id] == target is checked
                    end = mid -1
                else:
                    begin = mid +1

            else: # nums[mid] < nums[end]
                if nums[mid] < target and nums[end] >= target: # >=
                    begin = mid + 1
                else:
                    end = mid - 1

        return False
```

84. Largest Rectangle in Histogram

https://www.youtube.com/watch?v=vcv3REtIvEo (https://www.youtube.com/watch?v=vcv3REtIvEo)

```
# For every bar ‘x’, we calculate the area with ‘x’ as the smallest bar in the rect
angle. If we calculate such area for every bar ‘x’ and find
the maximum of all areas, our task is done.
# Now how to calculate area with ‘x’ as the smallest bar? We need to know index of
the first smaller (smaller than ‘x’) bar on left of ‘x’ and index of first smaller
bar on right of ‘x’.
# next smaller is strictly smaller while previous smaller also considers equal elem
ents (in addition to smaller elements)

# We traverse all bars from left to right, maintain a stack of bars. Every bar is p
ushed to stack once. A bar is popped from stack when a bar of smaller height is see
n.
# When a bar is popped, we calculate the area with the popped bar as smallest/leftm
ost bar for the bars in stack. How do we get left and right indexes of the popped b
ar – the current index (bar of smaller height than popped element) tells us the ‘ri
ght index’ and index of previous item than popped element - 1(width of popped bar i
tself) in stack is the ‘left index’.

# E.g. [6,7,5], first max area = 7, next max area = 12, next max area = 15

class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        stack = [-1] # mark end of stack
        max_area = 0
        for i in range(len(heights)):
            while stack[-1] != -1 and heights[stack[-1]] >= heights[i]:
                current_height = heights[stack.pop()]
                current_width = i - stack[-1] - 1
                max_area = max(max_area, current_height * current_width)
            stack.append(i)

        # Fully traversed heights array but stack is not empty
        while stack[-1] != -1:
            current_height = heights[stack.pop()]
            current_width = len(heights) - stack[-1] - 1  # popped element as leftm
ost boundary and len(heights) as rightmost boundary
            max_area = max(max_area, current_height * current_width)
        return max_area
```

```
let maxArea = 0;
  for (let i = 0; i < heights.length; i++) {
    let currentHeight = heights[i];
    let width = nextSmaller[i] - previousSmaller[i] - 1;
    maxArea = Math.max(maxArea, currentHeight * width);
  }

  return maxArea
```

88. Merge Sorted Array

look solution 2 : with O(1) space complexity

```
# Compare from last and fill in from last
# Take care of the case when nums2 need to be added to the beginning of nums1

class Solution:
    def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:
        """
        Do not return anything, modify nums1 in-place instead.
        """
        p1 = m-1
        p2 = n-1
        p = m+n-1

        # while there are still elements to compare
        while p1 >= 0 and p2 >= 0:

            if nums2[p2] > nums1[p1]:
                nums1[p] = nums2[p2]
                p = p-1
                p2 = p2-1

            elif nums2[p2] <= nums1[p1]:
                nums1[p] = nums1[p1]
                p = p-1
                p1 = p1 -1
        # Imp: add missing elements from nums2 for below cases
        # if num1 = [0] and num2 = [1]  i.e. m=0, n=1
        # if num1 = [2,0] and num2 =[1]

        nums1[:p2 + 1] = nums2[:p2 + 1]

# Time Complexity = O (n + m)
# Space Complexity = O (1)
```

91. Decode Ways

https://leetcode.com/problems/decode-ways/solutions/4454037/97-43-easy-solution-with-explanation
(https://leetcode.com/problems/decode-ways/solutions/4454037/97-43-easy-solution-with-explanation)

```
# Initialize an array dp of size n + 1, where n = length of the input string.
# Set dp[0] = 1, as there is one way to decode an empty string and dp[1] = 0 if fir
st char is '0' else dp[1] = 1
# In general, dp[i] = number of ways to decode a string of length i i.e. from index
0 to index i-1
# be mindful of '0'


# Iterate through the string starting from index 2 up to n
# a) Convert the current one-digit and two-digit substrings to integers
# b) Check if valid single digit decode is possible -> If the one-digit substring i
s not '0', update dp[i] by adding dp[i - 1] since all the ways up to (i-1)th char n
ow lead up to i-th char too.
# c) Check if valid 2 digit decode is possible ->  If the two-digit substring is be
tween 10 and 26 (inclusive), update dp[i] by adding dp[i - 2] since all the ways up
to (i-2)th char now lead up to i-th char too
# d) result in dp[len(string)] = no. of ways to decode full string

class Solution:
    def numDecodings(self, s: str) -> int:
        # Initialize
        dp = [0 for _ in range(len(s) + 1)]

        dp[0] = 1
        dp[1] = 0 if s[0] == '0' else 1

        # Iterate from 2 to len of string
        for i in range(2, len(dp)):

            # Check if successful single digit decode is possible.
            if s[i - 1] != '0':
                dp[i] += dp[i - 1]

            # Check if successful two digit decode is possible.
            two_digit = int(s[i - 2 : i])
            if two_digit >= 10 and two_digit <= 26:
                dp[i] += dp[i - 2]

        return dp[len(s)]

# Time Complexity = O(n), one pass
# Space Complexity = O(n), space for dp array
```

94. Binary Tree Inorder Traversal

https://www.techiedelight.com/inorder-tree-traversal-iterative-recursive/ (https://www.techiedelight.com/inorder-tree-traversal-iterative-recursive/)

https://www.youtube.com/watch?v=nzmtCFNae9k (https://www.youtube.com/watch?v=nzmtCFNae9k)

```
# Do not put anything in stack before while loop

class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]:
        if root is None: # edge case
            return []

        stack = []
        cur = root

        result = []
        while stack or cur: # Imp: or
            if cur:
                stack.append(cur)
                cur = cur.left
            else:
                cur = stack.pop()
                result.append(cur.val)
                cur = cur.right

        return result
```

---

97. Interleaving String

```
# why 2 pointer approach will fail - https://www.youtube.com/watch?v=3Rw3p9LrgvE&t=
1s
# if a char in s3 matches both s1 and s2, which one it needs to pick bcz future mat
ch will depend on this pick

# https://leetcode.com/problems/interleaving-string/solutions/3956989/dp-easy-best-
approach/?envType=study-plan-v2&envId=top-interview-150

# https://www.youtube.com/watch?v=ih2OZ9-M3OM

# Top Down DP with base case

class Solution:
    def isInterleave(self, s1: str, s2: str, s3: str) -> bool:
           # If the total length of s1 and s2 is not equal to s3, it's impossible
        if len(s1) + len(s2) != len(s3):
            return False

                # Create a 2D DP array to store intermediate results
        dp = [[False] * (len(s2)+1) for _ in range(len(s1)+1)]

        # Base Case: empty strings can always interleave to form an empty string
        dp[0][0] = True

        # Fill the first row
        for j in range(1, len(s2)+1):
            if s3[j-1] == s2[j-1] and dp[0][j-1] == True:
                dp[0][j] = True
            else:
                dp[0][j] = False

        # Fill the first col
        for i in range(1, len(s1)+1):
            if s3[i-1] == s1[i-1] and dp[i-1][0] == True:
                dp[i][0] = True
            else:
                dp[i][0] = False

        # Fill the DP array from 1,1 position by comparing s1 or s2 with 2nd elemen
t of s3 and left or top dp array
                # which  in turn looks to see whether string s3 has 1st element cov
ered already
        for i in range(1, len(s1)+1):
            for j in range(1, len(s2)+1):
                dp[i][j] = (s3[i+j-1] == s1[i-1] and dp[i-1][j]) or (s3[i+j-1]== s2
[j-1] and dp[i][j-1])

        return dp[len(s1)][len(s2)]

# Time and Space Complexity = O(len_s1 * len_s2)
```

98. Validate Binary Search Tree

```
# Inorder traversal of binary tree is in ascending order => BST
# keep track of prev value and keep comparing
class Solution:
    def isValidBST(self, root: TreeNode) -> bool:
        if root is None:
            return True

        stack = []
        cur = root
        prev_val = float('-inf')
        while stack or cur:
            if cur:
                stack.append(cur)
                cur = cur.left

            else:
                cur = stack.pop()
                if cur.val <= prev_val: # question mentions < and >, for test cases
it is <= and >=
                    return False
                prev_val = cur.val

                cur = cur.right
        return True
```

100. Same Tree

https://leetcode.com/articles/same-tree/ (https://leetcode.com/articles/same-tree/)

```
# different binary trees can have the same inorder, preorder, or postorder traversa
l
class Solution:
    def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:

        # p and q are both None
        if p == None and q == None:
            return True
        # one of p and q is None
        if p == None or q == None:
            return False
        # p and q values do not match
        if p.val != q.val:
            return False
        return self.isSameTree(p.right, q.right) and self.isSameTree(p.left, q.lef
t)

# Time Complexity = O(n)
# Space Complexity = O(n) for recursion stack
```

101. Symmetric Tree

See BFS solution (iterative)

```
class Solution:
    def isSymmetric(self, root: TreeNode) -> bool:
        if root is None: # edge case
            return True

        queue = []
        queue.append(root)
        queue.append(root)

        while queue:
            t1 = queue.pop(0)
            t2 = queue.pop(0)

            if t1 is None and t2 is None:
                continue

            if t1 is None or t2 is None:
                return False

            if t1.val != t2.val:
                return False

            queue.append(t1.left)
            queue.append(t2.right)
            queue.append(t1.right)
            queue.append(t2.left)

        return True
```

102. Binary Tree Level Order Traversal

https://medium.com/@timpark0807/leetcode-is-easy-binary-tree-patterns-1-2-6e1793b76415
(https://medium.com/@timpark0807/leetcode-is-easy-binary-tree-patterns-1-2-6e1793b76415)

```
# Visit->Left-> Right
# take care of level, prev_level
# Add new list when encounter new level

class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if root is None: # edge case
            return []

        queue = []

        queue.append((root,0))

        result = []
        prev_level = -1
        while queue:
            cur, level = queue.pop(0)
            if level != prev_level: # if encounter new level, create new list
                result.append([])


            result[level].append(cur.val)
            if cur.left:
                queue.append((cur.left, level+1))

            if cur.right:
                queue.append((cur.right, level+1))

            prev_level = level

        return result

# Time Complexity = O(n)
# Space Complexity = O(n) for storing elements in queue
```

103. Binary Tree Zigzag Level Order Traversal

https://www.techiedelight.com/spiral-order-traversal-binary-tree/ (https://www.techiedelight.com/spiral-order-traversal-binary-tree/)

```
# [1,2,3,4,null,null,5]: if at a level only single child exists, usual approach of
appending in queue in
# opposite way (i.e. right first then left) at alternate levels will fail

# An easier approach is: perform level order traversal as usual and at last reverse
alternate list
from collections import deque
class Solution:
    def zigzagLevelOrder(self, root: TreeNode) -> List[List[int]]:
        if root is None:
            return

        result = []
        q = deque()
        q.append(root)
        flag = True

        while q:
            nodeCount = len(q)
            if flag:
                subresult = []
                while nodeCount > 0:
                    curr = q.popleft() # popleft
                    subresult.append(curr.val)

                    # left then right
                    if curr.left:
                        q.append(curr.left)
                    if curr.right:
                        q.append(curr.right)

                    nodeCount = nodeCount - 1
            else:
                subresult = []
                while nodeCount > 0:
                    curr = q.pop() # pop
                    subresult.append(curr.val)

                    # right then left + appendleft
                    if curr.right:
                        q.appendleft(curr.right)

                    if curr.left:
                        q.appendleft(curr.left)

                    nodeCount = nodeCount - 1

            result.append(subresult)
            flag = not flag

        return result
```

```
# Time Complexity = O(n)
# Space Complexity = O(n) for storing elements in queue

# As one can see, at any given moment, the node_queue would hold the nodes that are
at most across two levels.Therefore, at most, the size of the queue would be no mor
e than 2L, assuming L is the maximum number of nodes that might reside on the same
level. Since we have a binary tree, the level that contains the most nodes could oc
cur to consist all the leave nodes in a full binary tree, which is roughly L = N/2.
Therefore, 2*N/2 = N
```

104. Maximum Depth of Binary Tree

https://medium.com/@timpark0807/leetcode-is-easy-binary-tree-patterns-1-2-6e1793b76415 (https://medium.com/@timpark0807/leetcode-is-easy-binary-tree-patterns-1-2-6e1793b76415)

```
class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        if root is None: #edge case
            return 0

        queue = []
        queue.append((root,1))

        max_depth = 0
        while queue:
            cur, level = queue.pop(0)
            max_depth = max(max_depth, level)

            if cur.left:
                queue.append((cur.left, level+1))

            if cur.right:
                queue.append((cur.right, level+1))

        return max_depth

# Time Complexity = O(n)
# Space Complexity = O(n) for storing elements in queue
```

105. Construct Binary Tree from Preorder and Inorder Traversal

https://www.youtube.com/watch?v=PoBGyrIWisE (https://www.youtube.com/watch?v=PoBGyrIWisE) see comments - https://leetcode.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/discuss/34579/Python-short-recursive-solution (https://leetcode.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/discuss/34579/Python-short-recursive-solution)

```
# inorder + postorder or inorder + preorder are both unique identifiers of whatever
binary tree
# root comes from preorder (DLR)
# left and right subtrees come from inorder (LDR)

from collections import deque
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> TreeNode:
        preorder = deque(preorder)
        inorder_map = {num:index for index, num in enumerate(inorder)}

        def build_tree(start, end):
            if start > end: # stop condition for recursion
                return

            root_val = preorder.popleft()
            root = TreeNode(root_val)
            idx = inorder_map[root_val]
            root.left = build_tree(start, idx-1)
            root.right = build_tree(idx+1, end)
            return root

        return build_tree(0, len(inorder)-1)


# Time Complexity: deque(preorder) = O(n), inorder_map = O(n), build tree = O(n); t
otal = O(n)
# Space Complexity: deque = O(n), inorder_map = O(n), build_tree = O(n) to store en
tire tree; total = O(n)
```

108. Convert Sorted Array to Binary Search Tree

See solution Approach 1

```
# Inorder traversal is not a unique identifier of BST. Several BSTs can have same i
norder traversal(see
# example in solution)
# At the same time both preorder and postorder traversals are unique identifiers of
BST
# Approach: Always choose Left Middle Node as a Root

class Solution:
    def sortedArrayToBST(self, nums: List[int]) -> TreeNode:
        def helper(left, right):
            if left > right: # exit condition for recursion
                return

            mid = left + (right-left) //2

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

        return helper(0,len(nums)-1)

# Time Complexity = O(n) since each node is visited once
# Space Complexity = O(n) to construct the BST and O(log n) for recursion stack (he
ight-balanced)
```

110. Balanced Binary Tree

https://www.techiedelight.com/calculate-height-binary-tree-iterative-recursive/
(https://www.techiedelight.com/calculate-height-binary-tree-iterative-recursive/)

```
class Solution:
    def isBalanced(self, root: TreeNode) -> bool:
        if root is None: # edge case: empty tree is balanced by definition
            return True

        def get_height(node):
            if node is None:
                return 0

            left = get_height(node.left)
            right = get_height(node.right)

            if abs(left - right) > 1 or left == -1 or right == -1:  # -1 value bein
g bubbled up in recursion
                return -1
            return max(left, right) + 1



        return get_height(root) != -1


# Time Complexity = O(n)
# Space Complexity = O(n) for recursion stack, if the tree is completely skewed tow
ards one side
```

112. Path Sum

```
class Solution:
    def hasPathSum(self, root: TreeNode, sum: int) -> bool:
        if root is None:
            return []

        stack = []
        stack.append((root, sum - root.val))

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

            if remains == 0 and node.left is None and node.right is None: # remain
= 0 only at leaf node
                return True

            if node.left:
                stack.append((node.left, remains - node.left.val))

            if node.right:
                stack.append((node.right, remains - node.right.val))

        return False

# Time Complexity = O(n)
# Space Complexity = O(n)
```

---

118. Pascal's Triangle

```
# Step 1: Create a list of list with every list size of row+1, initialized with 0
# Step 2: Make first and last element of every list 1
# Step 3: Loop thru the list of list and every element that is 0, add value of [pre
v row] [prev col ] + [prev row][same col]

class Solution:
    def generate(self, numRows: int) -> List[List[int]]:
        triangle = []
        for row in range(numRows):
            triangle.append([0] * (row+1))
            triangle[row][0] = 1
            triangle[row][-1] = 1

        for i in range(numRows):
            for j in range(len(triangle[i])):
                if triangle[i][j] == 0:
                    triangle[i][j] = triangle[i-1][j-1] + triangle[i-1][j]

        return triangle

# Time and Space Complexity = O(size of triangle) or O(numRows ^2)
```

120. Triangle

```
# Column can be expressed in terms of rows

# 3 conditions starting from row 1 (row 0 remains as is) to get minimum
# If col == 0: There is only one cell above, located at (row - 1, col)
# If col == row: There is only one cell above, located at (row - 1, col - 1)
# if col != row and col != 0:  Take min of two cells above, located at (row - 1, co
l - 1) and (row - 1, col)

# keep running minimum sum in-place


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

        for row in range(1, len(triangle)):
            for col in range(row + 1):
                if col == 0:
                    triangle[row][col] += triangle[row-1][col]
                if col == row:
                    triangle[row][col] += triangle[row-1][col-1]
                if col != row and col != 0:
                    triangle[row][col] += min(triangle[row-1][col-1], triangle[row-
1][col])

        return min(triangle[-1])

# Time Complexity = O(size of triangle)
# Space Complexity = O(1), filling the triangle itself for min cum sum
```

---

121. Best Time to Buy and Sell Stock

```
# We need to find the largest peak following the smallest valley.
# We can maintain two variables - minprice and maxprofit corresponding to the small
est valley and
# maximum profit (maximum difference between selling price and minprice) obtained s
o far

# Input can never be negative int
# If the input is in descending order, max profit = 0

import sys
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        min_price = sys.maxsize
        max_profit = 0

        for i in range(len(prices)):
            if prices[i] < min_price:
                min_price = prices[i]
            elif prices[i] - min_price > max_profit:
                max_profit = prices[i] - min_price

        return max_profit

# Time Complexity = O(n)
# Space Complexity = O(1)
```

122. Best Time to Buy and Sell Stock II

```
# We can simply go on crawling over the slope and keep on adding the profit obtaine
d from every
# consecutive transaction, if there is any profit i.e. if prices[i] > prices[i-1]

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        max_profit = 0

        for i in range(1,len(prices)):
            if prices[i] > prices[i-1]:
                max_profit = max_profit + (prices[i] - prices[i-1])

        return max_profit

# Time Complexity = O(n)
# Space Complexity = O(1)
```

125. Valid Palindrome

```
class Solution:
    def isPalindrome(self, s: str) -> bool:
        if len(s) == 0:
            return True

        begin = 0
        end = len(s) -1

        while begin < end:
            if s[begin].isalnum() and s[end].isalnum():
                if s[begin].lower() != s[end].lower():
                    return False
                else:
                    begin = begin +1
                    end = end - 1

            if s[begin].isalnum() == False:
                begin = begin + 1
            if s[end].isalnum() == False:
                end = end - 1

        return True

# Time Complexity = O(n)
# Space Complexity = O(1)
```

127. Word Ladder

https://www.youtube.com/watch?v=h9iTnkgv05E&t=1008s (https://www.youtube.com/watch?v=h9iTnkgv05E&t=1008s)

```
# Shortest - BFS

# Create a dict with key =  one char masked from each word and value = all the word
s that satisfy that criteria
# Put the begin word in queue, pop it
# For all words formed by masking one char from this begin word and not visited, ad
d it to queue and mark them visited

# Stop condition: if word popped from queue matches end word

from collections import defaultdict
class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> in
t:
        if endWord not in wordList:
            return 0

        d = defaultdict(list)
        for word in wordList:
            for i in range(len(word)):
                d[word[:i] + "*" + word[i+1:]].append(word)

        visited = []
        q = deque()
        q.append([beginWord,1])

        while q:
            current_word, level = q.popleft()
            if current_word == endWord:
                return level

            for i in range(len(current_word)):
                for word in d[current_word[:i] + "*" + current_word[i+1:]]:

                    if word not in visited:
                        visited.append(word)
                        q.append([word,level+1])
        return 0


# Time Complexity: O(M^2 *N) where M = length of each word and N = total number of
words
# For each word in the word list, we iterate over its length to find all the interm
ediate words corresponding to it. Since the length of each word is M and # we have
N words, the total number of iterations the algorithm takes to create dict is M×N
# Additionally, forming each of the intermediate word takes O(M)time because of the
substring operation used to create the new string.

# Space Complexity = O(M^2 *N) where  N = total number of words and M = length of e
ach word
```

```
# Each word requires M intermediate words = O(M^2)
# Total N words
```

128. Longest Consecutive Sequence

```
# Brute force approach is to sort and find the longest streak of consecutive nums b
ut it is O(n log n)

# https://www.youtube.com/watch?v=P6RZZMu_maU&t=214s

# lookup is done using hash table/set hence O(1)
# convert nums list to nums set (for faster lookup O(1))

# Another approach is find the number of distinct sequences and their length
# Start of a sequence = if prev num is not found
# Initialize this num as current num and start the streak looking for next numbers
# keep track of max_streak



class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        nums = set(nums) # hash table lookup O(1)
        max_streak = 0

        for num in nums:
            if num-1 not in nums: # start of a new sequence
                curr_num = num  # start the streak
                current_streak = 1

                while curr_num + 1 in nums:
                    curr_num = curr_num + 1
                    current_streak = current_streak + 1

                max_streak = max(max_streak, current_streak)

        return max_streak

# Time Complexity = O(n), though it appears 'while' loop is inside 'for' loop but i
nner 'while'
# loop runs for nums for which outer 'for' loop doesn't, also creating set() is O
(n)
# Space Complexity = O(n) for creating set
```

130. Surrounded Regions

https://www.youtube.com/watch?v=9z2BunfoZ5Y (https://www.youtube.com/watch?v=9z2BunfoZ5Y)

```
# Find all regions/islands of Os which start from border, except these regions conv
ert all Os to X

# 3 steps:
# Do a DFS/BFS from all border cells with O value and mark this island with any tem
p char (e.g. T) since we don't want to flip them
# Iterate over entire grid and change all remaining Os to X
# Now change all border islands back to O

class Solution:
    def solve(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        ROWS = len(board)
        COLS = len(board[0])

        def dfs(row,col):
            if row < 0 or row >= ROWS or col < 0 or col >= COLS \
                or board[row][col] == "X" \
                or board[row][col] == "T":
                return

            board[row][col] = "T"
            directions = [[1,0], [-1,0], [0,1], [0,-1]]
            for direction in directions:
                nr, nc = row + direction[0], col + direction[1]
                dfs(nr, nc)

        # 1. (DFS) Capture unsurrounded regions (O -> T)
        for row in range(ROWS):
            for col in range(COLS):
                if board[row][col] == "O" and (row in [0, ROWS-1] or col in [0, COL
S-1]):
                    dfs(row,col)

        # 2. Capture surrounded regions (O -> X)
        for row in range(ROWS):
            for col in range(COLS):
                if board[row][col] == "O":
                   board[row][col] = "X"

        # 3. Uncapture unsurrounded regions (T -> O)
        for row in range(ROWS):
            for col in range(COLS):
                if board[row][col] == "T":
                   board[row][col] = "O"
```

131. Palindrome Partitioning

```
# Find all substrings of the given string and check each of them for palindrome pro
perty
# A bunch of substrings would be repeated so keep track of substrings that are chec
ked for palindrome property in a dp table to avoid duplication
# If the current substring ends at index end, end+1 becomes the start index for the
next recursive call

# note: this is different from combination backtracking questions in the sense that
continuous elements need to be checked
# E.g. aab is checked as a, a , b, ab, aa, b, aab

class Solution:
    def partition(self, s: str) -> List[List[str]]:
        res = [] # vector to store all possible partitions
        path = [] # vector to store current partition

        dp = [[0 for j in range(len(s))] for i in range(len(s))]

        # helper function that checks if a substring is a palindrome
        def isPalindrome(s, start, end):
            if dp[start][end] == 1:
                return True
            while start <= end: # iterate through the substring
                if s[start] != s[end]: # check if current characters are not equal
                    return False # if they are not, return false
                start += 1
                end -= 1

            return True # if we reach this point, the substring is a palindrome

                # imp snippet
        # helper function that uses recursion to find all possible partitions
        def backtrack(start_index):
            if start_index == len(s): # base case: if we have reached the end of th
e string
                res.append(path[:]) # add current partition to the result vector
                return

            for end_index in range(start_index, len(s)): # iterate thru all substri
ngs starting from the current index
                    if isPalindrome(s, start_index, end_index): # check if the curr
ent substring is a palindrome
                        dp[start_index][end_index] = 1
                        path.append(s[start_index:end_index+1]) # if it is, add it
to the current partition
                        backtrack(end_index+1) # call function recursively with the
next index as the starting point
                        path.pop() # backtrack to check for other partitions

        backtrack(0) # call backtrack function to start recursion
        return res
```

```
#Time Complexity : O(N*2^N), for a string of length N, 2^N possible substrings and
O(N) time needed to generate one substring and check for plaindrome
# Space Complexity : O(N *N) for dp table
```

133. Clone Graph

https://www.youtube.com/watch?v=mQeF6bN8hMk (https://www.youtube.com/watch?v=mQeF6bN8hMk)

```
# Original graph format -> node: [neighbor1, neighbor2, ..]
# Create a hashmap to maintain mapping between old nodes and corresponding new node
s created
# While creating neighbors of new node, it is possible that the neighbor might have
been created before

# From the original node's val, create a new node and add the mapping to hashmap
# Traverse the neighbors of original node, and append it to the neighbor list of ne
w node while checking whether the neighbor was already created using hashmap

class Solution:
    def cloneGraph(self, node: 'Node') -> 'Node':
        OldToNew = {}

        def clone(node):
            if node in OldToNew:
                return OldToNew[node]

            copy = Node(node.val)
            OldToNew[node] = copy

            for neighbor in node.neighbors:
                copy.neighbors.append(clone(neighbor))
            return copy

        return clone(node) if node else None

# Time Complexity = O(E +V)
# Space Complexity = O(V), OldToNew hashmap
```

134. Gas Station

if you start your journey from gas station 'a' and get stuck at gas station 'b', it means that you cannot reach station 'b' from any of the stations between 'a' and 'b'. In other words, for any intermediate station 'c' between 'a' and 'b', the total amount of gas available at 'c' and all subsequent stations is not sufficient to cover the cost of traveling from 'c' to the next station.

```
# No solution if gas[i] - cost[i] over all i is -ve starting from any index (will b
e same if you start from # any other index)
# if solution exists:
# Traverse through gas and cost. For each index i, increment total_gain and curr_ga
in by gas[i] - cost[i].
# If curr_gain is smaller than 0, we will test if station i + 1 is a valid starting
station by setting start_index as i + 1, resetting curr_gain to 0, and repeating
# When the iteration is complete, if total_gain is smaller than 0, return -1. Other
wise, return start_index



class Solution:
    def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:

        total_gain = 0
        current_gain = 0
        start_index = 0

        for i in range(len(cost)):
            total_gain = total_gain + gas[i] - cost[i]
            current_gain = current_gain + gas[i] - cost[i]

            if current_gain < 0: # restart
                current_gain = 0
                start_index = i + 1

        if total_gain < 0:
            return -1
        else:
            return start_index


# Time Complexity = O(n)
# Space Complexity = O(1)
```

135. Candy

```
# Going forward to reward higher rating on the right. Going backward for higher rat
ing on left
# Generally, there is only one rule: If you see a higher rating on the next child,
it deserves one more candies if it didn't


class Solution:
    def candy(self, ratings: List[int]) -> int:

        n = len(ratings)
        give = [1] * n

        for i in range(1,n):
            if ratings[i] > ratings[i-1]:
                give[i] = give[i-1] + 1

        for i in range(n-2, -1, -1):
            if ratings[i] > ratings[i+1] and give[i] <= give[i+1]:
                give[i] = give[i+1] + 1

        return sum(give)

# Time Complexity = O(n)
# Space Complexity = O(n) for give array
```

136. Single Number

```
# Method 1: Hash but will have O(n) time and space complexity

# Method 2 : XoR => a xor 0 = a, a xor a = 0 and a xor b xor a = b, space complexit
y will be O(1)
# XOR = If the bits are the same, the result is 0. If the bits are different, the r
esult is 1



class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        result = 0
        for num in nums:
            result = result ^ num
        return a
```

138. Copy List with Random Pointer

Similar to Clone graph LC:133

https://www.youtube.com/watch?v=OvpKeraoxW0 (https://www.youtube.com/watch?v=OvpKeraoxW0)

```
# deep copy means new copy at new address
# now if the 'pointed to' node does not exist => it's corresponding address also do
es not exist,
# then how to fill pointer values of current node ??

# Create a visited dictionary to hold old node reference as "key" and new node refe
rence as the
# "value"

class Solution:
    def __init__(self):
        self.visited = {}

    def getClonedNode(self, node):
        if node: # check for null
            if node in self.visited:
                return self.visited[node]
            else:
                self.visited[node] = Node(node.val, None, None) # create new node a
nd update dict
                return self.visited[node]
        return None

    def copyRandomList(self, head: 'Node') -> 'Node':
        if not head:
            return head

        old_node = head
        new_node = Node(old_node.val, None, None) # create first node
        self.visited[old_node] = new_node # update dict

        while old_node:
            new_node.random = self.getClonedNode(old_node.random)
            new_node.next = self.getClonedNode(old_node.next)

            old_node = old_node.next
            new_node = new_node.next

        return self.visited[head]

# Time Complexity = O(n) - one pass over original linked list
# Space Complexity = O(n) - create hash table for keeping track of mapping
```

139. Word Break

https://www.youtube.com/watch?v=Sx9NNgInc3A&t=761s (https://www.youtube.com/watch?v=Sx9NNgInc3A&t=761s)

Note: Time Complexity of BFS solution is much worse than DP solution

```
# Bottom-up DP

class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:

        # dp[i] = is the sub-string starting at index i found in dict? True or Fals
e
        dp = [False] * (len(s) + 1)
        # base case: sub-string starting after len of string = True
        dp[len(s)] = True

        for i in range(len(s), -1, -1):
            for w in wordDict:

                # think of "leetcode" string and "code" word in dict
                # within bounds and sub-string matches with word from word dict
                if i + len(w) <= len(s) and s[i:i+len(w)] == w:
                    dp[i] = dp[i + len(w)]

                # if dp[i] becomes true, no need to check any other word in dict
                if dp[i] == True:
                       break

        # if dp[0] == True, all the subwords of string are found in word dict
        return dp[0]

# Time Complexity = O(length of string  * length of word dict * avg length of word
in word dict), nested for loop
# Space Complexity = O(len(string)), to store dp[i] values
```

141. Linked List Cycle

```
# slow pointer moves 1 step, fast pointer moves 2 steps
# If they meet = cycle
# if fast pointer reaches end (both for odd length and even length list) = no cycle


# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None


class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        slow, fast = head, head

        while fast and fast.next: # length of list could be odd or even
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                return True
        return False



# Time Complexity = O(n)
# Space Complexity = O(1)
```

144. Binary Tree Preorder Traversal

https://www.techiedelight.com/preorder-tree-traversal-iterative-recursive/ (https://www.techiedelight.com/preorder-tree-traversal-iterative-recursive/) https://www.youtube.com/watch?v=elQcrJrfObg (https://www.youtube.com/watch?v=elQcrJrfObg)

```
class Solution:
    def preorderTraversal(self, root: TreeNode) -> List[int]:
        if root is None: #edge case
            return []

        stack = []
        stack.append(root)

        result = []
        while stack:
            cur = stack.pop()

            result.append(cur.val)

            # Stack is LIFO hence first right then left
            if cur.right:
                stack.append(cur.right)

            if cur.left:
                stack.append(cur.left)

        return result
```

145. Binary Tree Postorder Traversal

https://www.techiedelight.com/postorder-tree-traversal-iterative-recursive/ (https://www.techiedelight.com/postorder-tree-traversal-iterative-recursive/) https://www.youtube.com/watch?v=qT65HltK2uE (https://www.youtube.com/watch?v=qT65HltK2uE)

```python
# Iterative Preorder Traversal: Tweak the Order of the Output
# Two stacks (one to reverse at last)
# DLR in implementation (LRD in theory) + Reverse
class Solution:
    def postorderTraversal(self, root: TreeNode) -> List[int]:
        if root is None: # edge case
            return []

        stack = []
        stack.append(root)

        result = []
        while stack:
            cur = stack.pop()
            result.append(cur.val)

            if cur.left:
                stack.append(cur.left)

            if cur.right:
                stack.append(cur.right)

        return result[::-1]
```

146. LRU Cache

```
# get and put can be done using hashmap in O(1)
# however, deleting the first added key is tricky -> push whatever is touched (get
or put) to the end and then deleting head (which is least used) can be done using l
inked list in O(1) time

# OrderedDict in python = hashmap + linked list (inserts key in the order of insert
ion) with move_to_end(key) and popitem(last=False) [pop from beginning] works for t
his problem

from collections import OrderedDict
class LRUCache:

    def __init__(self, capacity: int):
        self.od = OrderedDict()
        self.capacity = capacity


    def get(self, key: int) -> int:
        # if key is not present
        if key not in self.od:
            return -1

        # if key is present, return value  and move it to end since we accessed it
        self.od.move_to_end(key)
        return self.od[key]


    def put(self, key: int, value: int) -> None:
        # if key is present, value needs to be updated and move it to end since we
accessed it
        # e.g. key already present but new value e.g. put[2,2] then put [2,1]
        if key in self.od:
            self.od.move_to_end(key)
            self.od[key] = value

        # if key not present, add key-value pair
        # orderedDict adds element to the last (append) hence move_to_end not neede
d for new entries
        else:
            self.od[key] = value

        # Capacity check, pop from start since accessed elements are moved to end
        if len(self.od) > self.capacity:
            self.od.popitem(last=False)

# Time Complexity - get(key), put(key,value) : O(1) since get/in/set/move_to_end/po
pitem can all be done in O(1)
# Space Complexity = O(capacity)
```

150. Evaluate Reverse Polish Notation

```
# In reverse Polish notation, the operators follow their operands. For example, to
add 3 and 4 together, the expression is 3 4 + rather than 3 + 4.
# The concept of a stack, a last-in/first-out construct, is integral to the left-to
-right evaluation of RPN. In the example 3 4 -, first the 3 is put onto the stack,
then the 4; the 4 is now on top and the 3 below it. The subtraction operator remove
s the top two items from the stack, performs 3 - 4, and puts the result of -1 back
onto the stack.

# // in case of division will not work as in case of -ve numbers it will go against
zero and not towards zero as specified in the problem, need to do int(left/right)

# Iterate thru the input from left to right
# if the element is not operator, append to stack
# if the element is operator, pop the last two elements from stack as right and lef
t operand + perform operation and put it back in stack

class Solution:
    def evalRPN(self, tokens: List[str]) -> int:

        operators = ['+', '-', '*', '/']
        stack = []

        def apply_operator(symbol, left, right):
            if symbol == '+':
                return left + right
            if symbol == '-':
                return left - right
            if symbol == '*':
                return left * right
            if symbol == '/':
                return int(left/right) # // will not work as in case of -ve

        for c in tokens:
            if c not in operators:
                stack.append(int(c))

            if c in operators:
                right_operand = stack.pop()
                left_operand = stack.pop()
                result = apply_operator(c, left_operand, right_operand)
                stack.append(result)

        return stack.pop()
```

151. Reverse Words in a String

```
# can't do in-place if the string is immutable in chosen language like python

# Reverse the Whole String and Then Reverse Each Word

class Solution:
    def trim_spaces(self, s: str) -> list:
        left = 0
        right = len(s) - 1

        # remove leading spaces
        while left <= right and s[left] == " ":
            left = left + 1
        # remove trailing spaces
        while left <= right and s[right] == " ":
            right = right - 1

        # reduce multiple spaces to single one in between words
        output = []
        while left <= right:
            if s[left] != " ":
                output.append(s[left])
            elif output[-1] != " ":   # useful trick
                output.append(s[left])
            left = left + 1
        return output


    def reverse(self, l: list, left: int, right: int) -> None:
        while left <= right:
            l[left], l[right] = l[right], l[left]
            left = left + 1
            right = right - 1

    def reverse_each_word(self, l: list) -> None:
        start = 0
        end = 0

       # note double while loop
        while start < len(l):
            # go to the end of the word, end is a counter so it can go beyond lengt
h of list
            while end < len(l) and l[end] != ' ':   # works for the end word as wel
l
                end = end + 1
             # reverse the word
            self.reverse(l, start, end - 1)
            # move to the next word, both start and end moves to end + 1
            start = end + 1
            end = end + 1

    def reverseWords(self, s: str) -> str:
```

```
        l = self.trim_spaces(s)
        self.reverse(l, 0, len(l)- 1)
        self.reverse_each_word(l)
        return "".join(l)

# Time Complexity = O(n)
# Space Complexity = O(n)
```

152. Maximum Product Subarray

https://www.youtube.com/watch?v=vtJvbRlHqTA (https://www.youtube.com/watch?v=vtJvbRlHqTA)

e.g. [4,-2,-3]

```
# If all numbers are +ve, max product of subarray = multiplication of all elements
in array
# Issue happens when some numbers are -ve
# depending on current num sign, prev max or prev min can become current max, there
fore we need to keep track of prev min
# E.g. -1,6,2,-2 => max product is 24, it can be seen why we want to keep track of
prev_min_prod

class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        curr_max_prod = nums[0]
        curr_min_prod = nums[0]
        result = nums[0]
        prev_max_prod = nums[0]
        prev_min_prod = nums[0]

        for i in range(1, len(nums)):
            curr_max_prod = max(prev_max_prod * nums[i], prev_min_prod * nums[i], n
ums[i])
            curr_min_prod = min(prev_max_prod * nums[i], prev_min_prod * nums[i], n
ums[i])
            result = max(result, curr_max_prod)
            prev_max_prod = curr_max_prod
            prev_min_prod = curr_min_prod

        return result

# Time Complexity = O(n)
# Space Complexity = O(1)
```

153. Find Minimum in Rotated Sorted Array

```
# No duplicates

# keep track of min value seen at every mid calculation (nums[mid] itself could be the rotation point e.g. [4,5,1,2,3]
# Find which side rotation happened by comparing mid to end and adjust begin (mid+1) and end (mid -1) pointers accordingly


class Solution:
    def findMin(self, nums: List[int]) -> int:
        start , end = 0, len(nums) - 1
        curr_min = float("inf")

        while start  <=  end :
            mid = start + (end-start) // 2
            curr_min = min(curr_min,nums[mid])

            # right has the min
            if nums[mid] > nums[end]:
                start = mid + 1

            # left has the min
            else:
                end = mid - 1

        return curr_min

# Time Complexity = O(log n)
# Space Complexity = O(1)
```

155. Min Stack

```
# Two stacks: 2nd stack to keep track of min val at every position in 1st stack
# if incoming value is lower than top most element in min_stack then push it on min
_stack else take the last element from min_stack and push it again on min_stack
# push () and pop() operations happen for both stacks

class MinStack:

    def __init__(self):
        self.stack = []
        self.min_stack = []


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


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


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


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

157. Read N Characters Given Read4

```
# readn using read4 and return the number of chars read + fill buf with the charact
ers read
# two cases: len(file) >= 4 and len(file) < 4
# Also note that buf4 (and buf) are populated after passing them as arg in fn calls

class Solution:
    def read(self, buf, n):
        buf4 = [''] * 4 # needs to be initialized
        copied_chars = 0

        chars_read = 0

        while copied_chars < n:
            chars_read = read4(buf4)
            if chars_read == 0: # e.g. file = "abc" and n = 4
                return copied_chars

            for i in range(chars_read):
                if copied_chars == n:
                    return copied_chars
                buf[copied_chars] = buf4[i]
                copied_chars = copied_chars + 1

        return copied_chars

# Time Complexity = O(n) to copy n chars
# Space Complexity = O(1), buf is given and buf4 is always of 4 chars independent o
f input
```

159. Longest Substring with At Most Two Distinct Characters

```
# Sliding window to keep only 2 distinct chars,
# As soon as num of distinct chars reaches 3, delete the leftmost char from hashmap
and adjust left ptr
# keep track of max substring length

# How to delete the leftmost char?
# a) keep track of last index seen in hashmap i.e. char:last index seen (this way w
e know which char is the the leftmost)
# b) move left ptr to this deleted char's last index seen+1

class Solution:
    def lengthOfLongestSubstringTwoDistinct(self, s: str) -> int:
        left = 0
        right = 0

        hashmap = defaultdict(int)
        max_len = 0

        while right < len(s):
            hashmap[s[right]] = right
            right = right + 1

            if len(hashmap) == 3:
                # delete the leftmost character
                del_idx = min(hashmap.values())
                del hashmap[s[del_idx]]
                # move left ptr to  deleted char's last index seen+1
                left = del_idx + 1

            max_len = max(max_len, right-left)

        return max_len

# Time Complexity = O(n)
# Space Complexity = O(1)
```

160. Intersection of Two Linked Lists

Core idea :

If tails of both list are

1) different = no intersection 2) same = intersection

To find the intersection node: a = length of list A b = length of list B k = difference of lengths = abs(a-b) if this difference is maintained and traversal happens one at a time for both lists: they meet = intersection

To maintain k difference between lists: When pA reaches the end of a list, then redirect it to the head of B (yes, B, that's right.); similarly when pB reaches the end of a list, redirect it the head of A.

https://leetcode.com/problems/intersection-of-two-linked-lists/discuss/49798/Concise-python-code-with-comments (https://leetcode.com/problems/intersection-of-two-linked-lists/discuss/49798/Concise-python-code-with-comments)

```
class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:
        a = headA
        b = headB

        while a is not b:
            if a is not None:
                a = a.next
            else:
                a = headB

            if b is not None:
                b = b.next
            else:
                b = headA

        return a

# if they didn't meet, they will hit the end at the same iteration, a == b == None
(since a and b would have traversed (len a + len b) nodes)
```

162. Find Peak Element

https://www.youtube.com/watch?v=kMzJy9es7Hc&t=402s (https://www.youtube.com/watch?v=kMzJy9es7Hc&t=402s)

```
# array is not sorted and has duplicates
# any one peak can be returned
# array[i] != array[i+1] mentioned in constraint, very important condition to remem
ber i.e. adjacent numbers cannot be same

# Modified Binary Search

# Find whether mid is on rising slope or falling slope by comparing with its right
and left neighbor
# If mid is on rising slope when comparing to its right neighbor i.e. nums[mid] < n
ums[mid+1] -> peak is on right side of mid i.e. start = mid + 1
# If mid is on declining slope when comparing to its left neighbori.e. nums[mid] <
nums[mid-1] -> peak is on left side of mid i.e. end = mid - 1
# If above 2 conditions are not true, mid is the peak element
# make sure mid+1 and mid-1 are valid indexes


class Solution:
    def findPeakElement(self, nums: List[int]) -> int:
        start = 0
        end = len(nums) - 1


        while start <= end:
            mid = start+(end-start)// 2

            if 0 <= mid+1 <= len(nums)-1 and nums[mid] < nums[mid+1]:
                start = mid + 1
            elif  0 <= mid-1 <= len(nums)-1 and nums[mid] < nums[mid-1]:
                end = mid - 1
            else: # mid is greater than both of its neighbors
                break

        return mid


# Time Complexity = O(log n)
# Space Complexity = O(1)
```

163. Missing Ranges

```
# 3 cases:
# missing range between lower and first element of list
# missing range between last element of list and upper
# missing range between elements of list: if nums[i+1] - nums[i] == 1, nothing is m
issing, else (nums[i] + 1, nums[i+1] -1) is the missing range

# Edge case list is empty

class Solution:
    def findMissingRanges(self, nums: List[int], lower: int, upper: int) -> List[Li
st[int]]:
        n = len(nums)
        missing_ranges = []
        if n == 0:
            missing_ranges.append([lower, upper])
            return missing_ranges

        # Check for any missing numbers between the lower bound and nums[0].
        if lower < nums[0]:
            missing_ranges.append([lower, nums[0] - 1])

        # Check for any missing numbers between successive elements of nums.
        for i in range(n - 1):
            if nums[i + 1] - nums[i] == 1:
                continue
            else:
                missing_ranges.append([nums[i] + 1, nums[i + 1] - 1])

        # Check for any missing numbers between the last element of nums and the up
per bound.
        if upper > nums[n - 1]:
            missing_ranges.append([nums[n - 1] + 1, upper])

        return missing_ranges

# Time Complexity = O(n)
# Space Complexity = O(1)
```

167. Two Sum II - Input Array Is Sorted

```
# We use two indices, initially pointing to the first and the last element, respect
ively.
# Compare the sum of these two elements with target.
# If the sum is equal to target, we found the exactly only solution.
# If it is less than target, we increase the smaller index by one.
# If it is greater than target, we decrease the larger index by one.

# Two pointer approach since input array is sorted
# Hash table can be used but will have space complexity = O(n), we can do better th
an that using two
# pointer approach

class Solution:
    def twoSum(self, numbers: List[int], target: int) -> List[int]:
        low = 0
        high = len(numbers)-1

        result = []

        while low<high:
            total = numbers[low] + numbers[high]

            if total < target:
                low = low+1

            elif total > target:
                high = high-1

            else: # only one unique solution with index incremented by 1 (specified
in question)
                result.append(low+1)
                result.append(high+1)
                return result

# Time Complexity = O(n)
# Space Complexity = O(1)
```

169. Majority Element

Approach 1 - Hashmap. O(n) time and O(n) space. Build a hashmap num: count then return key with max value
Approach 2 - Sort in place. O(n log n) time and O(n) space. Sort and majority element is found at index floor(n/2)

Approach 3 - Boyer-Moore Voting Algorithm. O(n) time and O(1) space https://www.geeksforgeeks.org/boyer-moore-majority-voting-algorithm/ (https://www.geeksforgeeks.org/boyer-moore-majority-voting-algorithm/) If Majority element occurs more than n//2 times then only you can apply Bayer Moore Voting Algo

This algorithm works on the fact that if an element occurs more than N/2 times, it means that the remaining elements other than this would definitely be less than N/2. So let us check the proceeding of the algorithm.

```
# If Majority element occurs more than n//2 times then only you can apply Bayer Moo
re Voting Algo

# If votes = 0 , choose a candidate from the given set of elements and increment vo
tes
# if votes != 0,
# a) if nums[i] = same as the candidate element, increase the votes
# b) Otherwise, decrease the votes( if votes become 0, select another new element a
s the new candidate)

# Two var: votes and candidate

class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        votes = 0
        candidate = None

        for i in range(len(nums)):
            if votes == 0:
                candidate = nums[i]
                votes = votes + 1
            else:
                if nums[i] == candidate:
                    votes = votes + 1
                else:
                    votes = votes - 1

        return candidate

# Time Complexity = O(n)
# Space Complexity = O(1)
```

179. Largest Number

https://leetcode.com/problems/largest-number/solutions/3212971/179-solution-with-step-by-step-explanation (https://leetcode.com/problems/largest-number/solutions/3212971/179-solution-with-step-by-step-explanation)

```
# lexicographic sorting
# The intuition here is to design a custom sorting function that compares two strin
gs—representations of the numbers—and decides which
# combination (either a + b or b + a) creates a larger number. For instance, when c
omparing 9 and 34, we check if 934 is larger than 349, so we # can position 9 befor
e 34 in the final arrangement.

# convert list of numbers to list of strings
# sort the list of strings using custom sorting function (overrides str class for <
operator (__lt__) )
# We define a  custom sorting func that compares two strings a ,b. we consider a bi
gger than b if a+b > b+a (e.g 10,2 -> 210 or 102)
# i.e. checks the pairwise concatenation and returns whichever concat results in hi
gher number e.g. 30, 3 -> 303 or 330
# edge case: if 0 is the leading digit return "0" else return the string version of
largest num formed

class LargerNumKey(str):
    def __lt__(x, y):
        # Compare x+y with y+x  to get descending order
        return x+y > y+x

class Solution:
    def largestNumber(self, nums: List[int]) -> str:
        # Convert the list of numbers to list of strings
        nums = [str(num) for num in nums]

        # Sort the list of strings using our custom sorting function
        nums.sort(key=LargerNumKey)

        # Join the sorted list of strings to form the final result
        largest_num = ''.join(nums)

        # If the largest number is 0, return "0"
        # Otherwise, return the largest number
        return "0" if largest_num[0] == "0" else largest_num

    # Time complexity = O(n log n)
    # Space Complexity = O(n)
```

189. Rotate Array

```
# edge case [1, 2,3] k = 4 => k = 1 (as 3 rotations will bring array back to its or
iginal position)

# k = k % len(nums)
# reverse the full list (swap)
# reverse from 0 to k-1
# reverse from k to len(nums) - 1

class Solution:
    def rotate(self, nums: List[int], k: int) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        def reverse(nums,start,end):
            while(start<end):
                temp = nums[end]
                nums[end] = nums[start]
                nums[start] = temp
                start = start + 1
                end = end - 1

        k = k % len(nums) # edge case
        reverse(nums,0,len(nums)-1)
        reverse(nums,0,k-1)
        reverse(nums,k,len(nums)-1)

# Time Complexity = O(n)
# Space Complexity = O(1)
```

198. House Robber

https://www.youtube.com/watch?v=73r3KWiEvyk&t=1s (https://www.youtube.com/watch?v=73r3KWiEvyk&t=1s)

```
# adjacent houses cannot be robbed i.e. if you choose house #1, cannot choose house
#2 but can choose any house from house #3, house #4, ... house #n
# max amount that can be robbed at house #3 -> either max amount robbed until house
#2 or  max amount robbed until house #1 + amount robbed at house #3

# to understand code: extend the input to left by 2 zero elements e.g. [2,7,9,3,1]
-> [0,0,2,7,9,3,1]

class Solution:
    def rob(self, nums: List[int]) -> int:
            # Initialize
        rob1 = 0
        rob2 = 0

        for num in nums:
            max_rob = max(rob2, num + rob1) # think of house #3

            rob1 = rob2
            rob2 = max_rob

        return max_rob

# Time Complexity = O(n) where n = number of elements in nums array
# Space Complexity = O(1)
```

199. Binary Tree Right Side View

See Approach 3: BFS + level size

```
# Put all children of a node, left to right in queue at a level
# Now the last element of queue is the right side view

class Solution:
    def rightSideView(self, root: TreeNode) -> List[int]:
        if root is None: # edge case
            return []

        queue = []
        queue.append(root)

        right_view = []

        while queue:
            level_length = len(queue)

            for i in range(level_length):
                cur = queue.pop(0)

                if i == level_length - 1:
                    right_view.append(cur.val)

                if cur.left:
                    queue.append(cur.left)

                if cur.right:
                    queue.append(cur.right)

        return right_view

# Time Complexity = O(n)
# Space Complexity = O(n)
```

200. Number of Islands

https://www.youtube.com/watch?v=pV2kpPD66nE (https://www.youtube.com/watch?v=pV2kpPD66nE)

```
# Usual DFS and BFS

# DFS - use visited set to keep track of all already visited nodes (in one DFS path
as well as all DFS paths)
# i) start DFS if node is land and not visited
# ii) core  DFS algo: 2 parts
# a) invalid conditions, return - out of bounds, already visited, not land
# b) valid condition - i) add this node to visited set ii) start DFS in valid direc
tions

class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        visited = set()
        ROWS = len(grid)
        COLS = len(grid[0])

        def dfs(row,col):
            if row < 0 or row >= ROWS or col < 0 or col >= COLS \
                    or grid[row][col] == "0" \
                    or (row,col) in visited:  # backtracking path when to return
                return

            visited.add((row,col))
            directions = [[1,0], [-1,0], [0,1], [0,-1]]
            for direction in directions:
                nr, nc = row + direction[0], col + direction[1]
                dfs(nr, nc)

        num_island = 0
        for row in range(ROWS):
            for col in range(COLS):
                if grid[row][col] == "1" and (row,col) not in visited:
                    num_island +=1
                    dfs(row,col)
        return num_island

# Time Complexity = O(m*n) where m = # of rows and n = # of cols
# Space Complexity = O(m*n) visited array
```

```
# BFS: not recursive, use visited set to keep track of all already visited nodes
# i) start BFS if node is land and not visited
# ii) put the starting BFS node in queue and add it to visited
# iii) while queue is not empty, pop from queue
# iv) for every neighboring node in valid directions from this popped node, check i
f it satisfies valid conditions (within bounds, not visited, land)
# a) add it to queue
# b) add it to visited


class SolutionBFS:
    def numIslands(self, grid: List[List[str]]) -> int:
        if not grid:
            return 0

        rows, cols = len(grid), len(grid[0])
        visited=set()
        islands=0

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

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

                 for dr,dc in directions:
                     r,c = row + dr, col + dc
                     if (r) in range(rows) and (c) in range(cols) and grid[r][c] ==
'1' and (r ,c) not in visited:

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

         for r in range(rows):
             for c in range(cols):

                 if grid[r][c] == "1" and (r,c) not in visited:
                     bfs(r,c)
                     islands +=1

         return islands
```

202. Happy Number

```
# Two parts:
# 1. Given a number n, what is its next number?
# 2. Follow a chain of numbers and detect if we've entered a cycle.

class Solution:
    def isHappy(self, n: int) -> bool:
        def get_next(n): # Time Complexity = O(log n)
            sum_sqr = 0
            while n > 0:
                n, digit = divmod(n,10) # extracting each digit
                sum_sqr = sum_sqr + digit ** 2

            return sum_sqr

        slow = n
        fast = get_next(n)

        while slow != fast and fast != 1: # break when cycle or fast becomes 1 -> a
ny 1 cond true
            slow = get_next(slow)
            fast = get_next(get_next(fast))

        return fast == 1

# number of digits in a number N is floor(log10(N)) + 1 e.g. 100 has 3 digits => lo
g 100 + 1 = 3
# Finding the next value for a given number has a cost of O(logn) because we are pr
ocessing each digit in the number, and the number of digits in a number is given by
log n.


# Time Complexity = O (log n), 2 cases: 1) fast becomes 1  2) slow catches up with
fast
# case 1: if there is no cycle, fast reaches 1 and slow will be halfway to 1 => O(2
log n) = O(log n)
# case 2: if there is cycle, fast will get one step closer to slow at each cycle. I
magine there are k numbers in the cycle. If they started at k - 1 places apart (whi
ch is the furthest apart they can start), then it will take k - 1 steps for the fas
t runner to reach the slow runner, which again is constant for our purposes. Theref
ore, the dominating operation is still calculating the next value for the starting
n, which is O(log n).

# Space Complexity = O(1)
```

```
# Approach 2:

class Solution:
    def isHappy(self, n: int) -> bool:

        seen = set()
        while n != 1:
            if n in seen:
                return False

            seen.add(n)
            n = sum([int(i) ** 2 for i in str(n)])

        return True


# Time Complexity = O(log n) : processing each digit of a number with n digits is l
og (n)
# Space Complexity = O(log n)
# Think about what would happen if you had a number with 1 million digits in it. Th
e first step of the algorithm would process those million digits, and then the next
value would be, at most (pretend all the digits are 9), be 81∗1,000,000=81,000,0008
1 ∗ 1,000,000 = 81,000,00081∗1,000,000=81,000,000. In just one step, we've gone fro
m a million digits, down to just 8. The largest possible 8 digit number we could ge
t is 99,9999,99999,9999,99999,9999,999, which then goes down to 81∗8=64881 ∗ 8 = 64
881∗8=648. And then from here, the cost will be the same as if we'd started with a
3 digit number.
```

205. Isomorphic Strings

```
# 1:1 mapping between both strings (not many to 1 and not 1 to many)
# 2 conditions need to be satisfied
# 1. count of unique chars should be same between both strings
# 2. count of unique char pair by zipping both strings should be same as #1

# If condition 2 is not present then if s = "bbbaaaba" and t = "aaabbbba", it will
return true but it should be false since at the start of the string b can be replac
ed by a but the second last b needs to be b and not replaced by a


class Solution:
    def isIsomorphic(self, s: str, t: str) -> bool:
        return len(set(s)) == len(set(t)) == len(set(zip(s, t)))
```

206. Reverse Linked List

```
# 4 step process

# Three pointers are needed -> prev, curr, head pointing to 3 consecutive nodes
# reverse link for 2nd node
# ptr pointing to 1st node moves to 2nd node

# At the end, return prev

# Order of operations
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        prev = None

        while head:
            curr = head
            head = head.next
            curr.next = prev
            prev = curr
        return prev

# Time Complexity = O(n)
# Space Complexity = O(1)
```

207. Course Schedule

https://www.youtube.com/watch?v=EgI5nU9etnU (https://www.youtube.com/watch?v=EgI5nU9etnU) (neetcode)

```
# Detect cycle in directed graph

# visiting set() is used to keep track of all nodes in a DFS path "starting from a
given node" i.e. all nodes that ae not fully explored
# When doing DFS from a node, it will either have 1. cycle  2. no cycle
# cycle: node found in visiting set (), return False
# no cycle:  node neighbors are empty (e.g. last node in graph), return True (and n
ot found in visiting set)


# When all the neighboring nodes of a node are explored and no cycle is found, (e.
g. last node)
# i) remove the node from visiting set ()
# ii) set neighbors of the node as [] so that it behaves like last node in graph to
avoid running DFS again from this node (optimization)


# In case graph is disconnected, need to start DFS from node in the other part

# A node remains in the DFS recursion stack until all of its branches (all nodes in
its subtree) have been explored.
# When we have examined all of a node's branches, i.e. visited all of the nodes in
its subtree, the node is removed from the DFS recursive stack

class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        preMap = {i: [] for i in range(numCourses)}

        # map each course to prereq list
        for crs, pre in prerequisites:
            preMap[crs].append(pre)

        visiting = set()

        def dfs(crs):
                    # Return conditions
            if crs in visiting:
                return False
            if preMap[crs] == []:
                return True

            # visiting: to keep track of nodes along current DFS path starting from
a node
            visiting.add(crs)

                        # explore all neighbors of a node
            for pre in preMap[crs]:
                if dfs(pre) == False:
                    return False

                         # if all the neighbors of a node were explored and no cycle w
```

```
as found, remove this node from visiting set()
                        # note: below code would run as part of backtracking for ea
ch neighbor node as well

            visiting.remove(crs)
            preMap[crs] = []  # no need to run DFS again from node from where DFS a
lready works
            return True

        for c in range(numCourses):
            if dfs(c) == False:
                return False
        return True
```

https://www.geeksforgeeks.org/detect-cycle-direct-graph-using-colors/ (https://www.geeksforgeeks.org/detect-cycle-direct-graph-using-colors/) https://github.com/harishvc/challenges/blob/master/graph-detect-cycles-DFS.py (https://github.com/harishvc/challenges/blob/master/graph-detect-cycles-DFS.py) Detect cycle in a graph

Use the following approach: consider we have three colors, and each vertex should be painted with one of these colors. "White color" means that the vertex hasn't been visited yet. "Gray" means that we've visited the vertex but haven't visited all vertices in its subtree. "Black" means we've visited all vertices in subtree and left the vertex. So, initially all vertices are white. When we visit the vertex, we should paint it gray. When we leave the vertex (that is we are at the end of the dfs() function, after going throung all edges from the vertex), we should paint it black. If you use that approach, you just need to change dfs() a little bit. Assume we are going to walk through the edge v->u. If u is white, go there. If u is black, don't do anything. If u is gray, you've found the cycle because you haven't left u yet (it's gray, not black), but you come there one more time after walking throung some path.

```
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        color_dict = {x:"WHITE" for x in range(numCourses)}
        #instead of visited -> 2 states, color -> 3 states is used which contains v
isited's 2 states
        adj_list = {x:[] for x in range(numCourses)}
        for x,y in prerequisites:
            adj_list[y].append(x)

        found_cycle = [False]  # issue with global when using recursive stack, use
list


        def dfs(start_node, adj_list, color_dict, found_cycle):

            if found_cycle[0] == True:
                return # Unwrapping recursion when condition is met

            color_dict[start_node] = "GRAY"

            for neighbor in adj_list[start_node]:
                if color_dict[neighbor] == "GRAY":
                    found_cycle[0] = True
                    return # Unwrapping recursion when condition is met
                if color_dict[neighbor] == "WHITE" and dfs(neighbor, adj_list, colo
r_dict, found_cycle) == True:
                    return True  # above 'and' is important with return value true


            color_dict[start_node] = "BLACK"




        for start_node in range(numCourses):     # need to iterate on all nodes in
case disconnected graph
            if color_dict[start_node] == "WHITE":
                dfs(start_node, adj_list, color_dict, found_cycle)
            if found_cycle[0] == True:
                break
        return not found_cycle[0] # Need to return complement value when cycle is f
ound i.e. False
```

208. Implement Trie (Prefix Tree)

https://medium.com/basecs/trying-to-understand-tries-3ec6bede0014#:~:text=A%20trie%20is%20a%20tree,are%20a%20relatively%20new%20thing (https://medium.com/basecs/trying-to-understand-tries-

3ec6bede0014#:~:text=A%20trie%20is%20a%20tree,are%20a%20relatively%20new%20thing).

https://www.youtube.com/watch?v=AXjmTQ8LEoI (https://www.youtube.com/watch?v=AXjmTQ8LEoI)

https://leetcode.com/problems/implement-trie-prefix-tree/discuss/58834/AC-Python-Solution (https://leetcode.com/problems/implement-trie-prefix-tree/discuss/58834/AC-Python-Solution)

```
# Trie has 2 elements: hashmap and a boolean var
# hashmap contains key: letter, value: address of this letter (another TrieNode)
# boolean conatins whether this letter marks the end of word

# traversing: (with and without get())

from collections import defaultdict
class TrieNode:
# Initialize your data structure here.
def __init__(self):
    self.children = defaultdict(TrieNode)
    self.is_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        current = self.root
        for letter in word:
            current = current.children[letter] # Imp -> create the key in parent ha
shmap and update the address
        current.is_word = True # set this flag to indicate that it is a valid word

    def search(self, word: str) -> bool:
        current = self.root
        for letter in word:
            current = current.children.get(letter) # Imp .get()
            if current is None: # if the word mismatches with traversal then False
                return False
        return current.is_word # if the word is longer than what we traversed upto
current then False

    def startsWith(self, prefix: str) -> bool:
        current = self.root
        for letter in prefix:
            current = current.children.get(letter)
            if current is None:
                return False
        return True # here return True instead of is_word value


# Time Complexity of insert, search and delete for trie = O(m) wher m is length of
word
# Space Complexity of insert = O(m)
# Space Complexity of search = O(1)
```

209. Minimum Size Subarray Sum

```
# can follow usual sliding window procedure since all input is +ve

# 2 pointers left and right, left to mark start of window and right to traverse the
array and keep calculating cum sum
# If cumsum < target: keep on expanding window using right ptr (keeping left ptr co
nstant)
# If cumsum >= target: in a while loop i) keep track of min length ii) shrink the w
indow by discarding value at left ptr and advancing left ptr (keeping right ptr con
stant)



class Solution:
    def minSubArrayLen(self, target: int, nums: List[int]) -> int:
        res = float('inf')
        l, total = 0, 0

        for r in range(len(nums)):
            total += nums[r]
            while total >= target:
                res = min(res, r - l + 1)
                total -= nums[l]
                l += 1
        return res if res != float('inf') else 0



# Time Complexity = O(n), each element can be visited at most 2 times, once by left
ptr and once by right ptr = 2n = O(n)
# Space Complexity = O(1)
```

210. Course Schedule II

Topological sort cannot be applied if graph has cycles In Topological Sort, the idea is to visit the parent node followed by the child node. If the given graph contains a cycle, then there is at least one node which is a parent as well as a child so this will break Topological Order.

A topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering. In a circle, through one # path u comes v and through another path v comes before u E.g. dependencies in build system

https://www.youtube.com/watch?v=Akt3glAwyfY (https://www.youtube.com/watch?v=Akt3glAwyfY)

```
# Topological sort  - cannot be done if graph has cycles
# The sequence generated by topological sort is not unique ( can have multiple topo
logical sort sequence)

# visiting set() is used to keep track of nodes in one DFS path starting from a giv
en node
# cycle: node found in visiting set (), return False
# no cycle:  node found in output list

# When all the neighboring nodes of a node are explored and no cycle is found, (e.
g. last node)
# i) remove the node from visiting set ()
# ii) add it to output list (note: adj list is created a->b i.e. if b needs to be t
aken before a hence dfs works)


class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[in
t]:
        preMap = {i: [] for i in range(numCourses)}

        for crs,pre in prerequisites:
            preMap[crs].append(pre)

        visited = set()
        output = []

        def dfs(crs):
            if crs in visited: # cycle detection
                return False
            if crs in output: # already added, no need to DFS again
                return True


            visited.add(crs)
            for pre in preMap[crs]:
                if dfs(pre) == False:
                    return False
            visited.remove(crs)
            output.append(crs)

            return True

        for crs in range(numCourses):
            if dfs(crs) == False:
                return []
        return output
```

211. Design Add and Search Words Data Structure

https://leetcode.com/problems/implement-trie-prefix-tree/ (https://leetcode.com/problems/implement-trie-prefix-tree/)

https://leetcode.com/problems/add-and-search-word-data-structure-design/discuss/59725/Python-easy-to-follow-solution-using-Trie (https://leetcode.com/problems/add-and-search-word-data-structure-design/discuss/59725/Python-easy-to-follow-solution-using-Trie).

```
from collections import defaultdict
class TrieNode:
    def __init__(self):
        self.children = defaultdict(TrieNode)
        self.is_word = False

class WordDictionary:

    def __init__(self):
        self.root = TrieNode()


    def addWord(self, word: str) -> None:
        curr = self.root
        for letter in word:
            curr = curr.children[letter]
        curr.is_word = True

    def search(self, word: str) -> bool:
        self.result = False # Imp: note use of this flag
        self.dfs(self.root, word)
        return self.result

    def dfs(self, node, word):
        if len(word) == 0 : # exhausted all the letters in the word
            if node.is_word == True:
                self.result = True
            return

        if word[0] == ".": # if letter is ., move ahead one node in trie and next l
etter in word
            for node in node.children.values():
                self.dfs(node, word[1:])
        else:
            node = node.children.get(word[0])
            if node is None: # next letter in word is not found in trie
                return
            self.dfs(node, word[1:])


# Time Complexity of search = O(m) where m is length of word
# Space Complexity of search = O(1)

# Time and Space Complexity of addWord = O(m)
```

213. House Robber II

```
# Extension to House Robber Problem:
# The only difference is that the first and the last houses are adjacent to each ot
her and therefore, if the thief has robbed the first house, they cannot steal the l
ast house and vice versa

# Either robbing i) House[1] to House[n-1] or  ii) House[2] to House[n], whichever
is max
```

215. Kth Largest Element in an Array

https://medium.com/basecs/learning-to-love-heaps-cef2b273a238 (https://medium.com/basecs/learning-to-love-heaps-cef2b273a238) https://www.youtube.com/watch?v=4hkJBcW5Ruk (https://www.youtube.com/watch?v=4hkJBcW5Ruk) https://www.geeksforgeeks.org/complexity-analysis-of-various-operations-of-binary-min-heap/ (https://www.geeksforgeeks.org/complexity-analysis-of-various-operations-of-binary-min-heap/)

```
# one approach is to use max heap: heapify all elements in array [O (n)] and pop k-
1 times max element [O((k-1) log n)] then heap[0] is kth largest
# better approach: maintain a min heap of size k [O (N log k)] i.e. n-k min element
s are popped then heap[0] is the kth largest

import heapq
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        heap = []
        for num in nums:
            heapq.heappush(heap, num)
            if len(heap) > k:
                heapq.heappop(heap)
        return heap[0] # Min heap of size k, heap[0] = kth largest element

# Time Complexity = O(N log k), adding one element to heap is log(size of heap i.e.
k), for loop is N times (heap is always limited to k, even when popping)
# Space Complexity = O(k) only k elements in heap
# Note: Buiding a heap is O(N) and not O(N log N) bcz the approach is different tha
n calling [heapify operation]
# heapq.heappush() N times
```

219. Contains Duplicate II

```
# Find 2 numbers in the array that are equal and are at most k apart from each othe
r

class Solution:
    def containsNearbyDuplicate(self, nums: List[int], k: int) -> bool:
        index_dict = {}


        for i, n in enumerate(nums):
            if n in index_dict and i - index_dict[n] <= k:
                return True

            index_dict[n] = i

        return False
```

221. Maximal Square

```
# Top down DP without any base case
# https://www.youtube.com/watch?v=6X7Ha2PrDmM&t=19s

# Consider every cell as the top leftmost cell of a square

# if the cell is 0, no square can be formed

# if the cell is 1:
# if right, down and diagonal elements are all 1, then a square of length 2 (area
4) can be formed
# if one of right, down and diagonal elements is 0 then a square of length 1 can on
ly be formed
# in equation format: 1 + min(down, right, diag)

class Solution:
    def maximalSquare(self, matrix: List[List[str]]) -> int:
        ROWS, COLS = len(matrix), len(matrix[0])
        cache = {}  # map each (r, c) -> maxLength of square

        def helper(r, c):
            if r >= ROWS or c >= COLS:
                return 0

            if (r, c) not in cache:
                down = helper(r + 1, c)
                right = helper(r, c + 1)
                diag = helper(r + 1, c + 1)

                cache[(r, c)] = 0
                if matrix[r][c] == "1":
                    cache[(r, c)] = 1 + min(down, right, diag)
            return cache[(r, c)]

        helper(0, 0)
        return max(cache.values()) ** 2

# Time and Space Complexity = O(mn)
```

226. Invert Binary Tree

```
class Solution:
    def invertTree(self, root: TreeNode) -> TreeNode:
        if root is None:
            return None

        queue = []
        queue.append(root)

        temp = 0
        while queue:
            cur = queue.pop(0)

            temp = cur.left
            cur.left = cur.right
            cur.right = temp

            if cur.left:
                queue.append(cur.left)

            if cur.right:
                queue.append(cur.right)

        return root

# Time Complexity = O(n)
# Space Complexity = O(n)
```

227. Basic Calculator II

```
# string always represents a valid expression
# All the integers in the expression are non-negative integers => 1st number can be
considered to have the sign + (0 or any +ve number)

# Stack . E.g. 2-3*4 = 2- 12 = -10
# * and / will take precedence over + and - (BODMAS rule)

# Append the input string with "+" since calculation is triggered by operator
# And according to question, 1st num can be considered to have sign + (All the inte
gers in the expression are non-negative integers)
# Essentially making the input expression: 2-3*4 => +2-3*4+

# High level solution: Collect current num and previous operator to perform operati
ons
# Prepend and append the input with "+" operator (prepend -> prev_operator initiali
zation and append -> iterate on input string with "+" concatenated) since operation
s are triggered by current operator based on prev operator

# Collect the current_num by checking if the current char is a digit (until it is n
ot)
# Collect operator "before the number" (previous operator) which will determine wha
t needs to be done: (by default first operator is "+")
# if + : push current_num on stack
# if - : push -current_num on stack
# if * or / : pop top value from stack , evaluate current expression and push it ba
ck to stack
# At the end, sum all values in stack

class Solution:
    def calculate(self, s: str) -> int:

        curr_num = 0
        stack = []
        prev_op = '+' # all nums are non-negative meaning 1st number can be conside
red to have + sign

        for char in s+"+": # append the input string with "+" since calculation is
triggered with operators
            if char == ' ':
                continue

            if char.isdigit():
                curr_num = (curr_num * 10) + int(char)  # for multiple digit number

            else:
                if prev_op == '+':
                    stack.append(curr_num) # first num is always +ve int
                elif prev_op == '-':
                    stack.append(-curr_num)
                elif prev_op == '*':
                    stack.append(stack.pop() * curr_num)
```

```
            elif prev_op == '/':
                stack.append(int(stack.pop() / curr_num))  # floor division

            # store operator for operation on next number and reset curr_num
            prev_op = char
            curr_num = 0


    return sum(stack)

# Time Complexity = O(n)
# Space Complexity = O(n)
```

230. Kth Smallest Element in a BST

```
class Solution:
    def kthSmallest(self, root: TreeNode, k: int) -> int:
        if root is None: # edge case
            return []

        stack = []
        cur = root

        result = []
        while stack or cur: # Imp: or, at the beginning stack is empty
            if cur:
                stack.append(cur)
                cur = cur.left
            else:
                cur = stack.pop()
                k = k - 1
                if k == 0:
                    return cur.val
                cur = cur.right
```

235. Lowest Common Ancestor of a Binary Search Tree

See solution

```
# Exploit the property of BST
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode')
-> 'TreeNode':
        node = root

        while node:

            node_val = node.val

            if p.val > node_val and q.val > node_val:
                node = node.right
            elif p.val < node_val and q.val < node_val:
                node = node.left
            else:
                return node

# Time Complexity = O(n)
# Space Complexity = O(1)
```

236. Lowest Common Ancestor of a Binary Tree

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode')
-> 'TreeNode':
        stack = []
        stack.append(root)

        # Dictionary for parent pointers
        parent = {root: None}

        # Iterate until we find both the nodes p and q
        while p not in parent or q not in parent:
            node = stack.pop()

            if node.left:
                parent[node.left] = node
                stack.append(node.left)

            if node.right:
                parent[node.right] = node
                stack.append(node.right)

        # Ancestors set() for node p
        ancestors = set()
        # Process all ancestors for node p using parent pointers
        while p:
            ancestors.add(p)
            p = parent[p]

        # The first ancestor of q which appears in p's ancestor set() is their lowe
st common ancestor
        while q not in ancestors:
            q = parent[q]

        return q

# Time Complexity = O(n)
# Space Complexity = O(n) for stack + O(n) for parent pointers dict + O(n) for ance
stor's set of p
```

237. Delete Node in a Linked List

```
# Replace current node with next node (both val and next)

class Solution:
    def deleteNode(self, node):
        """
        :type node: ListNode
        :rtype: void Do not return anything, modify node in-place instead.
        """
        node.val = node.next.val
        node.next = node.next.next
```

238. Product of Array Except Self

https://leetcode.com/problems/product-of-array-except-self/solution/ (https://leetcode.com/problems/product-of-array-except-self/solution/)

```
class Solution:
    def productExceptSelf(self, nums: List[int]) -> List[int]:
        L = [0] * len(nums)
        L[0] = 1
        for i in range(1,len(nums)):
            L[i] = L[i-1] * nums[i-1]

        R = [0] * len(nums)
        R[len(nums)-1] = 1
        for i in reversed(range(len(nums)-1)):
            R[i] = R[i+1] * nums[i+1]

        result = [0] * len(nums)
        for i in range(len(nums)):
            result[i] = L[i] * R[i]

        return result

# Time Complexity = O(n)
# Space Complexity = O(n)
```

```
# Method 2: Optimized space complexity as O(1)
# using only one answer array, instead of two intermediate array and an answer arra
y

# Calculate L[i] in answer array
# Iterating from rightmost position, initialize R = 1 (rightmost value) and multipl
y with answer[i] (original left array) to update final answer array
# keep updating R as current R * nums[i]

class Solution:
    def productExceptSelf(self, nums: List[int]) -> List[int]:

        n = len(nums)
        answer = [1] * n


        for i in range(1, n):
            answer[i] = answer[i-1] * nums[i-1]


        R = 1
        for i in range(n-1, -1, -1):
            answer[i] = answer[i] * R
            R = R * nums[i]

        return answer

# Space Complexity = O(1), result array is not counted in space complexity as menti
oned in problem description
# Time Complexity = O(n)
```

240. Search a 2D Matrix II

See Approach 4: Search Space Reduction

```
# Start from bottom-left
# if target > current element then increment col index
# if target < current element then decrement row index
# search till row and column remain within bounds else not found

class Solution:
    def searchMatrix(self, matrix, target):
        """
        :type matrix: List[List[int]]
        :type target: int
        :rtype: bool
        """
        row = len(matrix) - 1
        col = 0

        while row >= 0 and col < len(matrix[0]):
            if matrix[row][col] == target:
                return True

            if target > matrix[row][col]:
                col = col + 1

            elif target < matrix[row][col]:
                row = row - 1

        return False

# Time Complexity = O(m+n), note that row or col is decremented/incremented at ever
y iteration.
# Now row cannot be decremented more than number of rows and column cannot be incre
mented more
# than number of columns hence loop cannot run for more than m+n times

# Space Complexity = O(1)
```

242. Valid Anagram

```python
from collections import defaultdict

class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        d = defaultdict(int)

        for c in s:
            d[c] = d[c] + 1

        for c in t:
            d[c] = d[c] - 1

        for key in d.keys():
            if d[key] != 0:
                return False

        return True

# Time Complexity = O(n)
# Space Complexity = O(1)
```

249. Group Shifted Strings

https://leetcode.com/problems/group-shifted-strings/discuss/282285/Python-Solution-with-Explanation-(44ms-84) (https://leetcode.com/problems/group-shifted-strings/discuss/282285/Python-Solution-with-Explanation-(44ms-84))

```
# map each string in strings to a key in a hashmap
# this key is ord(i+1) - ord(i)
# hence the hash table with key: tuple and value = list of string
# for case such as az and ba to be clubbed together: z-a = 25 and a-b = -1, add +26
and take mod of 26
# (26+25) % 26 => 25 and (26+1-2) %26 => 25

from collections import defaultdict
class Solution:
    def groupStrings(self, strings: List[str]) -> List[List[str]]:
        d = defaultdict(list)
        for s in strings:
            key = ()
            for i in range(len(s)-1): # until second last since i+1 below
                circular_diff = 26 + ord(s[i+1]) - ord(s[i])
                key = key + (circular_diff % 26,) # concat tuple to tuple

            d[key].append(s)

        return d.values()

# Time complexity would be O(ab) where a is the total number of strings and b is th
e length of the longest string in strings.
# Space complexity would be O(ab), as the most space we would use is the space requ
ired for strings and the keys of our hashmap.
```

252. Meeting Rooms

```
# Check whether overlap happens or not

# sort by start time
# Compare two consecutive elements: If end time of first element is less than begin
time of next element

class Solution:
    def canAttendMeetings(self, intervals: List[List[int]]) -> bool:
        intervals.sort(key= lambda x: x[0])

        i = 0
        while i < len(intervals) - 1:
            if intervals[i][1] > intervals[i+1][0]:
                return False
            else:
                i = i + 1
        return True
```

253. Meeting Rooms II

See Solution 2: Chronological ordering

```
# Min number of non-overlapping intervals

# Build 2 lists: one for start time and another one for end time
# sort both lists and initialize 2 ptrs at the start of these 2 lists
# if start time pointer by start ptr >= end time pointed by end ptr => room has bec
ome free

# When we encounter an ending event, that means
i) that some meeting that started earlier has ended now and
ii) a previously occupied room has now become free.


class Solution:
    def minMeetingRooms(self, intervals: List[List[int]]) -> int:
        start = [interval[0] for interval in intervals]
        end = [interval[1] for interval in intervals]

        start.sort()
        end.sort()

        start_ptr = 0
        end_ptr = 0

        used_rooms = len(intervals) # max no. of rooms = total no of meetings
        while start_ptr < len(intervals):
            if start[start_ptr] >= end[end_ptr]:  # if start time of any meeting >=
end time of any meeting -> room free
                used_rooms = used_rooms - 1
                end_ptr = end_ptr + 1
                start_ptr = start_ptr + 1
            else:
                start_ptr = start_ptr + 1
        return used_rooms

# Time Complexity = O(n log n) for sorting
# Space Complexity = O(n), for start array O(n) and end array O(n)
```

257. Binary Tree Paths

See solution: Approach 2 Iterations

class Solution: def binaryTreePaths(self, root: TreeNode) -> List[str]: if root is None: # edge case return []

```
    stack = []
    stack.append((root,str(root.val)))

    paths = []
    while stack:
        node, path = stack.pop()
        if node.left is None and node.right is None:
            paths.append(path)

        if node.left:
            stack.append((node.left, path + '->' + str(node.left.val)))

        if node.right:
            stack.append((node.right, path + '->' + str(node.right.val)))

    return paths
```

259. 3Sum Smaller

```
# Let us look at an example ((2 sum smaller) nums = [1,2,3,4,8] and target=7
# if left points to 1 and right points to 8 then left+right > target ie. 1+8 > 7 =>
decrement right by 1
# if left points to 1 and right points to 4 then left+right < target ie. 1+4 < 7 (c
ondition satisfied)
# No. of points possible which are less than target => [1,2], [1,3], [1,4] ie. 3 =>
right_index - left_index
# Now increment left by 1,
# Now still left+right < 7 i.e. 2+4 < 7 => [2,3], [2,4] => right_index -

class Solution:
    def threeSumSmaller(self, nums: List[int], target: int) -> int:

        nums.sort()
        result = 0

        for i in range(len(nums)-2):
            left = i+1
            right = len(nums) - 1
            while left < right:
                total = nums[left] + nums[right] + nums[i]

                if total >= target:
                    right = right - 1

                else:
                    result = result + (right-left) # imp: all the points between le
ft and right index are less than target
                    left = left + 1

        return result

# Time Complexity = O(n^2)
# Space Complexity = O(n) due to sort
```

261. Graph Valid Tree

```
# No duplicate edges + Undirected graph

# For a graph to be tree - 1) No cycles 2) Fully connected
# For an undirected graph with no duplicate edges, if it has n-1 edges => no cycle
# Checking whether or not a graph is fully connected is straightforward - we simply
check if all
# nodes are reachable from a search starting at a single node => BFS or DFS

# counter to track count of nodes = visited list can be used

class Solution:
    def validTree(self, n: int, edges: List[List[int]]) -> bool:

        if len(edges) != n-1: # check cycle
            return False

        visited = [0 for _ in range(n)]
        g = {x :[] for x in range(n)}
        for x,y in edges:
            g[x].append(y)
            g[y].append(x)


        def dfs(node):
            if visited[node] == 1:
                return

            visited[node] = 1
            for neighbor in g[node]:
                dfs(neighbor)

        dfs(0)

        # count the number of connected nodes using visited list
        # counter used within dfs function will not be available outside dfs functi
on as dfs has no return value + global variable issue for recursive fns => so count
visited nodes here
        counter = 0

        for i in range(n):
            if visited[i] == 1:
                counter = counter + 1

        if counter != n: # check fully connected
            return False
        return True

# Time Complexity = O(V+E)
# Space Complexity = O(V+E)
```

266. Palindrome Permutation

```
# All chars should have even freq
# Only one char is allowed to have odd freq

from collections import defaultdict
class Solution:
    def canPermutePalindrome(self, s: str) -> bool:
        d = defaultdict(int)
        for c in s:
            d[c] = d[c] + 1

        count = 0
        for c in d.keys():
            if d[c] % 2 == 0:
                continue
            if d[c] % 2 == 1:
                count = count + 1

        return count <=1

# Time Complexity = O(n)
# Space Complexity = O(1) since keys can be atmost 26 (constant)
```

268. Missing Number

```
class Solution:
    def missingNumber(self, nums: List[int]) -> int:
        total = 0
        maximum = 0


        for num in nums:
            total = total + num

        cumulative_sum = len(nums) * (len(nums)+1) //2

        missing_num = cumulative_sum - total
        return missing_num
```

269. Alien Dictionary

https://leetcode.com/problems/alien-dictionary/discuss/156130/Python-Solution-with-Detailed-Explanation-(91) (https://leetcode.com/problems/alien-dictionary/discuss/156130/Python-Solution-with-Detailed-Explanation-(91))

270. Closest Binary Search Tree Value

```
# Binary search tree : left subtree <= node <= right subtree
# go left if target is smaller than current root value, and go right otherwise
# keep track of closest value at each step

class Solution:
    def closestValue(self, root: TreeNode, target: float) -> int:

        closest = root.val

        while root:
            closest = min(root.val, closest, key = lambda x: abs(target -x ))

            if target < root.val:
                root = root.left
            else:
                root = root.right
        return closest

# Time Complexity = O(height of tree)
# Space Complexity = O(1)
```

271. Encode and Decode Strings

https://www.youtube.com/watch?v=B1k_sxOSgv8&t=672s (https://www.youtube.com/watch?v=B1k_sxOSgv8&t=672s)

```
# The whole problem is about adding a delimiter between strings in the list of stri
ngs to demarcate
# The issue - The delimiter chosen could be a char in the string itself
# Solution 1: choose the delimiter outside ASCII chars (since question mentions cha
r can only be ASCII)
# Solution 2: instead of just joining all the strings together with a delimiter, we
would precede each string with its length, followed by a delimiter e.g. #, and then
the string itself.
# This way, even if our string contains the delimiter, we can correctly identify th
e string boundaries


class Codec:
    def encode(self, strs: List[str]) -> str:
        """Encodes a list of strings to a single string.
        """
        res = ""
        for s in strs:
            res += str(len(s)) + "#" + s

        return res

    def decode(self, s: str) -> List[str]:
        """Decodes a single string to a list of strings.
        """
        res = []
        i = 0

        while i < len(s):
            j = i
            while s[j] != "#": # length could be multi char e.g. 214
                j = j + 1
            length = int(s[i:j]) # first chars upto # would be numbers
            res.append(s[j + 1: j + 1 + length])
            i = j + 1 + length

        return res

# Time Complexity = O(size of list of strings)
# Space Complexity = O(number of strings in the list of strings) - each string intr
oduces a delimiter and length, result list of strings is not counted in space compl
exity
```

274. H-Index

```
# After sorting in reverse -> 1 paper above n1 citation, 2 papers above n2 citation
until n papers above # or equal to n citations

class Solution:
    def hIndex(self, citations: List[int]) -> int:

        citations.sort(reverse=True) # Sort the array in non-increasing order
        h = 0

        # Iterate through the sorted array and compare each citation count to the n
umber of papers that have at least that many citations
        for i in range(len(citations)):
            if citations[i] >= i+1: # If the citation count is greater than or equa
l to the number of papers with at least that many citations, we have found the h-in
dex
                h = i+1

        return h

# Time Complexity = O(n)
# Space Complexity = O(1) in place sorting
```

278. First Bad Version

```
lass Solution:
    def firstBadVersion(self, n):
        """
        :type n: int
        :rtype: int
        """
        low = 1
        high = n

        res = n+1 # initialize with some max value  since we need to find min

        while low <= high:
            pivot = low + (high-low) // 2

            if isBadVersion(pivot) == True:
                res = min(res, pivot) # keep track of min
                high = pivot - 1

            else: # i.e. isBadVersion(pivot) == False
                low = pivot + 1

        return res

# Time Complexity = O(log n)
# Space Complexity = O(1)
```

---

279. Perfect Squares

```
# Exact question as coin change

class Solution:
    def numSquares(self, n: int) -> int:
        # list of square numbers that are less than `n`
        square_nums = [i * i for i in range(1, int(n**0.5)+1)] # +1 since range exc
ludes last element

        queue = [(n,1)]
        visited = [False] * (n)
        while queue:

            remainder, level = queue.pop(0)
            for sqr_num in square_nums:
                if remainder - sqr_num is 0:
                    return level
                elif remainder - sqr_num > 0 and not visited[remainder - sqr_num]:
                    queue.append((remainder - sqr_num, level + 1))
                    visited[remainder - sqr_num] = True
        return -1
```

283. Move Zeroes

https://www.youtube.com/watch?v=XWaVlWRSDx8 (https://www.youtube.com/watch?v=XWaVlWRSDx8)

```
# maintain relative order of elements - tricky part

class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        for i in reversed(range(len(nums))): # imp: reverse traversal
            if nums[i] == 0:
                k = i
                while k < len(nums)-1:
                    nums[k], nums[k+1] = nums[k+1], nums[k]
                    k = k+1
```

286. Walls and Gates

https://www.youtube.com/watch?v=e69C6xhiSQE&t=1s (https://www.youtube.com/watch?v=e69C6xhiSQE&t=1s)

```
# Brute Force: BFS from each empty room until any gate is reached (nearest gate) -
Time Limit Exceeded

# Multi -source BFS from all gates (Multi-source BFS from all empty rooms won't wor
k since any cell once visited will not be visited again)
# Update distance from gate to cells in-place i.e. using rooms[r][c] matrix when po
pping from queue

class Solution:

    def walls_and_gates(self, rooms: List[List[int]]):
        ROWS, COLS = len(rooms), len(rooms[0])
        visit = set()
        q = deque()

        def addRooms(r, c):
            if (min(r, c) < 0
                or r == ROWS
                or c == COLS
                or (r, c) in visit
                or rooms[r][c] == -1
            ):
                return

            visit.add((r, c))
            q.append([r, c])

        # Add all gates to queue
        for r in range(ROWS):
            for c in range(COLS):
                if rooms[r][c] == 0:
                    q.append([r, c])
                    visit.add((r, c))

        # Update distance from gate to cells in-place i.e. using rooms[r][c] matrix
when popping from queue
        dist = 0
        while q:
            for i in range(len(q)):
                r, c = q.popleft()
                rooms[r][c] = dist
                addRooms(r + 1, c)
                addRooms(r - 1, c)
                addRooms(r, c + 1)
                addRooms(r, c - 1)
            dist += 1

# Time Complexity = O(mn), each room is visited at most once since multi -source BF
S
# Space Complexity = O(mn), max size of queue, we can insert at most mn points in q
ueue
```

287. Find the Duplicate Number

https://www.youtube.com/watch?v=wjYnzkAhcNk&t=1s (https://www.youtube.com/watch?v=wjYnzkAhcNk&t=1s)

```
# Cyclic Sort cannot be applied as input array cannot be modified
# Repeated number i.e. duplicate number can exist 2 or more times (not just 2), len
gth of array = n+1

# Multiple approaches but they won't satisfy the constraint i.e. O(n) time complexi
ty and O(1) space complexity
# Sort and compare all consecutive numbers: O(n log n) time and O(1) space complexi
ty
# Hashmap approach: O(n) time and O(n) space complexity

# Follow up: How to prove that at least one duplicate number exists? (given the arr
ay index is from 0 to n-1 and values are 1 to n)
# if one traverses following the value at index 0 i.e. nums[x], nums[nums[x]], nums
[nums[nums[x]]].. , and gets a value that is repeated (cycle),  it implies that arr
ay contains duplicates and the duplicate element is the cycle entrance.

# Optimal Solution: Find the start of cycle (Floyd algorithm)
# Step 1: find the intersection of fast and slow pointers (fast ptr moves 2 steps a
nd slow ptr moves only 1 step), intersection point need not be start of cycle (inte
rsection will only happen if duplicate exists)
# Step 2: Initialize a new slow pointer to the beginning of list ie. nums[0] and ta
ke one step only, find where this new slow pointer intersects with old slow pointer
(which is now at intersection point of prev slow pointer and fast pointer) which al
so takes only one step => start of cycle i.e. the duplicate element (to see why thi
s is the case, check above video)

# Since each number is in range[1,n] inclusive, they can be mapped to indices 1 thr
u n inclusive, meaning for every number there will be a valid index.
# So if we start with the number at 0th index, it will have another number at this
index but no number will point at 0th index since numbers are in range [1,n] i.e. 0
th index number cannot be the beginning of cycle => that's why slow and fst are ini
tialized to 0

class Solution:
    def findDuplicate(self, nums: List[int]) -> int:
            # find the intersection of fast and slow pointers (only possible if dup
licate exists)
        slow, fast = 0, 0
        while True:
            slow = nums[slow]
            fast = nums[nums[fast]]
            if slow == fast:
                break

        # find start of cycle (i.e. duplicate element)
        slow2 = 0
        while True:
            slow = nums[slow]
            slow2 = nums[slow2]
            if slow == slow2:
                return slow
```

```
# Time Complexity = O(n)
# Space Complexity = O(1)
```

289. Game of Life

```
# update to a cell can impact the other neighboring cells, but need to do simulaten
ously
# Approach 1: Use a copy of board and use it to update original board, space comple
xity is not O(1)

# Approach 2 (constant space complexity): Update the board in such as way that it r
eflects previous and new state together i.e. 1) live to dead: -1 2) dead to live: 2
# above representation of live to dead = -1 comes handy when counting live neighbor
s, abs value of cell == 1
# Finally, traverse the board again and change back the value to 1 and 0

# (Follow-up) if the board is infinite: since we need only above row and below row
to apply rules, keep only 3 rows in memory at a time
```

290. Word Pattern

```
# Same as 205. Isomorphic Strings with additional condition length of s and pattern
should also be equal
# Additional condition bcz of following test case:
# pattern = "aba" and s = "cat cat cat dog", expected false, without additional con
dition it is true

class Solution:
    def wordPattern(self, pattern: str, s: str) -> bool:
        s_list = s.split()

        return (len(set(pattern)) == len(set(s_list)) == len(set(zip(s_list, patter
n)))) and len(s_list) == len(pattern)
```

295. Find Median from Data Stream

```
# maintain two heaps: i) max-heap to store the smaller half of the input numbers i
i) min-heap to store the larger half of the input numbers
# Both the heaps should be nearly balanced i.e. smaller half max heap can have atmo
st 1 element more than larger half min heap

import heapq
class MedianFinder:

    def __init__(self):
        """
        initialize your data structure here.
        """
        self.small = []
        self.large = []

    def addNum(self, num: int) -> None:
        # when self.small is empty
        if len(self.small) == 0:
            heapq.heappush(self.small, -num) # max heap hence push -num
            return

        # if incoming num is smaller than self.small[0] max heap
        if num <= -self.small[0]:
            heapq.heappush(self.small, -num)
        else:
            heapq.heappush(self.large, num)

        # balancing
        if len(self.small) - len(self.large) == 2:
            heapq.heappush(self.large, -heapq.heappop(self.small))
        elif len(self.small) - len(self.large) == -2:
            heapq.heappush(self.small, -heapq.heappop(self.large))


    def findMedian(self) -> float:
        if len(self.small) == len(self.large):
            return (self.large[0] + (- self.small[0]))/2.0
        if len(self.small) > len(self.large):
            return -self.small[0]
        else:
            return self.large[0]

# Time Complexity addNum() -> O(log n), only heap insertions and heap pops
# Time Complexity findMedian() -> O(1) constant time
# Space Complexity = O(n) to hold inputs
```

297. Serialize and Deserialize Binary Tree

```
# Serialize = needs to convert to String (not any other data structure) + level-ord
er traversal with 'null'
# Deserialize = convert string to list first, create root node + left subtree + rig
ht subtree ( if not 'null')

# Serialize
# level order traversal +
# when no child node (null) +
# no check whether child exists or not when appending to queue

# Deserialize
# convert string to list +
# create root from first element +
# add it to queue and do level order traversal while creating left and right childs
(+ appending them to queue)
# lookahead ptr i is used to traverse string + check whether we haven't hit the end
of string

class Codec:

    def serialize(self, root):
        if root is None: # edge case
            return ''

        queue = []
        queue.append(root)

        result = ''
        while queue:
            node = queue.pop(0)
            if not node: # if node is not present add 'null,' as string
                result = result + 'null,' # Imp: adding null
                continue
            result = result + str(node.val) + ',' # Imp: adding comma
            queue.append(node.left)
            queue.append(node.right)
        return result


    def deserialize(self, data):
        if data == '': # edge case
            return None

        l = data.split(',')
        queue = []
        root = TreeNode(l[0]) # create a root node from val
        queue.append(root)

        i = 1 # lookahead 1 in list
        while queue and i < len(l): # Imp: i < len(l) condition
            node = queue.pop(0)  # parent
```

```
            if l[i] != 'null': # left subtree
                left = TreeNode(l[i])
                node.left = left
                queue.append(left)

            if i+1 < len(l):
                i = i+1

            if l[i] != 'null': # right subtree
                right = TreeNode(l[i])
                node.right = right
                queue.append(right)

            i = i+1

        return root

# Time Complexity = O(n) every node is visited once
# Space Complexity = O(n) space required to store node val in list and building a t
ree
```

300. Longest Increasing Subsequence

Number of subsequences possible in array of len L = 2 ^L

```
# Method 1: (Better time complexity in method 2)
# Rational: LIS at any index = LIS at any prev index + 1, if num at index is larger
than num at prev index

# Initialize dp[i] = 1 corresponding to all elements in nums
# dp[i] represents the length of the longest increasing subsequence that ends with
the element at index i

# Iterate i from 1 to length of nums.
# At each iteration, use a second for loop to iterate from j = 0 to i - 1 (all the
elements before i)
# Check if that element is smaller than nums[i]. If so, set dp[i] = max(dp[i], dp
[j] + 1)

class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:

        dp = [1] * len(nums)

        for i in range(1, len(nums)):
            for j in range(i):
                if nums[i] > nums[j]:
                    dp[i] = max(dp[i], dp[j]+1)

        return max(dp)

# Time Complexity = O(n^2), 2 for loops
# Space Complexity = O(n), dp array
```

```
# Method 2
# Rational: Progressively build a result array with increasing subsequence as you i
terate thru the nums array

# Initialize an array result which contains the first element of nums i.e. nums[0]
# Iterate through nums[i] from the second element. For each element in nums[i]:

# If nums[i] is greater than last element in result array (result is in sorted orde
r), then add nums[i] to result array
# Otherwise, perform a binary search in result array to find the position where num
s[i] can be replaced in sorted order i.e. replace the first element in result array
which is greater than or equal to nums[i]
# len of result array = LIS (result array may not be a valid subsequence but the le
n of result array is max subsequence len)

# Note: One thing to add: this algorithm does not always generate a valid subsequen
ce of the input, but the length of the subsequence will always equal the length of
the longest increasing subsequence. For example, with the input [3, 4, 5, 1], at th
e end we will have sub = [1, 4, 5], which isn't a subsequence, but the length is st
ill correct


class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        result = [nums[0]]

        for i in range(1, len(nums)):
            if nums[i] > result[-1]:
                result.append(nums[i])
            else:
                insert_pos = bisect_left(result, nums[i])
                result[insert_pos] = nums[i]

        return len(result)


# Time Complexity = O(n log n), n elements in nums and log(n) for binary search
# Space Complexity = O(n), for result array
```

301. Remove Invalid Parentheses

https://leetcode.com/problems/remove-invalid-parentheses/discuss/75028/Short-Python-BFS (https://leetcode.com/problems/remove-invalid-parentheses/discuss/75028/Short-Python-BFS) (see in comments, readable version)

```
# Minimum removal - remove one and check whether valid is found, if not then do so
recursively
# Remove one bracket at every position and check if some valids are found
# if valids are not found, recursively remove one more element and check if valid i
s found
class Solution:
    def removeInvalidParentheses(self, s: str) -> List[str]:
        def isValid(s):
            balance = 0
            for c in s:
                if c == '(':
                    balance = balance + 1
                elif c == ')':
                    balance = balance - 1
                    if balance < 0:
                        return False
            return balance == 0

        level = {s} # Only one element at the beginning, set is used here in order
to avoid duplicate

        while len(level) > 0:
            valid = []
            for elem in level:
                if isValid(elem):
                    valid.append(elem)
            if valid: # 2. if any valid found
                return valid

            # initialize an empty set
            new_level = set()
            # 1. BFS -> Remove one element at every position
            # recursive hence in next iteration, it will remove one more element, i
f valid is not found
            for elem in level:
                for i in range(len(elem)):
                    new_level.add(elem[:i] + elem[i + 1:])

            level = new_level

# Time Complexity = O(n * 2^n), all subsets ( 2^n) need to be searched (n)
# Space Complexity = O(n * 2^n)
```

311. Sparse Matrix Multiplication

3 loop solution is not acceptable by companies

```
# Create a hashmap with key as tuple of row, col index and value as matrix value fr
om non-zero elements

class Solution:
    def multiply(self, mat1: List[List[int]], mat2: List[List[int]]) -> List[List[i
nt]]:

        sparse_a = self.get_nonzero_cells_dict(mat1)
        sparse_b = self.get_nonzero_cells_dict(mat2)

        matrix_result = [[0] * len(mat2[0]) for _ in range(len(mat1))]

        # can optimize outer loop to be shorter sparse matrix
                # i*k matrix multiplied by k*j matrix, needs only 3 vars: i, k and
j
        for i,k in sparse_a.keys(): # instead of two loops, one loop does the job
            for j in range(len(mat2[0])): # imp: columns in mat2, not in sparse ver
sion
                if (k,j) in sparse_b.keys():
                    matrix_result[i][j] = matrix_result[i][j] + sparse_a[(i,k)] * s
parse_b[(k,j)]

        return matrix_result

    def get_nonzero_cells_dict(self, matrix):
        d = {}
        for i in range(len(matrix)):
            for j in range(len(matrix[0])):
                if matrix[i][j] != 0:
                    d[(i,j)] = matrix[i][j]
        return d

# Time Complexity: O(mk + kn + M) where M = total number of multiplications of non-
zero pairs
# Space Complexity: O(c1 + c2) where c1, c2 = number of non-zero values in matrix 1
and matrix 2
```

316. Remove Duplicate Letters

https://leetcode.com/problems/remove-duplicate-letters/solutions/4091060/video-how-we-think-about-a-solution-with-stack-python-javascript-java-c (https://leetcode.com/problems/remove-duplicate-letters/solutions/4091060/video-how-we-think-about-a-solution-with-stack-python-javascript-java-c)

```
# order of input needs to be maintained
# Understand by example: bcabc
# 4 options: bca, bac, cab, abc. Answer: abc
# bcabc -> abc

# Iterate thru the input
# if the incoming char is already in the stack, skip it since we dont want duplicat
es e.g. bbb
# while the incoming char is i) lexicographically smaller than the char at top of s
tack and ii) the char at top of stack is also present ahead ->  pop chars from stac
k i.e.  greedily take the 1st option to form lexicographically smaller string
# add incoming char to stack

class Solution:
    def removeDuplicateLetters(self, s: str) -> str:
        # Dictionary to store the last occurrence of each character
        last_occur = {}

        # Record the last occurrence of each character
        for i, char in enumerate(s):
            last_occur[char] = i

        stack = []  # Stack to store characters in the desired order

        seen = set()

        for i in range(len(s)):
            if s[i] in seen:
                continue  # Skip if the character is already visited

            # If the top of the stack is greater than s[i] and will occur later aga
in, remove from stack
            while stack and s[i] < stack[-1] and i < last_occur.get(stack[-1], -1):
               seen.remove(stack.pop())


            stack.append(s[i])  # Add to the stack
            seen.add(s[i])

        return ''.join(stack)

# Time Complexity = O(n)
# Space Complexity = O(1), stack can only have 26 chars (no duplicates)
```

322. Coin Change

Contrast this with "Combination Sum" Problem (contrast time complexity) https://leetcode.com/problems/coin-change/discuss/77361/Fast-Python-BFS-Solution (https://leetcode.com/problems/coin-change/discuss/77361/Fast-Python-BFS-Solution) (in comments)

```
# Initialize q with target amount and no. of coins = 0
# Substract all coins from amount and check i) remainder >= 0 ii) remainder not see
n (to remove
# redundancy

# level corresponds to # of coins
# parallel path checking = BFS

class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:

        q = deque()
        q.append([amount,0]) # amount, num of coins
        visited = [False] * (amount)  # to remove redundancy and take fewest coins

        while q:
            remainder, level = q.popleft()
            if remainder ==  0:
                    return level

            # most imp code snippet
            for coin in coins:
                if remainder - coin >= 0 and not visited[remainder - coin]:
                    q.append((remainder - coin, level + 1))
                    visited[remainder - coin] = True
        return -1

# Time Complexity =  O(len(coins)*amount) since at every level, in worst case, all
coins need to be checked and this can happen "amount" number of times
# Space Complexity = O(len(coins)*amount) + O(amount) for visited -> this is upper
bound as same remainder at lower level is not put in queue
```

323. Number of Connected Components in an Undirected Graph

https://www.youtube.com/watch?v=8f1XPm4WOUc&t=895s (https://www.youtube.com/watch?v=8f1XPm4WOUc&t=895s) (Union Find Solution) https://www.youtube.com/watch?v=ayW5B2W9hfo (https://www.youtube.com/watch?v=ayW5B2W9hfo) (Union find algo in 5 mins)

```
# Can be solved via DFS and BFS but union-find has better time-complexity O(Edges *
log/alpha number of vertices) where alpha = inverse Ackermann function
# 4 parts: parent, rank, find, union [find parent then i) parent same ii) parent di
fferent - 3 conditions based on rank]

# union all nodes in the input edge list [union(x,y): union the groups containing x
and y]
# find representative of every node [find (x): find the representative of x in a tr
ee structure i.e. root node]
# number of distinct representatives = number of connected components

class UnionFind:
    def __init__(self, n):

        # Initially every node is a parent of itself, index of this array represent
s node and value represents the parent/representative
        self.parent = [i for i in range(n)]

        # If ith index is the parent/representative of a set, rank is the number of
nodes in this set
        # Initially, rank is set to 1
        self.rank = [1] * n


    # Find the parent/root/representative of a node
    # if x is not the parent of itself, then find it recursively
    def find(self, x):

        if (self.parent[x] != x):
            self.parent[x] = self.find(self.parent[x])

        return self.parent[x]

    def union(self, x, y):

        # Find current parent/representative of x and y
        xset = self.find(x)
        yset = self.find(y)

        # If representatives are same
        if xset == yset:
            return

        # if ranks are different for parents, put smaller ranked item under bigger
ranked item and add ranks
        if self.rank[xset] < self.rank[yset]:
            self.parent[xset] = yset
            self.rank[yset] = self.rank[yset] + self.rank[xset]

        elif self.rank[xset] > self.rank[yset]:
            self.parent[yset] = xset
```

```
            self.rank[xset] = self.rank[xset] + self.rank[yset]

        # If ranks are same for parents, then move y under x (doesn't matter which
one goes where) and increment rank of x's tree
        else:
            self.parent[yset] = xset
            self.rank[xset] = self.rank[xset] + self.rank[yset]

class Solution:
    def countComponents(self, n: int, edges: List[List[int]]) -> int:
        uf = UnionFind(n)

        for e1, e2 in edges:
            uf.union(e1, e2)

        # find representative of every node
        # number of distinct representatives = number of connected components
        representative = set()

        for i in range(n):
            representative.add(uf.find(i))
        return len(representative)

# Time Complexity = O(E * log/alpha V) where alpha = inverse Ackermann function, lo
g since height of tree with n nodes is log n
# Space Complexity = O(V)
# More description about time complexity = https://leetcode.com/explore/featured/ca
rd/graph/618/disjoint-set/3843/
```

```
# DFS and BFS solution

class Solution:
    def countComponents(self, n: int, edges: List[List[int]]) -> int:

        visited = [0 for _ in range(n)]

        g = {x:[] for x in range(n)}
        for x,y in edges:
            g[x].append(y)
            g[y].append(x)

        #def dfs(i,g,visited):
            #visited[i] = 1
            #for neighbor in g[i]:
                            # if not visited[neighbor]:
                       #dfs(neighbor,g, visited)

        #counter = 0
        #for i in range(n):
            #if not visited[i]:
                #counter = counter + 1
                #dfs(i,g,visited)
        #return counter

        def bfs(i,g,visited):  # visit and adding to queue happens together
            visited[i] = 1
            q =[]
            q.append(i)

            while q:
                popped = q.pop(0)

                for neighbor in g[popped]:
                    if not visited[neighbor]: # neighbors can have common neighbors
                        q.append(neighbor)
                        visited[neighbor] = 1
        counter = 0
        for i in range(n):
            if not visited[i]:
                counter = counter + 1
                bfs(i,g,visited)

        return counter
```

329. Longest Increasing Path in a Matrix

https://www.youtube.com/watch?v=uLjO2LUlLN4&t=29s (https://www.youtube.com/watch?v=uLjO2LUlLN4&t=29s)

```
# DFS from top left node until last node from where no further path possible.
# LIP is tracked using a dict with key = (r, c) of cell and value = LIP
# LIP for the last node in the DFS path is set to 1 and backtracking this DFS path
fills other nodes in the path subsequently incremented by 1
# In case LIP is already filled for a node, no need to do DFS from this node


# dfs function signature: dfs(r, c, prevVal)

class Solution:
    def longestIncreasingPath(self, matrix: List[List[int]]) -> int:
        ROWS, COLS = len(matrix), len(matrix[0])
        dp = {}  # key = (r, c) value = Longest Increasing Path from this cell

        def dfs(r, c, prevVal):
            # return 0
            if r < 0 or r == ROWS or c < 0 or c == COLS or matrix[r][c] <= prevVal:
                return 0

            # if LIP is known for the node, equivalent to visited
            if (r, c) in dp:
                return dp[(r, c)]

            # LIP from every cell is atleast 1
            res = 1

            # take the max from 4 directions
            directions = [[1,0], [-1,0], [0,1], [0,-1]]
            for dr, dc in directions:
                new_r, new_c = r + dr, c + dc
                              # below dfs will return when no further path
                                # max value from all 4 dirs
                res = max(res, 1 + dfs(new_r, new_c, matrix[r][c]))

            # below code executes while backtracking, update LIP for all nodes in t
his path
            # when last node in a DFS path is reached, starts populating LIC from t
his last node to first node
            dp[(r, c)] = res
            return res

        for r in range(ROWS):
            for c in range(COLS):
                dfs(r, c, -1) # put a value that is lower than all values in matri
x, to start the dfs
        return max(dp.values())

# Time Complexity = O(mn)
# Space Complexity = O(mn)
```

332. Reconstruct Itinerary

https://leetcode.com/problems/reconstruct-itinerary/discuss/469225/Intution-to-solve-the-question (https://leetcode.com/problems/reconstruct-itinerary/discuss/469225/Intution-to-solve-the-question)

https://www.youtube.com/watch?v=ZyB_gQ8vqGA (https://www.youtube.com/watch?v=ZyB_gQ8vqGA)

```
# all airports need to be visited
# duplicate edges possible (i.e. we might have multiple flights with the same origi
n and destination)
# visited set cannot be used as duplicate edges possible

# The question states: all tickets form at least one valid itinerary

# topological sorting cannot be used as is as cycles are possible
# Modified DFS since we can end up at "dead end" i.e. no outgoing flights but some
destinations remaining

# while neighbors exist dfs + backtrack (visit all edges - eulerian path) : force d
fs to visit all edges in a directed connected graph with cycles

from collections import defaultdict
class Solution:
    def findItinerary(self, tickets: List[List[str]]) -> List[str]:

        adj_list = defaultdict(list)
        for x,y in tickets:
            adj_list[x].append(y)

        for key,value in adj_list.items(): # smallest lexical order
            adj_list[key] = sorted(value)

        def dfs(start_node, adj_list, itinerary):
            while len(adj_list[start_node]) > 0:
                neighbor = adj_list[start_node][0]
                adj_list[start_node].remove(neighbor)
                dfs(neighbor, adj_list, itinerary)


            itinerary.insert(0,start_node)


        itinerary = []
        start_node="JFK"
        dfs(start_node, adj_list,itinerary)
        return itinerary

# Time Complexity = O(E log E): DFS takes O(E), Sorting takes O (E log E), sorting
dominates since dfs and sorting not in same loop
# Space Complexity = O(V+E)
```

339. Nested List Weight Sum

```
# nestedList = sum([x.getList() for x in nestedList if not x.isInteger()], [])
# This will concatenate the all the lists inside the current nestedList.
# sum([[1,2],[3,4]],[]) will return [1,2,3,4]
# nice trick to flatten a list of lists.

class Solution:
    def depthSum(self, nestedList: List[NestedInteger]) -> int:

        depth, result = 1, 0
        while nestedList:
            result += depth * sum([x.getInteger() for x in nestedList if x.isIntege
r()])
            # just flatten for next iteration, do not add to result
                    # if the last arg below [] is not used, throws error
            # unsupported operand type(s) for +: 'int' and 'list'
            nestedList = sum([x.getList() for x in nestedList if not x.isInteger
()], [])
            depth += 1
        return result

# Time Complexity = O(total number of elements in the input list)
# Space Complexity = O(1)
```

346. Moving Average from Data Stream

```
# maintain of deque of given size, pop from left and append to right happens in O
(1)
# keep track of queue size and queue sum whenever new element is added
# 2 conditions:i)  when queue length < max queue capacity ii) otherwise

class MovingAverage:

    def __init__(self, size: int):
        self.max = size
        self.queue = deque()

        self.sum = 0
        self.length = 0


    def next(self, val: int) -> float:
        if self.length < self.max:
            self.queue.append(val)
            self.sum = self.sum + val
            self.length = self.length + 1
        else:
            left_val = self.queue.popleft()
            self.sum = self.sum - left_val
            self.queue.append(val)
            self.sum = self.sum + val

        return self.sum/self.length

# Your MovingAverage object will be instantiated and called as such:
# obj = MovingAverage(size)
# param_1 = obj.next(val)

# Time Complexity = O(1), popleft() and append() in deque is O(1) operation
# Space Complexity = O(size of queue)
```

347. Top K Frequent Elements

https://www.youtube.com/watch?v=YPTqKIgVk-k&t=1s (https://www.youtube.com/watch?v=YPTqKIgVk-k&t=1s)

```
# Method 1: dict + heap

# build a dict of num: count then build a min heap of size k using N elements (keep
tuple of freq, num as heap element)
# Here top k of 'freq' needs to be found out hence heap

from collections import Counter
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        d = Counter(nums) # O(n)
        return [num for num,freq in d.most_common(k)] # can use heap here with (fre
q, num) tuple

# Time Complexity = O(N log k) if k < N, if k=N then O(N)
# Space Complexity = O(N + k), hashmap + heap
```

```
# Method 2: Bucket sort idea, keep count and corresponding elements with those coun
ts (better time complexity)

# Step 1: Create a dict of value: count (note: count cannot be greater than len(inp
ut array) since at most 1 element can occur len(array) times)
# Step 2: Iterate over the above dict and create a list of lists where the index of
individual list refers to the count and append all the nums with a given count
# Step 3: Iterate in reverse from this list of lists until k elements are appended
to the result


class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        count = defaultdict(int)
        freq = [[] for i in range(len(nums) + 1)] # initialize an empty list of lis
ts

        for num in nums:
            count[num] = count[num] + 1

        for num, count in count.items():
            freq[count].append(num)

        res = []
        for i in range(len(freq) - 1, 0, -1):         # count will be at least 1
            for n in freq[i]:
                res.append(n)
                if len(res) == k:
                    return res

# Time Complexity = O(n)
# Space Conmplexity = O(n)
```

349. Intersection of Two Arrays

```
# Duplicates allowed in input but output should not have duplicates even if they ar
e common in
# both arrays


class Solution:
    def intersection(self, nums1: List[int], nums2: List[int]) -> List[int]:
        s1 = set(nums1)
        s2 = set(nums2)

        result = s1.intersection(s2)
        return list(result)

# Time Complexity = O(m+n) where m and n are array's length. O(m) and O(n) time is
used to convert  # list into set

# Space Complexity = O(m+n) space required to create hashmap of arrays with length
m and n
```

350. Intersection of Two Arrays II

Hash Table Approach:

1. If nums1 is larger than nums2, swap the arrays.
2. For each element in nums1: Add it to the hash map m. Increment the count if the element is already there.
3. Initialize the insertion pointer (k) with zero.
4. Iterate along nums2: If the current number is in the hash map and count is positive: Copy the number into nums1[k], and increment k. Decrement the count in the hash map.
5. Return first k elements of nums1.

Time Complexity = O(m+n) where m and n are the lengths of the arrays Space Complexity = O(min(m,n)) - we build a hashmap, which is smaller of the two arrays

```
# Duplicates allowed in input and output should have duplicates if they are common
in both arrays
# Two approaches (diff in complexity) - 1)hash map and 2) sort+two pointer approach
# Also see approach 1 using hashmap in solution/notes

class Solution:
    def intersect(self, nums1: List[int], nums2: List[int]) -> List[int]:
        result = []

        nums1.sort()
        nums2.sort()

        j = k = 0

        while j < len(nums1) and k < len(nums2): # can be improved to iterate for t
he length of smaller array
            if nums1[j] == nums2[k]:
                result.append(nums1[j]) # can be improved to use smaller array to s
tore results
                j = j+1
                k = k+1
            elif nums1[j] < nums2[k]:
                j = j+1
            elif nums1[j] > nums2[k]:
                k = k+1
        return result

# Time Complexity = O(m log m + n log n), sort both arrays + linear scan afterwards
# Space Complexity = O(m+n), ignore the space to store output but include space for
sort
```

362. Design Hit Counter

https://leetcode.com/problems/design-hit-counter/discuss/83511/Python-solution-with-detailed-explanation (https://leetcode.com/problems/design-hit-counter/discuss/83511/Python-solution-with-detailed-explanation)

373. Find K Pairs with Smallest Sums

```
# arrays are sorted and contain duplicates

# BFS on Min heap with visited set (sum,(array 1 index, array 2 index)) i.e. tuple
of 2 elements (sum,(index1, index2)) where 2nd element is a tuple as well
# arrays are sorted and we have to find sums in ascending order i.e. index combinat
ions 0,0 -> 0,1 or 1,0 -> (prev remaining pair) or 1,1 or 1,2 or 2,1 -> ...

# keep index of one array fixed and index of other array incremented by 1 (for both
arrays)  + sum them and put it back in min heap e.g.,01,10,11,
# visited set to avoid duplication of indexes e.g. (1,1), (2,2) etc
# pop from min heap and add it to result


class Solution:
    def kSmallestPairs(self, nums1: List[int], nums2: List[int], k: int) -> List[Li
st[int]]:
        result = []
        visited = set()

        heap = [(nums1[0]+ nums2[0], (0,0))]
        visited.add((0,0))

        while k > 0 and heap:
            val, (i,j) = heappop(heap)
            result.append([ nums1[i],nums2[j] ])

            # within bound and not visited
            if i+1 < len(nums1) and (i+1,j) not in visited:
                heappush(heap, ( nums1[i+1]+ nums2[j], (i+1,j) ))
                visited.add((i+1, j))

                     # within bound and not visited
            if j+1 < len(nums2) and (i,j+1) not in visited:
                heappush(heap, ( nums1[i]+ nums2[j+1], (i,j+1) ))
                visited.add((i, j+1))

            k = k-1

        return result



# Time Complexity = O(min(k log k, mn log (mn)))
# O(min(k, mn)) to get required number of pairs. Log factor for insertions into min
heap

# Space Complexity = O(min(k, mn)), for min heap and visited set
```

378. Kth Smallest Element in a Sorted Matrix

```
# https://leetcode.com/problems/kth-smallest-element-in-a-sorted-matrix/descriptio
n/comments/1565034

# Initialize min heap with 0th element and row,col = 0,0
# Add r+1, c and r,c+1 elements to min heap (slant/diagonal traversal)
# keep track of visited cells to avoid duplication
# Pop min elements until k > 0



class Solution:
    def kthSmallest(self, matrix: List[List[int]], k: int) -> int:

        visited = set()

        heap = [( matrix[0][0], (0,0))]
        visited.add((0,0))

        while k > 0 and heap:
            val, (i,j) = heappop(heap)


            # within bound and not visited
            if i+1 < len(matrix) and (i+1,j) not in visited:
                heappush(heap, ( matrix[i+1][j], (i+1,j) ))
                visited.add((i+1, j))

            # within bound and not visited
            if j+1 < len(matrix[0]) and (i,j+1) not in visited:
                heappush(heap, ( matrix[i][j+1], (i,j+1) ))
                visited.add((i, j+1))

            k = k-1

        return val

# Time Complexity = O(k log k)
# Space Complexity = O(k) for min heap and visited set
```

380. Insert Delete GetRandom O(1)

See Solution: Hash table - insert and delete is O(1) however to get random element requires to choose a random index and then retrieve an element with that index. There are no indexes in hashmap and hence to get true random value, one has first to convert hashmap keys in a list, that would take linear time.

The solution here is to build a list of keys aside and to use this list to compute GetRandom in constant time.

List has indexes and could provide Insert and GetRandom in average constant time, though has problems with Delete. To delete a value at arbitrary index takes linear time. The solution here is to always delete the last value: 1)Swap the element to delete with the last one. 2)Pop the last element out. For that, one has to compute an index of each element in constant time, and hence needs a hashmap which stores element -> its index dictionary.

```
# list + hashmap
# list stores val and it's index is used to populate value in hashmap [for populati
ng index in dict, utilize len(list)]
# hashmap stores val -> index

# list deletion: find the index of value to be deleted from hashmap and swap with l
ast value, then pop the last value


import random
class RandomizedSet:

    def __init__(self):
        self.list = []
        self.dict = {}


    def insert(self, val: int) -> bool:
        if self.dict.get(val) == None:
            self.list.append(val)
            self.dict[val] = len(self.list) -1
            return True
        else:
            return False

    def remove(self, val: int) -> bool:
        if self.dict.get(val) != None:
            swap_index = self.dict[val]
            # swap in both list and dict
            self.list[-1], self.list[swap_index] = self.list[swap_index], self.list
[-1]
            self.dict[self.list[-1]], self.dict[self.list[swap_index]] = self.dict
[self.list[swap_index]], self.dict[self.list[-1]]

            # delete in both list and dict
            self.list.pop()
            del self.dict[val]
            return True
        return False


    def getRandom(self) -> int:

        r = random.randint(0, len(self.list) - 1) # both ends are included
        return self.list[r]
```

384. Shuffle an Array

import python

random.random() -> returns a number b/w [0.0,1.0) random.randint(a,b) -> returns an int between [a,b] random.choice(seq) -> returns an element from seq, throws error if seq is empty

Random = Every element in list should be **equally likely** to be picked up

This question is basically asking to generate random numbers equal to the size of input list and these random numbers should not repeat (sampling without replacement)

Brute Force Approach:

1. Pick a random index, output the number corresponding to it, store in output array
2. Remove that index from input array and repeat step 1 This will require an extra array

Another efficient approach without requiring extra memory

1. Pick a random index and swap the number corresponding to the picked index with the first element of array
2. Now pick a random index from second to last index and swap the number corresponding to the picked index with the second second element of array. This way every index is equally likely to be picked up and the index that is picked in one iteration will not be a candidate in next iteration

Variation of this problem:

1) input has duplicates and output can contain duplicates - same as above 2) input has duplicates and output should not contain duplicates - if the random element chosen is already part of output, throw it away and repeat (rejection sampling)

Complexity Analysis: Time complexity : O(n) The Fisher-Yates algorithm runs in linear time, as generating a random index and swapping two values can be done in constant time.

Space complexity : O(n) Although we managed to avoid using linear space on the auxiliary array from the brute force approach, we still need it for reset, so we're stuck with linear space complexity.

https://leetcode.com/articles/shuffle-an-array/ (https://leetcode.com/articles/shuffle-an-array/)
http://www.radwin.org/michael/2015/01/13/unique-random-numbers-technical-interview-question/
(http://www.radwin.org/michael/2015/01/13/unique-random-numbers-technical-interview-question/)

```
import random

class Solution:

    def __init__(self, nums: List[int]):
        self.nums = nums
        self.original = nums[:] # deep copy


    def reset(self) -> List[int]:
        """
        Resets the array to its original configuration and return it.
        """
        self.nums = self.original # both will change together
        self.original = self.original[:] # deep copy
        return self.nums

    def shuffle(self) -> List[int]:
        """
        Returns a random shuffling of the array.
        """
        # sampling without replacement
        for i in range(len(self.nums)):
            random_index = random.randint(i,len(self.nums)-1)
            self.nums[i], self.nums[random_index] = self.nums[random_index], self.n
ums[i]
        return self.nums


# Time complexity : O(n)
# The time complexity of swapping two variables in Python is O(1)

# Space complexity : O(n): space for deep copy
```

392. Is Subsequence

```
# 2 ptr approach

class Solution:
    def isSubsequence(self, s: str, t: str) -> bool:
        if len(s) == 0:
             return True

        i,j = 0, 0

        while i < len(t) and j < len(s):
            if t[i] == s[j]:
                j = j+1
                i = i + 1
            else:
                i = i+1

        if j == len(s):
            return True
        else:
            return False
```

394. Decode String

https://leetcode.com/problems/decode-string/discuss/508115/Simple-python-with-stack-easy-to-understand (https://leetcode.com/problems/decode-string/discuss/508115/Simple-python-with-stack-easy-to-understand)

```
# Push in stack unless ending brackets
# Extract the string between closing and ending brackets, and digit (can be multipl
e) before
# opening brackets
# Push the digit times string back into stack

class Solution:
    def decodeString(self, s: str) -> str:

        stack = []

        for element in s: # Loop through string
            if element == ']':

                # temp holds the popped element
                temp = stack.pop()
                sub_result = ""
                while (temp != '[' ): # Extract between ] and [
                    sub_result = temp + sub_result
                    temp = stack.pop()

                # temp points to the top of stack and checks whether it is a digit
                temp_digit = stack[-1]
                digit = ""
                while(temp_digit.isdigit()): # If multiple digits
                    digit = stack.pop() + digit
                    if stack:  # check whether stack is empty
                        temp_digit = stack[-1]
                    else:
                        break

                sub_result = sub_result * int(digit)

                stack.append(sub_result) # Push back into stack (most important tri
ck)


            elif element != ']':
                stack.append(element)

        return "".join(stack)

# Time Complexity = O(n)
# Space Complexity = O(n)
```

398. Random Pick Index

https://gregable.com/2007/10/reservoir-sampling.html

Reservoir Sampling is an algorithm for sampling elements from a stream of data.

Your goal is to efficiently return a random sample of 1,000 elements evenly distributed from the original stream. How would you do it?

The right answer is generating random integers between 0 and N - 1, then retrieving the elements at those indices and you have your answer. If you need to be generate unique elements, then just throw away indices you've already generated (rejection sampling)

So, let me make the problem harder. You don't know N (the size of the stream) in advance and you can't index directly into it.

First, you want to make a reservoir (array) of 1,000 elements and fill it with the first 1,000 elements in your stream. That way if you have exactly 1,000 elements, the algorithm works. This is the base case.

Now we have to make probability of 1001th element being part of 1000 elements = probability that any element within 1000 elements remains in the set

prob that 1001th element becomes part of 1000 elements = 1000/1001 (i)

The probability of removing any one element is the probability of element 1,001th getting selected multiplied by the probability of that element getting randomly chosen as the replacement candidate from the 1,000 elements in the reservoir. That probability is:
1000/1001 * 1/1000 = 1/1001

the probability that the any one element survives this round is:
1 - 1/1001 = 1000/1001 (ii)

Since (i) = (ii)

```
# Brute force: store all numbers and their index which are equal to target, and the
n pick one randomly
# Brute force approach will need to scan for all elements in input and store all el
ements which are equal  to target beforehand
# -> we don't know size of array + too much extra space (input array is large with
duplicates)

# Reservoir Sampling (array size is too large)


# Algorithm:
# To randomly pick up k elements in an array S with very big size N with the same p
robability
# (1) Get the first k elements from S and put them into an array result[]
# (2) for j > k && j < N: (between k and N)
# i) generate a random number r between 0 and j
# ii) if this random number r is less than k: picked index is out, if the picked in
dex is one of the first k indexes, we replace the element at picked index with the
element at index j

# Proof: Prob that every item in stream has equal probability of k/N to be in reser
voir


# let's take: j = k+1
# total possibilities = k, if r < k -> replace (analogous to 1000/1001)




import random
class Solution:

    def __init__(self, nums: List[int]):
        self.nums=nums


    def pick(self, target: int) -> int:
        count = 0
        result = -1 # initialize with a value that result cannot take

        for i in range(len(self.nums)):
            if self.nums[i] != target:
                continue
            count = count + 1
            rand = random.randint(1,count)   # both bounds included
            if rand <= 1:  # inside for loop, the last index that stays is the resu
lt
                result = i

        return result
```

```
# Time Complexity = O(n)
# Space Complexity = O(1)
```

```
To those who don't understand why it works. Consider the example in the OJ
{1,2,3,3,3} with target 3, you want to select 2,3,4 with a probability of 1/3 each.

2 : It's probability of selection is 1 * (1/2) * (2/3) = 1/3
3 : It's probability of selection is (1/2) * (2/3) = 1/3
4 : It's probability of selection is just 1/3

So they are each randomly selected.

Thanks. A bit more explanation. Let's take index 2 for example.

The first time we saw target 3 is at index 2. The count is 0. Our reservoir only ha
ve 0 and we need to pick rnd.nextInt(++count) == 0. The probability is 1. Result =
2.
Then we went to index 3. The count is 1. Our reservoir is has [0,1]. We say if we g
et 0, we'll change the the result, otherwise we keep it. Then chance that we keep t
he result=2 is 1/2 which means we got 1 from the reservoir.
Then we went to index 4. count =2. Our reservoir has [0,1,2]. Same as before, if we
get 0, then we'll change the result. The chance we get 0 is 1/3, while the chance w
e didn't get is 2/3. i.e The chance we keep the result ==2 is 2/3.
The chance we get index=2 is 1*1/2*2/3=1/3
```

399. Evaluate Division

https://www.youtube.com/watch?v=Uei1fwDoyKk&t=589s (https://www.youtube.com/watch?v=Uei1fwDoyKk&t=589s)

```
# a/c = a/b * b/c i.e. 2 * 3 = 6
# Visualize it as a graph: a->b->c with weights 2 and 3. Now to find a/c, we have t
o traverse from a to c and keep multiplying the weights
# In reverse direction: c/a, we have to start from c and traverse to a and multiply
inverse of weights

# build adjacency list of dict from equations: list of 2 elements i.e. {a:[b, value
of a/b]}
# in reverse direction as well i.e. {b:[a, value of b/a]}
# BFS from source to destination with weights multiplication (for each query)
# edge case: if variable in query does not exist in graph i.e. adjacency list, resu
lt is -1

from collections import defaultdict, deque
class Solution:
    def calcEquation(self, equations: List[List[str]], values: List[float], querie
s: List[List[str]]) -> List[float]:
        # build adjacency list of dict: list of 2 elements i.e. {a:[b, value of a/
b]}
        # in reverse direction as well i.e. {b:[a, value of b/a]}
        adj = defaultdict(list)

        for i, eq in enumerate(equations):
            a, b = eq
            adj[a].append([b, values[i]])
            adj[b].append([a, 1/values[i]])

        def bfs(src, dst):
            if src not in adj or dst not in adj: #if src or dst not in adjacency li
st
                return -1

            q = deque()
            visited = set()
            q.append([src, 1]) # start weight as 1 since we have to multiply
            visited.add(src)

            while q:
                n, w = q.popleft()

                if n == dst:
                    return w

                for node, weight in adj[n]:
                    if node not in visited:
                        q.append([node, w * weight]) # cannot make it as 2 separate
statement [w = w * weight]
                        visited.add(node)

            return -1 # if src or dst exist but no path between them
```

```
        return[bfs(query[0], query[1]) for query in queries]
```

409. Longest Palindrome

```
# Create a hastable with freq of chars
# All chars with even frequency are taken
# All chars with odd freq = take their even component
# You can take one odd component, if there are any

from collections import defaultdict
class Solution:
    def longestPalindrome(self, s: str) -> int:
        d = defaultdict(int)
        result = 0
        for c in s:
            d[c] = d[c] + 1

        even_count = 0
        odd_count_present = 0
        # e.g. abababaaba -> 9
        for key in d.keys():
            even_count = even_count + d[key]//2
            if d[key] % 2 != 0:
                odd_count_present = 1

        result = (even_count * 2) + odd_count_present
        return result

# Time Complexity = O(n)
# Space complexity = O(1) as hash table keys are alphabets which can not exceed 26
(independent of input size)
```

415. Add Strings

ord() in python:

The ord() function in Python is a built-in function that accepts a string containing a single Unicode character and returns an integer representing the Unicode point of that character. For instance, ord('A') would return 65, the Unicode point for the character 'A'.

Given a string of length one, return an integer representing the Unicode point of the character Input : a Output : 97

ord('9') - ord('0') => 9

Imp: If the string length is more then one, and a **TypeError** will be raised.

zfill: The zfill() method adds zeros (0) at the beginning of the string, until the string reaches the specified length provided as len argument Syntax : string.zfill(len)

Also, result[::-1] does not reverse list in-place, it *returns * reversed list

```
# Make string lengths equal
# Iterate from end and extract the digits without converting it directly to int, ge
t the sum and carry
# if carry, remains at end
# reverse


class Solution:
    def addStrings(self, num1: str, num2: str) -> str:

                # Make string lengths equal
        max_len = max(len(num1), len(num2))
        num1, num2 = num1.zfill(max_len), num2.zfill(max_len)


        p = len(num1) - 1
        carry = 0
        result = []

                # Iterate from end and extract the digits without converting it dir
ectly to int, get the sum and carry
        while p >= 0:
            x1 = ord(num1[p]) - ord('0')
            x2 = ord(num2[p]) - ord('0')

            sum = (x1 + x2 + carry) % 10     # remainder
            carry = (x1 + x2 + carry) // 10    # quotient
            result.append(sum)
            p = p -1

                # if carry, remains at end
        if carry > 0:
            result.append(carry)

        # reverse
        return "".join(str(x) for x in result[::-1])

# Time and Space Complexity = O(max(n1,n2)) where n1 and n2 are lengths of num1 and
num2
```

416. Partition Equal Subset Sum

https://www.youtube.com/watch?v=IsvocB5BJhw&t=4s (https://www.youtube.com/watch?v=IsvocB5BJhw&t=4s)

```
# only +ve integers in input array
# subset != subarry != subsequence

# If sum of elements in input array = odd, half of it cannot be equal to any sum of
+ve integers
# 2 choices at every element in input array - take it or leave it
# base case: add 0 to the cum sum set
# Create a set which keeps track of cum sum = take or leave each element of input a
rray and keep adding to them with each successive element from input array


class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        if sum(nums) % 2:
            return False

        dp = set()
        dp.add(0)
        target = sum(nums) // 2

        # cannot update dp array while iterating hence nextDP array
        for i in range(len(nums)):
            nextDP = set()

            for t in dp:

                nextDP.add(t)

                if (t + nums[i] > target):
                    continue
                if (t + nums[i]) == target:
                    return True

                nextDP.add(t + nums[i])

            dp = nextDP

        return False

# Time Complexity = O(m * n) where m = subset sum and n = no of elements in input a
rray
# Space Complexity = O(m), set is used where max number of elements = subset sum
```

417. Pacific Atlantic Water Flow

https://www.youtube.com/watch?v=s-VkcjHqkGI (https://www.youtube.com/watch?v=s-VkcjHqkGI)

```
# Two sub-problems: i) find all cells where pacific water can reach ii) find all ce
lls where atlantic water can reach iii) result is common cells

# Initial condition
# Pacific ocean: All cells corresponding to topmost row and leftmost col can reach
pacific ocean
# Atlantic ocean: All cells corresponding to bottommost row and rightmost col can r
each atlantic ocean
# Perform DFS/BFS from all initial cells for both pacific and atlantic ocean

# keep visited set for both pacific and atlantic separately, pac and atl, to keep t
rack of all cells that can be reached via pacific or atlantic ocean correspondingly
# signature of dfs function:  dfs(r, c, visit, prevHeight) where prevHeight is the
height of cell from where DFS is called


class Solution:
    def pacificAtlantic(self, heights: List[List[int]]) -> List[List[int]]:
        ROWS, COLS = len(heights), len(heights[0])
        pac, atl = set(), set()

        def dfs(r, c, visit, prevHeight):
            if (
                (r, c) in visit
                or r < 0
                or c < 0
                or r == ROWS
                or c == COLS
                or heights[r][c] < prevHeight
            ):
                return
            visit.add((r, c))
            dfs(r + 1, c, visit, heights[r][c])
            dfs(r - 1, c, visit, heights[r][c])
            dfs(r, c + 1, visit, heights[r][c])
            dfs(r, c - 1, visit, heights[r][c])

        for c in range(COLS):
            dfs(0, c, pac, heights[0][c])
            dfs(ROWS - 1, c, atl, heights[ROWS - 1][c])

        for r in range(ROWS):
            dfs(r, 0, pac, heights[r][0])
            dfs(r, COLS - 1, atl, heights[r][COLS - 1])

        res = []
        for r in range(ROWS):
            for c in range(COLS):
                if (r, c) in pac and (r, c) in atl:
                    res.append([r, c])
        return res
```

```
# Time Complexity: O(mn)
# Space Complexity: O (mn), visited array
```

424. Longest Repeating Character Replacement

https://www.youtube.com/watch?v=gqXU1UyA8pk&t=1s (https://www.youtube.com/watch?v=gqXU1UyA8pk&t=1s)

```
# Similar to 1838. Frequency of the Most Frequent Element

# Hashmap to keep track of count of chars in the current window, char:count
# Expand window until: len(window) - count of most freq char <= k (valid window)
# if  len(window) - count of most freq char > k, increment left ptr and reduce the count of char at left ptr

class Solution:
    def characterReplacement(self, s: str, k: int) -> int:
        count = defaultdict(int)

        l = 0
        max_freq = 0
        ans = 0
        for r in range(len(s)):
            count[s[r]] += 1

            # to keep track of max count in dict
            max_freq = max(max_freq, count[s[r]])

            if (r - l + 1) - max_freq <= k:
                ans = max(ans, r - l + 1)

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

        return ans

# Time Complexity = O(n)
# Space Complexity = O(1) since only 26 chars
```

433. Minimum Genetic Mutation

```
class Solution:
    def minMutation(self, startGene: str, endGene: str, bank: List[str]) -> int:

        q = deque()
        q.append([startGene, 0])
        visited = set()
        visited.add(startGene)

        while q:
            gene, mutation = q.popleft()
            if gene == endGene:
                return mutation

            for i in range(8):
                for option in ['A', 'C', 'G', 'T']:
                    newgene = gene[:i] + option + gene[i+1:]
                    if newgene in bank and newgene not in visited:
                        q.append([newgene, mutation+1])
                        visited.add(newgene)

        return -1
```

435. Non-overlapping Intervals

```
# Min number of intervals to remove to make interval list non-overlapping = max num
ber of non-overlaping intervals

# Sort by start time
# Initialize first interval's end time as min_end and iterate thru interval list fr
om 1 to end
# If overlap happens, greedily take the min_end out of existing min_end and current
interval's end
# i.e. greedily remove the interval with larger end time

class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:

        intervals.sort(key=lambda x: x[0])

        res = 0
        begin = 0
        end = 1

        min_end = intervals[0][1]

        for i in range(1, len(intervals)):
            if intervals[i][begin] < min_end:
                min_end = min(min_end, intervals[i][end])
                res = res + 1
            else:
                min_end = intervals[i][end]

        return res
```

437. Path Sum III

https://leetcode.com/problems/path-sum-iii/discuss/350205/anybody-has-some-suggestions-on-implementing-prefix-sum-solution-iteratively (https://leetcode.com/problems/path-sum-iii/discuss/350205/anybody-has-some-suggestions-on-implementing-prefix-sum-solution-iteratively)

https://leetcode.com/problems/path-sum-iii/discuss/141424/Python-step-by-step-walk-through.-Easy-to-understand.-Two-solutions-comparison.-%3A- (https://leetcode.com/problems/path-sum-iii/discuss/141424/Python-step-by-step-walk-through.-Easy-to-understand.-Two-solutions-comparison.-%3A-))

https://leetcode.com/problems/path-sum-iii/discuss/91892/Python-solution-with-detailed-explanation (https://leetcode.com/problems/path-sum-iii/discuss/91892/Python-solution-with-detailed-explanation)

```
class Solution:
    def pathSum(self, root: TreeNode, sum: int) -> int:
        if not root:
            return 0

        prefix_sum = defaultdict(int)
        prefix_sum[0] = 1 # dict (map) that will be used to keep track of presum va
lues
        stack = [(root, 0, prefix_sum)]
        count = 0
        while stack:
            root, cur_sum, prefix_sum = stack.pop()
            cur_sum += root.val

            # check to see if we've found any path with given sum
            if cur_sum - sum in prefix_sum:
                count = count + prefix_sum[cur_sum - sum]

            # update prefix_sum
            prefix_sum[cur_sum] = prefix_sum.get(cur_sum, 0) + 1

            # go to subtrees
            # need to create copy of prefix_sum as when unwinding the stack prefix_
sum state needs to be restored from the point it was left

            if root.left:
                stack.append((root.left, cur_sum, dict(prefix_sum)))

            if root.right:
                stack.append((root.right, cur_sum, dict(prefix_sum)))

        return count
```

438. Find All Anagrams in a String

```
# Same as 567. Permutation in String
# Sliding Window

from collections import Counter
class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        if len(p) > len(s): # edge case
            return []

        p_dict = Counter(p)
        s_dict = Counter() # empty
        result = []
        for i in range(len(s)):
            # add one more letter on the right side of the window: build window of
size pattern + keep adding to it
            s_dict[s[i]] = s_dict[s[i]] + 1

            # remove one letter from the left side of the window: slide the window
            if i>=len(p):
                if s_dict[s[i-len(p)]] == 1:
                    del s_dict[s[i-len(p)]]
                else:
                    s_dict[s[i-len(p)]] = s_dict[s[i-len(p)]] -1 # e.g. s = abca an
d p = abc
            # compare
            if s_dict == p_dict:
                result.append(i-len(p)+1)
        return result

# Time Complexity = O(s + p) where s,p = length of string s and p (one-pass)
# Space Complexity = O(1), keys of dict cannot be more than 26 chars
```

442. Find All Duplicates in an Array

Approach 4: Mark Visited Elements in the Input Array itself

```
# The integers in the input array arr satisfy 1 ≤ arr[i] ≤ n, where n is the size o
f array
# All the integers present in the array are +ve
# The decrement of any integers present in the array must be an accessible index in
the array

# Iterate over the array and for every element x in the array, negate the value at
index abs(x)-1
# The negation operation effectively marks the value abs(x) as seen / visited

# Double negation i.e. +ve means the element was seen twice

class Solution:
    def findDuplicates(self, nums: List[int]) -> List[int]:
        result = []

        for num in nums:
            nums[abs(num)-1] = nums[abs(num)-1] * -1     # negate

        for num in nums:
            if nums[abs(num)-1] > 0:
                result.append(abs(num)) # Imp: add abs(x) in result
                nums[abs(num)-1] = nums[abs(num)-1] * -1  # negate again so as to a
void double counting


        return result

# Time Complexity = O(n)
# Space Complexity = O(1)
```

443. String Compression

```
# 3 ptrs:
# anchor: which points to the beginning of same set of chars
# write: which points to the position where next write (char or digit) needs to hap
pen
# read: which points to end of same set of chars

class Solution:
    def compress(self, chars):
        # the start position of the contiguous group of characters we are currently
reading
        anchor = 0
        # position to write next
        write = 0

        for read, c in enumerate(chars):
            # expand read ptr until it is not the same as anchor element or end of
string is reached
            # extract the char which is at anchor position
            if read + 1 == len(chars) or chars[read + 1] != chars[anchor]:
                chars[write] = chars[anchor]
                write += 1

                # if read > anchor then there are multiple chars of same element
                # extract the num of times a char is repeated
                if read > anchor:
                    repeated_times = read - anchor + 1
                    for digit in str(repeated_times):
                        chars[write] = digit
                        write += 1

                # reset anchor ptr to point to next set of chars
                anchor = read + 1

        return write

# Time Complexity = O(n)
# Space Complexity = O(1)
```

449. Serialize and Deserialize BST

https://leetcode.com/problems/serialize-and-deserialize-bst/discuss/93171/Python-O(-N-)-solution.-easy-to-understand (https://leetcode.com/problems/serialize-and-deserialize-bst/discuss/93171/Python-O(-N-)-solution.-easy-to-understand)

```
# Serialize and Deserialize binary tree method (LC 297) cannot be used as the quest
ion mentions encoded string should be 'as compact as possible' meaning you cannot u
se 'null' for empty left or right child (though we can use a delimiter to separate
out nodes)

# Unique BST could be constructed from preorder or postorder traversal only (not in
order or level order)

# That means that BST structure is already encoded in the preorder or postorder tra
versal and hence they are both suitable for the compact serialization

from collections import deque
class Codec:

    def serialize(self, root):
        vals = []

        def preOrder(node):
            if node:
                vals.append(node.val)
                preOrder(node.left)
                preOrder(node.right)

        preOrder(root) # preorder is DLR

        return ' '.join(map(str, vals))

    def deserialize(self, data):
        vals = deque(int(val) for val in data.split()) # put all elements in deque

        def build(minVal, maxVal):
            # left subtree - if the next element is between minVal and val
            # right subtree - if the next element is between val and maxVal
            if vals and minVal < vals[0] < maxVal:
                val = vals.popleft() # leftmost elemnt is popped out (removed)
                #print("val = {}".format(val))
                #print("minval = {}".format(minVal))
                #print("maxval = {}".format(maxVal))
                root = TreeNode(val)
                root.left = build(minVal, val)
                root.right = build(val, maxVal)
                return root

        return build(float('-infinity'), float('infinity'))

# Time Complexity = O(n)
# Space Complexity = O(n)

# Deserialization = deque takes O(n), build takes O(n) => O(n)
```

452. Minimum Number of Arrows to Burst Balloons

```
# Sort by end point
# Initialize with first end point
# If next start point begins after first end point, increase # of arrows and set th
e current end point as new end point
```

456. 132 Pattern

https://leetcode.com/discuss/study-guide/2347639/A-comprehensive-guide-and-template-for-monotonic-stack-based-problems (https://leetcode.com/discuss/study-guide/2347639/A-comprehensive-guide-and-template-for-monotonic-stack-based-problems)

```
# 132 pattern in subsequence (not neccessarily continguous)


# Find previous greater element for each number a.
# Once the previous greater element x is found, check the previous minimum element
for x
# If the previous minimum number is smaller than the number a, we know the pattern
exists
```

490. The Maze

```
# DFS time limit exceeded, BFS - shortest path

# usual BFS except:
# when exploring in any direction, roll the ball until it hits the wall i.e. while
loop (+ backtrack one step to reach the cell before wall)

# if the cell before hitting the wall is not visited, add it to visited set and beg
in BFS from this cell

from collections import deque
class Solution:
    def hasPath(self, maze: List[List[int]], start: List[int], destination: List[in
t]) -> bool:
        ROWS = len(maze)
        COLS = len(maze[0])
        visited = set()

        q = deque()
        q.append(start)
        visited.add((start[0], start[1]))

        while q:
            i,j = q.popleft()

            if (i,j) == (destination[0], destination[1]):
                return True

            directions = [[-1,0], [0,1], [1,0], [0,-1]]
            for direction in directions:
                next_i, next_j = i + direction[0], j + direction[1]

                # Roll the ball until it hits a wall
                while 0 <= next_i < len(maze) and 0 <= next_j < len(maze[0]) and ma
ze[next_i][next_j] == 0:
                    next_i = next_i + direction[0]
                    next_j = next_j + direction[1]

                # next_i,next_j hit a wall when exiting the above while loop, so we
need to backtrack 1 position
                next_i = next_i - direction[0]
                next_j = next_j - direction[1]

                # if not visited
                if (next_i, next_j) not in visited:
                    q.append([next_i, next_j])
                    visited.add((next_i,next_j))

        return False
```

496. Next Greater Element I

https://www.youtube.com/watch?v=sDKpIO2HGq0 (https://www.youtube.com/watch?v=sDKpIO2HGq0)

Awesome post - https://leetcode.com/discuss/study-guide/2347639/A-comprehensive-guide-and-template-for-monotonic-stack-based-problems#:~:text=Monotonic%20stacks%20are%20generally%20used,to%20solve%20variety%20of%20problems (https://leetcode.com/discuss/study-guide/2347639/A-comprehensive-guide-and-template-for-monotonic-stack-based-problems#:~:text=Monotonic%20stacks%20are%20generally%20used,to%20solve%20variety%20of%20problems).

i) Next Greater Element to the Right -> maintain a decreasing stack ii) Next Smaller Element to the Right -> maintain an increasing stack

i a) Previous Greater Element to the Left -> same as Next Greater Element to the Right but start from last element of array to begin (reverse order) ii a) Previous Smaller Element to the Left -> same as Next Smaller Element to the Right but start from last element of array to begin (reverse order)

---

503. Next Greater Element II

Awesome post - https://leetcode.com/discuss/study-guide/2347639/A-comprehensive-guide-and-template-for-monotonic-stack-based-problems#:~:text=Monotonic%20stacks%20are%20generally%20used,to%20solve%20variety%20of%20problems (https://leetcode.com/discuss/study-guide/2347639/A-comprehensive-guide-and-template-for-monotonic-stack-based-problems#:~:text=Monotonic%20stacks%20are%20generally%20used,to%20solve%20variety%20of%20problems).

i) Next Greater Element to the Right -> maintain a decreasing stack ii) Next Smaller Element to the Right -> maintain an increasing stack

i a) Previous Greater Element to the Left -> same as Next Greater Element to the Right but start from last element of array to begin (reverse order) ii a) Previous Smaller Element to the Left -> same as Next Smaller Element to the Right but start from last element of array to begin (reverse order)

```
# Iterate thru the nums array twice since if next greater is not found to the right
then we need to wrap around
```

---

515. Find Largest Value in Each Tree Row

https://medium.com/@timpark0807/leetcode-is-easy-binary-tree-patterns-1-2-6e1793b76415 (https://medium.com/@timpark0807/leetcode-is-easy-binary-tree-patterns-1-2-6e1793b76415)

```
class Solution:
    def largestValues(self, root: TreeNode) -> List[int]:
        if root is None: # edge case
            return []

        queue = []

        queue.append(root)


        result = []


        while queue:
            max_val = float('-inf')

            n = len(queue)
            for i in range(n):
                cur = queue.pop(0)

                if cur.val > max_val:
                    max_val = cur.val

                if cur.left:
                    queue.append(cur.left)

                if cur.right:
                    queue.append(cur.right)

            result.append(max_val)
        return result

# Time Complexity = O(n)
# Space Complexity = O(n) for storing elements in queue
```

518. Coin Change II

Bottom up dynamic programming solution in Editorial All the values of coins are unique.

```
# Create a 2D array called dp with n + 1 rows and amount + 1 columns where dp[i][j]
stores the number of ways to make up the j amount using the coins starting from ind
ex i
# Set dp[i][0] = 1 for all values of i as the base case

# Fill dp using two loops. The outer loop runs from i = n - 1 to 0. The inner loop
runs from j = 1 to amount
# In the nested loops, we perform the following:
# a) If the value of the current coin at index i exceeds j, we cannot use it. We se
t dp[i][j] = dp[i + 1][j].
# b) Otherwise, we add the total number of ways to make up j by using the current c
oin and ignoring it. We set dp[i][j] = dp[i + 1][j] + dp[i][j - coins[i]]


class Solution:
    def change(self, amount: int, coins: List[int]) -> int:
        n = len(coins)
        dp = [[0] * (amount + 1) for _ in range(n + 1)]
        for i in range(n):
            dp[i][0] = 1

        for i in range(n - 1, -1, -1):
            for j in range(1, amount + 1):
                if coins[i] > j:
                    dp[i][j] = dp[i + 1][j]
                else:
                    dp[i][j] = dp[i + 1][j] + dp[i][j - coins[i]]

        return dp[0][amount]

# Time complexity: O(n * amount) where  n = size of coins
# Space Complexity = O(n * amount), space for dp table
```

523. Continuous Subarray Sum

```
# Prefix Sum
# Usual sliding window won't work since it is tricky to make a decision if we want
to expand right or discard left

# whenever the same value is obtained for cumsum%k (i.e. remainder)  corresponding
to two indices i and j, it implies that sum of
# elements betweeen those indices is an integer multiple of k
# Example: [23,2,4] k = 6,
# cumsum = 23, 23%6 = 5
# cumsum = 25, 25%6 = 1
# cumsum = 29, 29%6 = 5. As soon as same remainder is encountered, it means a subar
ray exists that is a multiple of k. Here = [2,4]

# Create a hashmap of cumsum%k (remainder) : index (to count length of subarray)
# if at any point cumsum%k (remainder) already exists in map and index difference i
s > =2  (length of subarray should be atleast 2), return True


class Solution:
    def checkSubarraySum(self, nums: List[int], k: int) -> bool:

        # cumsum %k occurs at index: -1, also handles edge cases
        d = {0 :-1}
        cumsum = 0

        for index, num in enumerate(nums):
            cumsum = cumsum + nums[index]

                        # k should not be zero else division by zero error
            if k != 0:
                cumsum = cumsum%k

            if d.get(cumsum) != None:
                if index - d[cumsum] >= 2: # subarray size atleast 2
                    return True
            else:
                d[cumsum] = index

        return False

# Time Complexity = O(n)
# Space Compexity = O(min(n,k)) : HashMap can contain upto min(n,k) different pairi
ngs.
```

540. Single Element in a Sorted Array

```
# xor approach i.e. a xor a = 0 and a xor a xor b = b but it's O(n) time complexit
y. we need O (log n) solution

# Binary search (make use of the fact that input is sorted)
# if mid is even, then it's duplicate should be in next index or if mid is odd, the
n it's duplicate  should be in previous index => pattern until then in sequence is
not missed
# make sure mid+1 and mid-1 are within bounds (should be first condition in check)
# return left



class Solution:
    def singleNonDuplicate(self, nums: List[int]) -> int:
        left, right = 0, len(nums)-1

        while left <= right:
            mid = left + (right-left)//2
            if (0 <= mid -1 < len(nums) and mid % 2 == 1 and nums[mid - 1] == nums
[mid]) or (0 <= mid+1 < len(nums) and mid%2 == 0 and nums[mid] == nums[mid + 1]):
                left = mid + 1
            else:
                right = mid - 1
        return nums[left]

# Time Complexity = O(log n)
# Space Complexity = O(1)
```

542. 01 Matrix

```
# nearest cell - BFS

# Multi source BFS from 0s (not 1) + add them to visited set (visualize the problem
starting from 0 and not 1)
# first level unvisited neighbors would be at dist 1 and second level unvisited nei
ghbors would be at dist 2


# can use the original matrix to update dist

from collections import deque
class Solution:
    def updateMatrix(self, mat: List[List[int]]) -> List[List[int]]:
        ROWS = len(mat)
        COLS = len(mat[0])
        visited = set()


        q = deque()
        for i in range(ROWS):
            for j in range(COLS):
                if mat[i][j] == 0:
                    q.append([i,j,0])
                    visited.add((i,j))

        while q:
            for i in range(len(q)):
                r, c, level = q.popleft()

                directions = [[1,0], [-1,0], [0,1], [0,-1]]
                for dr, dc in directions:
                    new_r, new_c = r + dr, c + dc

                    if 0 <= new_r < ROWS and 0 <= new_c < COLS and (new_r, new_c) n
ot in visited:
                        visited.add((new_r, new_c))
                        q.append([new_r, new_c, level+1])
                        mat[new_r][new_c] = level + 1

        return mat

# Time Complexity = O(mn)
# Space Complexity = O(mn)
```

543. Diameter of Binary Tree

https://leetcode.com/problems/diameter-of-binary-tree/discuss/101145/Simple-Python
(https://leetcode.com/problems/diameter-of-binary-tree/discuss/101145/Simple-Python)

```
class Solution:
    def diameterOfBinaryTree(self, root: TreeNode) -> int:
        self.result = 0 # self is needed as it needs to keep track of max left+righ
t heights

        def get_height(node):
            if node is None:
                return 0

            left = get_height(node.left)
            right = get_height(node.right)
            self.result = max(self.result, left+right) # path thru this node is len
of left+righ subtree
            return max(left, right) + 1

        get_height(root)
        return self.result

# Time Complexity = O(n)
# Space Complexity = O(n) for keeping track of recursion stack
```

547. Number of Provinces

https://leetcode.com/problems/friend-circles/discuss/101431/Stupid-question%3A-How-is-this-question-different-from-Number-of-Islands (https://leetcode.com/problems/friend-circles/discuss/101431/Stupid-question%3A-How-is-this-question-different-from-Number-of-Islands)

https://leetcode.com/problems/friend-circles/discuss/228414/Wrong-problem-statement-made-me-waste-half-an-hour-looking-for-a-solution-for-a-complex-problem (https://leetcode.com/problems/friend-circles/discuss/228414/Wrong-problem-statement-made-me-waste-half-an-hour-looking-for-a-solution-for-a-complex-problem)

https://leetcode.com/problems/friend-circles/discuss/201096/Pythonthe-classic-DFS-super-easy-to-understand-comments-the-whole-shabang (https://leetcode.com/problems/friend-circles/discuss/201096/Pythonthe-classic-DFS-super-easy-to-understand-comments-the-whole-shabang)!

https://leetcode.com/problems/friend-circles/discuss/101349/Python-Simple-Explanation (https://leetcode.com/problems/friend-circles/discuss/101349/Python-Simple-Explanation)

```
# Adj Matrix and not adj list is given (square matrix)
# symmetric matrix
# Understand the input matrix representation

class Solution:
    def findCircleNum(self, M: List[List[int]]) -> int:
        # Create adj list using adj matrix
        g = {x:[] for x in range(len(M))}
        for i in range(len(M)):
            for j in range(len(M[0])):
                if M[i][j] == 1 and i != j: # Important: avoid self loop
                    g[i].append(j)

        visited = [0 for _ in range(len(M))]

        # usual dfs
        def dfs(node,g,visited):
            if visited[node] == 1:
                return
            visited[node] = 1

            for neighbor in g[node]:
                dfs(neighbor, g, visited)

        friend_circle = 0
        for start_node in range(len(g)):
            if visited[start_node] == 0:
                friend_circle = friend_circle + 1
                dfs(start_node, g, visited)

        return friend_circle
```

560. Subarray Sum Equals K

```
# Prefix Sum
# Sliding window does not work in this case as -ve integers are allowed in input
# main criteria for sliding window to work in any problem is if u move your left po
inter , it should never have a need to move left. Since this problem has negative i
ntegers, you cannot always move your left under the condition curr_sum>k. for some
new discovered element, your window might need to go left as well.


# if the cumulative sum upto two indices, say i and j is at a difference of k i.e.
if cumsum[i] - cumsum[j] = k, the sum of elements lying between indices i and j is
k.

# Create a map of cumsum: num of occurrence
# if cumsum-k exists in the map, it means num of occurrences of cumsum -k should be
added
# if cumsum-k == 0 exists in map,  add it's # of occurence (base case) which is 1


class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        d = {0:1} # to cover base case cumsum = 0 occurs 1 time, also for case when
cumsum == k
        cumsum = 0
        counter = 0

        for i in range(len(nums)):
            cumsum = nums[i] + cumsum

            # Need to check in dict first before creating dict
            if d.get(cumsum - k) != None:
                counter = counter + d[cumsum - k]

                  if d.get(cumsum) == None:
                d[cumsum] = 1
            else:
                d[cumsum] = d[cumsum] + 1

        return counter

# Time Complexity = O(n)
# Space Complexity = O(n)
```

566. Reshape the Matrix

```
class Solution:
    def matrixReshape(self, nums: List[List[int]], r: int, c: int) -> List[List[in
t]]:
        self.nums = nums
        ro = len(self.nums)
        co = len(self.nums[0])
        so = ro*co
        s = r*c
        if (so != s):
            return self.nums

        flatten = [x for y in self.nums for x in y] # Flatten matrix
        k = 0
        result = []
        for i in range(r):
            sub_result = []
            for j in range(c):
                sub_result.append(flatten[k])
                k = k+1
            result.append(sub_result)
        return result
```

567. Permutation in String

```
# Similar to 438. Find All Anagrams in a String

# 1. Create a dict of pattern, char: count
# 2.  Maintain a dict of main string char:count for the current window of size patt
ern
# build dict upto length of pattern
# if size of dict goes beyond size of pattern, remove one letter from the left side
of the window from this dict
# a) if char count at left side of window = 1, delete this entry
# b) if char count at left side of window = > 1, decrement the count
# 3. Compare 1 and 2

class Solution:
    def checkInclusion(self, s1: str, s2: str) -> bool:

        left = 0
        p_dict = Counter(s1)
        s_dict = Counter()

        for right in range(len(s2)):
            #  add letters on the right side of the window with count
            if s_dict.get(s2[right]) == None:
                s_dict[s2[right]] = 1

            else:
                s_dict[s2[right]] = s_dict[s2[right]] + 1

            # remove one letter from the left side of the window if len of window >
size of pattern
            if right >= len(s1):
                if s_dict[s2[left]] == 1:
                    del s_dict[s2[left]]
                    left = left + 1
                else:
                    s_dict[s2[left]] = s_dict[s2[left]] - 1
                    left = left + 1
            # compare
            if p_dict == s_dict:
                return True
        return False



# Time Complexity = O(len(s1) + len(s2))
# Space Complexity = O(1), keys of dict cannot be more than 26 chars
```

572. Subtree of Another Tree

First comment in: https://leetcode.com/problems/subtree-of-another-tree/discuss/102741/Python-Straightforward-with-Explanation-(O(ST)-and-O(S%2BT)-approaches (https://leetcode.com/problems/subtree-of-another-tree/discuss/102741/Python-Straightforward-with-Explanation-(O(ST)-and-O(S%2BT)-approaches))

Approach 2 in Solutions

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def isSubtree(self, s: TreeNode, t: TreeNode) -> bool:

        # checks if two trees are same
        def isMatch(s, t):
            if s is None and t is None:
                return True
            if (s is None and t is not None) or (s is not None and t is None):
                return False

            if s.val != t.val:
                return False

            return isMatch(s.left, t.left) and isMatch(s.right, t.right)

        if s is None: # check needed as below isSubtree uses s.left and s.right
            return False

        if isMatch(s, t): # Both are exact same trees
            return True

        # recursively check s left subtree with t and s right subtree with t
        if self.isSubtree(s.left, t) or self.isSubtree(s.right, t):
            return True
        else:
            return False

# Time Complexity = O(m*n) where m, n is number of nodes in t and s
# Space Complexity = O(n) where n number of nodes in s
```

621. Task Scheduler

```
# See approach 2 (math based)  in solutions

# Max of below
# 1. No idle slots: most frequent task is not frequent enough to force presence of
idle slots -> len(tasks)
# 2. Some Idle slots: most frequent task is frequent enough to force some idle slot
s -> (freq of most frequent task -1) * (cooling period +1) + count of most frequent
tasks

# need to calculate a) freq of most frequent task b) count of most frequent tasks

from collections import defaultdict

class Solution:
    def leastInterval(self, tasks: List[str], n: int) -> int:

        freq_dict = defaultdict(int)
        for task in tasks:
            freq_dict[task] = freq_dict[task] + 1

        max_freq = max(freq_dict.values())

        no_of_elements_with_max_freq = 0
        for key, value in freq_dict.items():
            if value == max_freq:
                no_of_elements_with_max_freq += 1

        return max(len(tasks), ((max_freq - 1) * (1+n)) + (no_of_elements_with_max_
freq))

# Time Complexity = O(n) where n is len of tasks
# Space Complexity = O(1) since dict has char as the key and only 26 chars are poss
ible
```

```
# List approach to calculate freq bcz sorting by dict key cannot be done in-place a
nd needs extra space
# calc initial total idle slots: (max freq -1) * cool-off time
# calc how much of idle time can be utilized with tasks with lower/equal freq than
the max freq task  : idle_slots = idle_slots - min(f_max - 1, freq.pop())

class Solution:
    def leastInterval(self, tasks: List[str], n: int) -> int:

        freq = [0] * 26

        for t in tasks:
            freq[ord(t) - ord('A')] = freq[ord(t) - ord('A')] + 1

        freq.sort()

        f_max = freq.pop()
        idle_slots = (f_max -1) * n

        while idle_slots > 0 and freq:
            # if top two tasks have same freq f_max-1 will be taken else freq.pop()
will be taken
            # e.g top two tasks with same freq (e.g. 2) and n= 1
            idle_slots = idle_slots - min(f_max - 1, freq.pop())

        idle_slots = max(0, idle_slots) # idle time cannot go below 0

        return idle_slots + len(tasks)


# Time Complexity = O(n) where n = total number of tasks. freq list is of length 26
irrespective of length of tasks list
# Space Complexity = O(1) since freq list is always of length 26
```

636. Exclusive Time of Functions

https://leetcode.com/problems/exclusive-time-of-functions/discuss/105100/Python-Straightforward-with-Explanation (https://leetcode.com/problems/exclusive-time-of-functions/discuss/105100/Python-Straightforward-with-Explanation)

```
# end timestamp is "ending at the end" of the timestamp hence while pop ts+1 used i
n substraction
# e.g. 0:start:0, 1:start:2, 1:end:5, 0:end:10

class Solution:
    def exclusiveTime(self, n: int, logs: List[str]) -> List[int]:
        result = [0] * n
        stack = []
        prev_ts = 0

        for log in logs:
            fn_id, event, ts = log.split(":")
            ts = int(ts)
            fn_id = int(fn_id) # needed else result[stack[-1]] will throw error: li
st indices must be int

            if event == 'start':
                if stack:
                    result[stack[-1]] = result[stack[-1]] + ts - prev_ts

                stack.append(fn_id)
                prev_ts = ts

            else:
                result[stack.pop()] += ts+1 - prev_ts  # use + else two pop() and w
ill throw error
                prev_ts = ts+1

        return result

# Time Complexity = O(number of elements in logs) i.e. len(logs)
# Space Complexity = O (len(logs) / 2), stackstack can grow upto a depth of atmost
n/2
```

647. Palindromic Substrings

https://www.techiedelight.com/find-possible-palindromic-substrings-string/
(https://www.techiedelight.com/find-possible-palindromic-substrings-string/)

```
# All "possible" palindrome substrings
# Substrings - continguous
# Notice that if [a, b] is a palindromic interval, then [a+1, b-1] is one too
# expand around center and count all valid palindromes

class Solution:
    def countSubstrings(self, s: str) -> int:
        def expand(low,high):
            count = 0
            while(low >= 0 and high <len(s) and s[low]==s[high]):
                low = low-1
                high = high +1

                count = count + 1
            return count


        counter = 0
        for i in range(len(s)):
            count_odd_len = expand(i,i)
            counter = counter + count_odd_len

            count_even_len = expand(i, i+1)
            counter = counter + count_even_len

        return counter

# Time Complexity = O(n^2) , at each position in n, expand till ends of string of l
en n
# Space Complexity = O(1)
```

658. Find K Closest Elements

```
# Solution 1: Heap

Class solution:
def findClosestElements(self, arr: List[int], k: int, x: int) -> List[int]:
    heap = []
        # Put first k elements in heap (elements are in sorted order)
    for i in range(k):
        heappush(heap, arr[i])

        # For the remaining elements in array, if they are closer to x than the min elemnt at heap[0], push them onto heap after popping the min element
        # i.e. maintain a heap of size k at all times
    for i in range(k, len(arr)):
        if abs(arr[i]-x) < abs(heap[0]-x):
            heappop(heap)
            heappush(heap, arr[i])

      # return sorted list
    return sorted(heap)

# Time Complexity = O(k log k)
# Space Complexity = O(k)

# Solution 2: Binary Search to find the left bound

# Find the leftmost index from where k elements begin
# if k elements in output then the biggest index the left bound could be is len(array) - k if the smallest index is 0
# If the element at arr[mid] is closer to x than arr[mid + k], then that means arr[mid + k], as well as every element to the right of it can never be in the answer,

# Using minimization template of Binary Search (https://leetcode.com/discuss/study-guide/2371234/#template)
# In minimization template, high ptr (here left_end) contains the answer

class Solution:
    def findClosestElements(self, arr: List[int], k: int, x: int) -> List[int]:
        # Initialize binary search bounds
        left_begin = -1
        left_end = len(arr) - k

        # Binary search against the criteria described since it is sorted array
        while left_begin + 1 < left_end:
            mid = left_begin + (left_end - left_begin) // 2
            if x - arr[mid] <= arr[mid + k] - x:
                left_end = mid
            else:
                left_begin = mid

        return arr[left_end:left_end + k]
```

```
# Time Complexity = O(log (n-k) + k), O(log (n-k)) for binary search and O(k) for s
licing array
# Space Complexity = O(1) constant amount of space for our pointers, space used for
the output does not count towards the space complexity
```

670. Maximum Swap

map(func, iter) -> Returns a list of the results(iterable) after applying the given function to each item of a given iterable

hence num = map(list, str(num)) will not work as each element of num will become list

https://leetcode.com/problems/maximum-swap/discuss/107066/Python-Straightforward-with-Explanation (https://leetcode.com/problems/maximum-swap/discuss/107066/Python-Straightforward-with-Explanation)

```
# Create map of each digit in num (key = digit and value = index)
# Iterate thru each digit in num and if any digit from 9 to x+1 is already in num a
nd if it is at a later  # position than the digit, swap and return result (max 1 sw
ap allowed)

# Be careful with converting int to list of strings

class Solution:
    def maximumSwap(self, num: int) -> int:
        A = list(str(num))  # list of string elements
        d = { int(x): i for i, x in enumerate(A) }

        for i,x in enumerate(A):
            for digit in range(9, int(x),-1):  # Imp: from 9,8,7...,x+1
                if d.get(digit) != None and d.get(digit) > i:

                    A[i], A[d[digit]] = A[d[digit]], A[i]
                    return int("".join(A))  # not break since two for loops

        return num # if the num is already in descending order e.g. 9973 (above ret
urn will not be hit)

# Time Complexity = O(n); iterating from 9 to x+1 in descending order is constant
(independent of input)
# Space Complexity = O(n) i.e. storage for A; d is constant as it will be max 10 fo
r any input
```

678. Valid Parenthesis String

```
# Two stack approach: one for keeping index of '(' and another one for '*'
# when ')' comes first check in '(' stack and then in '*' stack and pop them
# check if '(' is on the left hand side of '*' by comparing and poping indexes from
both stacks e.g. *( -> invalid (that's why keep index in stack)
# At the end, stack containing '(' brackets should be empty
# (stack containing * could not non-empty as it can be counted as empty string)

class Solution:
    def checkValidString(self, s: str) -> bool:

        stack = []
        star = []

        for idx, symbol in enumerate(s):

            if symbol == '*':
                star.append(idx)

            if symbol == '(':
                stack.append(idx)

            if symbol == ')':
                if stack:
                    stack.pop()
                elif star:
                    star.pop()
                else:
                    return False

        # consider the case "*(" i.e. * coming before ( -> invalid
        # '(' must be on the left hand side of '*' i.e. "(*" is valid bcz * can be
counted as closing bracket
        while stack and star:
            if stack[-1] > star[-1]:
                return False

            stack.pop()
            star.pop()

        # Accept when stack is empty, which means all braces are paired. Else, reje
ct
        if len(stack) == 0:
            return True
        else:
            return False

# Time Complexity = O(n)
# Space Complexity = O(n)
```

680. Valid Palindrome II

https://leetcode.com/problems/valid-palindrome-ii/discuss/107718/Easy-to-Understand-Python-Solution (https://leetcode.com/problems/valid-palindrome-ii/discuss/107718/Easy-to-Understand-Python-Solution)

```
# Two cases if letters do not match at left and right:
# s[start+1] == s[stop] or s[start] == s[stop-1] or both
# e.g. ebcbb ec ecabbac ec bbcbe" , "abc"

class Solution:
    def validPalindrome(self, s: str) -> bool:

        def isPalindrome(ss):
            begin = 0
            end = len(ss) - 1
            while begin < end:
                if ss[begin] != ss[end]:
                    return False
                begin = begin +1
                end = end -1
            return True

        left = 0
        right = len(s) - 1


        while left < right:
            if s[left] != s[right]:  # at most one deletion allowed
                return isPalindrome(s[left+1: right+1]) or isPalindrome(s[left:righ
t])

            left = left +1
            right = right -1

        return True

# Time Complexity = O(n)
# Space Complexity = O(n) due to slicing operation
```

684. Redundant Connection

https://www.youtube.com/watch?v=FXWRE67PLL0&t=3s (https://www.youtube.com/watch?v=FXWRE67PLL0&t=3s)

```
# DFS solution possible but it is O(n^2) bcz for each pair of nodes, we would run d
fs to see if path exists between edge e1 and edge e2
# i.e. run dfs from e1 to see if it is reachable to e2
# https://leetcode.com/problems/redundant-connection/solutions/3876792/python3-dfs-
with-clear-explanation/?envType=list&envId=x8sbwdxv

# union-find sol O(n)
# union nodes in each edge until nodes have same representative i.e. they are alrea
dy in the same set and the edge becomes redundant

class UnionFind:
    def __init__(self, n):

        # Initially every node is a parent of itself, index of this array represent
s node and value represents the parent/representative
        self.parent = [i for i in range(n)]

        # If ith index is the parent/representative of a set, rank is the number of
nodes in this set
        # Initially, rank is set to 1
        self.rank = [1] * n


    # Find the parent/root/representative of a node
    # if x is not the parent of itself, then find it recursively
    def find(self, x):

        if (self.parent[x] != x):
            self.parent[x] = self.find(self.parent[x])

        return self.parent[x]

    def union(self, x, y):

        # Find current parent/representative of x and y
        xset = self.find(x)
        yset = self.find(y)

        # If representatives are same, we have found the redundant connection
        if xset == yset:
            return False

        # if ranks are different for parents, put smaller ranked item under bigger
ranked item and add ranks
        if self.rank[xset] < self.rank[yset]:
            self.parent[xset] = yset
            self.rank[yset] = self.rank[yset] + self.rank[xset]

        elif self.rank[xset] > self.rank[yset]:
            self.parent[yset] = xset
            self.rank[xset] = self.rank[xset] + self.rank[yset]
```

```
        # If ranks are same for parents, then move y under x (doesn't matter which
one goes where) and increment rank of x's tree
        else:
            self.parent[yset] = xset
            self.rank[xset] = self.rank[xset] + self.rank[yset]

class Solution:
    def findRedundantConnection(self, edges: List[List[int]]) -> List[int]:
        uf = UnionFind(len(edges) + 1) # nodes are from 1 to n, not 0 to n-1
        for e1, e2 in edges:
            if uf.union(e1, e2) == False:
                return [e1,e2]

# Time Complexity = O(E * log/alpha V) where alpha = inverse Ackermann function, lo
g since height of tree with n nodes is log n
# Space Complexity = O(V)
# More description about complexity - https://leetcode.com/explore/featured/card/gr
aph/618/disjoint-set/3843/
```

688. Knight Probability in Chessboard

https://www.youtube.com/watch?v=PCNvgEnUbmY (https://www.youtube.com/watch?v=PCNvgEnUbmY)
https://knapsacklabs.netlify.app/course/coding_interview/dynamic_programming/knight_probability_in_chessboard (https://knapsacklabs.netlify.app/course/coding_interview/dynamic_programming/knight_probability_in_chessboard)

```
# Solution 1: BFS
# Starting from the initial position (prob = 1), perform a BFS along 8 directions
(if within bounds of board, prob = 1/8 for the landing position) for k moves
# Now if a position on board has already been arrived at by prev moves, prob of lan
ding at this position = prev prob * 1/8
# Sum all the prob within bounds of board to get final prob to remain on board
# Time Complexity: 8^k since 1st move -> 8 positions, now for each of the 8 positio
ns we can move 8 more positions in next/2nd move and so on i.e. 8, 8*8, 8*8*8,....
# Space Complexity = 8^k since the length of queue can grow like 8, 8*8, 8*8*8,.. 8
^k



Solution 2: DP (better time and space complexity)
# Start from top-left position on board and calculate/record prob of arriving at th
at position from 8 positions from prev move
# Note that a given position can be arrived at by multiple positions from prev move
s
# In that case, sum the (prob of arriving from each of the positions from prev mov
e/8)
# Since we only need, prob of positions from prev move, we can have 2 tables: one f
or current and one for prev and swap after each move
# At the end, sum the prob of each position on the board
# Time Complexity = k(n^2): k moves and for each move we iterate the entire chessbo
ard (n^2)
# Space Complexity = 2 (n^2): 2 tables to keep track of prob for prev and current m
ove for each of the positions on chessboard

class Solution:
    def knightProbability(self, n: int, k: int, row: int, column: int) -> float:
        # Define possible directions for the knight's moves
        directions = [(1, 2), (1, -2), (-1, 2), (-1, -2),
                      (2, 1), (2, -1), (-2, 1), (-2, -1)]

        # Initialize the previous and current DP arrays
        prev_dp = [[0] * n for _ in range(n)]
        curr_dp = [[0] * n for _ in range(n)]

        # Set the probability of the starting position to 1
        prev_dp[row][column] = 1

        # Iterate over the number of moves and update a new curr_dp using prev_dp
        for moves in range(k):
            # Iterate over the cells on the chessboard
            for i in range(n):
                for j in range(n):
                    # Reset the probability for the current cell
                    curr_dp[i][j] = 0

                    # Iterate over possible directions by calculating 8 prev positi
ons
                    for direction in directions:
```

```
                        prev_i, prev_j = i - direction[0], j - direction[1]
                        # Check if the previous cell is within the chessboard
                        if 0 <= prev_i < n and 0 <= prev_j < n:
                            # Update the curr probability table using prev prob tab
le, note division by 8
                            curr_dp[i][j] += prev_dp[prev_i][prev_j] / 8

            # Swap the previous and current DP arrays
            prev_dp, curr_dp = curr_dp, prev_dp

        # Calculate the total probability by summing prob
        total_probability = 0
        for i in range(n):
            for j in range(n):
                total_probability += prev_dp[i][j] # bcz of swap, use prev_dp to su
m up prob

        return total_probability
```

694. Number of Distinct Islands

```
# When we start a depth-first search on the top-left square of some island, the pat
h taken by our depth-first search will be the same if, and only if, the shape is th
e same
# Hash By Path Signature

# keep track of every DFS path direction-wise including backtracking
# dedupe direction paths
# dfs function signature: dfs(row, col, direction)

class Solution:
    def numDistinctIslands(self, grid: List[List[int]]) -> int:
        seen = set()
        unique_islands = set()

        # Do a DFS to find all cells in the current island.
        def dfs(row, col, direction):
            if row < 0 or col < 0 or row >= len(grid) or col >= len(grid[0]) or (ro
w, col) in seen or grid[row][col] == 0:
                return

            seen.add((row, col))
                        # keep track of path direction-wise
            path_signature.append(direction)

            dfs(row + 1, col, "D")
            dfs(row - 1, col, "U")
            dfs(row, col + 1, "R")
            dfs(row, col - 1, "L")
                        # keep track of backtracking direction as well
            path_signature.append("0")


        # Repeatedly start DFS's as long as there are islands remaining.
        for row in range(len(grid)):
            for col in range(len(grid[0])):
                if grid[row][col] == 1:
                    path_signature = [] # path_signature list will be available in
dfs function
                    dfs(row, col, "0") # to match dfs function signature
                    if path_signature:
                        unique_islands.add(tuple(path_signature))

        return len(unique_islands)
```

695. Max Area of Island

```
# Both DFS and BFS solution possible
# BFS:
# Every queue instance does 1 BFS from one source
# keep track of land encountered in every BFS and then take max

from collections import deque
class Solution:
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        ROWS = len(grid)
        COLS = len(grid[0])
        visited = set()
        result = []

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

            area = 1
            while q:
                r,c = q.popleft()
                directions = [[1,0], [-1,0], [0,1], [0,-1]]
                for dr, dc in directions:
                    new_r, new_c = r + dr, c + dc
                    if 0 <= new_r < ROWS and 0 <= new_c < COLS and (new_r, new_c) n
ot in visited and grid[new_r][new_c] == 1:
                        q.append((new_r, new_c))
                        area = area + 1
                        visited.add((new_r, new_c))

            result.append(area)


        island = 0
        for i in range(ROWS):
            for j in range(COLS):
                if grid[i][j] == 1 and (i,j) not in visited:
                    island = island + 1
                    bfs(i,j)
        return max(result) if island else 0

# Time Complexity = O(mn)
# Space Complexity = O (mn) for queue or/and visited array
```

713. Subarray Product Less Than K

```
# Sliding Window

class Solution:
    def numSubarrayProductLessThanK(self, nums: List[int], k: int) -> int:
        if k <= 1:
            return 0

        prod = 1
        ans = left = 0
        for right, val in enumerate(nums):
            prod *= val
            while prod >= k:
                prod /= nums[left]
                left += 1
            ans += right - left + 1
        return ans
```

---

721. Accounts Merge

https://www.youtube.com/watch?v=f17PKE8W2p8 (https://www.youtube.com/watch?v=f17PKE8W2p8)

```

# DFS for each connected component i.e. emails that are connected to each other

# Trickiest part: Building dicts/hashmaps

# 2 dicts: email to emails mapping (accountDict) and email to name mapping. The value of email to emails mapping should be kept as set to avoid duplicates

# Connect all emails for a given account. To do this, take every email and connect with all other emails in the same account (including itself) using hashmap (notice that the graph should be undirected and hence adjacency list has connections both ways)

# Now, it is possible that email in another account matches with one of the emails built until now, but since each email has its own hashmap, the emails in other account will be appended to already seen emails

# For every distinct email (i.e. email in email to name dict), run DFS and keep track of this DFS path
```

class Solution: def accountsMerge(self, accounts: List[List[str]]) -> List[List[str]]: #AccountDict stores doubly directed email linked to head of the email list accountDict = defaultdict(set) # value ->set() emailToNameDict = {} result = []

```
    #Build both the dictionaries
    for account in accounts:
        accName = account[0]
        emailHead = account[1]

        for i in range(1,len(account)):
            currEmail = account[i]

            accountDict[emailHead].add(currEmail)
            accountDict[currEmail].add(emailHead)
            emailToNameDict[currEmail] =  accName

    #Traverse dfs of the graph to find emails that are connected to this email
    def dfs(currEmail,mergedAcc):
        if currEmail in visited:
            return

        visited.add(currEmail)
        mergedAcc.append(currEmail)
        for neighbor in accountDict[currEmail]:
            dfs(neighbor,mergedAcc)

        return mergedAcc

    #visited set to ensure we do not visit and already visited node
    visited = set()
    for email in emailToNameDict:
        if email not in visited:
            result.append([emailToNameDict[email]] + sorted(dfs(email,[])))

    return result
```

Time Complexity = O(N K log NK)

N is the number of accounts and K is the maximum length of an account i.e. total number of email = N*K, need to sort t so log factor

Space Complexity = O(NK)

---

733. Flood Fill

```
# usual BFS

# Color of starting node should be matched with neighbors  (could be 0 or 1)
# If matched, then fill with requested color

from collections import deque
class Solution:
    def floodFill(self, image: List[List[int]], sr: int, sc: int, color: int) -> Li
st[List[int]]:
        ROWS = len(image)
        COLS = len(image[0])
        visited = set()

        q = deque()
        q.append([sr,sc])
        visited.add((sr,sc))

        match_color = image[sr][sc]

        while q:
            r, c = q.popleft()
            image[r][c] = color

            directions = [[1,0], [-1,0], [0,-1], [0,1]]
            for dr, dc in directions:
                new_r, new_c = r + dr, c + dc
                if 0 <= new_r < ROWS and 0 <= new_c < COLS and image[new_r][new_c]
== match_color and (new_r, new_c) not in visited:
                    q.append([new_r, new_c])
                    visited.add((new_r, new_c))

        return image

# Time Complexity = O(mn)
# Space complexity = O(mn), visited array and queue
```

739. Daily Temperatures

Awesome post - https://leetcode.com/discuss/study-guide/2347639/A-comprehensive-guide-and-template-for-monotonic-stack-based-problems (https://leetcode.com/discuss/study-guide/2347639/A-comprehensive-guide-and-template-for-monotonic-stack-based-problems)

i) Next Greater Element to the Right -> maintain a monotonic decreasing stack ii) Next Smaller Element to the Right -> maintain a monotonic increasing stack

i a) Previous Greater Element to the Left -> same as Next Greater Element to the Right but start from last element of array to begin (reverse order) ii a) Previous Smaller Element to the Left -> same as Next Smaller Element to the Right but start from last element of array to begin (reverse order)

```
# Next greater element to the right: maintain decreasing stack

# Use stack having element [temperature, index]
# If the incoming element greater than topmost element of stack, pop until this con
dition is not true anymore and finally push this incoming element onto stack
# If the incoming element is less than or equal to topmost element of stack, push i
t onto stack

# Initialize result array with 0 (default value)


class Solution:
    def dailyTemperatures(self, temperatures: List[int]) -> List[int]:
        res = [0] * len(temperatures)
        stack = []  # pair: [temp, index]

        for i, t in enumerate(temperatures):
            while stack and t > stack[-1][0]:
                stackT, stackInd = stack.pop()
                res[stackInd] = i - stackInd
            stack.append([t, i])

        return res

    # Time Complexity = O(n)
    # Space Complexity = O(n) for stack storage
```

746. Min Cost Climbing Stairs

```
# The "top of the floor" refers to beyond array bounds hence min cost array is len
(cost array) + 1
# minimum cost to reach the ith step is equal to minimumCost[i] = min(minimumCost[i
- 1] + cost[i - 1], minimumCost[i - 2] + cost[i - 2])
# Since we can start with 0 or 1 step, min cost to "reach" 0th or 1st step is 0

class Solution:
    def minCostClimbingStairs(self, cost: List[int]) -> int:
        # The array's length should be 1 longer than the length of cost since top f
loor can be considered as len(array) + 1
        minimum_cost = [0] * (len(cost) + 1)

        # Start iteration from step 2, since the minimum cost of reaching step 0 an
d step 1 is 0 (initialized before)
        for i in range(2, len(cost) + 1):
            take_one_step = minimum_cost[i - 1] + cost[i - 1]
            take_two_steps = minimum_cost[i - 2] + cost[i - 2]
            minimum_cost[i] = min(take_one_step, take_two_steps)

        # The final element in minimum_cost refers to the top floor
        return minimum_cost[-1]


# Time Complexity = O(N)
# Space Complexity = O(N) size of minimum_cost array,
# Space complexity can be optimized to O(1) since we need to keep track of only las
t two minimum cost

class Solution:
    def minCostClimbingStairs(self, cost: List[int]) -> int:
        down_one = down_two = 0
        for i in range(2, len(cost) + 1):
            temp = down_one
            down_one = min(down_one + cost[i - 1], down_two + cost[i - 2])
            down_two = temp

        return down_one
```

763. Partition Labels

https://www.youtube.com/watch?v=B7m8UmZE-vw (https://www.youtube.com/watch?v=B7m8UmZE-vw)

```
# 2 pass thru the input string
# Build a hashmap of char: last position seen

# Iterate thru the input string and keep track of max end seen until now
# At any point, max end seen equals the iterated position -> boundary found i.e. al
l chars before it are always within this boundary


class Solution:
    def partitionLabels(self, s: str) -> List[int]:

        last_occur = {}
        for i, letter in enumerate(s):
            last_occur[letter] = i

        size = 0
        end = 0
        result = []

        for i, char in enumerate(s):
            size = size + 1
            end = max(end, last_occur[char])

            if i == end:
                result.append(size)
                size = 0

        return result

# Time Complexity = O(n)
# Space Complexity = O(1) since only 26 chars
```

766. Toeplitz Matrix

```
class Solution:
    def isToeplitzMatrix(self, matrix: List[List[int]]) -> bool:
        for r, row in enumerate(matrix):
            for c, val in enumerate(row):
                if r > 0 and c > 0 and matrix[r-1][c-1] != matrix[r][c]:
                    return False

        return True

# Time complexity = O(M*N)
# Space Complexity = O(1)
```

767. Reorganize String

https://www.youtube.com/watch?v=v3f30xiaPVc (https://www.youtube.com/watch?v=v3f30xiaPVc)

from collections import Counter S = "ababbaa" print(Counter(S).most_common(1))

O/P:

[('a', 4)]

```
# Build a dict with char: freq and find the most common char and its count (linearl
y)
# No solution: if freq of most common char > (len(string) + 1) //2 i.e. solution po
ssible if most common char count <= (len(string) + 1) //2

# Otherwise solution exists
# Place most frequent char starting at 0th index with space (even indexes)
# Place rest of the letters in odd places, if further even places are not empty (wi
th wrap around)

# e.g. aaabb -> step1:  a [] a [] a, step2: a b a b a

class Solution:
    def reorganizeString(self, s: str) -> str:
        letter_count_map = Counter(s)

        max_count = 0
        most_freq_char = ''
        for letter, count in letter_count_map.items():
            if count > max_count:
                max_count = count
                most_freq_char = letter

        if max_count > (len(s)+1)//2:
            return ""

        ans = [''] * len(s)
        index = 0

                # Place most frequent char
                # make sure after this loop index points to the next even position
where char needs to be filled
                # decrement the count of most freq char in dict as you fill it so t
hat when you fill remaining chars from dict, the most freq char is not flled again
        while letter_count_map[most_freq_char] != 0:
            ans[index] = most_freq_char
            index +=2
            letter_count_map[most_freq_char] -=1

                # Place remaining chars
                # decrement the count in dict
                # wrap around from 1 if end is reached
        for char,count in letter_count_map.items():
            while count > 0:
                if index >= len(s):   # wrap around from 1 (starting odd index)
                    index = 1

                ans[index] = char
                index += 2
                count -= 1
```

```
        return ''.join(ans)


# Time Complexity = O(n)
# Space Complexity = O(1) the number of unique chars in counter, and number of uniq
ue chars is constant (26) hence O(1)
```

703. Kth Largest Element in a Stream

```
# kth largest in sorted order, not necessarily kth distinct element e.g. 1,2,2,3 an
d k = 3 => Ans: 2 and not 1

# KthLargest(int k, int[] nums) -> len(nums) could be < k, ==k or > k
# __init__(): heapify nums and pop until only k elements remain
# add(): push the new element on heap, if size of heap > k, pop and return heap [0]



# heappop(heap) -> pops root element or min element in min heap maintaining the hea
p invariant O(log N)
# heap[0] -> returns min element in min heap without popping or removing the elemen
t from heap O(1)

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

        while len(self.heap) > k:
            heapq.heappop(self.heap)

    def add(self, val: int) -> int:
        heapq.heappush(self.heap, val)
        if len(self.heap) > self.k:
            heapq.heappop(self.heap)
        return self.heap[0]


# Time Complexity:
# init (): O(N + N log N): Heapify operation is O(N) and heap pop happens for (N-K)
elements, each pop takes O(log size of heap) ~ O(N log N)
# add(): each call to add is O(log k) since k is the size of heap
# Space Complexity = O(N): N elements in heap
```

```
# Alternate way of implementing __init__  function (usual min heap way) with time c
omplexity O (N log k)

    def __init__(self, k: int, nums: List[int]):
        self.heap = []
        self.k = k
        for num in nums:
            heapq.heappush(self.heap,num)
            if len(self.heap) > self.k:
               heapq.heappop(self.heap)
```

704. Binary Search

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        low = 0
        high = len(nums) - 1

        while low<=high: # imp: <=
            middle = low + (high-low)//2

            if  target == nums[middle]:
                return middle

            if  target < nums[middle]:
                high = middle-1

            elif target > nums[middle]:
                low = middle + 1

        return -1

# Time Complexity = O(log n)
# Space Complexity = O(1)
```

778. Swim in Rising Water

https://www.youtube.com/watch?v=amvrKlMLuGY&t=976s (https://www.youtube.com/watch?v=amvrKlMLuGY&t=976s)

```
# Dijkstra (BFS with weights)
# Out of all paths possible from top-left to bottom-right, find the path with min o
f max height encountered in each path
# For all neighbors, add the max height between node and neighbor (equivalent to ti
me elapsed) to priority queue
# Pop the neighbor from priority queue/heap which has the min height and continue B
FS


class Solution:
    def swimInWater(self, grid: List[List[int]]) -> int:
        N = len(grid)
        visit = set()
        minH = [[grid[0][0], 0, 0]]  # (time/max-height, r, c)
        directions = [[0, 1], [0, -1], [1, 0], [-1, 0]]

        visit.add((0, 0))
        while minH:
            t, r, c = heapq.heappop(minH)
            if r == N - 1 and c == N - 1:
                return t
            for dr, dc in directions:
                neiR, neiC = r + dr, c + dc
                if (
                    neiR < 0
                    or neiC < 0
                    or neiR == N
                    or neiC == N
                    or (neiR, neiC) in visit
                ):
                    continue

                visit.add((neiR, neiC))
                heapq.heappush(minH, [max(t, grid[neiR][neiC]), neiR, neiC])
```

785. Is Graph Bipartite?

https://www.geeksforgeeks.org/bipartite-graph/ (https://www.geeksforgeeks.org/bipartite-graph/)

```
# BFS
# Bipartite graph is possible if first level BFS neighbors of a node are in differe
nt set than the node itself and so on
# To keep track of visited nodes along with their color, use hashmap

# 1. Assign RED color to the source vertex (putting into set U).
# 2. Color all the neighbors with BLUE color (putting into set V).
# 3. Color all neighbor’s neighbor with RED color (putting into set U).
# 4. This way, assign color to all vertices
# 5. While assigning colors, if we find a neighbor which is colored with same color
as source vertex, then the graph is not Bipartite

# Graph can be disconnected => each of its connected components should be bipartite
# TIP: avoid modular bfs i.e. bfs() sub function


from collections import deque, defaultdict

class Solution:
    def isBipartite(self, graph: List[List[int]]) -> bool:
        # dict of visited node and its color
        color = defaultdict()
        q = deque()


        # graph can be disconnected
        for node in range(len(graph)):
            if node not in color:

                q.append(node)
                # set color of starting node
                color[node] = 0

                while q:

                    popped_node= q.popleft()

                    for neighbor in graph[popped_node]:
                                            # if not visited, alternate the color a
nd put it back in queue
                        if neighbor not in color:
                            color[neighbor] = color[popped_node] ^ 1  # Bitwise XO
R: 1->0 and 0->1
                            q.append(neighbor)
                                                # if color matches
                        elif color[neighbor] == color[popped_node]:
                            return False
        return True
```

787. Cheapest Flights Within K Stops

```
# Dijkstra
# min heap to pop out the min cost neighbor each time
# steps to keep track of stops

class Solution:
    def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst: in
t, k: int) -> int:
        # create adjacency list, from: (to, price)
        adj_list = defaultdict(list)
        for frm, to, price in flights:
            adj_list[frm].append((to, price))

        best_visited = [float("inf")]* n # Initialized to maximum

        min_heap = [ (0, -1, src) ]  # cumulative cost until node, count of stops e
ncountered until node, node

        while min_heap:
            cost, stops, node = heapq.heappop(min_heap) # min cost neighbor is popp
ed

            if stops > k:  # More than k stops, invalid
                continue

            if node==dst:  # reached the destination (first element of min heap is
cost so this cost is the most minimum cost)
                return cost

            best_visited[node] = stops # Update stops

            for neighb, weight in adj_list[node]:
                if stops + 1 < best_visited[neighb]: # Push neighb into the heap, o
nly if stops+1 is better than the last time it was visited
                    heapq.heappush(min_heap, (cost + weight, stops + 1, neighb))

        return -1
```

796. Rotate String

https://www.youtube.com/watch?v=4jY57Ehc14Y (https://www.youtube.com/watch?v=4jY57Ehc14Y)

```
class Solution(object):
    def rotateString(self, A, B):
        return len(A) == len(B) and B in A+A

# Time Complexity = O(n^2), searching a substring in a string without specialized a
lgos such as KMP is n^2
# Space Complexity = O(2n)
```

836. Rectangle Overlap

```
class Solution:
    def isRectangleOverlap(self, rec1: List[int], rec2: List[int]) -> bool:
        start_x = 0
        start_y = 1
        end_x = 2
        end_y = 3

        # Conditions for overlap
        if rec1[start_x] >= rec2[end_x]: return False
        if rec1[end_x] <= rec2[start_x]: return False
        if rec1[start_y] >= rec2[end_y]: return False
        if rec1[end_y] <= rec2[start_y]: return False

        return True

# Time and Space Complexity = O(1)
```

852. Peak Index in a Mountain Array

```
# Exact same as Find Peak Element (LC 162)

class Solution:
    def peakIndexInMountainArray(self, arr: List[int]) -> int:

        l = 0
        r = len(arr) - 1

        while l <= r:
            mid = l + (r-l)//2

            if arr[mid] < arr[mid+1]:
                l = mid + 1
            elif arr[mid] < arr[mid-1]:
                r = mid - 1
            else:
                break
        return mid
```

710. Random Pick with Blacklist

We can use rejectionsampling but since the question mentions "Optimize it such that it minimizes the call to system's Math.random()." - avoid rejection sampling

https://leetcode.com/problems/random-pick-with-blacklist/discuss/144624/Java-O(B)-O(1)-HashMap (https://leetcode.com/problems/random-pick-with-blacklist/discuss/144624/Java-O(B)-O(1)-HashMap)

867. Transpose Matrix

```
class Solution:
    def transpose(self, A: List[List[int]]) -> List[List[int]]:
        return [[A[j][i] for j in range(len(A)) ] for i in range(len(A[0]))]

class Solution(object):
    def transpose(self, A):
        R, C = len(A), len(A[0])
        ans = [[None] * R for _ in xrange(C)]
        for r, row in enumerate(A):
            for c, val in enumerate(row):
                ans[c][r] = val
        return ans

# A-transpose = list(map(list, zip(*A)))
```

470. Implement Rand10() Using Rand7()

Two cases for this type of question:

1) M > N (this question) 2) M < N (rejection sampling alone is sufficient)

https://leetcode.com/problems/implement-rand10-using-rand7/discuss/338395/In-depth-straightforward-detailed-explanation.-Java-Solution (https://leetcode.com/problems/implement-rand10-using-rand7/discuss/338395/In-depth-straightforward-detailed-explanation.-Java-Solution).

See solution section of this problem

Extra Reading: (the product or sum of two uniform distributions is not uniform) https://www.youtube.com/watch?v=5nqHLvWh1Q8 (https://www.youtube.com/watch?v=5nqHLvWh1Q8) https://www.noudaldenhoven.nl/wordpress/?p=117 (https://www.noudaldenhoven.nl/wordpress/?p=117) https://www.quora.com/What-is-the-intuition-behind-the-acceptance-rejection-method-the-method-used-to-generate-random-numbers-for-specific-probability-distribution (https://www.quora.com/What-is-the-intuition-behind-the-acceptance-rejection-method-the-method-used-to-generate-random-numbers-for-specific-probability-distribution)

```
class Solution:
    def rand10(self):
        """
        :rtype: int
        """
        r = float("inf")  # any number above 40 to start the loop

                # repeat if r > 40, rejection sampling
        while(r>40):
            column = rand7()
            row = rand7()
            r = (row - 1) * 7 + column


        result = (r-1) % 10 + 1
        return result

# Time Complexity: O(1)average, but O(∞) worst case.
# Space Complexity: O(1)

# Rejection Sampling is a Geometric distribution, so to calculate the expected valu
e for the number of calls to rand7(), there is a very simple formula:
# E=1/p, so E=1/(40/49)=49/40, and for every success we need call 2 times, so E=2*
49/40=49/20=2.45
```

875. Koko Eating Bananas

```
# Binary search within min and max
# Min: minimum eating speed needs to be at least ceiling(sum(piles) / h)
# Max: Since the number of piles is smaller than h, Koko is guaranteed to finish al
l bananas if eating speed is equal to the largest pile

class Solution:
    def minEatingSpeed(self, piles: List[int], h: int) -> int:
        l = math.ceil(sum(piles)/h)
        r = max(piles)
        res = r # initialize with max value

        while l <= r:  # O(log m)
            k = l + (r-l) // 2

            totalTime = 0
            for p in piles: # O(n)
                totalTime += math.ceil(float(p) / k)

            if totalTime <= h: # any time k satisfies the conditions, it is a possi
ble candidate and minimum seen until now
                res = min(res,k)
                r = k - 1
            else:
                l = k + 1

        return res

# Time Complexity = O(n log m) where m = search space b/w min and max bounds and n
= size of piles array
# Space Complexity = O(1)
```

876. Middle of the Linked List

```
# Two pointer: one slow and another fast, fast moves twice than slow
class Solution:
    def middleNode(self, head: ListNode) -> ListNode:
        curr = head

        while head and head.next: # imp
            curr = curr.next
            head = head.next.next

        return curr
```

528. Random Pick with Weight

https://www.youtube.com/watch?v=fWS0TCcr-lE (https://www.youtube.com/watch?v=fWS0TCcr-lE)

```
http://blog.gainlo.co/index.php/2016/11/11/uber-interview-question-weighted-random-
numbers/

1.W is the sum of all the weights (length of the horizontal line)
2. Get a random number R from [0, W] (randomly select a point)
=> This ensures all points are equally likely
3. Go over each element in order and keep the sum of weights of visited elements. O
nce the sum is larger than R, return the current element. This is finding which are
a includes the point.

https://leetcode.com/problems/random-pick-with-weight/discuss/154475/Python-1-liner
s-using-builtin-functions
```

```
# Weighted Sampling
# Naive soln (extra space)=> expand list by freq of index and choose random element
e.g. [1,2] -> [0,1,1]


# array = [1,2,3]
# prefix sum = [1,3,6] representing buckets: (0,1], (1,3], (3,6]
# random number = 4
# prefix sum where 4 falls -> (3,6], index of this prefix sum bucket = 2

# Prefix sum + Binary search

import bisect
import random
class Solution:

    def __init__(self, w: List[int]): # Time & Space Complexity = O(n)
        self.w = w
        self.prefix_sum = [ 0 ] * (len(self.w))
        self.prefix_sum[0] = self.w[0]
        for i in range(1,len(self.w)):
             self.prefix_sum[i] = self.prefix_sum[i-1] + self.w[i]


    def pickIndex(self) -> int: # Space Complexity = O(1)

        random_number = random.randint(1,self.prefix_sum[-1]) # Time Complexity = O
(1), start from 1(inclusive) since w[i] is always >=1

        result = bisect.bisect_left(self.prefix_sum, random_number) # Time Complexi
ty = O(log n), index of prefix sum where random number falls

        return result
```

896. Monotonic Array

```
# e.g. [6,5,4,4] or [2,3,3,4] -> True

class Solution:
    def isMonotonic(self, A: List[int]) -> bool:

        asc = desc = True

        # if adj values are same, do nothing and iterate ahead
        for i in range(len(A)-1):
            if A[i] < A[i+1]:
                desc = False
            if A[i] > A[i+1]:
                asc = False

        return asc or desc

# Time Complexity = O(n)
# Space Complexity = O(1)
```

909. Snakes and Ladders

BFS solves the shortest path problem for unweighted graphs, and Dijkstra's algorithm solves it for weighted graphs. https://www.youtube.com/watch?v=6lH4nO3JfLk&t=4s (https://www.youtube.com/watch?v=6lH4nO3JfLk&t=4s)

```
class Solution:
    def snakesAndLadders(self, board: List[List[int]]) -> int:
        length = len(board)

        board.reverse()
        def squareToPos(square):
            r = (square - 1) //length
            c = (square - 1) % length
            if r % 2 == 1: # odd rows
                c = length - 1 -c
            return r, c


        q = deque()
        q.append([1,0]) # [square, moves]
        visited = set()
        visited.add(1)

        while q:
            square, moves = q.popleft()
            for i in range(1,7):
                nextsquare = square + i

                r, c = squareToPos(nextsquare) # Check this snake/ladder before che
cking end condition
                if board[r][c] != -1:
                    nextsquare = board[r][c]

                if nextsquare == length * length:
                    return moves + 1

                if nextsquare not in visited:
                    q.append([nextsquare, moves+1])
                    visited.add(nextsquare)

        return -1
```

918. Maximum Sum Circular Subarray

```
# https://leetcode.com/problems/maximum-sum-circular-subarray/solutions/3066098/kad
ane-algo-easy-video-explaination-o-n-o-1/?envType=study-plan-v2&envId=top-interview
-150

# 3 cases:
# 1st case: No wrap around,  find max sum subarray (kadane's algo)
# 2nd case: Wrap around, find max sum subarray using kadane's algo and then check i
f there's a better sol by total sum - minimum subarray sum  [replace max by min in
KA algo]
# overall: return  max(maxSum, totalSum - minSum)

# 3rd corner case: all array elements are -ve, return max -ve element

class Solution:
    def maxSubarraySumCircular(self, nums: List[int]) -> int:
        max_sum = cum_max = min_sum = cum_min = nums[0]

        count_neg = 0
        if nums[0] < 0:
            count_neg = 1

        total_sum = nums[0]

        for i in range(1, len(nums)):
            if nums[i] < 0:
                count_neg = count_neg + 1
            total_sum = total_sum + nums[i]

            cum_max = max(nums[i],cum_max + nums[i])
            max_sum = max(max_sum, cum_max)
            cum_min = min(nums[i], cum_min + nums[i])
            min_sum = min(min_sum, cum_min)

        if count_neg == len(nums):
            return max(nums)

        else:
            return max(max_sum, total_sum - min_sum)

# Time Complexity = O(n)
# Space Complexity = O(1)
```

921. Minimum Add to Make Parentheses Valid

```
# Iterate over bracket string and keep count of opening brackets
# If closing bracket is encountered, 2 conditions:
# i) if opening bracket count is non-zero: decrement opening bracket count
# ii) else if opening bracket count is zero: increment closing bracket count
# return opening bracket count + closing bracket count

class Solution:
    def minAddToMakeValid(self, s: str) -> int:

        unmatched_open = 0
        unmatched_close = 0

        for bracket in s:
            if bracket == '(':
                unmatched_open =  unmatched_open + 1
            else:
                if unmatched_open:
                    unmatched_open =  unmatched_open - 1
                else:
                    unmatched_close = unmatched_close + 1

        return unmatched_close + unmatched_open

# Time Complexity = O(n)
# Space Complexity = O(1)
```

---

934. Shortest Bridge

```
# Shortest path in unweighted graph - BFS
# Shortest path in weighted graph - Djikstra

# DFS/BFS + multi-source BFS
# DFS/BFS to find one island (and keep marking them as 2 to distinguish from anothe
r island) and put all nodes of this island in queue to begin multi-source BFS until
any node of another island is found (i.e.1) (while keeping track of distance)
```

---

938. Range Sum of BST

```
# Binary Search Tree
# if node value > L -> move left
# if node value < R -> move right

class Solution:
    def rangeSumBST(self, root: TreeNode, L: int, R: int) -> int:
        sum = 0
        stack = [root]
        while stack:
            node = stack.pop()

            if L <= node.val <= R:
                sum = sum + node.val


            if node.val > L and node.left:
                stack.append(node.left)
            if node.val < R and node.right:
                stack.append(node.right)

        return sum

# Time Complexity = O(n) worst case, though we skip nodes whose values lie outside
[L,R]
# Space Complexity = O(height of tree)
```

939. Minimum Area Rectangle

https://leetcode.com/problems/minimum-area-rectangle/discuss/240341/Python-O(n2)-easy-to-understand.-Good-for-beginners (https://leetcode.com/problems/minimum-area-rectangle/discuss/240341/Python-O(n2)-easy-to-understand.-Good-for-beginners)

```
# For each pair of points in the array, consider them to be the long diagonal of a
potential
# rectangle. We can check if all 4 points are there using a Set.
# There could be duplicate points
# area could be zero in which case it is not a rectangle

import sys
class Solution:
    def minAreaRect(self, points: List[List[int]]) -> int:
        min_area = sys.maxsize
        s = set()
        for x,y in points:
            s.add((x,y))

        for x1,y1 in s:
            for x2,y2 in s:
                if x1 > x2 and y1 > y2: # only look at pairs already not seen (poin
ts are sorted)
                    if x1 == x2 or y1 == y2: # area becomes zero
                        continue
                    else:
                        if (x1,y2) in s and (x2,y1) in s:
                            area = abs(x2-x1) * abs(y2-y1)
                            min_area = min(area, min_area)

        if min_area == sys.maxsize:
            return 0
        else:
            return min_area
```

953. Verifying an Alien Dictionary

```
# hash + 2 ptr
# Two strings are lexicographically ordered if first non-matching char is in order

class Solution:
    def isAlienSorted(self, words: List[str], order: str) -> bool:
            # build a dict of char and its position/index from the order of alien l
anguage alphabet
        char_order = {c:i for i,c in enumerate(order)}
        char_order['#'] = -1 # padding char order should be lower than lowest one t
o handle cases such as "apple, "app" -> False

         # Iterate thru the list of words and compare 2 consecutive words at a time
                for i in range(len(words) -1):
            word1 = words[i]
            word2 = words[i+1]

            # Make the length of both strings equal with padding
            if len(word1) < len(word2):
                word1 = word1 + "#" * (len(word2) - len(word1))
            elif len(word1) > len(word2):
                word2 = word2 + "#" * (len(word1) - len(word2))

            for j in range(len(word1)): # both words are now of same length
                if word1[j] != word2[j]: # As soon as non-matching char is found
                    if char_order[word1[j]] > char_order[word2[j]]:
                        return False
                                        # Imp: if the first non-matching char follo
ws correct order break the inner loop since the two words are in order
                    break

        return True

# Time Complexity = O(nk) + O(d) where n = no. of words and k = max number of chars
in any string, and d = size of dictionary to store order
# Space Complexity = O(d)
```

973. K Closest Points to Origin

```
# Top k - heap
# closest points - max heap (python has min heap so -ve = max heap)
# heap element would be tuple of (dist, x, y) since we have to return x,y point

# note: if heap element is a tuple then heap operations happen by first element of
this tuple (if tie then second element and so on.. )

import heapq
class Solution:
    def kClosest(self, points: List[List[int]], K: int) -> List[List[int]]:
        heap = []
        for x,y in points:
            dist = -(x*x + y*y) # Top k smallest - Max heap
            heapq.heappush(heap, (dist,x,y))

            if len(heap) > K: # When heap size > K -> pop
                heapq.heappop(heap)

        return [[x,y] for dist,x,y in heap]

# Time Complexity = O(N log k)
# Space Complexity = O(k)
```

981. Time Based Key-Value Store

```
# Imp constraint mentioned in question: All the timestamps of set are strictly incr
easing i.e. sorted hence binary search
# Hashmap + Binary Search

class TimeMap:

    def __init__(self):
        self.time_map = defaultdict(list)


    def set(self, key: str, value: str, timestamp: int) -> None:

        self.time_map[key].append([timestamp, value])


    def get(self, key: str, timestamp: int) -> str:
        # Edge condition: key not present
        if key not in self.time_map:
            return ""

        # Edge condition: no lesser or equal timestamp value available in the list
corresponding to key
        if timestamp < self.time_map[key][0][0]:
            return ""


        # Binary search on the list corresponding to key, keeping track of max seen
below or equal to timestamp value
        left = 0
        right = len(self.time_map[key]) - 1
        res = float("-inf")

        while left <= right:
            mid = left + (right-left)//2

            if  self.time_map[key][mid][0] <= timestamp:
                res = max(res, mid) # closest we have seen so far, timestamps are i
n sorted increasing order

                left = mid + 1

            else: # self.time_map[key][mid][0] > timestamp:
                right = mid - 1

        return self.time_map[key][res][1]
```

986. Interval List Intersections

```
# Each list is sorted and non-overlapping

# Overlapping area = [Max of starting points, Min of ending points]
# Overlapping happens: if max of starting points <= min of ending points
# Whichever point ends first, go to the next point in the same list (whether overla
p happens or not)

class Solution:
    def intervalIntersection(self, A: List[List[int]], B: List[List[int]]) -> List
[List[int]]:

        i=j=0
        result = []

        while i < len(A) and j < len(B):
            low = max(A[i][0], B[j][0])
            high = min(A[i][1], B[j][1])
            if low <= high: # result could be [5,5]
                result.append([low,high])

            if A[i][1] < B[j][1]:
                i = i+1
            else:
                j = j+1

        return result

# Time Complexity = O(m+n) where m,n = size of list A and B
# Space Complexity = O(1), if space required by result is not considered, else O(m+
n)
```

994. Rotting Oranges

https://www.youtube.com/watch?v=y704fEOx0s0 (https://www.youtube.com/watch?v=y704fEOx0s0)

```
# This is not simple BFS, as all rotten one will contaminate neighbors parallely (m
ulti-source BFS)
# No need of visited set since fresh ones are converted to rotten

# Count number of fresh oranges and collect list of all rotten oranges + add them a
ll to queue to start multi-source BFS
# while q and fresh count > 0:
# Deque all elements of q in a loop,
# inspect neighbors -  if in bounds and nonrotten, make rotten, add it to q and dec
rement fresh count
# Increment time when all elements of q are dequed in a loop

# Time Complexity = O(mn)
# Space Complexity = O(mn) for queue
```

1004. Max Consecutive Ones III

```
# Sliding window
# keep expanding the right ptr until limit of 0s is reached (left ptr constant)
# In a while loop, keep contracting the left ptr until count of 0s is within limit
(right ptr constant)

class Solution:
    def longestOnes(self, nums: List[int], k: int) -> int:
        left = 0
        answer = 0
        counts = {0: 0, 1: 0}

        for right, num in enumerate(nums):
            counts[num] += 1

            while counts[0] > k:
                counts[nums[left]] -= 1
                left += 1

            curr_window_size = right - left + 1
            answer = max(answer, curr_window_size)

        return answer

# Time Complexity = O(n), each element can be visited at most 2 times, once by left
ptr and once by right ptr = 2n = O(n)
# Space Complexity = O(1)
```

1011. Capacity To Ship Packages Within D Days

https://www.youtube.com/watch?v=ER_oLmdc-nw&t=1s (https://www.youtube.com/watch?v=ER_oLmdc-nw&t=1s)

```
# Binary search between min and max bound of capacity
# Min bound of capacity: max weight among packages since we need a ship of at least
this capacity to ship
# Max bound of capacity: sum of weights of packages since it will ship all pacakges
in one day

# Helper function to find number of days/ships needed to ship all pacakges given ca
pacity of ship

class Solution:
    def shipWithinDays(self, weights: List[int], days: int) -> int:
        l = max(weights)
        r = sum(weights)
        min_cap = r  # initialize with max possible

        # most important snippet
       def canShip(cap):

            ships = 1
            total_capacity =  cap

            for w in weights:
                total_capacity = total_capacity - w

                if total_capacity >= 0:
                    continue

                                # total_capacity < 0
                else:
                    # get a new ship, reset total capacity and substract current we
ight
                                        ships += 1
                    total_capacity = cap
                    total_capacity -= w

            return ships <= days


        while l <= r:
            cap = l + (r-l) // 2
            if canShip(cap):
                min_cap = min(min_cap, cap)
                r = cap - 1
            else:
                l = cap + 1

        return min_cap

# Time Complexity = O(n log m),
# O(log m) time taken to iterate thru upper bound and lower bound of weights in bin
```

```
ary search
# Space Complexity = O(1)
```

1060. Missing Element in Sorted Array

See solution

```
# Build missing list
# i.e. number of elements missing upto each index in the list: nums[index] - nums
[0] - index
# Since this missing list is directly a function of index and constant value at num
s[0], use lambda fn
# else time and space complexity will be O(n)

# Find the first 'index' in the missing list where the missing list element >= k

# Find the kth smallest element: nums[index-1] + k - missing[index -1]

# E.g. [4,7,9,10] k = 1 : ans => 5, if k= 3 : ans => 8

# Array is sorted: binary search

import bisect
class Solution:
    def missingElement(self, nums: List[int], k: int) -> int:

        missing = lambda index : nums[index] - nums[0] - index

        # Edge case: If kth missing number is larger than the last element of the a
rray
        if k > missing(len(nums) - 1):
            return nums[-1] + k - missing(len(nums)- 1)


        left = 0
        right = len(nums) - 1

        while left < right:
            pivot = left + (right-left) // 2

            if missing(pivot) < k:
                left = pivot + 1
            else:
                right = pivot

        return nums[left-1] + k - missing(left - 1)



# Time Complexity  = O(log n)
# Space Complexity = O(1)
```

1099. Two Sum Less Than K

```
# Sort + 2 ptrs

# The pointers are initially set to the first and the last element respectively.
# We compare the sum of these two elements with the target.
# If it is larger than or equal to the target (we want less than target), we decrem
ent the right pointer
# If it is smaller than the target, we keep track of the max value seen below targe
t and increment the left # pointer

class Solution:
    def twoSumLessThanK(self, nums: List[int], k: int) -> int:
        nums.sort()

        left = 0
        right = len(nums) - 1
        max_sum = -1

        while left < right:
            sum = nums[left] + nums[right]
            if sum >= k:
                right = right - 1
            elif sum < k:
                max_sum = max(max_sum, sum)
                left = left + 1

        return max_sum

# Time Complexity = O(n log n) for sorting
# Space Complexity = O(n) for sorting
```

1046. Last Stone Weight

```
# entire problem is getting and removing the largest stone in most efficient way,
O(log N) time -> max heap
# python has min heap, so make all stone wt -ve

# Since we have to remove top 2 heaviest stones, iterate until number of stones >=
2
# remove the two heaviest stones, smash them together, and insert the result back i
nto the heap if it is non-zero
# if a stone is left at last, convert the stone wt back to +ve and return


class Solution:
    def lastStoneWeight(self, stones: List[int]) -> int:

        # Make all the stones negative *in place*, to keep the space complexity of
this algorithm at O(1)
        for i in range(len(stones)):
            stones[i] *= -1

        # Heapify all the stones
        heapq.heapify(stones)

        while len(stones) >= 2:
            stone_1 = heapq.heappop(stones) # min of -ve = max of +ve
            stone_2 = heapq.heappop(stones)
            if stone_1 != stone_2:
                heapq.heappush(stones, stone_1 - stone_2)

        # Check if there is a stone left to return. Convert it back  to positive
        return -heapq.heappop(stones) if stones else 0


# Time complexity : O(N log N), heapify operation is O(N), heappop() is log (N)
# Space complexity = O(1) In Python, converting a list to a heap is done in-place,
requiring O(1) auxillary space (Jave requires O(N))
```

1048. Longest String Chain

https://www.youtube.com/watch?v=7b0V1gT_Tlk (https://www.youtube.com/watch?v=7b0V1gT_Tlk)

```
# sort based on string len
# Delete a char and lookup using hash map (word: length of chain)
# To avoid duplication, cache the results of sub-problems/sub strings

class Solution:
    def longestStrChain(self, words: List[str]) -> int:
        from collections import defaultdict
        # cache len of chain
        cache = defaultdict(int) # word: length of chain
        # initiate data
        data = sorted(words, key=lambda w: len(w))

        for word in data:
            # from word, the len of chain is 1 at 1st
            cache[word] = 1
            for index in range(len(word)):
                # construct predecessor
                predecessor = word[:index] + word[index + 1:]
                if predecessor in cache:
                    cache[word] = max(cache[word], cache[predecessor] + 1)

        return max(cache.values())
```

1197. Minimum Knight Moves

```
# usual BFS from (0,0)
# keep track of number of moves
```

1091. Shortest Path in Binary Matrix

https://leetcode.com/problems/shortest-path-in-binary-matrix/discuss/312827/Python-Concise-BFS (https://leetcode.com/problems/shortest-path-in-binary-matrix/discuss/312827/Python-Concise-BFS)

```
# Shortest path => BFS
# Usual BFS
# Edge case: Check start and end node to be 0, if not return -1
```

1143. Longest Common Subsequence

https://www.youtube.com/watch?v=Ua0GhsJSlWM&t=630s (https://www.youtube.com/watch?v=Ua0GhsJSlWM&t=630s)

Number of subsequences possible in string of len L = 2 ^L

```
# Bottom-up DP

class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        dp = [[0 for j in range(len(text2) + 1)] for i in range(len(text1) + 1)]

        for i in range(len(text1) - 1, -1, -1):
            for j in range(len(text2) - 1, -1, -1):
                if text1[i] == text2[j]:
                    dp[i][j] = 1 + dp[i + 1][j + 1]  # 1 + diagonal value
                else:
                    dp[i][j] = max(dp[i][j + 1], dp[i + 1][j])    # max of right an
d own

        return dp[0][0]

# Time and Space Complexity = O(len of string 1 * len of string 2)
```

1249. Minimum Remove to Make Valid Parentheses

```
# 1: extra ')' - when no complement exists in stack  and 2: extra '(' remaining in
stack at the end

class Solution:
    def minRemoveToMakeValid(self, s: str) -> str:
        indexes_to_remove = set()
        stack = []

        # Get indices of all extra ')' and '('
        for i, c in enumerate(s):
            if c not in "()":
                continue

            if c == "(":
                stack.append(i)
            elif c == ")":
                if len(stack) == 0: # extra ')'
                    indexes_to_remove.add(i)
                else:
                    stack.pop()

        indexes_to_remove = indexes_to_remove.union(set(stack)) # extra '(' remaini
ng in stack

        result = []
        for i, c in enumerate(s):
            if i not in indexes_to_remove:
                result.append(c)
        return "".join(result)

# Time Complexity = O(n)
# Space Complexity = O(n)
```

1428. Leftmost Column with at Least a One

```
# Start from top-right
# encounter 0 => move down, encounter 1=> move left
# Edge case : if all elements are 0

class Solution:
    def leftMostColumnWithOne(self, binaryMatrix: 'BinaryMatrix') -> int:
        rows,cols = binaryMatrix.dimensions()

        current_col = cols - 1
        current_row = 0

        while current_row < rows and current_col >= 0:
            if binaryMatrix.get(current_row, current_col) == 0:
                current_row = current_row + 1
            else:
                current_col = current_col - 1

        # if all elements are zero
        if current_col == cols - 1:
            return -1
        else:
            return current_col + 1 #Imp: +1

# Time Complexity = O(row + cols)
# Space Complexity = O(1)
```

1268. Search Suggestions System

https://www.youtube.com/watch?v=D4T2N0yAr20&t=790s (https://www.youtube.com/watch?v=D4T2N0yAr20&t=790s)

```
# Sort + 2 ptr approach

# Step 1: sort list of products so as to make them in lexicographical order
# Step 2: left ptr points to begin product and right ptr points to last product
# Step 3: Iterate thru every char in search word and adjust left and right ptr to p
oint to valid window
# i.e. move left and right ptr if l <= r and (length of product <= index of searchW
ord or char of searchWord do not match product's char )
# Step 4: From this valid window between left and right ptr, form the result list
a) if # of suggestions > 3 b) if # of suggestions <= 3

class Solution:
    def suggestedProducts(self, products: List[str], searchWord: str) -> List[List
[str]]:

        # sort
        products.sort()
        l = 0
        r = len(products) - 1

        result = []

        # iterate thru every char of searchWord
        for i in range(len(searchWord)):
            c = searchWord[i]

           # find valid window
            while l <= r and (len(products[l]) <= i or products[l][i] != c):
                l = l + 1
            while l <= r and (len(products[r]) <= i or products[r][i] != c):
                r = r - 1


            no_of_suggestions = r - l + 1

            if no_of_suggestions > 3:
                result.append([products[l], products[l+1], products[l+2]])

            else:
                result_temp = []
                for j in range(no_of_suggestions):
                    result_temp.append(products[l+j])

                result.append(result_temp)

        return result

# Time Complexity = O(n log n) + O(n + m) where n is size of products  and m is siz
e of searchword
# Space Complexity = O(n) for sorting
```

1293. Shortest Path in a Grid with Obstacles Elimination

```
# BFS
# While exploring all paths from source to destination, keep track of remaining "qu
ota" of obstacles
# If the path has remaining quota of obstacles >= 0 then only append it to queue
# For removing duplication, only add the next state (row, col, obstacles remaining)
in queue if this state has not been seen already
# keep track of steps along the way



class Solution:
    def shortestPath(self, grid: List[List[int]], k: int) -> int:
        rows, cols = len(grid), len(grid[0])
        target = (rows - 1, cols - 1)

        # (row, col, remaining quota to eliminate obstacles)
        state = (0, 0, k)
        # (steps, state)
        queue = deque([(0, state)])
        seen = set([state])

        while queue:
            steps, (row, col, k) = queue.popleft()

            # we reach the target here
            if (row, col) == target:
                return steps

            # explore the four directions in the next step
            for new_row, new_col in [(row, col + 1), (row + 1, col), (row, col -
1), (row - 1, col)]:
                # if (new_row, new_col) is within the grid boundaries
                if (0 <= new_row < rows) and (0 <= new_col < cols):
                    quota_remaining = k - grid[new_row][new_col]
                    new_state = (new_row, new_col, quota_remaining)
                    # add the next move in the queue if it qualifies
                    if quota_remaining >= 0 and new_state not in seen:
                        seen.add(new_state)
                        queue.append((steps + 1, new_state))

        # did not reach the target
        return -1

# Time Complexity = O(nk) where n = no. of cells in grid and k = quota of obstacles
# each cell can be visited with k different quotas of obstacles

# Space Complexity = O(nk)
```

1306. Jump Game III

```
# BFS from start index and in bounds on both sides of array
# use visited set to avoid duplication and cycles

class Solution:
    def canReach(self, arr: List[int], start: int) -> bool:

        q = deque()
        q.append(start)
        visited = set()
        visited.add(start)

        while q:
            cur_index = q.popleft()

            if arr[cur_index] == 0:
                return True


            for jump_index in [(cur_index + arr[cur_index]), (cur_index - arr[cur_i
ndex])]:
                if 0 <= jump_index < len(arr) and jump_index not in visited:
                    q.append(jump_index)
                    visited.add(jump_index)

        return False

# Time Complexity: O(N)
# Space complexity: O(N)
```

1539. Kth Missing Positive Number

```
# sorted array and only +ve integers
# Brute force : put the nums in hashset and start looking in this hashset from 1 to
until kth number is not found

# another solution involves cyclic sort but time complexity would be O(n)

# Let's say input array is: [2,3,4,7,11],
# Compare it with an array with no missing +ve integers: [1,2,3,4,5]
# The number of missing integers is a simple difference between the corresponding e
lements of these two arrays
# e.g. Before 2 in input array, there is 2 - 1 = 1 missing integer. Before 7, there
are 7 - 4 = 3 missing integers
# The number of positive integers which are missing before the arr[idx] is equal to
arr[idx] - idx - 1

# Binary search:
# The loop will break when left = right + 1
# i) The number of integers missing before arr[right] is arr[right] - right - 1
# ii) If no number was missing then arr[right] would be existing arr[right] + k
# so ii) - i) = arr[right] + k - (arr[right] - right - 1) = k + right + 1 = k+left
(since left = right+1)


class Solution:
    def findKthPositive(self, arr: List[int], k: int) -> int:
        left, right = 0, len(arr) - 1
        while left <= right:
            pivot = left +(right - left) // 2

            # If number of positive integers which are missing before arr[pivot] is
less than k --> continue to search on the right
            if arr[pivot] - pivot - 1 < k:
                left = pivot + 1

            # Otherwise, go left (> and =, both conditions)
            else:
                right = pivot - 1

        # At the end of the loop, left = right + 1 and the kth missing is in-betwee
n arr[right] and arr[left].
        # The number of integers missing before arr[right] is arr[right] - right -
1 and the number to return is arr[right] + k - (arr[right] - right - 1) = k + left

        return left + k

# Time Complexity = O( log N)
# Space Complexity = O(1)
```

1570. Dot Product of Two Sparse Vectors

2 Solutions: Solution 2 is better in time complexity

```
# Solution 1: List of List (data structure)  + Binary Search (lookup)

class SparseVector:
    def __init__(self, nums: List[int]):
        # Create a list of list [index, value] for non-zero elements
        self.sparse_list = []
        for i in range(len(nums)):
            if nums[i] != 0:
                self.sparse_list.append([i, nums[i]])

    # Return the dotProduct of two sparse vectors
    def dotProduct(self, vec: 'SparseVector') -> int:

        # In python, = just creates a new variable that shares the reference of the
original object
                sparse_list1 = self.sparse_list
        sparse_list2 = vec.sparse_list

                res = 0

        # The shorter list should be iterated on bcz it will save iteration cycles
compared to longer list
        if len(sparse_list1) > len(sparse_list2):
            sparse_list1, sparse_list2 = sparse_list2, sparse_list1
        for i in range(len(sparse_list1)):
            left = 0
            right = len(sparse_list2) - 1
            # Binary search to look up index in the shorter list
            # Note both lists are sorted by index as that's how they were built
            # Hashmap solution has O(1) lookup so no binary search needed
            while left <= right:
                mid = (left + right) // 2
                if sparse_list2[mid][0] == sparse_list1[i][0]:
                    res += sparse_list2[mid][1] * sparse_list1[i][1]
                    break
                if sparse_list2[mid][0] < sparse_list1[i][0]:
                    left = mid + 1
                else:
                    right = mid - 1
        return res



# Time Complexity of __init__():O(L1 + L2) where L1, L2 = length of first and secon
d list resp.
# Space Complexity of __init(): O(l1 + l2) where l1,l2 = length of longer and short
er sparse list resp

# Time Complexity of dotProduct(): l2 * log (l1) [binary search]
# Space Complexity of dotProduct(): O(1) [binary search]
```

```
# Solution 2: Hashmap (data structure): Lookup is already O(1) hence binary search
is not needed
# Note: some interviewers do not prefer hashmap based solution bcz of practical res
trictions in creating such a big hashmap

class SparseVector:
    def __init__(self, nums: List[int]):
        # Create hashmap of index:value for non-zero elements
        self.d = {i: x for i, x in enumerate(nums) if x}

    # Iterate on shorter vector to save iteration cycles compared to longer vector
    # dict.get(key,0) returns 0 when key is not found
    def dotProduct(self, vec: 'SparseVector') -> int:
        dot_product = 0
        if len(self.d) <= len(vec.d):
            for key in self.d:
                dot_product = dot_product + self.d[key]*vec.d.get(key, 0)
        else:
            for key in vec.d:
                dot_product = dot_product + vec.d[key]*self.d.get(key, 0)

        return dot_product

# Time Complexity of __init__():O(L1 + L2) where L1, L2 = length of first and secon
d list resp.
# Space Complexity of __init(): O(l1 + l2) where l1,l2 = length of longer and short
er sparse list resp

# Time Complexity of dotProduct(): l2 (hashmap lookup is O(1))
# Space Complexity of dotProduct(): O(1)
```

1631. Path With Minimum Effort

https://www.youtube.com/watch?v=XQlxCCx2vI4 (https://www.youtube.com/watch?v=XQlxCCx2vI4)

1762. Buildings With an Ocean View

```
# Next greater to the right
# If for any element there is no next greater to the right then that element has oc
ean view


class Solution:
    def findBuildings(self, heights: List[int]) -> List[int]:

        stack = []
        for i, ht in enumerate(heights):
            while stack and ht >= stack[-1][0]:
                stack.pop()
            stack.append([ht, i])

        res = [i  for h, i in stack]
        return res
```

1838. Frequency of the Most Frequent Element

```
# Similar to 424. Longest Repeating Character Replacement

# Sort + Sliding Window

# Its basically asking from the current sliding window e.g. [1, 2,  4], how much do
you need to convert it in [4, ,4 ,4], you would need
5 (nums[r] times the window length (r - l + 1) - the sum of the current window).
# So nums[r]*(r-l+1) - sum, if this is affordable with budget k (nums[r]*(r-l+1) -
sum <= k) then a frequency can be achieved with an amount equal to the window lengt
h. (e.g. 1 + 2 + 4 = 7, 4 + 4 + 4 = 12, you need 5 to convert it, which is within b
udget)

class Solution:
    def maxFrequency(self, nums: List[int], k: int) -> int:
        nums.sort()
        left = 0
        curr_sum = 0

        ans = 0
        for right in range(len(nums)):
            curr_sum += nums[right]

            if (right - left + 1) * nums[right] - curr_sum <= k:
                ans = max(ans, right - left + 1)


            if (right - left + 1) * nums[right] - curr_sum > k:
                curr_sum -= nums[left]
                left += 1

        return ans


# Time complexity: O(n·logn)
# Space Complexity = O(n), python's in-built sort takes O(n) space
```